Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik

Bachelorarbeit

Migration von Subversion nach Mercurial und Einsatz dezentraler Versionskontrolle in Unternehmen

Autor:

Christoph Mewes

1. August, 2011

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake
Dipl.-Inform. Thomas Thüm

Institut für Technische und Betriebliche Informationssysteme

Mewes, Christoph:
Migration von Subversion nach Mercurial und Einsatz dezentraler Versionskontrolle in Unternehmen
Bachelorarbeit, Otto-von-Guericke-Universität Magdeburg, 2011.

Inhaltsangabe

Diese Arbeit beschäftigt sich mit den grundlegenden Unterschieden zwischen zentralen und dezentralen Versionskontrollsystemen am Beispiel von Subversion und Mercurial und geht dabei besonders auf die Anforderungen im Unternehmenseinsatz ein.

Nach einem Vergleich der Systeme wird erläutert, wie eine konkrete Migration von Subversion zu Mercurial gestaltet werden kann. Den Abschluss bilden Tipps und Tricks zum Umgang mit Mercurial sowie ein Ausblick auf weitere Möglichkeiten, das System einzusetzen.

Bei der Migration mussten eine Reihe von Problemen gelöst werden, insbesondere was die Struktur der einzelnen Projekte, als auch die verwendeten Arbeitsabläufe betrifft. Die dafür nötigen Änderungen zahlten sich bereits wenige Wochen nach ihrer Umsetzung in Form von besser dokumentierten und strukturierten Projekten aus, die häufiger und mit weniger Aufwand aktualisiert wurden. Gleichzeitig begünstigte der Umstieg eine automatisierte Veröffentlichung der Open Source Projekte im Internet.

Subversion gilt als Industriestandard (vgl. [Col09]) und etabliert, wohingegen Mercurial in Unternehmen noch kaum Beachtung findet, im Bereich von Open Source Software jedoch zunehmend an Bedeutung gewinnt. So setzen einige große Projekte wie Mozilla, NetBeans, OpenJDK, OpenOffice, Opera (Dragonfly), Python, das W3C und weitere auf Mercurial als Versionskontrollsystem.[1]

[1] http://mercurial.selenic.com/wiki/ProjectsUsingMercurial

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface, Programmierschnittstelle
CMS	Content Management System
DVCS	Dezentrales Versionskontrollsystem
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated Development Environment
IT	Informationstechnik
MVC	Model-View-Controller
PHP	Hypertext Preprocessor, eine auf Entwicklung von Webanwendungen spezialisierte Programmiersprache
RCS	Revision Control System
SCCS	Source Code Control System
SSH	Secure Shell
UI	User Interface
VCS	Versionskontrollsystem

Glossar

AddOn	Eigenständige Komponente für SallyCMS, die Funktionen im Backend und Frontend bereitstellen kann.
Annotation	Eine Annotation ist die Anzeige, welcher Autor zuletzt welche Zeile einer bestimmten Datei geändert hat; auch bekannt als „Blame“.
Arbeitskopie	Kopie einer bestimmten Revision, an der der Entwickler Änderungen vornimmt, um sie als neuen Commit in das Repository zu übertragen.
Backend	Verwaltungsoberfläche für Webseiten, in der Redakteure die Inhalte einpflegen können.
Blame	siehe Annotation
Branch	Ein Branch entsteht, wenn ein Commit mehr als ein Child besitzt.
Bug	Fehler in einem Programm.
Child	Der Nachfolger eines Commits; Commits können beliebig viele Children besitzen.
Commit	Ein Eintrag in der Versionsgeschichte eines Projekts; enthält Delta, Autor, Zeitpunkt und ggf. weitere Informationen zu einer Änderung an einer Menge von Dateien.
Cronjob	Ein in regelmäßigen Abständen automatisch ausgeführtes Programm.
Delta	Änderung am Inhalt oder den Attributen einer Datei.
Diff	Enthält die Änderungen zwischen zwei Revisionen
Fork	Aus technischer Sicht ein Klon; wird häufig verwendet, wenn Dritte ein Repository klonen.
Frontend	Öffentlich sichtbarer Teil einer Webseite; stellt die im Backend eingegebenen Inhalte für Besucher dar.
Head	Bezeichnet den zuletzt erstellten Commit (die aktuellste Revision); wird auch als „tip“ oder „HEAD“ bezeichnet.

Hook	Funktion oder Programm, die von Mercurial beim Eintreten eines Ereignisses automatisch ausgeführt wird.
Hunk	Ein Teil eines Diffs, enthält meist noch eine bestimmte Menge an Zeilen davor und danach, die als Kontext dienen.
Interface	Schnittstelle, die ein AddOn anbietet, um mit ihm zu interagieren.
Klon	Eine vollständige und exakte Kopie eines Repositories, meist mit dem Englischen Begriff „Clone“ bezeichnet.
Merge	Ein Merge entsteht, wenn zwei oder mehr Branches zusammengeführt werden; ein Merge kann eine Mergecommit erzeugen, der mehrere Parents besitzt.
Modul	Inhaltstragendes Element einer Seite, zum Beispiel ein Fließtext, eine Galerie oder ein Formular.
Parent	Der Vorgänger eines Commits; jeder Commit hat mindestens einen Parent (der Parent des ersten Commits existiert nur virtuell).
Patch	Textdatei, die einen oder mehreren Diffs sowie eine Beschreibung und ggf. weitere Informationen wie den Autor
Pull	Empfangen von Commits aus einem anderen Repository, meist über HTTP, um beide zu synchronisieren.
Push	Senden von Commits an ein anderes Repository, meist über HTTP, um beide zu synchronisieren.
Refactoring	Vorgang, bei dem die Struktur von Quellcode verändert wird, ohne das externe Programmverhalten zu ändern. Ziele sind u.a. bessere Wartbarkeit oder Performance.
Repository	Spezielle Datenbank, die alle Commits eines Projekts enthält. Kann als gerichteter Graph von Commits aufgefasst werden.
Revision	Eine bestimmte Version des Projekts, die durch das Anwenden einer Reihe von Commits entstanden ist.
sed	Stream Editor, ein Unix-Programm zum Manipulieren von Dateien.
Stdout	Name des Datenstroms für die Standardausgabe in der Programmiersprache C.
Symlink	Kurzname für einen „symbolischen Link“; eine Verknüpfung in einem Dateisystem (Dateien und Verzeichnisse), die auf eine andere Datei oder ein anderes Verzeichnis verweist (vgl. [Wik11b]).

Tag	Alternativname für eine bestimmte Revision
Template	Designvorlage einer einzelnen Unterseite in SallyCMS-Projekten.
Trunk	Übliche Bezeichnung für die Hauptentwicklungslinie eines Projekts.
Wiki	System zur Verwaltung von Inhalten, das darauf aufgebaut ist, dass jeder die Inhalte bearbeiten kann.

1. Einleitung

Versionskontrolle ist der Vorgang, Änderungen an Dokumenten aufzuzeichnen und damit nachvollziehbar zu machen. Dazu werden neben der eigentlichen Änderung auch der Autor sowie das Datum und die Uhrzeit (sowie ggf. weitere Metadaten) in einer Datenbank gespeichert. Insbesondere im IT-Bereich kommt Versionskontrolle zum Einsatz, um die Entwicklung von Quelltexten oder Spezifikationen zu dokumentieren, jedoch sind auch Anwendungen in Wikis oder Content-Management-Systemen (CMS) möglich.

Neben der eigentlichen Aufzeichnung bieten derartige Systeme oft noch weitere Funktionen:

- Wiederherstellung des Zustands eines Projekts zu jedem beliebigen Zeitpunkt.
- Annotieren von Dokumenten, um pro Zeile den jeweiligen Autor zu ermitteln.
- Koordinierung gleichzeitiger Änderungen durch mehrere Benutzer.
- Möglichkeit, ein Projekt in verschiedenen Zweigen parallel zu bearbeiten.
- Integritätsprüfung durch kryptografisch sichere Prüfsummen.

Ein Versionskontrollsystem kommt vor allem bei der Entwicklung von Software zum Einsatz um die schon angesprochenen Quelltexte zu versionieren. Dies ist sowohl dem Umstand geschuldet, dass sein Einsatz fundierte Kenntnisse im Umgang mit ihm erfordert, als auch, dass die zugrunde liegende Technik auf reine Textdateien optimiert ist: So ist ein Vergleich zweier Bilddateien aufgrund ihres binären Aufbaus nicht ohne weitere Zusatzsoftware möglich, während zum Ermitteln der Unterschiede von reinem Text die geänderten Zeilen genügen.

Hintergrund

Diese Arbeit entstand auf Basis einer Migration von Subversion auf Mercurial. Dabei wurde die Entwicklung von Websites durch ein kleines Team von neun Entwicklern nach halbjähriger Vorbereitung und Evaluation innerhalb einer Woche umgestellt. Die Umstellung betraf dabei sowohl die Entwicklerrechner als auch den internen Server, auf dem die Projekte lagerten.

Die gesammelten Erfahrungen sollen hier in Form eines Leitfadens zusammengefasst werden. Damit sollen andere Unternehmen ermutigt werden, eine Migration zu wagen, ohne die gleichen Fehler zu begehen oder ähnliche Probleme erneut lösen zu müssen.

Gliederung der Arbeit

Die vorliegende Arbeit ist in fünf Abschnitte aufgeteilt, wobei die ersten beiden in das Thema einführen und insbesondere für Leser gedacht sind, die sich erst in das Thema und die Motivation einlesen wollen. Die folgenden beiden Abschnitte sind dagegen so verfasst, dass sie eine bereits gefasste Entscheidung mit konkreten Anleitungen zur Migration unterstützen.

Bevor auf die Unterschiede der Systeme und die konkrete Migration eingegangen wird, wird in Kapitel 2 auf der nächsten Seite kurz die Entwicklung der Versionskontrolle am Beispiel freier Software (vgl. [Fre08]) skizziert. Dies bietet gleichzeitig den Rahmen, in dem die zentralen Begriffe eingeführt werden, die für das Verständnis der nachfolgenden Kapitel erforderlich sind.
Nach dem historischen Abriss werden in Kapitel 3 auf Seite 11 grundlegende Unterschiede zwischen zentralen und dezentralen Systemen behandelt. Dazu werden erst das Konzept, dann eine jeweils konkrete Implementierung und zuletzt die GUIs miteinander verglichen. Exemplarisch werden dabei Subversion (zentral) und Mercurial (dezentral) behandelt. Im Kontext des Vergleichs wird direkt auf die jeweiligen Vor- und Nachteile eingegangen, die den Wechsel von Subversion nach Mercurial beschreiben.

Daran anschließend wird in Kapitel 4 auf Seite 31 eine konkrete Migration beschrieben, bei der eine Reihe von gleichartigen Projekten konvertiert und Arbeitsabläufe wie das Anlegen von Projekten behandelt werden. Dabei werden für die einzelnen Aufgaben jeweils Beispiel-Quellcodes in PHP angegeben, die die verwendeten Algorithmen demonstrieren. Kapitel 5 auf Seite 55 geht dann auf die erweiterten Möglichkeiten ein, mit den Projekten zu arbeiten und diskutiert, wie Änderungen aus Projekten in die Basis zurückgespielt und Projekte nachträglich kontinuierlich aktualisiert werden können.

Den Abschluss bildet Kapitel 6 auf Seite 69, das die Ergebnisse der vorangegangenen Kapitel zusammenfasst, bewertet und einen Ausblick auf weitere Techniken bietet.

2. Hintergrund

Versionskontrollsysteme existieren seit den frühen 1970er Jahren. Im Lauf der Zeit haben sich die Systeme stetig weiterentwickelt, nicht zuletzt um mit den immer größer werdenden Projekten, die mit ihnen verwaltet wurden, mitzuhalten. In diesem Kapitel soll ihre Geschichte skizziert werden. Dies schafft einen Überblick über ihre Entwicklung und bietet gleichzeitig eine Gelegenheit, die für das Verständnis der nachfolgenden Kapitel nötigen Begriffe und Konzepte einzuführen[1].

Die Entwicklung von Versionskontrollsystemen hat bisher über 35 kommerzielle und freie Programme hervorgebracht (vgl. [Wik11a]), von denen im Folgenden jedoch nur die freien (vgl. [Fre08]) Vertreter betrachtet werden sollen, um nicht den Rahmen der Einführung zu sprengen. Die kommerziellen Vertreter bieten oft noch speziellere Funktionen wie eine tiefer gehende Integration (vgl. [Mic11]) in die vorhandene Infrastruktur, Rechteverwaltung oder vorgefertigte Reports (vgl. [Mic06]).

Die folgenden Abschnitte beschreiben in chronologischer Reihenfolge die drei primären Typen von Versionskontrollsystemen.

2.1 Einzelplatzsysteme

Die ersten Vertreter von Versionskontrollsystemen arbeiteten auf einzelnen Rechnern und boten noch keine Netzwerkunterstützung. Sie verwendeten jedoch bereits Konzepte, die auch heute noch aktuell sind.

2.1.1 SCCS - Source Code Control System

Die erste Software zur Versionskontrolle entstand 1972 in den Bell Laboratories und wurde von Marc Rochkind entwickelt. Das Source Code Control System (SCCS, vgl. [Roc75]) genannte Programm verwaltete die Änderungen an einzelnen Dateien (statt wie heute üblich von einem ganzen Verzeichnisbaum) und konnte die Log-Informationen bereits mit einer einfachen Prüfsumme (der Summe über alle geänderten Zeichen) versehen, um versehentliche oder mutwillige Änderungen an der

[1] Für eine Kurzbeschreibung der wichtigsten Begriffe sei auf das Glossar verwiesen.

Versionsgeschichte zu bemerken (vgl. [All], Abschnitt 10.2). Die Versionsgeschichte einer Datei `X` wurde in einer weiteren Datei `X,v` abgelegt. Dabei bezeichnet man `X` auch als *Arbeitskopie* und `X,v` als *Repository* (auch wenn diese Begriffe erst später geprägt wurden).

Definition 2.1. *Das **Repository** enthält alle Commits eines Projekts und kann als spezielle Form einer Datenbank betrachtet werden. (vgl. [Mas06], S. 19)*

Da in SCCS Dateien einzeln versioniert werden, existiert ein Repository pro Datei. In späteren Systemen wie CVS enthält ein einzelnes Repository die Daten aller im Projekt enthaltenen Dateien.

Definition 2.2. *Die **Arbeitskopie** wird aus einer beliebigen Revision im Repository erstellt und ist der Satz an Dateien, mit dem der Entwickler arbeitet. Die Arbeitskopie kann jederzeit auf die letzte (oder eine beliebige andere) Revision zurückgesetzt werden, wobei eventuelle Änderungen verworfen werden. (vgl. [Mas06], S. 22)*

Abbildung 2.1 veranschaulicht die Struktur von Arbeitskopie und Repository. Jede Datei enthält eine eigene Arbeitskopie und wird unabhängig von den anderen versioniert. Die Pfeile demonstrieren den Check-in-Prozess, bei dem die Änderungen in das Repository übertragen werden.

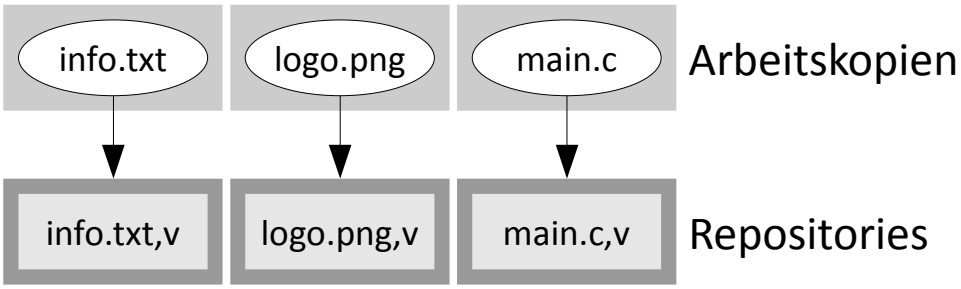

Abbildung 2.1: Arbeitskopien und Repositories in SCCS und RCS

Die einzelnen versionierten Änderungen wurden als *Check-in* bezeichnet, die häufig auch *Commit* oder *Changeset* genannt werden.

Definition 2.3. *Ein **Commit** ist ein Eintrag in die Versionsgeschichte eines Projekts und enthält die Änderungen (Delta) am Inhalt oder den Attributen einer oder mehrerer Dateien in der Arbeitskopie gegenüber dem letzten Projektzustand im Repository, zusammen mit dem Autor, dem Zeitpunkt, einer Commitnachricht und ggf. noch weitere Informationen. (vgl. [Cha11], S. 8)*

Für gewöhnlich sind Commits unveränderlich. Commits werden vom Autor angelegt, wenn die gemachten Änderungen so inhaltlich abgeschlossen sind, um sie permanent zu speichern (und damit meist auch anderen Entwicklern zur Verfügung zu stellen). Die Commitnachricht dient dabei dazu, die Änderungen zu beschreiben.

Definition 2.4. *Eine **Revision** entsteht beim Anlegen eines neuen Commits. Je nach System wird dazu ein Zähler inkrementiert oder eine für das gesamte Projektarchiv eindeutige Prüfsumme berechnet. Revisionen bezeichnen also den Projektzustand nach dem Anwenden einer Reihe von Commits. (vgl. [Mas06], S. 18)*

Um die jeweils letzte Revision immer unter dem gleichen Namen ansprechen zu können, existiert in den meisten Systemen ein Aliasname für sie.

Definition 2.5. *Der **Head** ist der Commit, der keine Nachfolger (Child) hat. (vgl. [Ott09], S. 2)*

SCCS legte damit die Grundlagen für alle folgenden Versionskontrollsysteme. Die verwendeten Datenstrukturen finden noch heute Verwendung.

2.1.2 RCS - Revision Control System

Auf SCCS folgte circa 10 Jahre später das von Walter F. Tichy entwickelte Revision Control System (RCS, vgl. [TT85])[2], das sowohl konzeptionell als auch aus Sicht der Funktionalität mit SCCS vergleichbar ist, allerdings keine Prüfsumme verwendet und damit anfälliger für unbemerkte Änderungen in der Versionsgeschichte ist.

Beide Programme boten damit keine Funktionen für das verteilte oder parallele Arbeiten an Dokumenten. Sie werden jedoch noch heute eingesetzt, um zum Beispiel einzelne Konfigurationsdateien zu versionieren.
Um Änderungen zwischen Rechnern zu übertragen, boten sich bereits damals *Patches* an, die die einzelnen *Diffs* übertragbar machten.

Definition 2.6. *Ein **Diff** ist der Unterschied zwischen zwei beliebigen Revisionen. Diffs drücken Änderungen zeilenweise mit Angabe hinzugefügter und gelöschter Zeilen aus. Ein Diff muss jedoch nicht zwischen zwei Revisionen ermittelt werden, sondern kann auch zwischen zuletzt erstellten Commit (dem sog. „Head“) und aktuellem Projektstand berechnet werden (und fungiert damit als Commit-Vorschau). (vgl. [Ott09], S. 3)*

Diffs können auch ohne Versionskontrollsystem zwischen beliebigen Dateien erstellt werden. Das dafür notwendige Programm `diff` ist für viele Plattformen verfügbar.

Definition 2.7. *Ein **Patch** enthält neben einem Diff (der beliebig viele Dateien umfassen kann) meist noch weitere Informationen wie den Autor, das Datum und die Commitnachricht. Da es sich um einfache Textdateien handelt, können sie neben der automatischen Verarbeitung durch ein VCS auch von Menschen gelesen werden. (vgl. [Mas06], S. 95)*

Patches können ebenfalls ohne Versionskontrollsystem verwendet werden. Das gleichnamige Programm ermöglicht das automatische Anwenden eines Patches auf die von ihm betroffenen Dateien.
Patches ermöglichen es, Änderungen auf einfache Weise über Rechnergrenzen hinweg zu verteilen, indem sie zum Beispiel per E-Mail verschickt oder auf Datenträger kopiert werden.

[2] http://www.gnu.org/software/rcs/rcs.html

2.2 Client-Server-Systeme

Durch die zunehmende Vernetzung von Rechnern und den steigenden Umfang von Projekten (sowohl in Hinblick auf Daten als auch die beteiligten Mitarbeiter) wurde es bald nötig, die Übertragung von Änderungen zwischen Rechnern besser zu automatisieren, um einen koordinierten Projektablauf zu ermöglichen. Dies führte zur Entwicklung eines neuen Versionskontrollsystems.

2.2.1 CVS - Concurrent Versions System

Mitte der 1980er Jahre folgte das Concurrent Versions System (CVS, vgl. [Gru86]) von Dick Grunes, das noch heute weitverbreitet ist. CVS begann als Sammlung von Scripts und entwickelte sich bis zum Ende der 80er Jahre zu einer eigenständigen Software (vgl. [Pri05]). Erstmals wurden ganze Verzeichnisse versioniert.

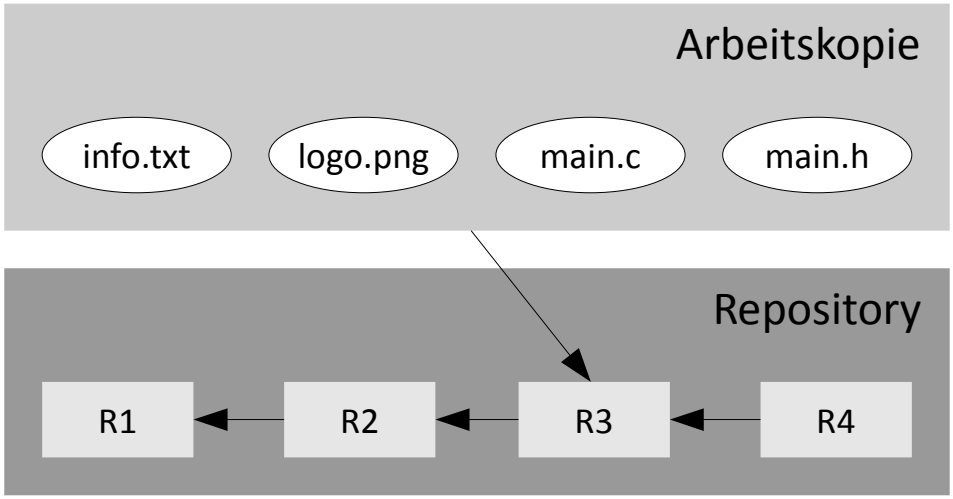

Abbildung 2.2: Arbeitskopie und Repository in CVS und späteren Systemen

In Abbildung 2.2 wird gezeigt, dass es für die Menge aller Dateien nur noch eine Arbeitskopie gibt (pro Client, wobei Clients auch mehrere Arbeitskopien parallel erhalten können), die als Gesamtes betrachtet wird, wenn ein Commit stattfinden soll. Gleichzeitig gibt es auch nur noch ein Repository für alle Dateien. Der Pfeil auf `R3` verdeutlicht, dass die Arbeitskopie nicht notwendigerweise auf der letzten Revision basieren muss, sondern auf jeden Projektstand zurückgesetzt werden kann.

Commits werden an einen zentralen Server geschickt. Damit eignet es sich bereits wesentlich besser zur Software-Entwicklung, da bereits parallele Arbeitszweige (sog. *Branches*) möglich sind.

Definition 2.8. *Wenn sich die Entwicklung eines Projekts in parallele Zweige aufspaltet, spricht man von Branching. Dies tritt immer dann auf, wenn ein Commit mehr als einen Nachfolger (Child) hat. Von einem Commit können beliebig viele Branches ausgehen. Der Begriff leitet sich von dem üblichen Begriff für die Hauptentwicklungslinie, dem* ***Trunk*** *(dt. Stamm), ab, von dem sich dann die Zweige abspalten. (vgl. [Mas06], S. 30)*

Branches werden oft verwendet, um abseits der Hauptentwicklung an neuen Funktionen zu arbeiten oder ein Refactoring vorzunehmen. Dies bedeutet, dass die Änderungen aus einem Zweig zu einem späteren Zeitpunkt auch wieder in den Trunk zurückfließen müssen. Dieses Verschmelzen von Branches bezeichnet man als *Merge*.

Definition 2.9. *Beim Mergen werden zwei oder mehr Branches zusammengeführt. Der dabei entstehende Commit wird als „Mergecommit“ bezeichnet (vgl. [Mas06], S. 32). Merges können für Änderungen an verschiedenen Stellen einer Datei automatisch durchgeführt werden. Wird die gleiche Stelle einer Datei in mehr als einem Branch geändert, muss dieser* ***Konflikt*** *manuell bereinigt werden. Dabei werden dem Benutzer beide Änderungen sowie die Ausgangsversion der Datei angezeigt und er muss entscheiden, welche der Seite übernommen wird.*

Abbildung 2.3 stellt einen beispielhaften Revisionsgraphen dar, wie er in einem Repository verwaltet werden kann. Der gerichtete Graph enthält dabei auch die Information, dass z. B. Revision `R2` auf Revision `R1` basiert (sie ist der Child von `R1`, wobei `R1` ihr Parent ist).

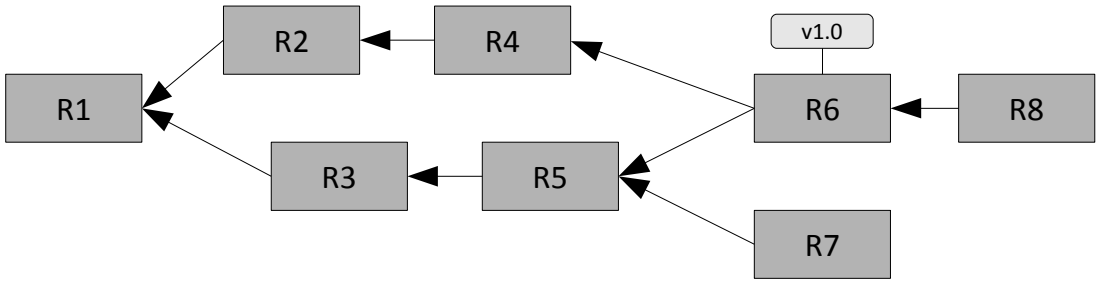

Abbildung 2.3: Beispielhafter Revisionsgraph

Nach `R1` verzweigt sich die Projektgeschichte in zwei Branches (beginnend mit `R2` und `R3`). `R6` ist der Mergecommit, indem beide wieder zusammengeführt werden. Gleichzeitig wird bei `R5` durch `R7` ein neuer Branch eröffnet.

Auch wenn CVS bereits die Geschichte von mehreren Dateien in einem Repository verwaltet, werden die enthaltenen Dateien einzeln versioniert (sodass das Repository mehrere Revisionsgraphen wie in Abbildung 2.3 enthält).

In Abbildung 2.2 auf der vorherigen Seite würde ein neuer Commit einen Branch eröffnen, da sein Parent nicht der Head, sondern `R3` wäre.

Die Verwendung von Branches macht es notwendig, die in Definition 2.5 auf Seite 5 gegebene Definition eines *Heads* zu erweitern:

Definition 2.10. *Der* ***Head*** *ist ein Commit, der keine Nachfolger (Children) hat. Jeder offene (d. h. noch nicht zurückgemergte) Branch besitzt genau einen Head.*

`R7` und `R8` sind in Abbildung 2.3 die beiden Heads. Neue Heads werden damit implizit durch das Anlegen eines neuen Branches erstellt.

CVS unterstützt erstmals auch das Vergeben von *Tags*.

Definition 2.11. *Ein **Tag** ist eine Alternativbezeichnung für eine bestimmte Revision. So kann eine Revision nicht nur unter ihrer Nummer/Prüfsumme adressiert werden, sondern auch über einen sprechenderen Namen wie „v1.0“. (vgl. [Cha11], S. 8)*

Obwohl CVS gravierende Mängel (wie fehlende Transaktionssicherheit) aufwies, entwickelte es sich zum Industriestandard (vgl. [Pri06]). Die Probleme führten schließlich zur Entwicklung von Subversion.

2.2.2 Subversion

Um das Jahr 2000 begann die Entwicklung eines freien Ersatzes von CVS: Subversion sollte die Schwächen von CVS beseitigen, konzeptionell aber keine Neuerungen einführen (vgl. [Mas06], S. 17). Ab Mitte 2001 versionierte Subversion sich selbst, drei Jahre später (im Februar 2004) wurde Version 1.0 veröffentlicht.
Auch wenn es noch heute viele CVS-Nutzer gibt, hat sich Subversion doch klar als Standard für Versionskontrolle etabliert (vgl. [Res09]).

Im Gegensatz zu CVS verwaltet Subversion eine Revisionsgeschichte für alle Dateien. So kann über die Angabe „Revision 5“ der Zustand des ganzen Projekts statt nur einer Datei beschrieben werden. Weiterhin verwendet Subversion einen lokalen Cache, der zwar den Speicherplatzverbrauch der Arbeitskopie verdoppelt, dafür aber Operationen wie Diffs beschleunigt und es ermöglicht, beim Commit nur die Änderungen zu übertragen (und nicht wie bei CVS die vollständigen Inhalte aller geänderten Dateien). [BCS08] geht auf weitere Unterschiede in der Architektur beider Programme ein.

2.3 Dezentrale Systeme

Die steigende Akzeptanz von Open Source führte zur gleichen Zeit bereits zu Problemen mit dem zentralen Ansatz: Das inhärent verteilte Modell von Open Source lässt sich mit zentralisierter Software nur unzureichend abbilden. Das führte zur Entwicklung von Alternativen: 2001 wurde GNU Arch (vgl. [Tai08a]) von Thomas Lord initiiert, das allerdings nach einigen Abspaltungen und dem Rücktritt von Lord als Projektmaintainer (vgl. [Lor05]) bald nicht mehr konkurrenzfähig war und im März 2008 (bis auf sicherheitsrelevante Korrekturen) eingestellt wurde (vgl. [Tai08b]).
Dennoch diente es als Inspiration für eine Reihe von Nachfolgern, darunter Darcs, Monotone, Bazaar (das zum Nachfolger von GNU Arch erhoben wurde), Git und Mercurial.

Allen dezentralen Systemen ist gleich, dass Projekte nicht in Form von Arbeitskopien, sondern als vollständige Kopien des Repositories (sog. „Clones“) verteilt werden. Damit erhält jedes Projektmitglied eine Kopie der gesamten Projektgeschichte und kann unabhängig von anderen arbeiten. Einen zentralen Server gibt es in diesen Systemen nicht mehr. In Abschnitt 3.1.2 auf Seite 13 wird weiter auf die Möglichkeiten eingegangen.

2.4 Zusammenfassung

Obwohl neuere Systeme meist eine Obermenge der Funktionen ihrer Vorfahren enthalten, kommen RCS und SCCS auch heute noch vereinzelt zum Einsatz. So wurde das 1972 entwickelte Programm SCCS Teil des POSIX-Standards und ist noch heute Bestandteil moderner UNIX-Distributionen.
Aus diesem Grund lässt sich auch nur schwer eine genaue zeitliche Einordnung erstellen. Ginge man von Releases aus, wäre RCS aus dem Jahr 1982 eine noch heute aktuelle Software, während die letzte CVS-Version (Version 1.11.23) 2008 veröffentlicht wurde. In Abbildung 2.4 sind daher primär die Jahresangaben für die jeweils ersten Releases von Bedeutung.

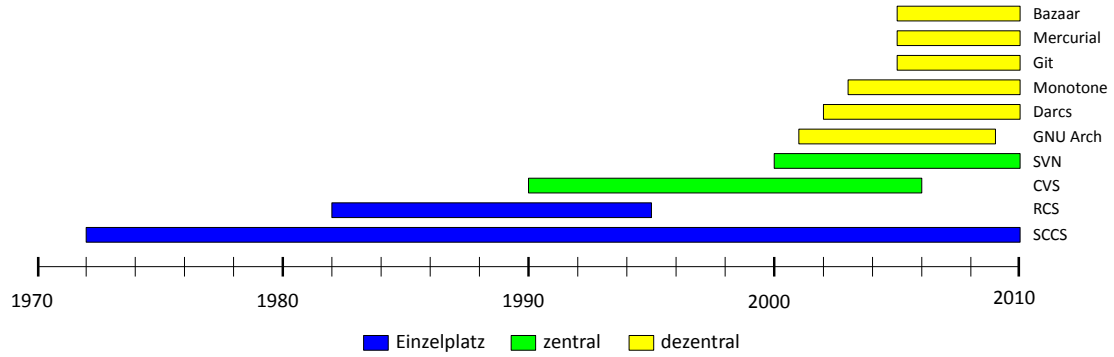

Abbildung 2.4: Verlauf der Entwicklung von Versionskontrollsystemen (vgl. [Mon11])

Ausblick

Allein die Menge dezentraler Systeme zeigt, in welche Richtung sich Versionskontrolle im Moment entwickelt: Seit Subversion wurden, was Open Source angeht, keine nennenswerten neuen, zentralen Ansätze mehr verfolgt, da die dezentrale Konkurrenz bereits eine Obermenge der Funktionen enthält.

> First, we think that this will probably be the „final“ centralized system that gets written in the open source world — it represents the end-of-the-line for this model of code collaboration.[3]

[3] http://blog.red-bean.com/sussman/?p=90

3. Vergleich zwischen zentraler und dezentraler Versionskontrolle

In diesem Kapitel werden die Unterschiede zwischen den beiden Welten, repräsentiert durch Subversion und Mercurial, erläutert. Der Vergleich erstreckt sich über drei Abschnitte, die mit zunehmender Spezialisierung die Systeme vergleichen.

Im ersten Abschnitt werden die theoretischen Grundlagen von zentraler und dezentraler Versionskontrolle gegenübergestellt. Auch wenn sicherlich nicht jedes Versionskontrollsystem die Theorien exakt so umsetzt, wie sie hier beschrieben werden, so stellt dieser Abschnitt jedoch das Fundament dar, auf dem die ihm folgenden Abschnitte aufbauen.
Daran anschließend werden die beiden konkreten Implementierungen verglichen. Aus Benutzersicht handelt es sich dabei um die Kommandozeilen-Werkzeuge von Subversion und Mercurial. Dieser Abschnitt zeigt damit die Fähigkeiten auf, die beide Systeme unabhängig von einer grafischen Oberfläche besitzen.
Den Abschluss bildet ein Vergleich zweier möglicher GUIs. Dabei wurden die Programme TortoiseSVN und TortoiseHg gewählt, da es sich in beiden Fällen um Erweiterungen des Windows Explorers handelt, was einen Vergleich fair macht und aufzeigt, wie sehr sich Unterschiede im Aufbau auf die möglichen Arbeitsabläufe auswirken können. Im gleichen Atemzug wird auf die Verfügbarkeit von Eclipse-Plugins eingegangen, die ähnliche Funktionen wie die Explorer-Erweiterungen bieten, dafür aber in die Entwicklungsumgebung integriert sind.

In den einzelnen Abschnitten werden die Probleme, die zu einem Wechsel des Versionskontrollsystems führten, dargelegt. Dies macht es einfacher ersichtlich, welche Probleme konzeptbedingt und welche implementierungsabhängig sind. Die Ausführungen dienen gleichzeitig als Einführung für die in Abschnitt 4.2 auf Seite 36 gegebenen Ziele der Migration.

3.1 Zentrale und dezentrale Konzepte

Der Hauptunterschied zwischen zentralen und dezentralen Systemen ist der Ort des Repositories: Während es bei zentralen auf einem Server liegt, mit dem sich alle Projektteilnehmer verbinden und synchronisieren müssen, besitzt bei dezentralen Systemen jeder Teilnehmer eine eigene, unabhängige und vollständige Kopie. Dies hat teils gravierende Auswirkungen auf die möglichen Arbeitsabläufe.

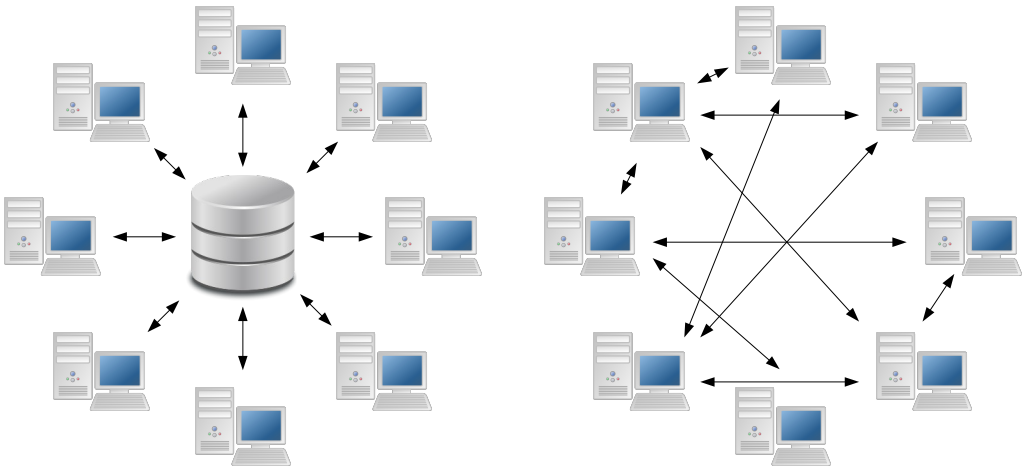

Abbildung 3.1: Schema eines zentralen und dezentralen Versionskontrollsystems

In Abbildung 3.1 wird dieser Unterschied verdeutlicht. Auf der linken Seite ist der Aufbau eines klassischen, zentralen Versionskontrollsystems zu sehen. Alle Clients müssen sich mit dem gleichen Server verbinden, um Änderungen auszutauschen. Auf der rechten Seite ist der Aufbau eines dezentralen Systems illustriert. Hierbei fehlt der Server ganz; die Clients können direkt miteinander kommunizieren, um sich zu synchronisieren. Dies soll nicht darüber hinwegtäuschen, dass mit einer dezentralen Software auch ein wie links dargestelltes Entwicklungsmodell umgesetzt werden kann.

3.1.1 Zentrales Konzept

Um in einem zentralen System Informationen über das Projekt zu erhalten (wie einen Log oder die letzten Änderungen an einer Datei), muss der Teilnehmer mit dem Server verbunden sein.[1]

Bei einem zentralen System kann ein Commit erst stattfinden, wenn eventuelle Änderungen im Repository in die eigene Arbeitskopie übernommen wurden. Man spricht deswegen auch von „Merge Before Commit“. Diese Beschränkung beruht auf dem Gedanken, dass jeder Autor einer Änderung selbst dafür verantwortlich ist, diese kompatibel mit dem aktuellen Projektstand zu halten. Wäre ein Commit ohne vorheriges Mergen möglich, muss unter Umständen eine Person den Merge durchführen, die mit den Änderungen eigentlich nichts zu tun hatte. Das fehlende Know-how kann dann zu einem qualitativ schlechteren Merge führen, bei dem neue Fehler entstehen können.

[1] Heutige Implementierungen verwenden meist einen Cache, um Repository-Daten vorzuhalten. Dieser enthält jedoch im Falle von Subversion nur Teile des Logs und zum Beispiel keine Deltas.

Ein Ausfall des Servers oder eine nicht vorhandene Netzwerkverbindung sind kritisch und bringen nicht nur die eigene Arbeit an einem Projekt zum Erliegen, sondern verhindern auch eine Synchronisation zwischen Projektmitgliedern. Dies wird umso problematischer, je weiter die Clients vom Server entfernt sind: Eine über den Globus verteilte Entwicklung stößt schnell an ihre Grenzen, wenn die Bandbreite des Servers ausgelastet ist oder die Verbindung ganz zusammenbricht.
Neben diesem technischen „Single Point of Failure“ birgt das zentrale Modell besonders für Open Source Projekte Probleme mit sich: Während ein öffentlicher Lesezugriff meist kein Problem darstellt, führt die Frage, wer Änderungen am Projekt vornehmen darf, oft zu langen Diskussionen und unnötig komplexen Regelungen (vgl. [Spa09], [Med11], [Col11], [The], [Gav06]). So müssen sich Mitwirkende erst über einen mehr oder weniger langen Zeitraum aktiver Mitarbeit als „würdig“ erweisen, um direkten Schreibzugriff auf das Repository zu erhalten. In dieser Zeit können sie selbst jedoch die verwendete Software kaum nutzen, um ihre eigene Arbeit angemessen zu verwalten. Patches sind als kleinster gemeinsamer Nenner dann oft das Mittel der Wahl, Änderungen beizusteuern.

Auf der anderen Seite führt ein zentrales Repository meist dazu, dass die einzelnen Mitglieder niemals weiter als eine Revision vom jeweils aktuellsten Projektstand entfernt sein können, da sie vor jedem Commit die zwischenzeitlich stattgefundenen Änderungen anderer bei sich einspielen müssen. Dies setzt natürlich voraus, dass Mitglieder regelmäßig ihre Änderungen committen. Selbst wenn einzelne Entwickler eigene Branches verwenden, sind die Änderungen doch auf dem zentralen Server verfügbar und können immerhin von anderen in die Hauptentwicklungslinie eingespielt werden. Es besteht somit nur eine geringe Gefahr, dass die Entwicklungen eines Einzelnen über längere Zeit von der Hauptentwicklungslinie abweichen. Außerdem gibt es immer genau einen Ort, an dem der aktuelle Stand zu finden ist.

Ein weiterer Vorteil zentraler Systeme ist die leichte Umsetzbarkeit von Richtlinien innerhalb eines Unternehmens. So können Commits hinsichtlich ihrer Nachricht oder ihres Inhalts geprüft und so beispielsweise Fehler (falsche Formatierung von Quellcode, zu große (Binär-)Dateien, fehlgeschlagene Unit-Tests) frühzeitig abgelehnt werden.

3.1.2 Dezentrales Konzept

In dezentralen Systemen erhält jeder Mitarbeiter eine vollständige Kopie des Repositories. Dies erlaubt es, wesentlich freier an einem Projekt zu arbeiten. Ein Projektmitglied kann für sich allein seine Änderungen versionieren und entscheidet erst danach, ob und wann er sich mit anderen Mitgliedern synchronisieren möchte. Man spricht daher auch von „Commit Before Merge“. Da Commits nicht sofort für alle Projektmitglieder sichtbar sind und auch wieder gelöscht werden können, spricht man auch von privaten Commits. Um Änderungen an andere Repositories zu übertragen, wird ein sog. *Push* ausgeführt, der alle beim Ziel noch nicht vorhandenen Commits überträgt. Um die noch nicht im eigenen Repository enthaltenen Commits zu empfangen, wird hingegen ein *Pull* ausgeführt.

Ein sofort spürbarer Vorteil der Repository-Kopie ist die allgemeine Geschwindigkeit, mit der das System auf Befehle reagieren kann. Da die Versionsgeschichte einer Datei nicht erst von einem Server abgerufen werden muss, steht sie schneller

zur Verfügung. Selbst wenn ein zentrales System Anfragen in unter einer Sekunde beantworten könnte, verhindert bereits die Netzwerkkommunikation mit ihm ein effizientes Arbeiten, da jede Zeit, die ein Entwickler mit Warten auf die Ausführung eines Befehls verbringt, bereits eine Ablenkung und damit eine Störung seiner Konzentration darstellt.

Die Erfahrungen, die beim Umstieg auf Mercurial gesammelt wurden, zeigen dabei, dass der Effekt, dass Kommandos keine spürbare Verzögerung aufweisen, hierbei nicht unterschätzt werden darf: Wenn das Annotieren (die zeilenweise Anzeige der Autoren einer Datei[2]) einer Datei oder das Anzeigen der Projektgeschichte ohne Warten möglich ist, werden diese Funktionen auch automatisch häufiger genutzt (vgl. [Tor05] ab Minute 50).
Gleichzeitig nützen die verschiedenen Kommandos eines VCS nur dann etwas, wenn bei der Nutzung bereits auf sinnvolle Commit-Nachrichten und eine gewisse inhaltliche Abgeschlossenheit geachtet wird. So führt die unmittelbare Verfügbarkeit von Repository-Daten in einem gewissen Rahmen direkt zu einer besseren Nutzung des Versionskontrollsystems: Commits werden häufiger gemacht und besser beschrieben, allein schon weil es in der Regel einfacher ist, die Änderungen von einer Stunde Arbeit in einem Satz zusammenzufassen, als für alle Änderungen eines ganzen Tages (beim „Feierabend-Commit“).

Dezentrale Systeme erlauben es, Commits nachträglich zu verändern.[3] Sie können umsortiert, bearbeitet, gelöscht oder sogar an andere Stellen in der Versionsgeschichte eingefügt werden, bevor sie mit anderen Projektmitgliedern geteilt werden. Dies macht es möglich, die eigene Arbeit flexibler zu strukturieren und mehr zu experimentieren, da Änderungen nicht sofort die Arbeit Anderer beeinflusst: Branches können lokal angelegt und wieder entfernt werden, ohne sie veröffentlichen zu müssen.

Die größere Freiheit in der Organisation der Entwicklung birgt allerdings auch Gefahren: So kann der fehlende Zwang, Änderungen in eine gemeinsame Basis zu übertragen, zu einem Inseleffekt führen, bei dem Commits erst nach einer relativ langen Zeitspanne veröffentlicht werden. Dieses lange Zurückhalten kann zu Problemen führen, wenn die restlichen Projektmitglieder die Änderungen nicht Stück für Stück integrieren können, sondern plötzlich mit einem signifikanten Umbau des Projekts konfrontiert werden.

Da alle Klone eines Repositories technisch gleichwertig sind, erfordern dezentrale Systeme meist genaue Regelungen, welcher Klon als Referenz angesehen wird. Zwar können Änderungen beliebig zwischen Teilnehmern ausgetauscht werden, für gewöhnlich gibt aber ein Master-Repository, von dem dann das fertige Produkt erzeugt wird.

Im Unterschied zu zentralen Systemen ist das Umsetzen von Richtlinien, was den Inhalt der Commits angeht, in dezentralen Systemen wesentlich schwieriger. Eine

[2] Je nach System ist diese Funktion auch als „blame“ (beschuldigen) bekannt.

[3] Die meisten Systeme arbeiten mit kryptografischen Prüfsummen und ermöglichen es daher nicht, im wörtlichen Sinne einen Commit zu verändern. Stattdessen entsteht durch die Änderung ein neuer Commit, der den alten ersetzt. In Abschnitt 3.2.2 auf Seite 17 wird auf diesen Umstand näher eingegangen.

Richtlinie könnte beispielsweise prüfen, ob eine neue Funktion in einem Quellcode kommentiert wurde, ob neue Dateien die korrekten Zeilenumbrüche verwenden oder der Quellcode keine Syntax-Fehler aufweist. Es gibt zwei Orte, an denen die Richtlinien theoretisch gehandhabt werden könnten:

Weiterhin beim Master-Klon des Repositories
: Dies würde es erlauben, die Routinen ebenfalls auf zentrale Art zu verwalten und sie automatisch für jeden, der Änderungen zum Master senden will, verpflichtend zu machen. Das größte Problem hierbei ist jedoch, dass Clients nicht gezwungen sind, jeden Commit sofort zu senden, was im Extremfall dazu führen kann, dass ein Client mehrere Commits senden will, doch der erste bereits problematisch ist. In diesem Fall müsste der Client (wenn überhaupt möglich) in der Revisionsgeschichte zurückgehen und von dem ersten fehlerhaften Commit an die Geschichte ändern.
Dieses Vorgehen ist problematisch, da es die Gefahr birgt, dass bereits getätigte Commits nachträglich noch einmal bearbeitet werden müssen, um die Richtlinien zu erfüllen. Für gewöhnlich arbeiten Versionskontrollsysteme jedoch nach dem Schema, dass neue Änderungen auch in neuen Commits resultieren sollen (anstatt dass bereits bestehende Commits immer wieder geändert werden). So würde ein Ablehnen von Commits direkt diese Arbeitsweise stören und so nicht mehr mit, sondern gegen das eingesetzte Versionskontrollsystem arbeiten. Abgesehen von dieser erzwungenen Fehlbedienung bringt dieses Verfahren auch noch einen hohen Mehraufwand für die einzelnen Projektmitglieder mit sich.
Der Gefahr, Commits ändern zu müssen, kann dadurch begegnet werden, dass nicht die Commits einzeln geprüft, sondern nur die durch ihre Anwendung entstehende Revision geprüft wird. Damit wäre es bei Problemen möglich, diese durch neue Commits zu beheben.

Auf den einzelnen Clients
: Die zweckmäßigere Variante wäre, die Prüfroutinen auf den Rechnern der Projektmitglieder auszuführen, sodass fehlerhafte Commits gar nicht erst entstehen können und im Nachhinein korrigiert werden müssten. Hierbei kann allerdings die Ausführung der Prüfungen nicht garantiert werden, da Entwickler die Routinen im Zweifelsfall deaktivieren könnten. Allerdings ist es möglich, die Prüfroutinen sowohl auf dem Client als auch beim Master-Repository durchführen zu lassen. Dabei wären Probleme, die erst beim Übertragen zum Master entdeckt werden, dann von dem jeweiligen Mitglied selbst verschuldet.
Ein weiteres Problem ist die Verteilung der Prüfroutinen. Sie erfordern nicht nur, dass alle Mitglieder eine Kopie von ihnen erhalten (und diese auch aktuell halten), sondern auch, dass sie auf den Clientrechnern lauffähig sind. Je nach eingesetzter Software kann dies zu Lizenzkosten oder Imkompatibilitäten führen. Die Verteilung kann im einfachsten Fall über ein weiteres Repository erfolgen, das ein Cronjob (ein in regelmäßigen Abständen automatisch ausgeführtes Programm) in einem sinnvollen Intervall aktualisiert.

Natürlich ist es möglich, beide Strategien zu kombinieren, indem beim Client nur ein Proxy liegt, der zur eigentlichen Überprüfung des Commits entweder die Prüfroutinen vom Server bezieht oder gar die Änderungen dorthin sendet. Dies würde

jedoch den Vorteil, dass Commits ohne spürbare Verzögerung getätigt werden können, relativieren.

3.1.3 Bewertung

Die vorangegangenen Ausführungen haben ein deutliches Problem mit dem zentralen Konzept aufgezeigt. Es ist insofern hinderlich, dass es die experimentelle Entwicklung von Funktionen dadurch verhindert, dass alle Branches und Commits sofort veröffentlicht werden müssen und damit die anderen Projektmitglieder beeinflussen. So müssen diese die gemachten Änderungen erst in ihre eigene Arbeit einspielen (mergen), bevor sie ihre Arbeit versionieren können. Gleichzeitig treten durch die Zentralität Namenskonflikte auf, wenn kurzlebige Branches erzeugt werden sollen, da es nur einen Branch namens „Test“ geben kann. So ist es impraktikabel, häufig Branches zu verwenden, um eben nicht die Arbeit anderer direkt zu beeinflussen.

Ein dezentrales System würde die Open Source-Ausrichtung des Unternehmens begünstigen, da Fragen wie Schreibzugriff auf öffentliche Repositories entfielen: Jeder kann einen eigenen Klon eines Repositories besitzen und hat auf ihm volle Zugriffsrechte. Zu diesem Zweck würde es reichen, Klone der Repositories zu veröffentlichen, die eventuelle Dritte ihrerseits klonen und zur Entwicklung verwenden können. Die Frage, wie genau die einzelnen Repositories unter Verwendung des zentralen Systems veröffentlicht werden sollten (und wie mit Änderungen Dritter umgegangen werden sollte), blieb dagegen über Monate trotz häufiger Diskussionen ungelöst.

3.2 VCS-Implementierungen

Nachdem im vorangegangenen Abschnitt die Grundzüge zentraler und dezentraler Systeme beschrieben wurden, sollen nun zwei konkrete Implementierungen verglichen werden. Zu diesem Zweck werden Subversion (zentral) und Mercurial (dezentral) gegenübergestellt.

	Subversion	Mercurial
Hersteller	Apache Foundation	Matt Mackall et al.
erste stabile Version	Februar 2004	März 2008
aktuelle Version	1.6.16	1.8.2
implementiert in	C	Python
Systemunterstützung	Unix, Win32, MacOS X, ...[4]	wo Python verfügbar
Lizenz	Apache License, Version 2.0	GPL 2.0

Tabelle 3.1: Vergleich zwischen Subversion und Mercurial

In Tabelle 3.1 werden einige statistische Fakten zu beiden Programmen aufgelistet, die allerdings zum Verständnis der folgenden Abschnitte nicht relevant sind.

3.2.1 Subversion (SVN)

Subversion hat sich in den letzten Jahren als klarer Industriestandard zur Versionierung entwickelt. Im Jahr 2009 wurde es laut [Col09] auf über 400.000 öffentlichen Servern eingesetzt.

[4] http://subversion.apache.org/packages.html

Subversion verwaltet üblicherweise mehrere Projekte in einem Repository. Dies macht es möglich, identische Inhalte in verschiedenen Projekten effizient zu speichern. Beim Erstellen einer Arbeitskopie wird dann ein (beliebiger) Teilbaum des Repositories abgerufen („Checkout“).
Diese Struktur ermöglicht es, jedes Verzeichnis als eigenen Checkout zu betrachten. Gleichzeitig führt sie dazu, dass Commits in einem „Projekt“ die globale Revisionsnummer für das gesamte Repository inkrementieren.

Ein Problem dieser Herangehensweise ist, dass Subversion in jedem Verzeichnis der Arbeitskopie ein verstecktes Verzeichnis für seine eigenen Metadaten ablegt. Diese (meist .svn benannten) Verzeichnisse enthalten neben Informationen zur Verbindung mit dem Repository auch eine Kopie aller Dateien auf Stand des letzten Checkouts. Dies beschleunigt die Berechnung Diffs und macht es möglich, auch ohne Verbindung zum Repository die Arbeitskopie zurückzusetzen.
Allerdings verdoppelt dieses Verfahren den für eine Arbeitskopie benötigten Speicherplatz und macht Dateisystemoperationen wie das Löschen von Verzeichnissen erheblich schwerer: Werden für das Löschen oder Umbenennen von Verzeichnissen nicht die Kommandos von Subversion verwendet, kann die Arbeitskopie irreparabel beschädigt werden und muss in diesem Fall aufwendig manuell zurückgesetzt werden. Beispielsweise würde das Löschen und Neuerzeugen eines Verzeichnisses einen Konflikt erzeugen, da die Client-Software einen `add`-Befehl ausführen will, der Subversion-Server dies aber ablehnt, da das Verzeichnis bereits existiert (und damit einen `update`-Befehl erwartet). Dieses Verhalten wurde bewusst gewählt und verhindert es, dass erweiterte Dateimanager oder Skripte in einer Arbeitskopie verwendet werden.[5]

Subversion kennt kein Konzept von Branches und Tags, stattdessen werden diese üblicherweise über ein grundlegendes Repository-Layout nachgerüstet, bei dem Kopien von Verzeichnissen mit besonderer Semantik versehen werden. So soll im Verzeichnis `trunk` die Hauptentwicklung stattfinden, in `branches` Kopien des Trunks liegen, mit der Absicht, die dort getätigten Änderungen zu einem späteren Zeitpunkt in den Trunk zurückzumergen und in `tags` jeweils Kopien liegen, die im Nachhinein nicht mehr verändert werden sollten und so effektiv einen Alternativnamen für eine bestimmte Revision bereitstellen.

3.2.2 Mercurial (hg)

Mercurial wird seit 2004 federführend von Matt Mackall als leichtgewichtiges und dennoch leistungsfähiges Versionskontrollsystem entwickelt. Es weist große Parallelen zu Git auf, das sich in dem gleichen Zeitraum entwickelte, aber eher für die Verwaltung des Linux Kernels geschaffen wurde, als als allgemeines VCS (vgl. [Tor05] und [Ott09], S. 8). Das Kürzel „hg“ ist eine Referenz auf das Elementsymbol von Quecksilber (engl. mercurial) und deutet auf die Eigenschaften des Programms hin: „fast, fluid, flexible“ (vgl. [Mac11b]).

Mercurial orientiert sich bewusst an den üblichen Termini und verwendet daher auch Befehlsnamen wie „Commit“ und „Merge“, um den Ein- und Umstieg einfacher zu gestalten.[6]

[5] http://subversion.apache.org/faq.html#wc-change-detection
[6] http://mercurial.selenic.com/about/

In Mercurial dienen SHA-1-Prüfsummen als eindeutige Bezeichner einzelner Commits (vgl. [Div11d]). In die Prüfsumme werden neben den eigentlichen Änderungen auch Autor, Datum und Commit-Nachricht einbezogen, ebenso wie die Prüfsummen der Parents. Dieses Verfahren sorgt dafür, dass auch bei verteilter Arbeit an einem Projekt keine Konflikte bei Versionsnummern auftreten und jede Änderung einen eineindeutigen Identifier besitzt. Im Unterschied zu Git bietet Mercurial jedoch auch eine pro Repository eindeutige, fortlaufende Nummer an, die die Arbeit mit Commits erleichtert. Außerdem müssen von Prüfsummen nur die so viele Stellen angegeben werden, um eindeutig zu sein; in den meisten Fällen genügen die ersten 8 Zeichen der hexadezimalen Darstellung (z. B. `e9263eee` statt `e9263eee0d8d79927a1983deff15831a8d27fa0a`).

Das Bilden der Prüfsummen stellt außerdem sicher, dass keine (böswilligen) Veränderungen an der Revisionsgeschichte vorgenommen werden können. Dies ist dem rekursiven Aufbau geschuldet, der in Formel 3.1 beschrieben ist. Dabei gibt der Index die Nummer des Commits an. d_n ist der Diff zwischen Commit n und $(n-1)$, m_n ist die Commitnachricht, t_n die Zeit, a_n der Name des Autors und h die Hashfunktion, die ihre Argumente konkateniert und mit SHA-1 verarbeitet. Damit ergibt sich das in Formel 3.1 dargestellte Schema für die Berechnung der Prüfsummen, wobei $hash_0$ die Prüfsumme des virtuellen Null-Commits ist.

$$\begin{aligned} h(d, m, t, a, hash) &= sha1(d + m + t + a + hash) \\ hash_0 &= 0 \\ hash_n &= h(d_n, m_n, t_n, a_n, hash_{n-1}) \end{aligned} \tag{3.1}$$

Eine Änderung an der Prüfsumme des Commits x beeinflusst damit direkt die Prüfsummen aller folgenden Commits ($hash_y$ mit $y > x$). Dies ermöglicht es, die Integrität eines Repositories sicherzustellen, indem die Prüfsumme des letzten Commits berechnet und mit einem Referenzwert (zum Beispiel aus einem Backup) verglichen wird. Auf diese Weise können auch unbeabsichtigte Übertragungsfehler beim Klonen eines Repositories ausgeschlossen werden. Im Zweifelsfall kann die Integrität manuell über den Aufruf des `verify`-Befehls von Mercurial geprüft werden.

Dass alle Inhalte des Repositories mit Prüfsummen versehen werden, führt allerdings auch dazu, dass es nicht möglich ist, nur einen Teil der Inhalte abzurufen. Würde man beispielsweise nur ein einzelnes Verzeichnis aus einem Repository abrufen, würde das neue Repository auch nur einen Teil der Diffs enthalten. Dies führt gemäß Formel 3.1 zu neuen Hashes (da sich jeweils der Parameter d ändert) und damit zu einem neuen Repository.

Beim Klonen ist es daher nur möglich, das gesamte Repository zu übertragen. Es ist jedoch meist ohne Weiteres möglich, im Anschluss daran die gewünschte Teilmenge der Dateien durch einen Konvertierprozess (vgl. [Div11a]), bei dem einzelne Dateien oder Verzeichnisse ausgeschlossen werden können, oder das Exportieren von Commits (vgl. [Mac11a], Abschnitt „archive“) zu erhalten. Dabei entsteht jedoch in den meisten Fällen ein neues Repository, sodass auch bei dieser zweistufigen Vorgehensweise nicht von einem partiellen Klon gesprochen werden kann. In manchen

Anwendungsfällen kann es allerdings ausreichend sein, nach dem Klon die relevante Teilmenge über einen weiteren Prozess zu erhalten, zum Beispiel wenn Repositories aus Platzmangel auf andere Systeme übertragen werden können.

Die Einschränkung, immer das komplette Repository zu übertragen, sollte bei der Entscheidung, wie einzelne Projekte organisiert werden, unbedingt berücksichtigt werden.

Im Gegensatz zu Subversion hat Mercurial ein Verständnis von Branches und Tags. Branches entstehen gemäß Definition 2.8 auf Seite 6 immer dann, wenn ein Commit mehr als ein Child hat. Dabei können Branches unbenannt innerhalb eines Repositories verwaltet, explizit benannt oder durch das Verwenden mehrerer Repositories verwendet werden. Insofern stellt jeder Klon eines Repositories einen eigenen Branch dar. Tags hingegen sind gemäß Definition 2.11 auf Seite 8 Alternativnamen und werden in einer regulären, versionierten Datei namens `.hgtags` im Repository verwaltet, in der zu jedem Tag die dazugehörige Prüfsumme des Commits notiert ist. Tags sind pro Repository eindeutig und müssen beim Mergen von Commits gesondert betrachtet werden.

3.2.3 Git

Git[7] ist ebenfalls ein dezentrales Versionskontrollsystem, das sich in etwa im gleichen Zeitraum wie Mercurial entwickelte und viele Gemeinsamkeiten mit ihm teilt (wie zum Beispiel die Verwendung von SHA-1 für die Prüfsummen von Commits). Git wurde speziell dafür geschaffen, den Linux-Quellcode zu verwalten und die Entwicklung durch Hunderte Entwickler zu unterstützen.

Der Fokus auf die Linux-Entwicklung führt jedoch dazu, dass Git unter Windows weniger performant und komfortabel zu bedienen ist. So fehlt Windows eine mächtige Shell, die es ermöglichen würde, die einzelnen Git-Kommandos miteinander zu verbinden und so die volle Flexibilität von Git auszunutzen.

Außerdem ist Git teils komplizierter zu bedienen als Mercurial. So stellt insbesondere der Git-Index (eine virtuelle Schicht zwischen Arbeitskopie und Repository, die zur Vorbereitung von Commits dient; vgl. [Cha11], S. 17) eine Hürde beim Einstieg dar, da Commits nicht wie in anderen Systemen über einen einzelnen Befehl, sondern eine Reihe von Kommandos ausgeführt werden.

Allgemein ist Git eher auf größtmögliche Flexibilität ausgerichtet und gibt dem Benutzer Werkzeuge an die Hand, um die Versionsgeschichte auch nachträglich noch zu verändern. Mercurial beschränkt sich dagegen auf einfachere Kommandos, die eine Fehlbedienung erschweren und so Datenverlust schwerer machen (vgl. [Div11c]).

3.2.4 Benchmarks

Die getätigten Aussagen über die höhere Effizienz von Mercurial sollen nun an einem Beispiel belegt werden. Dazu wurde das Projekt FeatureIDE[8] von Subversion nach Mercurial konvertiert. Das Projekt befand sich in Revision 891 vom 24. Februar 2011. Zu Testzwecken wurde nur der Trunk betrachtet.

[7] http://git-scm.com/

[8] https://faracvs.cs.uni-magdeburg.de/svn/tthuem-FeatureIDE/trunk

	Subversion	Mercurial
Dateien in der Arbeitskopie	10.881	3.199
Verzeichnisse in der Arbeitskopie	16.803	835
Größe der Arbeitskopie	202 MiB	100 MiB
Größe inkl. Repository	N/A	240 MiB

Tabelle 3.2: Platzvergleich zwischen Subversion und Mercurial

Die Daten zeigen, dass Mercurial die gesamte Projektgeschichte inklusive einer Arbeitskopie mit nur 38 MiB Overhead speichert. Gleichzeitig wird aufgrund der fehlenden verstreuten Metadaten nur ein Bruchteil der Verzeichnisse verwendet.

Ein Benchmark derjenigen Befehle, die in Subversion eine Kommunikation mit dem Repository erfordern, soll an dieser Stelle nicht durchgeführt werden. Zum einen hängt die benötigte Zeit von der Auslastung des Servers, zum anderen von der verfügbaren Bandbreite ab. Ein derartiger Test hätte aufgrund dieser Faktoren keine praktische Relevanz.

3.2.5 Bewertung

Subversion wurde ursprünglich als Versionskontrollsystem gewählt, da es sich vergleichsweise leicht aufsetzen und administrieren lässt und eine ausgereifte Integration in die verwendete Entwicklungsumgebung (Eclipse) bietet. Außerdem waren die Entwickler mit dem Einsatz bereits vertraut, sodass kein Schulungsaufwand erforderlich war.

Der Subversion-Cache ist als Performance-Verbesserung gedacht, stellte sich jedoch als eher hinderlich heraus. Dadurch, dass die Entwickler mit einem Server im Haus kommunizierten, war der Geschwindigkeitsvorteil durch den Cache verschwindend gering, die von ihm hervorgerufenen Probleme jedoch sehr deutlich: So verhindert der Cache, wie bereits erwähnt, den Einsatz gewohnter Werkzeuge zur Dateiverwaltung. So ist es nicht problemlos möglich, ein Verzeichnis zu löschen und es im Anschluss neu anzulegen. Gleichzeitig blähte er Projekte um mehr als das Doppelte auf. Dieser Effekt wurde dadurch verstärkt, dass die versionierten Projekte meist aus vielen kleinen Dateien bestanden. Die Blockgröße der Festplatten in den Entwicklerrechnern (üblicherweise 4 KiB) förderte damit eine starke Fragmentierung oder Vergeudung von Speicherplatz.

Mercurial

Obwohl Mercurial weit weniger populär als Git ist, stellt es die bessere Wahl für das Unternehmen dar.[9] Das einfachere Interface und vor allem die bessere Unterstützung für Windows sind starke Argumente für Mercurial. So wurde insbesondere der Git-Index als Verständnisproblem von den Entwicklern wahrgenommen. Weiterhin eröffnet sich die ganze Flexibilität von Git erst beim Einsatz einer Unix-Shell, die auf den Windows-Maschinen der Entwickler nicht zur Verfügung stand.

[9] Laut einer Statistik bei [Bla11], bei dem im Mai 2011 35% aller auf ohloh.net registrierten Projekte Git verwendeten (verglichen mit 1% Mercurial-Repositories).

Dies soll nicht darüber hinwegtäuschen, dass die zugrunde liegenden Konzepte beider Systeme sehr ähnlich sind und beide daher in etwa die gleichen Möglichkeiten bieten, wenn auch bei Mercurial für bestimmte Operationen (teils mitgelieferte) Erweiterungen notwendig sind (vgl. [RS10]).

Die anderen, verteilten Systeme wie Darcs[10], Monotone[11] oder Bazaar[12] wurden wegen ihrer geringeren Verbreitung nicht weiter in Betracht gezogen. Es war klar, dass es für ein populäres System einfacher sein würde, im Problemfall Hilfe zu erhalten.

3.3 Grafische Oberflächen

Die beiden grafischen Oberflächen (GUIs), die im Folgenden verglichen werden sollen, integrieren sich nicht in eine bestimmte Entwicklungsumgebung, sondern in den Windows Explorer und dessen Kontextmenü. Dies erlaubt es, sie unabhängig von bestimmten Entwicklungsmodellen und Projekten zu verwenden.
Der Namensbestandteil „Tortoise“ wurde von der ersten Shell-Erweiterung, TortoiseCVS, übernommen und hat sich seither für diese Art von Versionskontrollclients etabliert.[13]

	TortoiseSVN[14]	TortoiseHg
Hersteller	Stephan Küng, Lübbe Onken	Steve Borho et al.
erste stabile Version	März 2004	März 2010
aktuelle Version	1.6.15	2.0.3
implementiert in	C++	Python
Systemunterstützung	Windows ab 2000 SP2	Windows ab XP / GNOME
64 Bit-Version	verfügbar	verfügbar
Lizenz	GPL 2.0	GPL 2.0

Tabelle 3.3: Vergleich zwischen TortoiseSVN und TortoiseHg

In Tabelle 3.4 auf Seite 29 werden einige statistische Fakten zu beiden Programmen aufgelistet, die allerdings zum Verständnis der folgenden Abschnitte nicht relevant sind.

Der Vergleich der GUIs soll sich auf die wesentlichen Funktionen beschränken, anstatt alle möglichen Fenster abzubilden und gegenüberzustellen. Dies ist dem Umstand geschuldet, dass im Arbeitsalltag viele Funktionen eines Versionskontrollsystems nicht benötigt werden und der Fokus dieser Arbeit nicht auf dem Umgang mit den GUIs der Systeme liegt.

3.3.1 TortoiseSVN

TortoiseSVN bindet sich in den Windows Explorer ein und stellt neben einem umfangreichen Kontextmenü auch Overlays für die Verzeichnis- und Dateisymbole bereit, die den jeweils aktuellen Status (geändert, hinzugefügt, zum Löschen vormarkiert, ...) verdeutlichen.

[10] http://darcs.net/
[11] http://www.monotone.ca/
[12] http://bazaar.canonical.com/en/
[13] http://en.wikipedia.org/wiki/TortoiseCVS
[15] http://tortoisesvn.net/

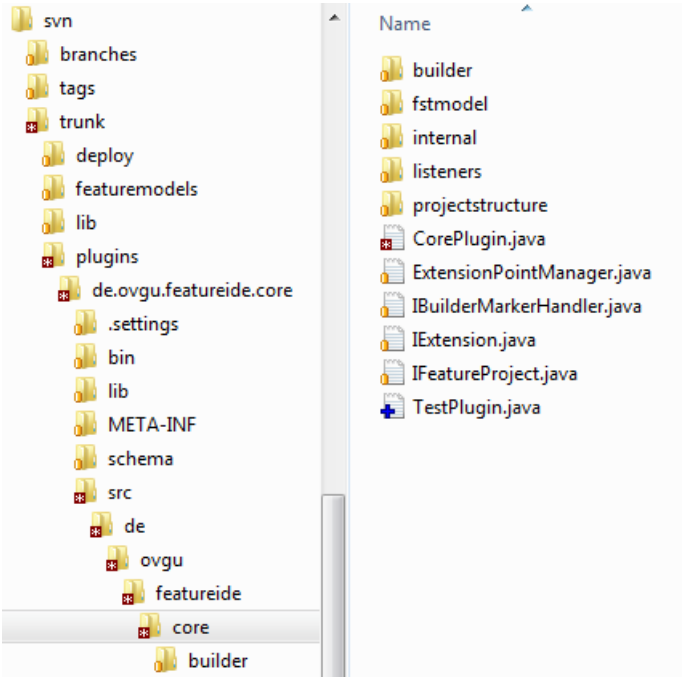

Abbildung 3.2: Overlay-Icons von TortoiseSVN im Windows Explorer

Abbildung 3.2 zeigt einen Ausschnitt des Windows Explorers mit den Overlay-Icons von TortoiseSVN. Die Datei `CorePlugin.java` wird als *geändert* angezeigt, während die Datei `TestPlugin.java` als *neu* markiert wird. Die Markierungen wirken sich dabei auch auf die Verzeichnissymbole aus. Dateien wie `IExtension.java` werden als *unverändert* markiert.

Commit- und Log-Ansicht

Die zentralen Elemente der Oberfläche sind die Commit- und die Log-Ansicht.

Abbildung 3.3 auf der nächsten Seite zeigt den Commit-Dialog von TortoiseSVN. Im oberen Bereich kann die Commit-Nachricht angegeben werden, während darunter die Liste der geänderten Dateien dargestellt wird. Dabei können einzelne Dateien über die Auswahlfelder vor ihren Namen vom Commit ausgeschlossen werden. Der Nutzer muss jedoch auf einzelne Dateien einen Doppelklick ausführen, um die eigentlichen Änderungen zu betrachten, die sich in einem neuen Fenster mit einem benutzerdefinierten Diff-Programm öffnen würden.

Es ist nicht ohne Weiteres möglich, aus Dateien nur einzelne Teile (sog. Hunks) auszuwählen, das heißt, es wird immer die gesamte Datei committet. Der schon angesprochene lokale Cache der aktuellsten Revision sorgt hierbei dafür, dass die Diffs sofort zur Verfügung stehen.

Anstatt Änderungen zu speichern, können Sie auch direkt rückgängig gemacht oder als Patch exportiert werden.

In der Log-Ansicht (zu sehen in Abbildung 3.4 auf der nächsten Seite) kann der Nutzer die bisher getätigten Änderungen in zumeist chronologischer Folge betrachten, um sich so einen Überblick zu verschaffen. Neben der Commit-Nachricht werden Autor, Datum und die Liste der geänderten Dateien angezeigt. Auch hier sind

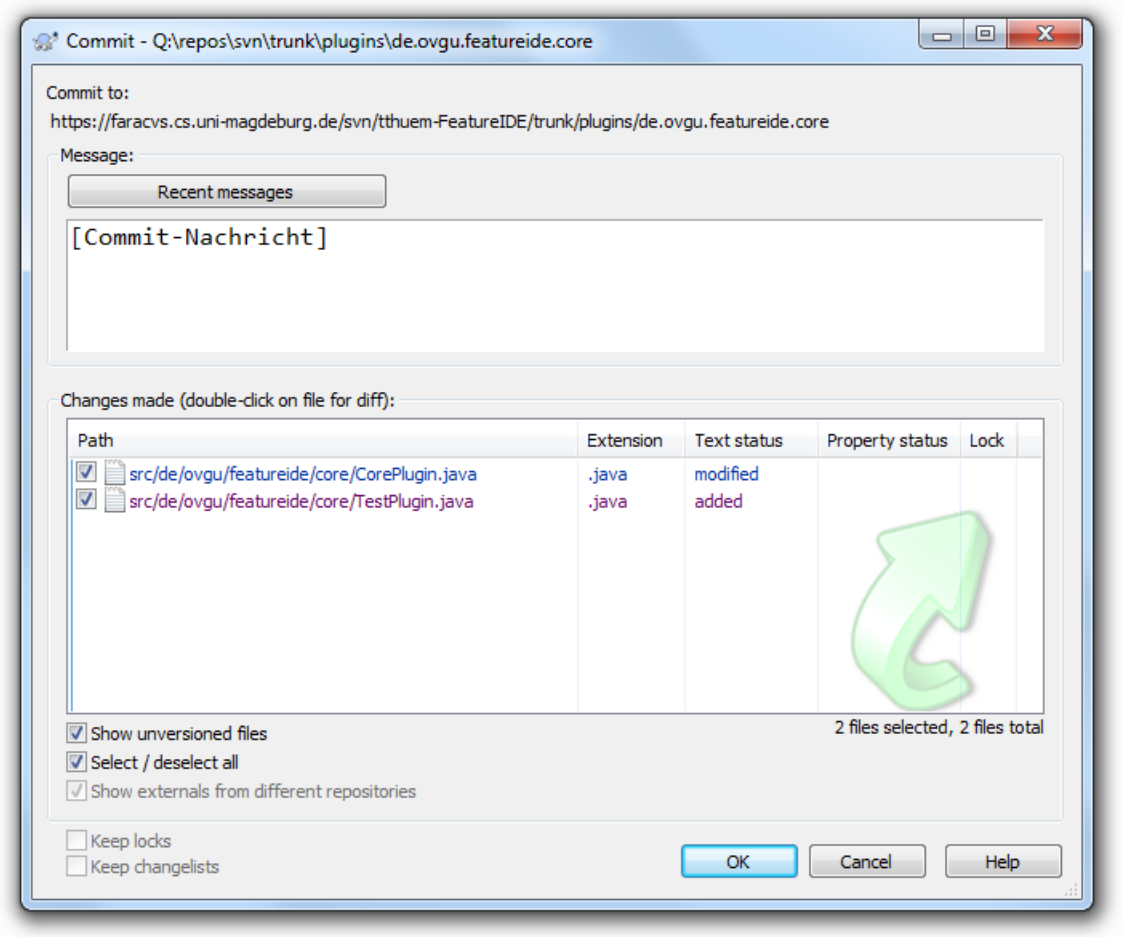

Abbildung 3.3: Commit-Ansicht von TortoiseSVN

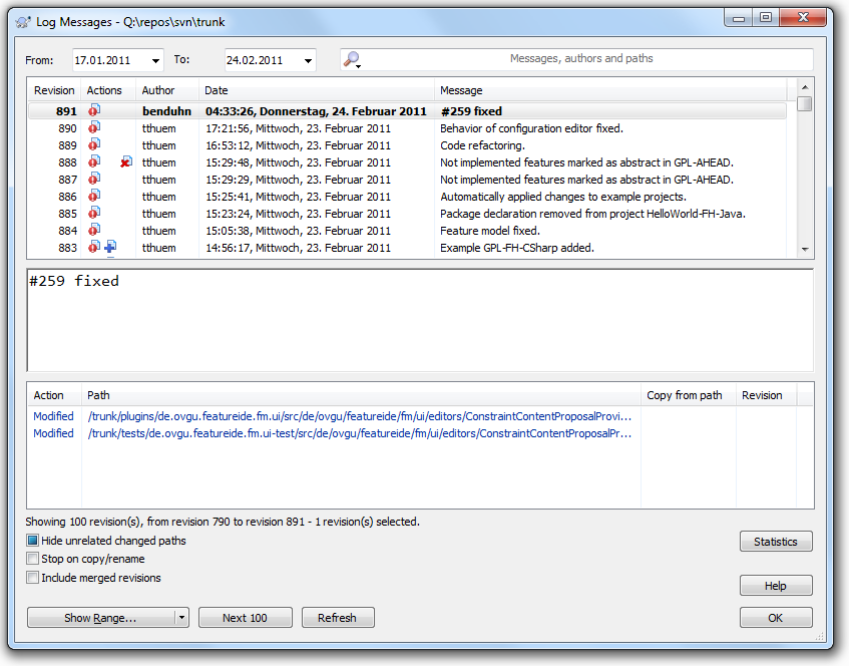

Abbildung 3.4: Log-Ansicht von TortoiseSVN

die einzelnen Diffs nur per Doppelklick auf einen Dateinamen verfügbar und müssen jeweils vom Repository abgerufen werden. Die Informationen wie Autor und Commit-Nachricht werden hingegen lokal vorgehalten. Über diverse Filter können die angezeigten Revisionen einschränkt werden.

Repository-Browser

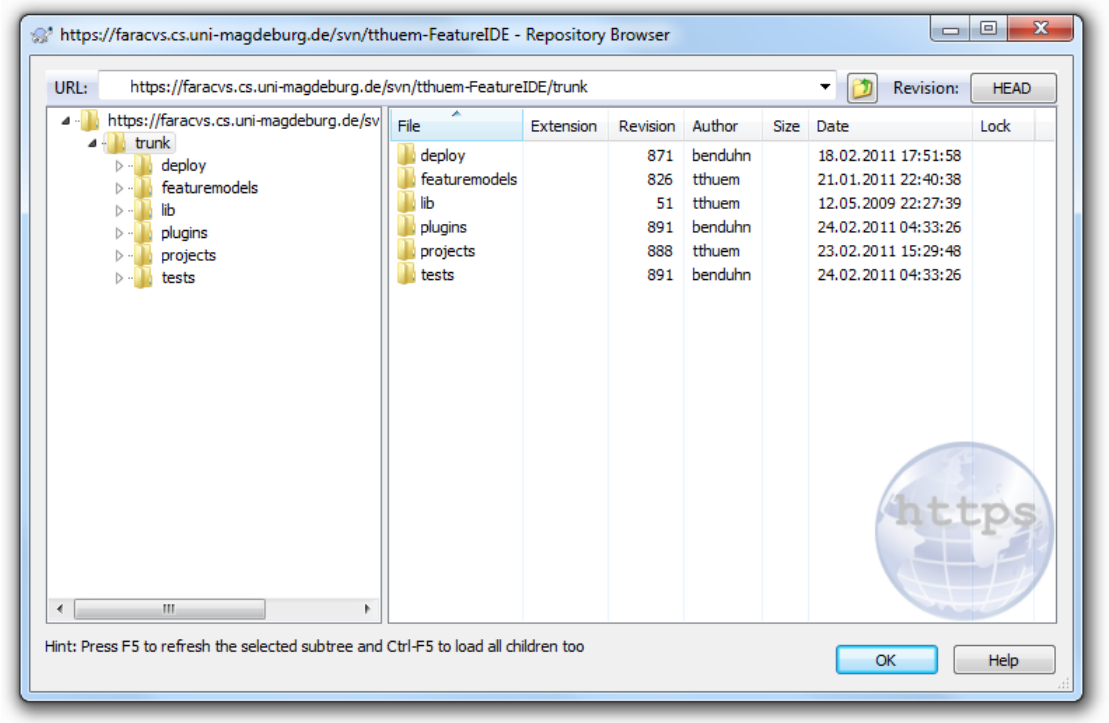

Abbildung 3.5: Repository-Browser von TortoiseSVN

Im Repository-Browser können Operationen wie das Löschen oder Verschieben von Verzeichnissen direkt auf dem Server vorgenommen werden, anstatt über lokale SVN-Kommandos, die dann als Commit gesendet würden. Dies kann bei größeren Operationen (wie dem Anlegen eines neuen Projekts) massiv Netzwerkverkehr einsparen und bedeutend komfortabler sein, als die Änderungen lokal durchzuführen. Abbildung 3.5 zeigt einen geöffneten Repository-Browser für das Repository von FeatureIDE, der links die Verzeichnisstruktur (inklusive des besonderen Verzeichnisses `trunk`) und rechts die Dateien und Verzeichnisse innerhalb des ausgewählten Verzeichnisses. Neben jeder Datei ist die aktuelle Revisionsnummer sowie der Benutzername des letzten Autors zu sehen.

3.3.2 TortoiseHg

TortoiseHg teilt sich den grundlegenden Aufbau als Explorer-Erweiterung mit TortoiseSVN, legt aber einen größeren Fokus auf die Projektgeschichte. Da für die Overlay-Icons dieselbe Bibliothek zum Einsatz kommt, soll hier auf Abbildungen verzichtet werden.

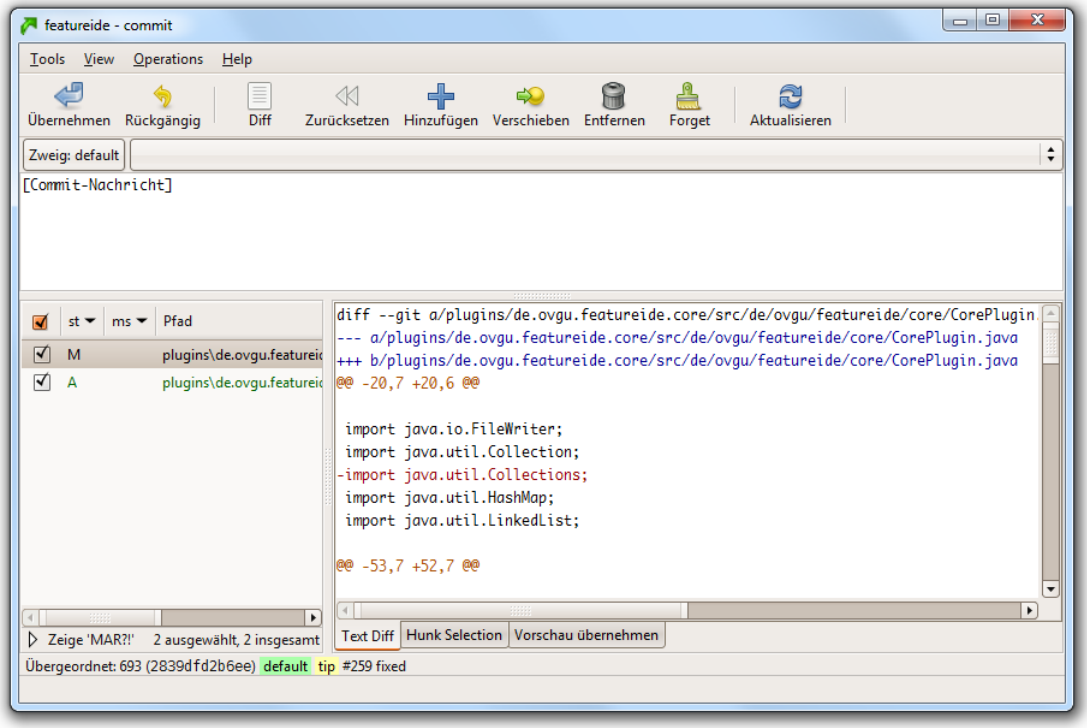

Abbildung 3.6: Commit-Ansicht von TortoiseHg

Commit-Ansicht

Die Commit-Ansicht von TortoiseHg zeigt in Abbildung 3.6 neben den Änderungen im Dateisystem auch direkt die zu versionierenden Diffs an. Dies macht es einfacher, die eigene Arbeit noch einmal zu kontrollieren und zum Beispiel Testcode zu entfernen, bevor er im Repository gespeichert wird. Außerdem ist es wie bei TortoiseSVN möglich, Dateien vom Commit auszuschließen oder frühere Commit-Nachrichten wiederzuverwenden.

Repository Explorer

Das Kernelement der GUI ist der Repository Explorer, über den die verschiedenen Management-Funktionen wie der Commit-Dialog, das Zwischenlager für Änderungen oder die Einstellungen von TortoiseHg, zugänglich sind. Außerdem dient er zum Synchronisieren mit anderen Repositories.

In Abbildung 3.7 auf der nächsten Seite wird der Projektverlauf inklusive einer Visualisierung der Branches dargestellt. Ebenso wie bei TortoiseSVN sind pro Commit die geänderten Dateien sichtbar, mit dem Unterschied, dass die Diffs sofort im selben Fenster angezeigt werden. Auf diese Weise können getätigte Änderungen wesentlich effizienter nachvollzogen werden, als wenn die Diffs pro Datei einzeln abgerufen werden müssten. Die grafische Darstellung des Revisionsgraphs ermöglicht es, den Commitverlauf schnell zu überblicken. Über die Eingabeelemente auf der oberen Seite des Fensters kann die Anzeige der Commits auf eine bestimmte Zeitspanne oder Verzeichnis eingeschränkt werden.

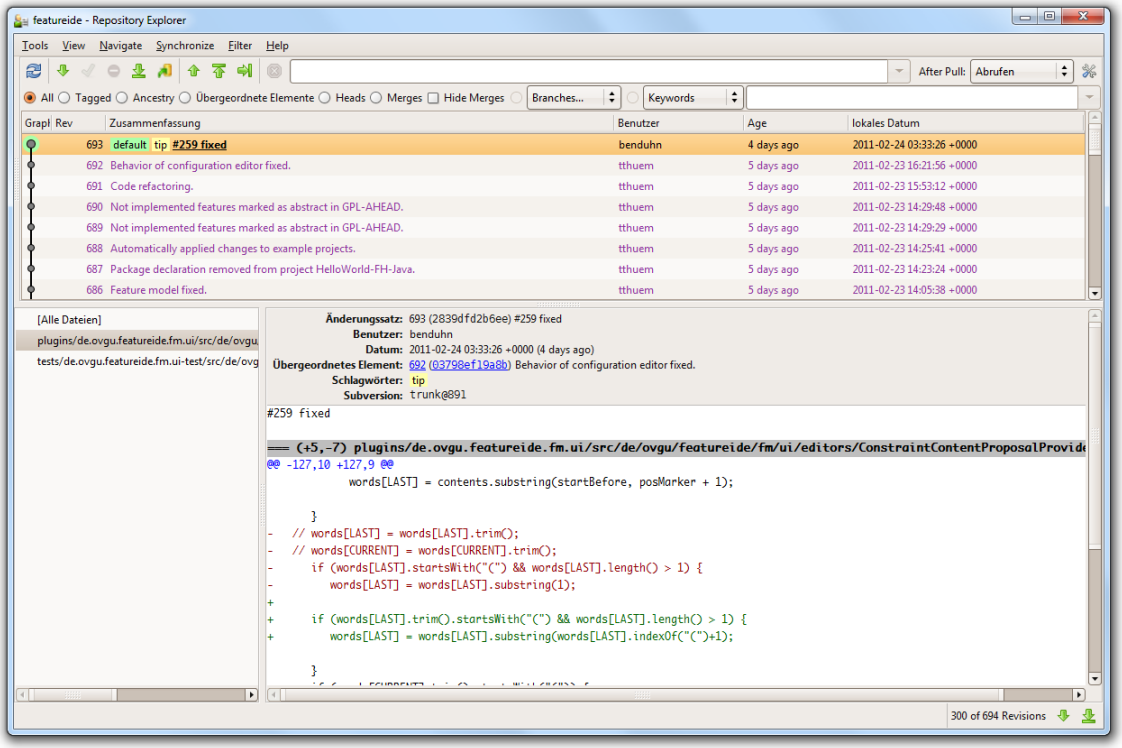

Abbildung 3.7: Repository-Explorer von TortoiseHg

Im Repository Explorer können direkt Änderungen aus anderen Repositories eingespielt oder versendet werden, außerdem stehen Exporte, Merges und weitere über Mecurial-Erweiterungen nachrüstbare Funktionen bereit.

Der Explorer darf nicht mit dem Repository-Browser von TortoiseSVN verwechselt werden: Während dieser auf einem entfernten Repository arbeitet, dient der Explorer dem Management eines lokal zugänglichen Repositories.

3.3.3 Bewertung

Auch wenn sich die Integrationen in das Kontextmenü des Windows Explorers beider Programme stark ähneln, führen die Unterschiede im Detail zu einer deutlich geänderten Nutzung der GUIs. Gerade die Commit- und Log-Ansicht von TortoiseSVN stellten sich im Vergleich zu TortoiseHg als große Probleme heraus.

Dadurch, dass in dem normalen UI von TortoiseSVN (der verwendeten GUI, vgl. Abschnitt 3.3.1 auf Seite 21) nie die eigentlichen Änderungen aufgezeigt wurden, und gleichzeitig die Abfrage von Logs, Annotationen und älteren Dateirevisionen mit spürbaren Verzögerungen einhergingen, wurden diese Funktionen nie verwendet. Und da die Funktionen nicht genutzt wurden, wurden auch keine sinnvollen (d. h. nichtleeren) Commit-Nachrichten angegeben.

Ohne die Anzeige eines Diffs beim Commit war es zu aufwendig, die eigenen Änderungen in logisch getrennte Commits aufzutrennen. So wurden häufig querschneidende Änderungen in einem einzelnen Commit gespeichert, was teils auch mit dem vorangegangenen Punkt zusammengehängt: Wenn Commits nicht direkt angezeigt

und/oder sofort zur Verfügung stehen, werden sie nicht genutzt. Also war es nicht sinnvoll, sich über eine Aufteilung der Änderungen in einzelne Commits Gedanken zu machen.

Die verschwindend geringe Sorgfalt beim Anlegen von Commits führte die Projektentwicklung in einen „Teufelskreis“: Da Commits nicht genutzt wurden, um mithilfe der aus ihnen konstruierbaren Informationen Änderungen nachzuvollziehen, wurden keine sinnvollen Commits getätigt. Stattdessen wurden Änderungen häufig ein einziges Mal zum Feierabend hin kommentarlos eingecheckt. Das Fehlen sauberer Commits wiederum verhinderte, dass irgendein Nutzen aus Subversion gezogen werden konnte, abgesehen von der zentralen Speicherung von Dateien und der Synchronisation zwischen den Entwicklern. Das Nachvollziehen von Änderungen fiel schwerer, da keine Commitnachricht vorhanden war, ebenso wie beim Aktualisieren der Arbeitskopie mit den Änderungen eines Kollegen keine Möglichkeit vorhanden war, die getätigten Änderungen ohne Blick in den Code zu überblicken.

3.4 Eclipse-Integrationen

Eclipse war die zum fraglichen Zeitpunkt der Migration, Ende 2009, im Unternehmen genutzte IDE zur Projektentwicklung. Dies lag sowohl an der guten Subversion-Unterstützung mit dem Plugin Subversive als auch an der Integration von Bugzilla[15].

Da die Plugins für Eclipse in etwa die gleiche Funktionalität wie die besprochenen Explorer-Erweiterungen bieten und im Anschluss an die Migration keiner der Entwickler weiterhin Eclipse verwendete, soll an dieser Stelle nur kurz auf die Plugins eingegangen werden.

3.4.1 Subversive

Subversive[16] integriert sich in die Datei- und Projektverwaltung von Eclipse und bietet Zugriff auf eine Reihe von Funktionen wie Commits, Diffs, Merges. Die Menge der Funktionen ist mit der von TortoiseSVN vergleichbar, abgesehen von den Overlay-Icons im Windows Explorer. Das Plugin erbt somit die gleichen Stärken (gelungene Integration in die Arbeitsabläufe) und Schwächen (keine Diffs beim Erstellen von Commits) von TortoiseSVN.

3.4.2 merclipse & MercurialEclipse

Um die Einarbeitung in eine neue GUI zu vermeiden, sollte ein Ersatz für Subversive gefunden werden. Zwei Plugins standen dafür zur Verfügung: merclipse und MercurialEclipse. Sie sollen im Folgenden kurz angesprochen werden.

merclipse

merclipse[17] ist ein Plugin für Eclipse 3.3 und 3.4, das von Whitney Sorenson entwickelt wurde. Es unterstützt die Grundfunktionen von Mercurial sowie Annotations und das Bereitstellen eines Repositories über HTTP (`hg serve`).

[15] http://www.bugzilla.org/
[16] http://www.eclipse.org/subversive/

Da das Plugin zum Zeitpunkt der Migration bereits seit zwei Jahren nicht mehr weiterentwickelt wurde und zudem einige Bugs aufwies, konnte es nicht für den produktiven Einsatz empfohlen werden. Insbesondere, da es nicht mit aktuellen Eclipse-Versionen kompatibel ist.

MercurialEclipse

Im Gegensatz zu merclipse wird MercurialEclipse[18] aktiv weiterentwickelt und steht zurzeit in Version 1.8 für Eclipse 3.6 bereit. Es stellt eine wesentlich ausgereiftere Integration zur Verfügung und kann wie TortoiseHg nicht nur Kernbefehle, sondern auch Erweiterungen wie `mq` benutzen.

Zum fraglichen Zeitpunkt (Ende 2009) stand Version 1.4 zur Verfügung. Diese ließ sich jedoch nicht installieren, sodass die noch ältere Version 1.3 getestet wurde. Wenngleich sie stabiler als merclipse war, schlossen auch hier Bugs wie Probleme beim Einlesen der Mercurialkonfiguration und die Inkompatibilität mit dem Pluginsystem von Eclipse einen produktiven Einsatz aus.
So wurde der in Mercurial konfigurierte Editor nicht gestartet, wenn für ihn weitere Kommandozeilen-Optionen als nur der Pfad zur Programmdatei angegeben waren. Außerdem war es unmöglich, das Plugin sauber zu deinstallieren oder wenigstens zu deaktivieren.

Nachdem der Einsatz von MercurialEclipse verworfen wurde, wurden keine weiteren Tests mit aktuelleren Versionen durchgeführt. Für eine Entscheidung, ob das Plugin heute einen gleichwertigen Ersatz von TortoiseHg darstellt, sollte daher unbedingt die aktuelle Version evaluiert werden.

3.4.3 Bewertung

Merclipse und MercurialEclipse erfüllten nicht die Anforderungen an Komfort und Stabilität, um produktiv eingesetzt zu werden. Gleichzeitig übertraf TortoiseHg sie in Sachen Funktionalität, sodass der Wunsch nach einer Eclipse-Integration recht schnell aufgegeben wurde.

Im Verlauf der nächsten Monate sind zudem die meisten Entwickler von Eclipse auf NetBeans[19] oder andere Editoren umgestiegen, sodass die Frage, welches Eclipse-Plugin die Nachfolge von Subversive antreten solle, schnell an Bedeutung verlor. Stattdessen stellte sich der Einsatz einer IDE-unabhängigen Software, die von allen Entwicklern genutzt und für die nur ein einmaliger Schulungsaufwand nötig war, als Vorteil heraus. Denn obwohl NetBeans eine integrierte Unterstützung für Mercurial mitliefert, wird diese von keinem Entwickler gegenüber TortoiseHg bevorzugt. Außerdem gab keiner der Entwickler in einer im Unternehmen durchgeführten Umfrage an, wegen der fehlenden Unterstützung in Eclipse nach NetBeans gewechselt zu haben.

[17] http://goldenhammers.com/merclipse/
[18] http://javaforge.com/project/HGE
[19] http://www.netbeans.org/

3.5 Zusammenfassung

Die folgende Tabelle soll die Erkenntnisse aus den vorangegangenen Abschnitten noch einmal zusammenfassen. Der Übersichtlichkeit halber soll hier kein Unterschied zwischen Konzept, Implementierung und grafischer Oberfläche gemacht werden, obwohl natürlich zum Beispiel andere Subversion-Oberflächen weitere oder weniger Funktionen bieten können.
Außerdem wird jeweils von der Standard-Konfiguration der Software ausgegangen.

	Subversion	Mercurial
Systemtyp	zentral	dezentral
partieller Checkout	ja	nein
Integritätsprüfung	nein	ja
private Commits/Branches	nein	ja
Wartung entfernter Repositories	ja	nein[20]
Versionierung leerer Verzeichnisse	ja	nein
Vers. von Copy/Move Operationen	ja, explizit	ja, implizit
Commit einzelner Hunks	nein	ja
Integration fremder Repositories	ja, über Externals	ja, über Subrepositories

Tabelle 3.4: Gegenüberstellung von Subversion und Mercurial

Zusammenfassend kann man sagen, dass Subversion Vorteile bietet, wenn die verwalteten Projekte einen vollständigen Klon des Repositories aus Platz- oder Zeitgründen nicht möglich sind. Außerdem ermöglicht der Repository-Explorer das einfache Verwalten entfernter Repositories. Ein weiterer wichtiger Unterschied zwischen beiden Systemen ist, dass Subversion in der Lage ist, auch leere Verzeichnisse zu versionieren, wohingegen Mercurial immer nur Dateien (und damit nur implizit Verzeichnisse) versioniert.

Mercurial auf der anderen Seite bietet eine implizite Integritätsprüfung durch das Verwenden der kryptografisch sicheren Prüfsummen und kann so Datenkorruption ohne weitere Software erkennen. Die Möglichkeit für Entwickler, Änderungen in privaten Branches zu versionieren und diese erst zu einem späteren Zeitpunkt zu veröffentlichen, ermöglicht einen flexibleren Projektablauf und mehr „Experimente“ mit dem Projekt. Da Änderungen bis in ihre einzelnen Hunks aufgeteilt werden können, können Commits sauberer (das heißt mit weniger querschneidenden Belangen) erstellt werden.

TortoiseHg bietet einen detaillierteren und vor allem auch präsenteren Blick auf das Projekt und erhöht so die Übersichtlichkeit. Gleichzeitig zeigt es den Wert sauber benannter und logisch getrennter Commits auf, da es nur mit diesen als Voraussetzung seine Stärken (wie den einfachen Export von Commits als Patches) ausspielen kann. Dies erfordert und ermuntert gleichzeitig zu einem disziplinierteren Umgang mit der Versionskontrollsoftware.

[20] Die Wartung ist nur über Umwege möglich und nicht direkt eine Funktion von Mercurial: Über einen SSH-Tunnel können Befehle an ein entferntes Repository gesendet werden.

4. Migration von Subversion nach Mercurial

Im Folgenden sollen die in Kapitel 3 erworbenen Kenntnisse in einen praktischen Kontext übertragen werden, bei dem eine bestehende Infrastruktur auf Subversion-Basis nach Mercurial migriert werden sollte. Zu diesem Zweck werden zuerst die Anforderungen und dann die jeweiligen Lösungen für die aufgeworfenen Probleme diskutiert. Dabei wird auf die jeweils eingesetzte Software (oder Mercurial-Erweiterung) und die möglichen Arbeitsabläufe eingegangen. Die entwickelten Algorithmen werden in Form von PHP-Quellcodes gegeben. Im Anschluss werden die Lösungen bewertet und ein Ausblick auf mögliche Verbesserungen und Erweiterungen gegeben.

Konventionen

Befehle, die auf der Shell eingegeben werden müssen, werden durch ein vorangestelltes `$` und `nichtproportionale Schrift` gekennzeichnet:

```
$ hg version
```

Zeilen, die dabei keine führenden Dollarzeichen besitzen, sind Ausgabe des jeweiligen Programms. Befehle, die für eine einzelne Zeile zu lang sind, werden umbrochen und eingerückt, wie in dem folgenden Beispiel.

```
$ hg convert --filemap [FILEMAP] --authormap [AUTHORMAP] ↩
    --branchmap [BRANCHMAP] SRC [DST]
```

Verzeichnisbäume werden in einer schematisierten Form dargestellt, wie am folgenden Beispiel zu sehen ist.

```
/
+-- directory1
+-- directory2
|   +-- subdirectory
|   +-- file2.txt
|   +-- file3.txt
+-- file.txt
```

4.1 Ausgangslage

Bevor auf die eigentlichen Schritte eingegangen werden soll, die für eine Migration notwendig sind, sollen die Vorbedingungen beschrieben werden. Diese sind für das Verständnis der in den kommenden Abschnitten erläuterten Maßnahmen von nicht zu unterschätzender Bedeutung.

Bei den betroffenen Projekten handelt es sich um Webseiten, die mit dem Open Source CMS SallyCMS[1] umgesetzt wurden, das auf PHP und MySQL basiert. Jedes Projekt besteht aus dem Kernsystem sowie einer Reihe zusätzlicher Komponenten, AddOns genannt. Da Projekte in der Regel kleiner als 10 MiB waren, wurden sie in einem einzelnen, großen Repository verwaltet, wobei jedes ein einzelnes Verzeichnis im Trunk einnahm. Das folgende Schema verdeutlicht den Aufbau des Repositories, das in diesem Dokument über die fiktive URL `https://repo/` zu erreichen sein soll.

```
/
+-- branches
+-- tags
+-- trunk
    +-- projekt1
    +-- projekt2
    +-- projekt3
    +-- ...
```

Zur Versionskontrolle wurde Subversion 1.4 eingesetzt. Da diese Version nicht in der Lage war, inkrementelle Merges durchzuführen, ohne interne Konflikte zu riskieren (vgl. [Ben08]), wurden Änderungen zwischen einem Projekt und dem Basis-Repository nur manuell zwischen den betroffenen Dateien kopiert. Dies erforderte zudem keinen Schulungsaufwand für Mitarbeiter im Umgang mit Branches und Merges. Hinzu kommt die Tatsache, dass nur wenige Projekte parallel entwickelt wurden und so die das teils zeitaufwendige Kopieren von Dateien zu keinen spürbaren Verzögerungen in der Entwicklung führte.

Tags kamen bei der Entwicklung nicht zum Einsatz. Die Koordinierung zwischen den Projekten erforderte keine Tags und eine Veröffentlichung von AddOns fand nicht kontrolliert statt (so wurden nur in unregelmäßigen Abständen manuell Versionen zum Download auf der Unternehmenswebsite bereitgestellt).

Im folgenden Abschnitt wird nun näher auf die Struktur eines einzelnen Projekts eingegangen.

4.1.1 Projektaufbau

Jedes Projekt besteht aus einer Kopie des SallyCMS-Basissystems, einer variablen Menge von zusätzlichen Komponenten (AddOns genannt) sowie projektspezifischen Designs (Templates) und Inhalten (Module und Assets wie CSS[2] und JavaScript). Im Folgenden werden Templates und Module als Develop-Daten bezeichnet.

[1] http://www.sallycms.de/
[2] Cascading Stylesheets

Die Struktur eines Projekts folgt einem immer gleichen Aufbau, den das folgende Schema veranschaulicht. Je nach Projektumfang und -anforderungen können weitere Verzeichnisse oder Dateien hinzukommen.

```
/trunk/projekt1
+-- develop
|   +-- templates
|   +-- modules
+-- assets
|   +-- css
|   +-- js
|   +-- images
+-- sally
    +-- include
    |   +-- addons
    |   |   +-- addon1
    |   |   +-- addon2
    |   |   +-- addon3
    |   +-- lib
    |   +-- views
    |   +-- ... (weitere Verzeichnisse der Basis)
    +-- media
```

Ein Projekt lässt sich grob in zwei Bereiche unterteilen.

projektunabhängig
: Die Bestandteile, die in jedem Projekt (nicht notwendigerweise in der gleichen Version) enthalten sind, befinden sich im Verzeichnis `sally`. Dort sind sowohl das Basissystem als auch die einzelnen AddOns (in `sally/include/addons`) abgelegt. Auch wenn sich dabei die Auswahl der AddOns pro Projekt unterscheidet, finden in der eigentlichen Umsetzung eines Projekts an ihnen keine Änderungen statt (sie sind damit statisch und könnten zwischen mehreren Projekten via Symlinks geteilt werden). AddOns stellen dabei Funktionalitäten über eine Schnittstelle bereit, die beim Entwickeln der projektspezifischen Dateien genutzt werden können.

projektspezifisch
: Die Dateien, die pro Projekt verschieden sind, werden in `develop` und `assets` abgelegt. Dabei handelt es sich um das Design sowie andere relevante Implementierungen (Formulare, Galerien, ...). Sie werden mit versioniert, aber nicht zwischen Projekten synchronisiert und entsprechen immer den jeweiligen Anforderungen des Projekts.

Die folgenden Abschnitte behandeln die Versionierung und das Zusammenstellen der *projektunabhängigen* Bestandteile.

4.1.2 AddOn-Entwicklung

AddOns stellen allgemeine Funktionen (wie zum Beispiel ein Gästebuch, Benutzerverwaltung, Shop, Suchfunktion etc.) zur Verfügung, die über die des Basissystems (Artikel- und Inhaltsverwaltung) hinausgehen. Sie werden so flexibel wie möglich entwickelt, um innerhalb eines Projekts nicht verändert, sondern nur konfiguriert, eingebunden und über ihr Interface angesprochen zu werden.

AddOns wurden ebenfalls in dem Repository verwaltet, das auch die Projekte beinhaltet. Sie wurden in `https://repo/trunk/sally_development/addons` in einem Verzeichnis pro AddOn versioniert und bei Bedarf während der im nächsten Abschnitt näher besprochenen Projekterstellung in ein Projekt kopiert.

4.1.3 Projekterstellung

Zum Erzeugen eines neuen Projekts wurde der in Abschnitt 3.3.1 auf Seite 21 angesprochene Repository-Browser von TortoiseSVN verwendet. Dieser unterstützt über Drag&Drop das Kopieren von Verzeichnissen, wobei serverseitig keine echten Kopien, sondern nur Verknüpfungen erstellt werden. Damit werden die Inhalte nicht dupliziert und der Speicherbedarf des Repositories steigt nur unmerklich.
Auf diese Weise wurde im ersten Schritt das Basisprojekt (ein leeres Projekt, das nur SallyCMS selbst enthielt) kopiert. Anschließend wurden die jeweils benötigten AddOns, Templates und Module auf die gleiche Weise in das neue Projekt kopiert.

Dieses Vorgehen hat den Vorteil, dass die Revisionsgeschichte der einzelnen Komponenten erhalten blieb: So zeigt das folgende Subversion-Kommando nicht nur die Kopier-Operation des AddOns `addon1` an, sondern auch die Änderungen, die sich vorher im Quellverzeichnis ergaben.

```
$ svn log https://repo/trunk/projetkt1/sally/include/addons/addon1
```

Die obige Kommandozeile ist damit gleichbedeutend mit[3]

```
$ svn log https://repo/trunk/sally_development/addons/addon1
```

Projekte als Branches

Eine weitere Möglichkeit, die Projektentwicklung zu organisieren, wäre die Verwendung von Branches gewesen. Dabei hätte das Basisprojekt als Trunk verwendet werden und die einzelnen Projekte jeweils Branches von diesem sein können. Auf diese Weise wäre es möglich gewesen, Projekte nachträglich durch einen Merge mit dem Trunk (der zwischenzeitlich weiterentwickelt wurde und neue Funktionen ohne Fehlerkorrekturen enthält) zu aktualisieren. Das gleiche Konzept hätte auch auf die einzelnen AddOns angewandt werden können.

Da aber, wie bereits in Abschnitt 4.1 auf Seite 32 besprochen, Subversion 1.4 zum Einsatz kam, wurde von dieser Möglichkeit abgesehen. Zwar sind die Kopien, die im vorangegangenen Abschnitt besprochen wurden, technisch gesehen auch Branches, allerdings wurden diese nicht als solche behandelt.

[3] Die Befehle sind nicht vollständig identisch: Die History des Projekts enthält neben den Commits aus dem AddOn auch noch den Commit, der beim Kopieren entstand.

Alternative: Externals

Subversion bietet einen weiteren Weg, einzelne Komponenten in ein Projekt (sowohl in echte Repositories als auch in einzelne Verzeichnisse) zu importieren: Über sog. „Externals“ können fremde Repositories verknüpft werden, wie es auch bei Symlinks in den Dateisystemen wie ext2, NTFS oder ReiserFS möglich ist.

Der Weg über die Kopien der Komponenten (also Kopien der Verzeichnisse im SVN-Repository) wurde aus zwei Gründen gewählt:

1. Subversion führt nur eine oberflächliche Kopie durch; die Dateiinhalte werden nicht kopiert, sondern nur verknüpft. Der Platzbedarf ist damit ohnehin minimal.

2. Die Kopie erlaubt es, projektspezifische Anpassungen an Komponenten zu tätigen, die bewusst nicht für weitere Projekte geeignet sind. Derartige Änderungen würden mit Externals das Nutzen von Branches im verknüpften Repository erfordern. Da auch bereits in Projekten keine Branches verwendet wurden (siehe vorangegangen Abschnitt), wurde diese Option wegen dem zusätzlichen organisatorischem wie Schulungsaufwand ausgeschlossen.

4.1.4 Arbeitsabläufe

Projekte wurden trotz der enthaltenen Komponenten als in sich geschlossen behandelt. Das bedeutet, dass Änderungen an einzelnen Komponenten (wenn nötig) nur in seltenen Fällen geplant in die Basis zurückgespielt wurden. Einzelne Korrekturen wurden so häufig übersehen und nicht aus dem Projekt in das jeweilige AddOn übertragen. Dieser Umstand ist auf die fehlende einfache Möglichkeit, Änderungen zwischen zwei Projekten oder zwischen einem Projekt und der Basis zu mergen zurückzuführen.

Für die Entwicklung bedeutete dies, dass eventuelle Korrekturen oft mehrfach in Projekten durchgeführt werden mussten. Das manuelle Mergen wurde in der Regel über einen einfachen Diff zwischen der Komponente im Projekt und der Basisversion durchgeführt; teilweise wurde die Basisversion auch vollständig durch die Version aus dem Projekt ersetzt, wenn die Änderungen zu umfangreich waren. Dabei sind sämtliche einzelne Commits verloren gegangen und die Revisionsgeschichte der Komponente wurde nur um einen einzelnen Sammelcommit der Form „Version aus Projekt X“ erweitert.

In die andere Richtung, beim Aktualisieren eines Projekts mit den jeweils aktuellen Basis-AddOns, verliefen Aktualisierungen von AddOns ähnlich: Es wurde ein vollständiger Diff erzeugt, der manuell übertragen wurde, oder das gesamte AddOn als Einheit kopiert. Das Basissystem, SallyCMS, eines Projekts wurde nie aktualisiert, da der Aufwand den zu erwartenden Verbesserungen nicht gerecht wurde.

Die durch die Sammelcommits verloren gegangene Semantik der Änderungen führte oft zu dem Problem, dass zwei unterschiedliche Versionen einer Datei nicht ohne genaue manuelle Prüfung korrekt zusammengeführt werden konnten. Durch das Übertragen der Änderungen in Sammelcommits ging die logische Reihenfolge derjenigen

Commits verloren, die in ihnen zusammengefasst wurden. Der Aufwand, in alten Projekten nach den Nachrichten der einzelnen Commits, war zu hoch und wurde daher selten betrieben, um beim Mergen von Korrekturen zurück in die Basis zu assistieren.

4.2 Ziele & Herausforderungen

Nachdem in den vorangegangenen Abschnitten die bestehende Infrastruktur beschrieben wurde, sollen nun die Ziele dargelegt werden, die bei der Migration nach Mercurial erfüllt werden sollten. Jedem Ziel wird ein gesonderter Abschnitt gewidmet, indem die Anforderung näher erläutert und eine Lösung präsentiert wird.

Die Ziele, die vom Unternehmen im Vorfeld aufgestellt wurden, lauten dabei in Reihenfolge ihrer Umsetzung wie folgt:

- Das zugrunde liegende Modell sollte weiterhin zentral ausgelegt sein. Dabei sollen weiterhin die Repositories auf einem internen Server vorgehalten werden, um jederzeit jedem Mitarbeiter zugänglich zu sein. Dies führt auch dazu, dass es immer genau eine Stelle gibt, an der die „offizielle“ Version eines Projekts abrufbar ist.

- Mitarbeiter sollten weiterhin eigenständig Projekte anlegen können. Ohne diese Möglichkeit würde ein Engpass bei der Projekterstellung entstehen, da nur einzelne, geschulte Mitarbeiter in der Lage wären, sie durchzuführen.

- Projekte sollten weiterhin in sich geschlossene Einheiten sein, in denen die einzelnen Komponenten als Kopien enthalten sind. Dies vereinfacht die Handhabung von Korrekturen und vor allem projektspezifischen Änderungen, die bewusst nicht in die Basis-Repositories zurückgespielt werden sollen.

- Die vollständige Versionsgeschichte sowohl der Basis als auch aller Komponenten sollten in jedem Projekt enthalten sein. Damit wird es möglich, Fehler und die Commits, die zu ihnen führten, direkt im Projekt zu finden, statt dafür die Versionsgeschichte in den jeweiligen Quell-Repositories durchsuchen zu müssen. Diese Erleichterung soll damit die Nutzung der Geschichte, die in jedem Repository vorliegt, fördern.

- Eine Integration in Eclipse sollte verfügbar sein, um die Einarbeitung in eine neue GUI möglichst zu vermeiden. Eclipse war die verwendete IDE zur Projektentwicklung und der Wechsel des Versionskontrollsystems sollte die Entwickler nicht auch noch zu einem Wechsel der IDE und ggf. weiterer Werkzeuge zwingen.

- Projekte sollten einfacher aktualisiert, das heißt mit den jeweils letzten Änderungen aus den Basis-Repositories von SallyCMS und den AddOns gemergt, werden können. Korrekturen aus anderen Projekten sollen somit häufiger in parallel entwickelten Projekten erscheinen. Dies sollte es ermöglichen, neue Funktionen und Korrekturen bereits während der Entwicklung des Projekts zu integrieren.

- Änderungen sollen häufiger und kontrollierter in die Basis zurückfließen. Dies ist eine notwendige Voraussetzungen, um Korrekturen möglichst schnell in die parallel entwickelten Projekte zu übertragen und so das mehrfache Korrigieren desselben Problems zu verhindern.

- AddOns sollten einfach und automatisierbar als Repository veröffentlicht werden können. Eventuelle Dritte sollten leicht Änderungen beisteuern können. Diese Forderung leitet sich aus dem Wunsch, alle AddOns als Open Source zu entwickeln, her und beeinflusst die interne Projektentwickler daher kaum.

In den folgenden Abschnitten sollen die einzelnen Punkte betrachtet und Lösungswege präsentiert werden. Die Forderung nach einem Eclipse-Plugin wurde recht schnell fallen gelassen, nachdem die Vorteile von TortoiseHg offensichtlich wurden. Die verfügbaren Plugins wurden in Abschnitt 3.4 auf Seite 27 beschrieben und für unzureichend geeignet befunden. Das obige Ziel, Eclipse-Integrationen zu finden, wurde nur der Vollständigkeit halber aufgeführt und hat im Folgenden keine weitere Bedeutung.

4.2.1 Vorbereitungen

Als erste Maßnahme wurde das große Subversion-Repository in eine Reihe kleinerer Repositories pro Projekt zerlegt. Dies ist notwendig, damit Mitarbeiter Projekte einzeln abrufen können und nicht immer ein großes Repository mit Daten zu allen Projekten vorhalten müssen. Außerdem würde bei der Verwendung eines einzelnen, großen Repositories die Entwicklung mehrerer Projekte quasi unmöglich werden, da jeder Commit durch die in Abschnitt 3.2.2 auf Seite 17 angesprochenen Prüfsummen das gesamte Repository beeinflusst und Mitarbeiter auf diese Weise erst Änderungen aus anderen Projekten mergen müssen, bevor sie ihre (inhaltlich unabhängigen) Commits durchführen können.

Mercurial liefert die für die Konvertierung benötigten Werkzeuge in Form der `convert`-Erweiterung[4] bereits mit:

```
$ hg convert --filemap [FILEMAP] --authormap [AUTHORMAP] ↩
     --branchmap [BRANCHMAP] SRC [DST]
```

`SRC` gibt dabei die URL zum Quellsystem (u. a. Subversion, Git oder CVS) an, während `DST` das Verzeichnis ist, in dem das konvertierte Repository abgelegt wird. Bis auf `SRC` sind alle Parameter des Befehls optional.

Über die `FILEMAP` können Dateien ausgeschlossen oder umbenannt werden. Die angegebene Datei enthält pro Zeile eine Anweisung, wobei die drei Aktionen `rename`, `include` und `exclude` möglich sind. Das folgende Beispiel würde die Datei `a.txt` umbenennen und die Datei `lizenz.txt` ausschließen.

```
rename a.txt b.txt
exclude lizenz.txt
```

[4] [Div11a]

Die `AUTHORMAP` dient der Übersetzung von Nutzerkennungen und wurde verwendet, um die Subversion-Namen (beispielsweise „christoph“) in das übliche Mercurial-Format zu bringen:

```
christoph = Christoph <christoph@webvariants.de>
admin = Admin <admin@webvariants.de>
```

Da die `convert`-Erweiterung bei Subversion als Quellsystem den letzten Pfadbestandteil als Branchnamen ansieht, würde eine Konvertierung wie

```
$ hg convert https://repo/trunk/myproject myproject-converted
```

ein neues Repository mit einem einzelnen Branch namens `myproject` anlegen. Um die spätere Verarbeitung der Repositories zu erleichtern, wurden bei der Konvertierung diese Branches in `default` (dem Mercurial-Standardbranch) umbenannt.

```
myproject default
anotherproject default
```

4.3 Zentralisierte Entwicklung

Wie bereits in Abschnitt 3.1.2 auf Seite 13 erwähnt, unterstützen dezentrale Systeme eine Vielzahl verschiedener Entwicklungsmodelle. Da das Modell nicht grundlegend verändert werden sollte, sei an dieser Stelle auf [Cha09] und [Ott09] für eine Erläuterung weiterer Möglichkeiten wie mehrstufige Review-Prozesse oder Strukturen, die die Organisation des Unternehmens in einzelne Abteilungen widerspiegeln, verwiesen.

Da ein zentraler Ansatz weiterhin sinnvoll erschien, um immer eine definierte Masterversion eines Projekts zu erhalten (die gleichzeitig für alle Mitarbeiter erreichbar ist), wurden die Repositories auf dem gleichen Server, der auch schon Subversion betrieb, abgelegt. Damit werden mithilfe einer dezentralen Software (Mercurial) weiterhin ein zentrales Modell implementiert und so die Vorteile aus beiden Welten (zentrale Anlaufstelle und selbst organisierte Entwicklung) kombiniert.
Gleichzeitig wurde die Gelegenheit genutzt, die Projekte stärker zu strukturieren, sodass sie nun wie folgt abgelegt wurden:

```
/
+-- Sally-Projekte
|   +-- Projekt1
|   +-- Projekt2
+-- Sally-Basis
|   +-- AddOns
|   |   +-- AddOn1
|   |   +-- AddOn2
|   +-- sally_base
|   +-- sally_base-0.1
|   +-- sally_base-0.2
+-- Internes
    +-- Projekt3
    +-- Projekt4
```

Die so vorgehaltenen Repositories werden über das von Mercurial mitgelieferte hgwebdir[5] veröffentlicht und stehen sowohl intern als auch für externen Zugriff bereit. Ein eigenständiger Webserver, in diesem Fall Apache[6], wird zur Authentifikation eingesetzt.

4.3.1 Konfiguration eines Apache-Webservers

Der Webserver stellt die Basis des Zugriffs über HTTPS bereit, dient zur Zugriffskontrolle und kann wie in Listing A.1 auf Seite 79 konfiguriert werden. Die dabei wichtigste Einstellung ist, dass alle Requests auf Port 10109 zu `localhost:8001` weitergeleitet werden (`ProxyPass`) und die Benutzer über Basic Auth authentifiziert werden. Damit leitet Apache die Anfragen direkt an hgwebdir weiter, wenn eine gültige Authentifizierung vorliegt.

Die Benutzer werden in der Datei `/srv/hg/htpasswd` konfiguriert und dort zeilenweise mit Benutzernamen und Passwort eingetragen. Dabei ist zu beachten, dass die Benutzernamen von Apache nichts mit den Autoren der Commits in den Repositories zu tun haben.

4.3.2 Konfiguration von hgwebdir

hgwebdir ist eine Erweiterung für Mercurial, die eine Menge von Repositories über HTTP oder HTTPS zur Verfügung stellt und fungiert damit als Server. Die Konfiguration erfolgt wie bei Mercurial über eine einfache ini-Datei, die wie in A.2 auf Seite 80 gezeigt aufgebaut sein muss.

Die angeführte Konfiguration erlaubt einen Push von jedem Benutzer (`allow_push`) und dabei auch, kein SSL zu verwenden (`push_ssl`). Da Apache bereits die Benutzer authentifiziert hat, müssen diese hier nicht mehr validiert werden. Da der Proxy die Requests auf HTTP weiterleitet, muss diese Option gesetzt werden.
Im Abschnitt `[paths]` wird festgelegt, dass die HTTP-Wurzelebene auf `/srv/hg` zeigen soll und dort rekursiv alle Verzeichnisse nach Repositories durchsuchen soll.

Der Mercurial-Prozess kann wie folgt gestartet werden:

```
$ /usr/bin/python2.6 /usr/bin/hg serve -d -p 8001 --webdir-conf config
```

Im Anschluss können Repositories über `https://hg.webvariants.de:10109/` abgerufen werden.

4.3.3 Vorteile von Apache & hgwebdir

Die Kombination mit einem Webserver ermöglicht eine einfache Verwaltung von Accounts und stellt gleichzeitige eine performante Lösung dar, da Apache bereits auf dem Server zum Einsatz kam und so kein zweiter Webserver ausgeführt werden muss.

Mitarbeiter können über das von hgwebdir erzeugte Webinterface einzelne Projekte auswählen und über deren URL das Repository klonen. Für kleine Aufgaben (wie

[5] http://mercurial.selenic.com/wiki/HgWebDirStepByStep
[6] http://httpd.apache.org/

das Überprüfen des letzten Commits oder den Inhalt einer Datei abzurufen) reichen die Fähigkeiten von hgwebdir bereits vollständig aus und stellen so, was den lesenden Zugriff angeht, in etwa den gleichen Funktionsumfang bereit, den auch Subversion bzw. TortoiseSVN anbieten.

4.3.4 Einschränkungen

Wie bereits in Abschnitt 3.5 auf Seite 29 angesprochen, bietet Mercurial keinen direkten Weg, um auf einem Server Repositories anzulegen. Zwar könnten per SSH-Tunnel Mercurial-Kommandos auf dem Server ausgeführt werden, dies würde aber entweder einen allen bekannten, gemeinsamen Benutzeraccount erfordern oder einen Account pro Mitarbeiter. Da beide Wege einem Entwickler zu viel Macht gäben, musste ein anderer Weg gefunden werden, der im Folgenden beschrieben wird.

4.4 Administration des Servers

Im vorangegangenen Abschnitt wurde erwähnt, dass es nicht möglich (bzw. nicht praktikabel) ist, mit Mercurial Kommandos auf einem entfernten System auszuführen. Um Projekte zu erzeugen ist jedoch mindestens das Anlegen von Repositories als Klone des Basisprojekts sowie das Pullen und Mergen von AddOns notwendig. Um diese Aufgaben serverseitig umzusetzen, musste eine Schnittstelle geschaffen werden, die idealerweise per Webbrowser zu bedienen ist und die notwendigen Aufgaben erledigen kann.

Diese Anforderung wurde über ein selbst entwickeltes Webinterface umgesetzt, das bei der Verwaltung der Repositories unterstützt und für die häufig benötigten Abläufe Assistenten anbietet. Ein Benutzer muss dabei keinerlei Mercurial-Kenntnisse besitzen. Die dafür im Unternehmen im Rahmen dieser Arbeit entwickelte Software namens „Dexter“ arbeitet auf Basis von PHP 5.2 und kommuniziert über `exec()`[7] mit Mercurial. PHP wurde gewählt, da es die Kernkompetenz des Unternehmens darstellt, einfach zu implementieren und ideal für ein Webinterface zu verwenden ist. So ist der Code, der nötig ist, um Mercurial anzusteuern, in einer einzelnen Funktion wie in Listing A.3 auf Seite 80 zu schreiben. Die eigentliche Anwendung basiert auf EuropaPHP[8] 1.0.2, einem sehr minimalen MVC-Framework.

Repositories anlegen

Dexter bietet unter anderem einen Assistenten an, der in der Lage ist, Repositories anzulegen. Dazu werden neben dem Namen auch noch weitere Informationen wie der Name einer Kontaktperson und eine Beschreibung (beide optional) abgefragt. hgwebdir verwendet diese Angaben und zeigt sie, wenn vorhanden, in der Liste der Repositories an. Sie müssen innerhalb des neuen Repositories in die Datei `.hg/hgrc` eingetragen werden. Der dazu nötige Code sieht in einer auf das Wesentliche reduzierten Version aus wie in Listing A.4 auf Seite 81.

[7] Eine in PHP vordefinierte Funktion, die das Ausführen von externen Programmen ermöglicht und deren Rückgabewert sowie Ausgabe an das PHP-Skript übergibt.

[8] http://europaphp.org/

4.4.1 Vorteile

Einzelne Entwickler können über Dexter ohne Zusatzprogramme auf ihren Rechnern Repositories und Projekte (siehe nächster Abschnitt) anlegen. Webbrowser sind im Unternehmen verständlicherweise zur Genüge vorhanden.[9] Durch die Verwendung eines Frameworks kann die Anwendung schnell und einfach um neue Funktionen erweitert werden, wie zum Beispiel um eine Auswertung, welche Commits in Projekten wieder in die jeweilige Basis zurückgemergt werden müssen (vgl. Abschnitt 5.2 auf Seite 59).

4.4.2 Einschränkungen

Auch wenn Dexter die Projekterzeugung und einige weitere, sehr spezielle Funktionen unterstützt, bietet es keinen vollständigen Ersatz für TortoiseSVN. Änderungen an den Arbeitsabläufen müssen daher in die Implementierung übertragen werden und können nicht einfach in einer angepassten Nutzung eines generellen Management-Werkzeugs wie eben TortoiseSVN resultieren.

Für viele Aufgaben (wie das Überprüfen einzelner Commits ohne das Repository klonen zu müssen, der Überblick über alle Repositories, sowie das Betrachten einzelner Dateien) kommt weiterhin hgwebdir zum Einsatz. Es bietet mit seinem zumindest lesenden Zugriff auf die Repositories einen brauchbaren Ersatz für TortoiseSVN.

4.5 Projekte erzeugen

Nachdem nun besprochen wurde, wie Mercurial prinzipiell mit PHP angesteuert werden kann, sollen diese Funktionen nun konkret zur Projekterzeugung verwendet werden. Da dieser Vorgang recht komplex ist und für sein Verständnis eine Reihe weiterer Mercurial-Kenntnisse erforderlich sind, sollen diese nun noch kurz erläutert werden, bevor mit der eigentlichen Implementierung begonnen wird.

Wiederholung: Aufbau eines Repositories

Wie bereits in Abschnitt 3.2.2 auf Seite 17 besprochen, sind Repositories in Mercurial unveränderlich, was ihren Inhalt betrifft. Das bedeutet, dass bei einer Änderung an einem oder mehreren Commits ein neues Repository entsteht. Dies tritt auch auf, wenn zum Beispiel alle Dateien in einem Repository verschoben werden, wie es im folgenden Prozess notwendig sein wird. Das neue Repository erhält dabei neue Prüfsummen, die sich von denen des alten Repositories unterscheiden.

Wiederholung: Tags

Tags sind Alternativnamen für Revisionen. Innerhalb eines Repositories sind sie immer eindeutig und werden in einer einfachen Textdatei namens `.hgtags` im Wurzelverzeichnis des Repositories verwaltet und mitversioniert. In dieser Datei steht eine zeilenweise Abbildung der Prüfsummen auf das jeweilige Tag.

[9] Das Unternehmen entwickelt Webseiten.

4.5.1 Grundidee

Gemäß den Anforderungen aus Abschnitt 4.2 auf Seite 36 sollen Projekte durch jeweils ein einzelnes Repository verwaltet werden. Dieses Repository ist ein Klon des Basissystems (SallyCMS), der noch keine AddOns oder projektspezifische Inhalte (wie Templates oder Module, vgl. Abschnitt 4.1.1 auf Seite 32) enthält. Erst im Anschluss an das Klonen werden die benötigten AddOns dem Projekt hinzugefügt. Dabei werden ihre einzelnen Repositories in das Projekt-Repository gemergt.

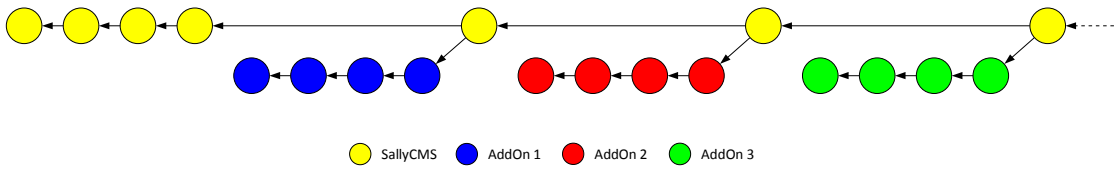

Abbildung 4.1: Revisionsgraph eines Projekts mit drei AddOns

In Abbildung 4.1 wird die entstehende Struktur im Repository verdeutlicht. Die ersten vier gelben Kreise spiegeln die Commits im SallyCMS-Repository wider. Diese werden ungeändert beim Klonen übernommen. Im Anschluss werden die AddOns einzeln (in keiner bestimmten Reihenfolge) in den Klon gemergt, wobei jeweils ein neuer Mergecommit entsteht. Am Ende enthält das Repository die Basis und drei AddOns.

4.5.2 Konzept der AddOn-Umbenennungen

AddOns werden in einzelnen Repositories versioniert. Ihr Aufbau ist dabei immer ähnlich und folgt dem untenstehenden Schema. Dabei ist zu beachten, dass die Dateien eines AddOns immer im Wurzelverzeichnis des jeweiligen Repositories liegen.

```
/
+-- assets
+-- lib
|   +-- ClassA.php
|   +-- ...
+-- config.inc.php
+-- version
```

Damit ergibt sich die Frage, wie verschiedene AddOns in einem Projekt gemergt werden können und wie sie an ihren Bestimmungsort (das Verzeichnis `sally/include/addons/`) gelangen. Würde man die Repositories direkt mergen, würden unter anderem die Dateien `config.inc.php`, `version` usw. zu Konflikten führen. Außerdem lägen dann alle Inhalte der AddOns im Wurzelverzeichnis des Projekts.

Zu diesem Zweck werden die Repositories, bevor sie in ein Projekt gemergt werden, erst noch vorverarbeitet, d. h. konvertiert. Diese Konvertierung wird dabei ausschließlich die enthaltenen Dateien in das passende Unterverzeichnis verschieben, sodass sich ein neues Repository wie folgt ergibt:

```
/
+-- sally
    +-- include
        +-- addons
            +-- addon1
                +-- assets
                +-- lib
                |   +-- ClassA.php
                |   +-- ...
                +-- config.inc.php
                +-- version
```

Zu sehen, dass beispielsweise die Datei `config.inc.php` des AddOns in `sally/include/addons/addon1/config.inc.php` umbenannt (verschoben) wurde. *Dieser Umbenennungsschritt ist das Kernelement der gesamten Projekterzeugung.* Die so vorbereiteten AddOns können konfliktfrei in ein Projekt gemergt werden.

Alternative: AddOns direkt im Unterverzeichnis versionieren

Die Alternative dazu, AddOns vor einer Projekterzeugung erst zu konvertieren, wäre, in den AddOn-Repositories bereits die oben gezeigte Pfadstruktur widerzuspiegeln. Dieses Vorgehen wurde jedoch aus folgenden Gründen verworfen:

1. Sollte sich das Pfadschema ändern, müssten die Inhalte verschoben oder das Repository konvertiert werden. Dies trat in der früheren Entwicklung bereits auf, als das Verzeichnis, in dem AddOns liegen, von `redaxo/include/addons` in `sally/include/addons` geändert wurde.[10] Dies würde evtl. vorhandene Klone nutzlos machen (vgl. Abschnitt 3.2.2 auf Seite 17).

2. Die drei zusätzlichen Verzeichnisebenen wurden als unnötiger Ballast empfunden.

3. Bei der Veröffentlichung der AddOns müssten die Verzeichnisse wieder entfernt werden, da sie in einem zum Download bereitgestellten Archiv nicht enthalten sein sollten.

Schema der Projekterzeugung

Der diskutierte Merge-Prozess ist automatisierbar. Da das Projekt-Repository die Ziel-Verzeichnisse für die AddOns noch nicht enthält (dies wird per Konvention während der Arbeit am Basissystem sichergestellt), können keine Konflikte auftreten. Die Repository-Inhalte sind also disjunkt (sowohl zwischen der Basis und einem AddOn als auch zwischen zwei AddOns).

Abbildung 4.2 auf der nächsten Seite enthält eine schematische Darstellung des Prozesses. Es zeigt, wie die Original-Repositories erst konvertiert werden, bevor sie durch einen Pull und einen Merge in das Projekt integriert werden können. Unter den Repositories ist der Pfad einer Beispieldatei, der Lizenz in `LICENSE`, angegeben.

[10] SallyCMS ist ein Fork des CMS REDAXO. Bei der Entwicklung der Version 0.3 wurde das `redaxo`-Verzeichnis in `sally` umbenannt.

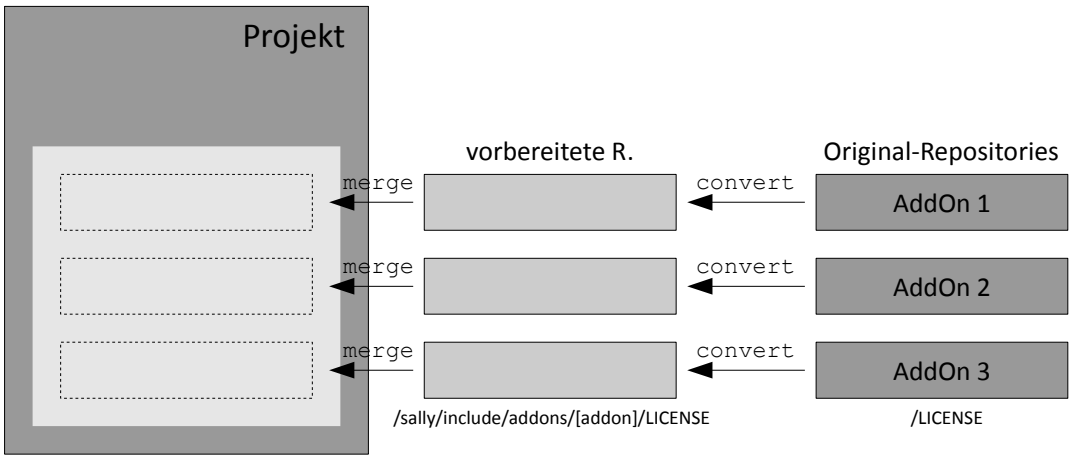

Abbildung 4.2: Schema der Projekterzeugung

4.5.3 AddOns vorbereiten

Das besprochene Verfahren zum Vorbereiten der AddOns soll nun in die Praxis umgesetzt werden. Zu diesem Zweck kommt die schon in Abschnitt 4.2.1 auf Seite 37 genutzte `convert`-Erweiterung zum Einsatz. Diesmal wird allerdings nicht das Versionskontrollsystem gewechselt, sondern von Mercurial nach Mercurial konvertiert. Zusätzlich kommt eine Filemap zum Einsatz, die die eigentliche Umbenennung vornimmt.

Die benötigte Filemap umfasst eine einzelne Zeile und sieht für ein AddOn `test` wie folgt aus:

```
rename . sally/include/addons/test
```

Da Filemaps nicht in dem Aufruf von Mercurial direkt angegeben werden können, müssen sie in Dateien abgelegt werden. Ein Skript könnte dazu beispielsweise eine temporäre Datei (z. B. in `/tmp`) anlegen. Die Datei muss dann wie folgt verwendet werden:

```
$ hg convert --filemap=/tmp/map_test.txt test test-converted
```

Dieser Befehl würde ein Repository `test` nach `test-converted` verarbeiten und dabei die Dateien verschieben. Dieser Vorgang muss einmal pro AddOn ausgeführt werden.

4.5.4 Sonderfall Tags

In Abschnitt 3.2.2 auf Seite 17 wurde bereits angesprochen, dass Mercurial Tags in einer regulären Datei namens `.hgtags` verwaltet. Sie enthält pro Zeile ein Tag mit der dazugehörigen Prüfsumme und kann wie folgt aussehen:

```
d7b759d93c40836ef14ec51a087ece253a871809 v2.0.0
399f38c6b8a4e49d56e5c1091fac524726c4d380 v2.0.2
07a16a24ffc947f06a3e6109d427a26afcccdb11 v2.2.0-rc1
5196c7f827f027d602dd69016167bf6a95e46a45 v2.2.0-rc2
```

Das genaue Format wird erst in einem späteren Kapitel von Bedeutung sein, im Moment genügt das Wissen um den zeilenweisen Aufbau.

Die `convert`-Erweiterung wird die `.hgtags`-Datei wie jede andere im Repository behandeln und sie beim Umbenennen ebenfalls nach `sally/include/addons` ↩ `/test/.hgtags` verschieben. Da sie dort allerdings nicht mehr von Mercurial ausgewertet wird, muss sie noch einmal gesondert behandelt werden. Es genügt hierbei jedoch nicht, nur eine Kopie im Wurzelverzeichnis des neuen Repositories abzulegen, da sich durch das Verschieben der Dateien die Prüfsummen aller Commits geändert haben.

Komfortablerweise kümmert sich die angesprochene Mercurial-Erweiterung bereits vollautomatisch um die Verarbeitung der Tags. Dabei werden alle im Original-Repository vorhandenen Tags migriert. Dazu werden die Original-Prüfsummen den jeweils neuen zugewiesen, sodass aus der obigen Datei beispielsweise die folgende entstehen könnte:

```
7df8d81ceb9dae873f7d662ef8a31424850a2328 v2.0.0
9c1c2f98825c1bc96837ed535a121cc421b47c40 v2.0.2
f284a5ae760c0f74c3c7471b77f2e73791da89b5 v2.2.0-rc1
14e2120acb87d1e44cdb85fb03a86145b188f326 v2.2.0-rc2
```

Die Übersetzung der alten in die neuen Prüfsummen findet in einem neuen, von der Erweiterung angelegten Commit mit der Nachricht „update tags“ statt. Auf diese Weise werden die Tags in das neue Repository übernommen. Übrig bleibt eine wertlose Datei `sally/include/addons/test/.hgtags`, die im Folgenden als letzter Schritt entfernt werden kann. Sie ist nicht von Bedeutung, da Mercurial die `.hgtags`-Datei nur im Wurzelverzeichnis eines Repositories auswertet.

Die Tags stellen die Projekterzeugung jedoch vor ein Problem: Falls mehrere AddOns die gleichen Tags definieren, würden diese beim Mergen zu Konflikten führen. Außerdem stellt sich die Frage, welchen Wert ein Tag „v1.0“ in einem Projekt hat, wenn man nicht weiß, ob es sich auf das Basissystem, ein AddOn oder gar das Projekt bezieht.

Beide Probleme, die Mehrdeutigkeit und die überflüssige Datei, können über einen einzelnen, zusätzlichen Commit behandelt werden. Im Folgenden soll das Vorgehen allgemein und anschließend ein optimiertes Vorgehen erläutert werden, bei dem der von Mercurial eingefügte „update tags“-Commit wiederverwendet wird.

Ad-hoc-Lösung

Um zu vermeiden, dass zwei gleiche Tags[11] beim Merge kollidieren, kann man sie einfach mit dem Namen des AddOns versehen, sodass aus „v1.0“ das Tag „test v1.0“ wird. Im resultierenden Projekt-Repository vermeidet dies nicht nur die Kollision, sondern erhöht auch noch die Übersicht, da ein Tag sofort einem AddOn zugeordnet werden kann.
Es besteht hierbei allerdings die Gefahr, dass in dem Basisprojekt bereits ein derartiges Tag gesetzt ist. Dies würde das automatische Mergen wesentlich erschweren und sollte daher beim Aufsetzen eines Systems bedacht werden. Die Wahrscheinlichkeit,

dass im Basis-Repository (d. h. in SallyCMS) ein Tag namens „test v1.0“ gesetzt wird, wird als gering genug angesehen, um auf diesen Fall nicht weiter einzugehen. Im Unternehmen trat dieser Fall bisher nicht ein, auch da Tags sparsam und nur für Versionsnummern in der Form „vX.Y.Z“ verwendet werden.

Das Ziel ist also, aus einer Datei wie

```
399f38c6b8a4e49d56e5c1091fac524726c4d380 v2.0.2
d7b759d93c40836ef14ec51a087ece253a871809 v2.0.0
07a16a24ffc947f06a3e6109d427a26afcccdb11 v2.2.0-rc1
5196c7f827f027d602dd69016167bf6a95e46a45 v2.2.0-rc2
```

eine Datei wie

```
399f38c6b8a4e49d56e5c1091fac524726c4d380 test v2.0.2
d7b759d93c40836ef14ec51a087ece253a871809 test v2.0.0
07a16a24ffc947f06a3e6109d427a26afcccdb11 test v2.2.0-rc1
5196c7f827f027d602dd69016167bf6a95e46a45 test v2.2.0-rc2
```

zu erzeugen. Der dazu notwendige Algorithmus ist denkbar einfach: Jede Zeile muss anhand des ersten Leerzeichens in zwei Teile (`TEIL1` und `TEIL2`) aufgetrennt werden. Danach wird die Zeile durch `TEIL1 addon TEIL2` ersetzt, wobei `addon` der Name des jeweiligen AddOns ist. Der Algorithmus ist in Listing A.5 auf Seite 82 beispielhaft als PHP-Funktion implementiert.

Auf einem Unix-System kann diese Aufgabe mit einem einzelnen Aufruf von `sed` (ein Programm zum Manipulieren von Dateien) erledigt werden, wie es in Listing A.6 auf Seite 82 gezeigt wird. Der Code im Listing erledigt gleichzeitig noch die Aufgabe, die überflüssige Version der `.hgtags`-Datei zu entfernen, bevor der Commit durchgeführt wird.

Nachteilig ist hierbei jedoch, dass ein neuer Commit erzeugt wird, der eigentlich nicht nötig ist. Im folgenden Abschnitt wird daher eine Möglichkeit erläutert, die den schon vorhandenen „update tags“-Commits wiederverwendet und so einen Commit einsparen kann.

Lösung mit Mercurial Queues

Bisher wurde erklärt, dass einmal getätigte Commits in einem Mercurial-Repository unveränderlich sind. Diese Tatsache hat auch weiterhin Bestand. Allerdings gibt es eine Möglichkeit, die Commits nicht als finale Einträge in der Revisionsgeschichte zu betrachten, sondern als Folge von noch veränderbaren Patches. Diese Funktion wird von der Mercurial-Erweiterung `mq`[12] (Kürzel für „Mercurial Queues“) bereitgestellt, die in der Lage ist, einen „Stapel“ von Patches auf einem Repository zu verwalten. Dies soll nicht darüber hinwegtäuschen, dass Änderungen an den Patches

[11] Das Tag `tip` besteht nur virtuell und zeigt immer auf die letzte Revision. Es wird daher bei Merges nicht zu Konflikten führen und taucht auch nicht in `.hgtags` auf.

auch die Prüfsummen der Commits beeinflussen, die aus ihnen erzeugt werden. Die genaue Funktionsweise der Erweiterung soll nicht Thema dieser Arbeit sein, sodass für weitere Informationen auf die in der Fußnote verlinkte Ressource verwiesen sei.

Prinzipiell versetzt die Erweiterung den Benutzer unter anderem in die Lage, die Commits wieder in einen veränderbaren Zustand zu versetzen. Dazu werden sie als Patches innerhalb des Verzeichnisses `.hg/patches` abgelegt. Über die bereitgestellten Befehle kann zwischen den Patches gewechselt, weitere Commits in Patches überführt oder Patches wieder als Commit abgeschlossen werden. Dieses Vorgehen soll nun auf den bereits genannten „update tags“-Commit angewandt werden. Dazu soll kurz auf die benötigten Befehle eingegangen werden, die von `mq` zur Verfügung gestellt werden.

`hg qimport`
: Überführt einen oder mehrere bestehende Commits aus der Revisionsgeschichte in Patches und macht sie so wieder veränderbar. Die Commits bestehen dabei weiterhin in der Revisionsgeschichte des Repositories, werden aber über spezielle Tags markiert, damit die Erweiterung sie identifizieren kann. So ist es möglich, die Patches im Anschluss zu löschen, ohne die ursprünglichen Commits zu verlieren.

`hg qrefresh`
: Aktualisiert den aktuellen (zuletzt erzeugten) Patch mit den Änderungen aus der Arbeitskopie. Dabei wird der eigentliche Patch mit dem aktuellen Diff aktualisiert, ebenso wie der dazugehörige Commit aktualisiert wird. Damit einher geht eine Änderung der Prüfsumme des Commits.

`hg qfinish`
: Beendet die Arbeit an einem oder mehreren Patches und entfernt sie damit aus der Liste der Patches. Die Änderungen stehen danach nur noch als reguläre Commits im Repository bereit. Sie können jederzeit wieder in die Patch-Serie importiert werden, indem `qimport` ausgeführt wird.

Zur Nachbearbeitung der Tags kann dies genutzt werden, um den letzten Commit („update tags“) als Patch zu importieren und damit veränderbar zu machen. Listing A.7 demonstriert den Einsatz der oben genannten Kommandos. Damit entsteht ein neuer „update tags“-Commit (da sich dessen Diff ändert), der den alten ersetzt.

Wie bereits angesprochen stellt der Konvertierprozess eine konstante Abbildung von Commits dar. So entsteht (wenn die Parameter des Befehls identisch sind) beispielsweise aus einem Commit `2fd4f31e` immer ein neuer Commit `e6aad0fa`, unabhängig davon wie oft oder wann die Konvertierung durchgeführt wird. Einzige Ausnahme ist der „update tags“-Commit, da dieser erst „live“ im Anschluss an die Konvertierung erzeugt wird. Für ihn ergibt sich damit jedes Mal eine neue Prüfsumme. Dies führt dazu, dass zwei Projekte, die nacheinander erzeugt werden, für jeden der „update tags“-Commits jeweils verschiedene Prüfsummen enthalten. Damit ist es auch unerheblich, um welche Prüfsummen es sich dabei handelt. Und da diese somit irrelevant sind, ist es auch kein Problem, sie durch die Behandlung mit `mq` noch einmal zu verändern.

[12] http://mercurial.selenic.com/wiki/MqExtension

Abschluss

Es wurden zwei Möglichkeiten gezeigt, die Tags so zu korrigieren, dass sie konfliktfrei gemergt werden können. Die Variante, die `mq`-Erweiterung zu verwenden, sei dabei immer empfohlen, da sie einen unnötigen Commit vermeidet und den bereits vorhandenen „update tags“-Commit mit den Änderungen versieht, die gemäß seiner Commit-Nachricht auch in ihm geschehen sollten.

4.5.5 AddOns mergen

Nachdem alle AddOns vorbereitet wurden, können diese nun in das Basisprojekt gemergt werden. Dazu müssen die entsprechenden Commits erst in das Projekt importiert werden. Zu diesem Zweck wird `hg pull` verwendet:

```
$ cd baseproject
$ hg pull ../test-converted
pulling from ../test-converted
searching for changes
abort: repository is unrelated
```

Der obige Pull-Befehl schlägt allerdings fehl, da die beiden Repositories nichts miteinander zu tun haben (sie haben keine gemeinsame Revision). Mercurial verbietet das Pullen von Änderungen aus einem Repository, das nichts mit dem Ziel zu tun hat. Dies verhindert, dass aus Versehen aus einem falschen Repository gepullt wird.

Um diese Sicherheitsprüfung zu umgehen, kann der Parameter `-f` angegeben werden, um `pull` zum Weitermachen zu zwingen:

```
$ hg pull -f ../test-converted
pulling from ../test-converted
searching for changes
warning: repository is unrelated
adding changesets
adding manifests
adding file changes
added 60 changesets with 149 changes to 52 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
```

Die Ausgabe des Befehls zeigt am Ende noch, wie viele Commits (in Mercurial „changesets“ genannt) in das Repository übertragen wurden. Außerdem enthält sie die Information, dass nun ein weiterer Head im Repository vorhanden ist, und fordert zum Merge auf.

In Abbildung 4.3 auf der nächsten Seite wird die Struktur des Repositories illustriert. Die eben importierten Commits sind vorhanden, wirken sich aber noch nicht auf die Arbeitskopie (engl. „Working Copy“, WC) aus. Schematisch besteht das Repository nun also aus zwei unabhängigen Branches. Da die Arbeitskopie nur auf einer Revision basieren kann, muss das AddOn noch mit der Projektgeschichte gemergt werden, um in der Arbeitskopie aufzutauchen.

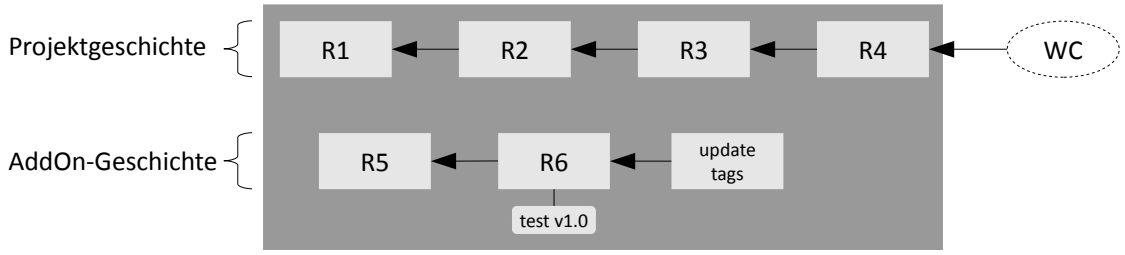

Abbildung 4.3: Projekt-Repository vor dem Merge des ersten AddOns

```
$ hg merge
merging .hgtags
52 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
```

Hierbei ist zu beachten, dass dies pro AddOn einzeln durchgeführt werden muss, da in Mercurial ein Merge nur genau zwei Parent-Commits haben kann.[13] Es ist also nicht möglich, alle AddOns auf einmal zu mergen. Nach dem Merge stellt sich das Repository wie in Abbildung 4.4 dar:

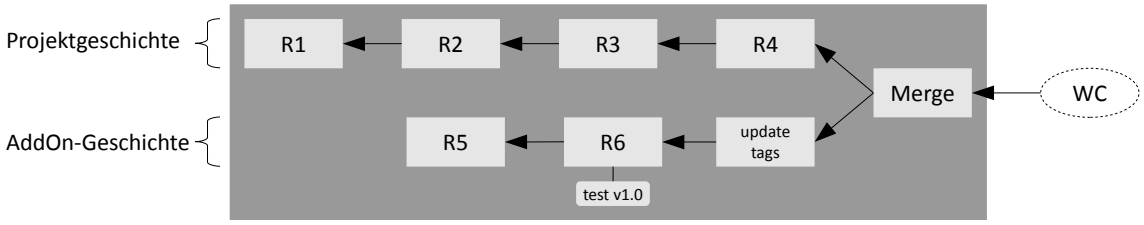

Abbildung 4.4: Projekt-Repository nach dem Merge des ersten AddOns

Da die beiden Branches sich in maximal einer Datei überschneiden, kann nur ein einzelner Konflikt auftreten: Mercurial ist nicht in der Lage, die `.hgtags` selbst zu mergen.[14] Im Folgenden soll daher gezeigt werden, wie dieser Konflikt automatisch aufgelöst werden kann. Er wird immer auftreten, wenn in zwei Repositories jeweils Tags gesetzt wird, da sich bereits Zeile 1 der jeweiligen `.hgtags`-Dateien unterscheiden werden, da sie verschiedene Prüfsummen enthalten.

Konfliktbereinigung

Um zwei Tagdateien automatisch zu mergen, ist es wichtig, ihren Aufbau zu verstehen:

- Beim Hinzufügen eines Tags wird eine Zeile mit der Prüfsumme und dem Namen des Tags ergänzt.
- Wird ein bestehendes Tag auf eine neue Revision gesetzt, werden zwei Zeilen hinzugefügt: Die erste enthält die Prüfsumme, auf die das Tag vorher zeigte, die zweite die neue Ziel-Prüfsumme.

[13] Dies ist einer der bedeutendsten Unterschied zu Git, das beliebig viele Parents für einen Commit unterstützt.

[14] http://mercurial.selenic.com/wiki/Tag

- Beim Löschen eines Tags wird wie beim Ändern eines Tags verfahren, nur dass die neue Prüfsumme die Nullsumme (`000..000`) ist.

Zusätzlich zu den obigen Hinweisen sollte bedacht sein, dass eine Revision mehrere Tags erhalten kann. Eine Abbildung von Prüfsumme auf Tag muss damit eine Abbildung auf eine *Liste* von Tags sein.

Der Konflikt, der in diesem Fall auftreten wird, ist kein „klassischer“ Konflikt, bei dem zwei Zeilen unterschiedlich sind und der Entwickler entscheiden muss, welche der beiden übernommen werden soll. Stattdessen haben die Zeilen inhaltlich nichts miteinander zu tun und befinden sich nur aufgrund des Aufbaus der Tag-Dateien an der gleichen Stelle. Die Lösung ist daher nicht, eine der beiden Zeilen auszuwählen, sondern *beide* beizubehalten und zu konkatenieren. Da Mercurial die Datei zeilenweise auswertet, führt das bereits zu dem gewünschten Effekt, dass im Anschluss die Tags beider Repositories erhalten bleiben.

Das Konkatenieren kann unter unixoiden Betriebssystemen von dem Programm `cat` ausgeführt werden, das die zwei Tag-Dateien aneinanderfügt und in die neue Datei schreibt.

Ein erweiterter Algorithmus ist jedoch in der Lage, die Datei zusätzlich aufzuräumen und nach Tags zu sortieren. Dazu reicht es prinzipiell, ein Dictionary zu befüllen, das eine Abbildung $tag \rightarrow hash$ enthält, und nach dem Einlesen beider Dateien das Ergebnis in eine neue `.hgtags` zu schreiben. In PHP implementiert könnte ein Programm wie Listing A.8 auf Seite 83 aussehen.[15] Das dargestellte (oder ein äquivalentes) Programm kann in Mercurial wie in Listing A.10 auf Seite 86 gezeigt, konfiguriert werden, um automatisch aufgerufen zu werden.

Die gezeigten Programme ermöglichen ein automatisches Lösen des Tag-Konflikts ohne menschliches Eingreifen. Sie bilden damit die Basis für eine erfolgreiche Projekterstellung. Die automatische Bereinigung der Tagdateien ist dabei kein Muss, aber mit steigender Projektgröße und Umfang der AddOns ein Nice-to-Have Feature, um dafür zu sorgen, dass die Tagdateien nicht übermäßig stark anwachsen. Dabei sollte jedoch bedacht sein, dass Mercurial auch mit großen Tagdateien keine prinzipiellen Probleme hat und die Performance nur minimal leiden würde.

4.5.6 Abschluss

Die gezeigten Techniken zur Vorbereitung der AddOns, Verarbeitung der Tags und das automatische Mergen derselben reichen aus, um bereits ein Projekt zu erzeugen. Listing A.11 auf Seite 87 enthält das Skript einer beispielhaften Projekterzeugung, bei der zwei AddOns (`usermngt` und `feeds`) in die Basis übernommen werden. Die Ausgabe der Befehle wurde auf die relevanten Stellen gekürzt, um nicht den Rahmen der Arbeit zu sprengen.

Es kann sich lohnen, als letzten Schritt noch ein zusätzliches Tag zu setzen, um die letzte Revision vor der eigentlichen Entwicklung in dem neuen Repository leichter zu identifizieren:

[15] Der Quellcode für eine C#-Anwendung findet sich ebenfalls im Anhang.

```
$ hg tag project_created
```

Weiterhin empfiehlt es sich, nach der Projekterzeugung die Arbeitskopie zu entfernen. Sie wird auf einem Server nicht benötigt, bei Push-Operationen nicht aktualisiert und verbraucht somit nur unnötig Speicherplatz:

```
$ hg up null
```

Das Projekt ist nun fertig erzeugt und steht zum Klonen durch die Mitarbeiter bereit. Die Projektentwicklung kann nun beginnen.

4.6 Revisionsgeschichte der AddOns

Die im vorangegangenen Kapitel gezeigte Technik erfüllt bereits die Forderung aus Abschnitt 4.2 auf Seite 36, sämtliche Revisionen der einzelnen AddOns beizubehalten. Die Revisionsgeschichte der AddOn steht weiterhin über `hg log` zur Verfügung, muss allerdings unter Angabe des Pfads eines AddOns gefiltert werden.

```
$ hg log sally/include/addons/test/
changeset:   1618:f5f3b6d09107
user:        convert-repo
date:        Tue Apr 19 12:12:17 2011 +0000
summary:     update tags

changeset:   1617:3834b48da67b
user:        Christoph <christoph@webvariants.de>
date:        Wed Dec 15 19:33:56 2010 +0100
summary:     correctly handle cache event (closes ticket #123)
```

Alternativ kann der `log`-Befehl auch im Verzeichnis des AddOns aufgerufen werden. Da er kontextsensitiv arbeitet, filtert er damit automatisch die History und zeigt nur die Änderungen des AddOns an.

```
$ cd sally/include/addons/test
$ hg log
```

Allerdings ist hierbei anzumerken, dass es sich nicht um die „echten“ Revisionen des AddOns handelt, sondern um konvertierten Entsprechungen. Durch den Umbenennungsvorgang aus Abschnitt 4.5.3 auf Seite 44 haben sich die Prüfsummen der einzelnen Commits geändert.

Projekte zurücksetzen

Wie bereits in Abschnitt 4.5.5 auf Seite 48 beschrieben, kann die Arbeitskopie eines Projekts immer nur auf genau einer Revision basieren. Da die einzelnen AddOns als eigenständige Branches in das Repository eingefügt werden, ist es damit nicht möglich, sie sinnvoll zurückzusetzen. Ein Blick auf Abbildung 4.4 auf Seite 49 verdeutlicht das Problem: Möchte man ein AddOn auf eine frühere Revision zurücksetzen (z. B. auf `R5` statt `R6`), so würde die Arbeitskopie auf dieser Revision basieren und damit ausschließlich die Dateien des AddOns, nicht aber die der Basis oder anderer AddOns enthalten.

Um Probleme durch Zurücksetzen zu identifizieren, sollte im einfachsten Fall das entsprechende AddOn in der Arbeitskopie gelöscht und an dessen Stelle ein Klon des Original-Repositories treten, mit dem derartige Operationen dann möglich sind. Mercurial wird ein Repository innerhalb der Arbeitskopie nicht beachten und während der Fehlersuche nur angeben, dass die gelöschten Dateien des AddOns fehlen. In dem AddOn-Repository kann die Arbeitskopie dann beliebig zurückgesetzt werden, ohne die Arbeitskopie des Projekts zu beeinflussen. Dabei kommt dem Vorgehen zugute, dass keine Änderungen in AddOns stattfinden und Konfiguration und Inhalte getrennt in einem anderen Verzeichnis liegen.

Nach Abschluss der Fehlersuche kann der Klon gelöscht und die Arbeitskopie dann wieder zurückgesetzt werden. Dabei werden die fehlenden AddOn-Dateien restauriert. Nun kann der Fehler entweder im Projekt als neuer Commit korrigiert oder in dem Original-Repository des AddOns korrigiert und dann über ein Update wie in Abschnitt 5.1.2 auf Seite 56 beschrieben in das Projekt eingespielt werden.

4.7 Zusammenfassung

In den vorangegangenen Abschnitten wurde gezeigt, wie Projekte, die aus mehreren Repositories zusammengesetzt werden, erstellt werden können. Es wurde besprochen, dass die einzelnen Projekte in einzelnen Repositories verwaltet werden sollten, und wie diese konkret erstellt werden.

Die Konvertierung der bestehenden Projekte aus dem Subversion-Repository stellt dabei einen einmaligen Prozess dar, der nichts mit der Erzeugung neuer Projekte zu tun hat. Die dabei generierten Repositories unterscheiden sich noch grundlegend von den neuen, da in ihnen jeweils nur ein Branch vorkommt, der sämtliche Änderungen (Basis, AddOns und Projektentwicklung) enthält. Diese Projekte können über die im folgenden Kapitel gezeigten Techniken nicht aktualisiert werden und wurden nur aus dem Subversion-Repository extrahiert, um nicht sowohl mit Subversion als auch Mercurial gleichzeitig arbeiten zu müssen, wenn an ihnen Wartungsarbeiten vorgenommen werden.

Der in Abschnitt 4.3 auf Seite 38 beschriebene Prozess zeigt, wie neue Projekte auf einem zentralen Server erzeugt werden können. Zu diesem Zweck werden die einzelnen Repositories, die die AddOns und das Basissystem enthalten, miteinander gemergt, um ihre Revisionsgeschichte zu vereinen. Dabei werden die AddOns vor dem Merge erst vorbereitet, um im entsprechenden Ziel-Verzeichnis zu erscheinen. Der Aufbau der Projekte stellt dann sicher, dass die durchzuführenden Merges fast

konfliktfrei ablaufen können. Der einzige Konflikt, der auftreten und von Mercurial nicht automatisch bereinigt werden kann, betrifft die Tags, die jeweils in den Repositories gesetzt sind. Um den Konflikt aufzulösen, wurde ein Algorithmus vorgestellt und implementiert, der die Tags beider Repositories beibehält und so semantisch korrekt mergen kann.

Im folgenden Kapitel wird die Projektentwicklung behandelt, die sich an die Erzeugung der Repositories anschließt. Dabei wird auf die Frage, wie Projekte nachträglich mit Korrekturen aus den Basis-Repositories versorgt und wie Änderungen aus dem Projekt in die Basis zurückfließen können, eingegangen.

5. Projektentwicklung

Das vorangegangene Kapitel hat gezeigt, wie Projekte erzeugt werden können, die aus mehreren, unabhängigen Komponenten bestehen. Nach der Erzeugung können die Repositories geklont und in ihnen gearbeitet werden, um das eigentliche Projekt umzusetzen. Die dabei entstehenden Commits werden als „Projektentwicklung“ bezeichnet und stellen (abgesehen von der Auswahl der Komponenten) den eigentlich kreativen Teil eines Projekts dar.

Im folgenden Kapitel werden nun weitere Techniken beschrieben, die die letzten drei Ziele aus Abschnitt 4.2 auf Seite 36 betrachten und es ermöglichen, ein Projekt zu aktualisieren, Änderungen zurückzuspielen und die Komponenten im Internet zu veröffentlichen.

Als grundlegende Richtlinie sei für die beschriebene Art der Projektentwicklung empfohlen, *keine Änderungen an der Basis (SallyCMS) oder den AddOns vorzunehmen.* Dies ermöglicht es, wie der folgende Abschnitt zeigen wird, konfliktfrei und damit automatisierbar zu mergen. Da sich diese Richtlinie in der Praxis jedoch selten als durchsetzbar erwiesen hat, geht Abschnitt 5.2 auf Seite 59 auf Möglichkeiten ein, gemachte Änderungen wieder in die Basis-Repositories zurückzuspielen.

5.1 Aktualisierung von Projekten

Bei der Aktualisierung eines Projekts werden Änderungen, die sich zwischenzeitlich in den Basis-Repositories ergeben haben, in das Projekt importiert. In den meisten Fällen werden damit Programmfehler korrigiert oder neue Funktionen hinzugefügt.

In Kapitel 4 auf Seite 31 wurde die Struktur und die Erzeugung von Projekten behandelt. Einer der großen Vorteile des Umstiegs auf Mercurial ist, dass Aktualisierungen der Projekte durch das nachträgliche Mergen der Quell-Repositories einfach und effizient möglich sind. So ist es im Vergleich zu Subversion problemlos möglich, Änderungen häufig zu mergen, ohne dabei Konflikte mit den internen Properties zu riskieren (vgl. [Ben08]). Auf diese Weise können Fehlerkorrekturen und neue Funktionen häufig und ohne großen Aufwand in Projekte eingespielt werden.

Für die bisher besprochenen Projekte ergeben sich zwei verschiedene Strategien, die im Folgenden behandelt werden sollen. Die beiden Strategien behandeln das Basissystem (SallyCMS) und die Aktualisierung der im Projekt enthaltenen AddOns.

5.1.1 Basis

Projekte sind gemäß des in Abschnitt 4.5 auf Seite 41 beschriebenen Verfahrens jeweils Klone des Basis-Repositories (SallyCMS). Damit ist es möglich, ohne spezielle Vorkehrungen einen regulären Pull durchzuführen, um sie zu aktualisieren. Der Ablauf ähnelt dem Hinzufügen eines AddOns:

```
$ hg pull https://repos/Sally-Basis/sally_base
$ hg merge
$ hg commit -m "updated base system"
```

Durch ein Update eines Projekts entsteht ein Revisionsgraph, wie er in Abbildung 5.1 dargestellt ist. Die Grafik zeigt ein Projekt mit einem AddOn. Nach der Projekterzeugung (gekennzeichnet durch das Tag „project_created“) fanden zwei Commits statt, in denen beispielsweise das Design einer Website oder eine bestimmte Funktionalität umgesetzt wurde (PProjektentwicklungïn der Abbildung). Im Anschluss wurde ein Update durchgeführt, bei dem drei neue Commits aus der Basis eingespielt (gemergt) wurden.

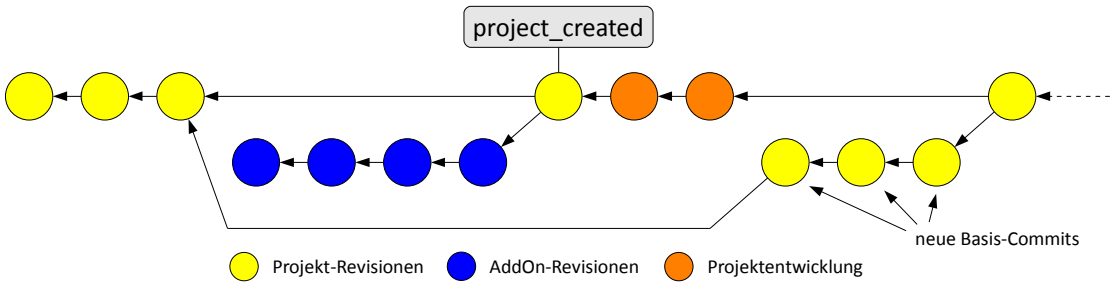

Abbildung 5.1: Revisionsgeschichte nach einem Basis-Update

Geht man davon aus, dass in einem Projekt keine Dateien des Basissystems geändert werden, ist ein Update konfliktfrei und damit automatisierbar. Bei Änderungen an der Basis müssen eventuell entstehende Konflikte manuell bereinigt werden.

5.1.2 AddOns

Das Aktualisieren von AddOns ist ein wesentlich komplexerer Vorgang als bei der Basis, da hierbei einige Besonderheiten berücksichtigt werden müssen. Während der Projekterzeugung wurden nicht die Original-Repositories in das neue Projekt gemergt, sondern konvertierte Versionen (vgl. Abschnitt 4.5.3 auf Seite 44). Damit ist es nicht möglich, Änderungen direkt aus den Originalen zu pullen, da sie sich in den Prüfsummen unterscheiden. Mercurial würde einen Pull daher ablehnen, wie das folgende Beispiel zeigt:

```
$ hg pull ../test
pulling from ../test
searching for changes
abort: repository is unrelated
```

AddOns können nicht von den Original-Repositories aktualisiert werden, da diese rein technisch nichts mit den Repositories gemein haben, die in das Projekt gemergt wurden. Hier müssen wieder die konvertierten Versionen zum Einsatz kommen:

```
$ hg pull ../test-converted
pulling from ../test-converted
searching for changes
no changes found
```

Das Konvertieren eines Repositories ist eine stabile Abbildung (d. h. es kann mehrmals durchgeführt werden und führt immer zu dem gleichen Ergebnis), wenn man den automatisch ergänzten „update tags“-Commit nicht betrachtet (da dieser jeweils am Ende des Prozesses angefügt wird, ändert sich seine Prüfsumme aufgrund des geänderten Timestamps). Es ist somit problemlos möglich, das für ein Update notwendige, konvertierte Repository zu einem AddOn zu erzeugen. Im Folgenden werden diese speziellen Repositories „Update-Repositories“ genannt.

In einem zentralen Modell kann dies automatisch auf dem Server geschehen. Dazu kann die Hook-Funktionalität von Mercurial genutzt werden, die es ermöglicht, beim Eintreten bestimmter Ereignisse benutzerdefinierte Programme auszuführen.[1] Die Mercurial-Dokumentation liefert eine Übersicht über alle verfügbaren Hooks.

Zum Erstellen der Update-Repositories sollte ein Hook auf das Ereignis `changegroup` registriert werden. Dieses wird pro Push auf ein Repository einmal ausgelöst. Da die Konvertierung wie bereits gezeigt mehr als einen einzelnen hg-Befehl umfasst (siehe Abschnitt 4.5.4 auf Seite 44), eignet sich ein kurzes PHP-Skript für diese Aufgabe. Es kann wie in Listing A.12 auf Seite 90 konfiguriert werden. Seine Aufgabe ist es, zu prüfen, ob der Commit auf ein passendes Repository stattfand (da der Hook für jedes Repository ausgeführt wird, aber nur bei AddOns für diesen Anwendungszweck von Bedeutung ist) und dann die eigentliche Konvertierung durchzuführen.

Die Erzeugung der Update-Repositories entspricht der in Abschnitt 4.5.3 auf Seite 44 gezeigten Vorgehensweise. Da in diesem Fall jedoch das zu erzeugende Repository bereits existieren kann, gibt es zwei Möglichkeiten, vorzugehen: Die Ad-hoc-Lösung beschreibt, wie die bestehenden Repositories entfernt werden. Der danach beschriebene, erweiterte Ansatz macht sich die Fähigkeit Mercurials, inkrementell zu arbeiten zunutze, um die Repositories einfach zu erweitern.

Ad-hoc-Lösung

Um eine neue Version des Update-Repositories zu erzeugen, sollte zuerst ein eventuell bestehendes Repository gelöscht werden. Daran anschließend kommt der schon

[1] http://www.selenic.com/mercurial/hgrc.5.html#hooks

erwähnte Algorithmus zur Erzeugung des Repositories und der Anpassung der Tags zum Einsatz. Listing A.13 auf Seite 90 zeigt eine beispielhafte Implementierung dieser Herangehensweise.

Problematisch an dieser Lösung ist jedoch, dass die Zeit für einen Push linear mit dem Alter des Repositories anwächst, da ältere Commits immer und immer wieder neu konvertiert werden müssen. Neben der benötigten Zeit wird dabei auch der Server immer stärker belastet. Damit ist klar, dass dieses Vorgehen keine dauerhafte Lösung darstellen kann.

Erweiterter Algorithmus

Um die angesprochenen Probleme zu lösen, kann die Fähigkeit der `convert`-Erweiterung, inkrementell zu arbeiten, ausgenutzt werden. Anstatt eine unterbrochene Konvertierung fortzuführen, wird die Funktion dazu genutzt, ein bestehendes Update-Repository um die neuen Commits zu erweitern. Auf diese Weise muss bei jedem Push nur die Arbeit verrichtet werden, die wirklich nötig ist.

Zu bedenken ist jedoch, dass dazu alle Commits im Update-Repository, die keine Entsprechung im Quell-Repository haben, entfernt werden müssen. Dazu zählen neben dem „update tags“-Commit auch alle weiteren, die durch eventuelle Operationen am Quell-Repository entfernt wurden (`hg strip`, vgl. Abschnitt 6.2.1 auf Seite 71). Ein Beispiel soll die Problematik verdeutlichen.

Wenn ein Benutzer einen Commit erstellt und zum Server pushed, wird durch den Hook das Update-Repository neu erzeugt. Später stellt sich heraus, dass in dem Commit Daten enthalten sind, die nicht hätten committet werden dürfen (zum Beispiel aus Lizenzgründen). Ein Administrator entfernt den Commit nun wieder aus dem Repository, indem er `hg strip` anwendet. Der Benutzer führt den gleichen Schritt in seinem Klon durch.
Im Anschluss wird wieder ein neuer Commit und Push durchgeführt. Die dabei angestoßene Konvertierung kann jedoch nicht ausgeführt werden, da im Update-Repository noch der fehlerhafte Commit enthalten ist und somit kein inkrementelles Update möglich ist.

Um den Vorgang robuster zu gestalten, sollte vor der Konvertierung nicht nur der letzte Commit, sondern jeder entfernt werden, dessen Ursprung nicht mehr im Quell-Repository vorhanden ist. Zu diesem Zweck kann die Revisionsgeschichte des Update-Repositories rückwärts durchgegangen werden. Die convert-Erweiterung legt in der Datei `.hg/shamap` im Repository eine Zuordnung zwischen Quell- und Ziel-Revisionen in Form einer einfachen Textdatei, die die Prüfsummen enthält, ab. Der Algorithmus lässt sich in PHP wie in Listing A.16 auf Seite 92 gezeigt implementieren. Im Anschluss kann die besprochene Konvertierung ohne weitere Anpassungen durchgeführt werden.

Updatevorgang

Zur Aktualisierung eines AddOns kann nun ein regulärer Pull verwendet werden, bei dem alle Änderungen importiert werden. Allerdings entsteht nach jedem Push zu einem AddOn mindestens ein neuer „update tags“-Commit, der wiederum bei

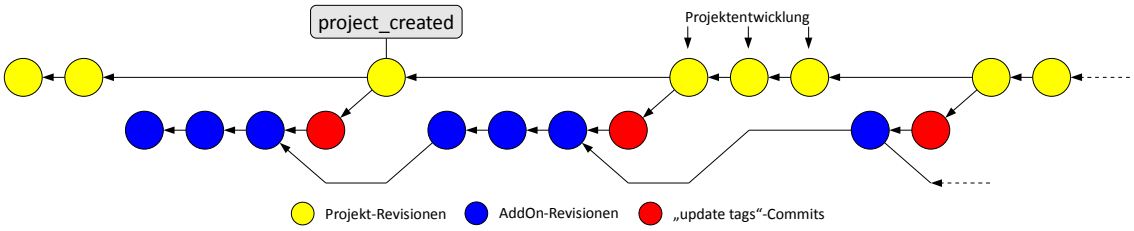

Abbildung 5.2: Revisionsgeschichte nach mehreren Updates

jedem Update eines Projekts importiert würde. Im Laufe der Zeit entstünde so eine Revisionsgeschichte, wie sie in Abbildung 5.2 dargestellt ist.

Da der „update tags“-Commit keine zusätzlichen Informationen enthält, wenn keine neuen Tags vergeben wurden, stellt er bei Updates nur unnötigen Ballast dar und sollte daher ausgelassen werden. Dies kann bewerkstelligt werden, indem die eingehenden Änderungen erst mit `hg incoming` untersucht und dann mit `hg pull -r X` nur bis zur vorletzten Revision (deren Prüfsumme mit `X` im vorangegangenen Befehl angegeben wurde) gepullt werden.

5.1.3 Zusammenfassung

Es wurde gezeigt, wie Aktualisierungen der Basis und aller AddOns vorgenommen werden können und welche Voraussetzungen dafür erfüllt werden müssen. Die Vorbereitungen können vollständig automatisiert werden, sodass Entwickler sich ausschließlich auf die eigentliche Projektentwicklung konzentrieren können und dennoch ständig aktuelle Repositories für Updates bereitstehen.

In einer idealen Entwicklung würden innerhalb eines Projekts keine Änderungen an der Basis oder den AddOns vorgenommen werden. Dies würde dann sogar vollständig automatisierbare (weil garantiert konfliktfreie) Updates ermöglichen.
Im realen Projektalltag lassen sich kleine Korrekturen, Anpassungen oder Erweiterungen jedoch nicht immer vermeiden. Hier stellt sich nun die Frage, wie diese möglichst effizient in die einzelnen Quell-Repositories zurückgespielt werden können und wie man erkennt, welche Commits davon betroffen sind. Der folgende Abschnitt widmet sich dieser Fragestellung.

5.2 Änderungen zurückspielen

Der deutlichste Nachteil des bisher gezeigten Verfahrens ist, dass die Repositories der einzelnen Komponenten verschwinden und so Änderungen in Projekten nicht ohne Weiteres in die jeweiligen Quellen zurückfließen können. So ist es nicht möglich, von der Basis (SallyCMS) oder einem AddOn aus einen Pull auf ein Projekt durchzuführen: Bei der Basis würde auf diese Weise die gesamte Projektgeschichte inklusive aller AddOns importiert werden, bei den AddOns würde ein Pull abgelehnt werden, da durch die Konvertierung keine Verbindung zwischen Projekt und AddOn besteht.
Im Folgenden sollen daher Wege gezeigt werden, wie betroffene Änderungen erkannt und möglichst einfach verarbeitet werden können.

5.2.1 Änderungen erkennen

Um Änderungen automatisch zu finden, muss klar sein, welche Commits damit genau gemeint sind. Es lohnt sich, zwischen Änderungen an der Basis und an AddOns zu unterscheiden, wobei hier im Speziellen auf die Struktur von SallyCMS-Projekten eingegangen werden soll (siehe dazu auch Abschnitt 4.1.1 auf Seite 32).

- Basis-Änderungen umfassen alle Dateien, die nicht `develop`, `assets` oder einem AddOn-Verzeichnis liegen.
- AddOn-Änderungen umfassen hingegen die Dateien, die in den jeweiligen Verzeichnissen der AddOns liegen.

Mercurial bietet von Haus aus bereits genug Möglichkeiten, derartige Änderungen zu finden. So kann der `log`-Befehl auf Verzeichnisse und Teilbäume der Revisionsgeschichte beschränkt werden (vgl. [Mac11a], Abschnitt „log“).
Das Vorgehen soll nun an einem Beispiel verdeutlicht werden. Dazu sollen aus dem Repository aus Abbildung 5.3 alle interessanten Revisionen ermittelt werden. Das Projekt besteht aus der Basis sowie zwei AddOns, von denen das zweite bereits einmal aktualisiert wurde. Die Commits $\{G, H, J, L\}$ stellen die Projektentwicklung und damit diejenigen Commits dar, in denen im Idealfall keine Änderungen vorhanden sein sollten.

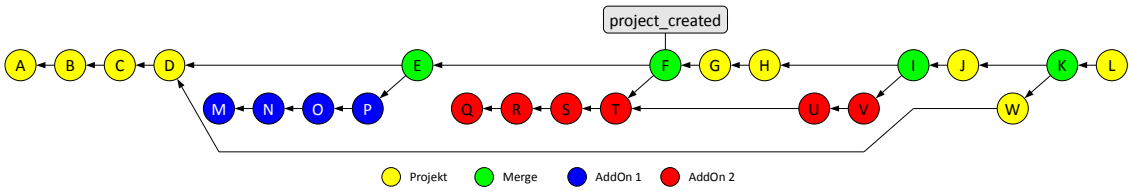

Abbildung 5.3: Revisionsgraph eines Beispielprojekts

Die Wirkung einzelner Anfragen soll nun demonstriert werden.

`log --rev descendants(0)` (nur Basissystem)
$= \{A, B, C, D, E, F, G, H, I, J, K, L, W\}$

`log --rev "descendants(project_created)"` (Änderungen ab `project_created`)
$= \{F, G, H, I, J, K, L\}$

`log --rev "children(descendants(project_created))"` (Kinder der obigen Query)
$= \{G, H, I, J, K, L\}$

`log -m` (nur Merges)
$= \{E, F, I, K\}$

`log -M` (keine Merges)
$= \{A, B, ..., V, W\} \setminus \{E, F, I, K\}$

`log -M --rev "children(descendants(project_created))"`
$= \{G, H, J, L\}$

Die letzte Anfrage gibt die gewünschten Revisionen zurück. Untersucht man diese Revisionen nach den entsprechenden Pfaden, kann man leicht ermittelt, ob Änderungen stattfanden, die zurückfließen müssen.

Die finale Anfrage, um Änderungen an der Basis zu ermitteln, lautet demnach wie folgt:

```
$ hg log -M --rev "children(descendants(project_created))" ↩
    -I "index.php" -I "sally/index.php" -I "sally/include/lib/**" ...
```

Hierbei ist zu beachten, dass aufgrund der Struktur eines Projekts eine längere Liste aller interessanten Stellen aufgezählt werden muss, von der jede Datei und jedes Verzeichnis über den Parameter `-I` angegeben werden muss.

- `index.php`
- `sally/index.php`
- `sally/include/lib/**`
- `sally/include/views/**`
- `sally/media/**`
- `...`

Für AddOns gestaltet sich die Anfrage einfacher, da jedes AddOn nur in genau einem Verzeichnis liegt.

```
$ hg log -M --rev "children(descendants(project_created))" ↩
    -I "sally/include/addons/**"
```

In den obigen Beispielen wurde davon ausgegangen, dass das schon angesprochene Tag `project_created` verwendet wurde, um im Repository den Zeitpunkt zu markieren, ab dem die projektspezifische Entwicklung begann. Um nach einem Zurückspielen der Änderungen diese nicht mehr zu selektieren, bietet es sich hier an, ein weiteres Tag namens `merged_back` zu verwenden, das immer auf die letzte Revision zeigt, bis zu der Änderungen bereits zurückgeflossen sind. So muss nur dieses Tag umgesetzt werden, um die Liste stets aktuell zu halten.
Um das Tag neu zu setzen, muss der Parameter `-f` (für *force*) angegeben werden:

```
$ hg tag merged_back
abort: tag 'merged_back' already exists (use -f to force)

$ hg tag -f merged_back
```

In der automatisierten Auswertung der Commits sollte nun der beste Startpunkt für die Revisionen wie in Listing A.14 ermittelt werden. Dabei werden nacheinander die Tags `merged_back`, `project_created` und schlussendlich die Nullrevision durchprobiert, bis eine ideale Revision gefunden wird.

Die so gefundenen Commits können, um sie leichter programmseitig auszuwerten, über ein sog. Template umformatiert werden. Templates legen in Mercurial das Format fest, mit dem ein Befehl seine Ausgabe erzeugt. Um eine zeilenweise Ausgabe zu erhalten, könnte der Befehl wie folgt aussehen:

```
$ hg log --template="{rev} ({date|rfc822date}): {desc|firstline} [{author|person}]\n"
```

Dies würde eine Ausgabe wie diese erzeugen:

```
2870 (Thu, 31 Mar 2011 12:14:37 +0200): fixed retrieval of section metainfo for ↩
    iewarning [Christoph]
2869 (Thu, 31 Mar 2011 02:06:45 +0200): added error page [Christoph]
2868 (Thu, 31 Mar 2011 02:06:37 +0200): fixed inclusion of error page [Christoph]
2867 (Thu, 31 Mar 2011 01:33:34 +0200): Merge [Christoph]
2866 (Wed, 23 Mar 2011 23:52:09 +0000): update tags [convert-repo]
```

Über `hg help templating` (vgl. [Mac11a]) können weitere Platzhalter und Filter eingesehen werden.

Nachdem nun diejenigen Commits identifiziert werden können, die Kandidaten für ein Zurückmergen sind, stellt sich noch die Frage, wie genau das Mergen durchgeführt werden kann.

5.2.2 Änderungen zurückmergen

Es gibt drei wichtige Methoden, Änderungen zurückfließen zu lassen: Patches, Pulls und manuelles Übertragen. Im Folgenden sollen die Vor- und Nachteile der einzelnen Verfahren erläutert werden.

Patches

Getätigte Änderungen an der Basis lassen sich am einfachsten in Form eines Patches zurückspielen. Dazu kann in TortoiseHg die betreffende Revision ausgewählt und über *Export / Export Patch...* im Kontextmenü ein Patch erstellt werden. Dabei bleiben Zeitpunkt, Autor und Commit-Nachricht erhalten.

Zu beachten ist hierbei jedoch, dass der Patch alle vom Commit betroffenen Dateien enthält. Wenn kein manuelles Bereinigen gewünscht ist, sollte bereits beim Committen darauf geachtet werden, dass nur diejenigen Dateien verarbeitet werden, die später übertragen werden sollen. Im Zweifelsfall sollte man Änderungen, die später zurückgemergt werden müssen, immer als einzelnen Commit anlegen. Ansonsten kann man mit einem einfachen Texteditor auch ungewünschte Diffs aus dem Patch entfernen.

Patches können auch über die Kommandozeile erstellt werden:

```
$ hg export --git -r REV > bugfix.patch
```

`REV` gibt hierbei die zu exportierende Revision an, während der Schalter `--git` eine erweiterte Ausgabe erzeugt, die auch rename/copy-Befehle sowie Änderungen an Binärdateien korrekt repräsentiert (im Standard Patchformat ist dies nicht möglich). Patches im Git-Format können auch dauerhaft in der Mercurial-Konfiguration aktiviert werden, wie in Listing A.15 auf Seite 91 gezeigt wird.

Änderungen an AddOns sind komplexer, da hier die Pfadänderung beachtet werden muss, die beim Konvertieren eingeführt wurde. So würde ein exportierter Patch eine Änderung an der Datei `sally/include/addons/test/lib/Class.php` enthalten, die in dem AddOn-Repository jedoch nicht vorhanden ist. Dort ist sie unter ihrem Originalpfad, `lib/Class.php` zu finden. Ein erzeugter Diff in einem Patch könnte (auf den Header gekürzt) wie folgt aussehen:

```
diff --git a/sally/include/addons/test/lib/Class.php ↩
    b/sally/include/addons/test/lib/Class.php
--- a/sally/include/addons/test/lib/Class.php
+++ b/sally/include/addons/test/lib/Class.php
@@ -364,8 +364,8 @@
...
```

In den meisten Fällen reicht es, die exportierten Patches nachträglich mit einem Editor zu bearbeiten und per Suchen & Ersetzen den Pfad zum AddOn zu entfernen, sodass ein Diff wie folgt entsteht:

```
diff --git a/lib/Class.php b/lib/Class.php
--- a/lib/Class.php
+++ b/lib/Class.php
@@ -364,8 +364,8 @@
...
```

Die Methode Patches zu verwenden ist für die meisten Änderungen sehr brauchbar und gleichzeitig einfach zu verwenden. Commits können selektiv zurückgespielt werden, wobei im Einzelfall auch Änderungen an den Patches denkbar sind (wenn zum Beispiel eine Änderung nicht vollständig übertragen werden soll).
Nachteilig ist jedoch, dass dieses Verfahren einen spürbaren manuellen Aufwand mit sich bringt und fehlerhaft konfigurierte Editoren beim Bearbeiten eines Patches diesen aus Versehen so modifizieren können (Encoding, Zeilenende, leere Zeilen), dass ein konfliktfreies Anwenden erschwert wird.

Pull-Operationen

Der ideale Weg, das Zurückmergen zu automatisieren, wären offensichtlich Pulls, bei denen Mercurial alle anstehenden Arbeiten selbst übernimmt. Das zur Projekterzeugung gewählte Verfahren macht es leider nur in sehr begrenztem Umfang möglich, Pulls zu benutzen: So sind bei einem Zurückmergen in die Basis die Commits der AddOns überschüssig und dürfen nicht übertragen werden, während beim Zurückmergen der AddOns die Pfade durch das Konvertieren nicht stimmen und erst korrigiert werden müssten.

Bei der Basis müssten mindestens die Merges mit den AddOns sowie alle projektspezifischen Commits ausgelassen werden. Dies kann über eine Konvertierung erreicht werden, wobei „konvertieren“ in diesem Fall eher als „filtern“ zu verstehen ist, da weder das Versionskontrollsystem gewechselt noch die Pfade geändert werden.
Eine Filemap wie die folgende würde dies bewerkstelligen:

```
exclude sally/include/addons
exclude develop
exclude assets
(hier weitere projektspezifische Verzeichnisse notieren)
```

Über den schon aus Abschnitt 4.2.1 auf Seite 37 bekannten convert-Befehl kann dann ein bereinigtes Repository erstellt werden:

```
$ hg convert --filemap=filemap.txt project project-cleaned
```

Von dem Basis-Repository aus können nun ein Pull und ein Merge ausgeführt werden, um die Änderungen zu übertragen.

```
$ hg pull ../project-cleaned
$ hg merge
$ hg commit -m "imported changes from project X"
```

Im Anschluss kann das bereinigte Repository (`project-cleaned`) entfernt werden. Die dabei übertragenen Commits werden, wenn das Projekt aktualisiert werden soll, als neue Commits importiert werden.

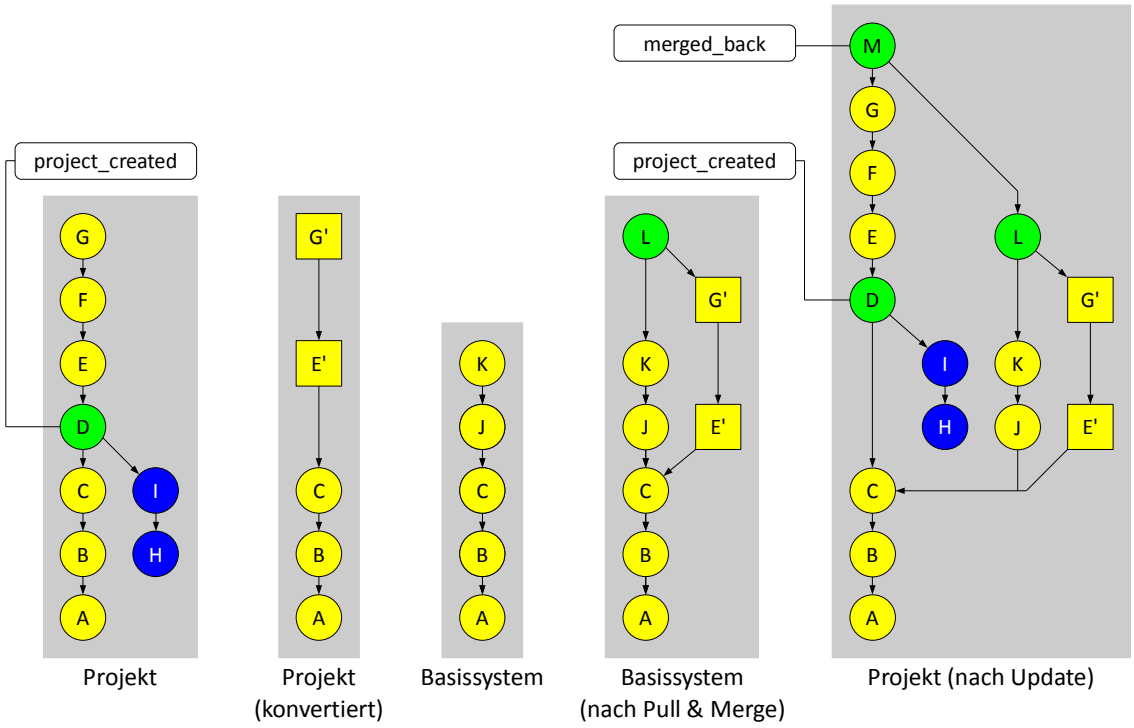

Abbildung 5.4: Revisionsgrpahen für ein Projekt vor und nach einem Update

Abbildung 5.4 veranschaulicht die entstehende Projektstruktur. Die Grafik ist von links nach rechts zu lesen. Der erste Graph stellt ein Projekt dar, in dem in den Commits $\{E, F, G\}$ die Projektentwicklung stattfand, wobei in E und G Änderungen an der Basis vorgenommen wurden. Der zweite Graph stellt das Repository dar, nachdem es konvertiert und die nicht relevanten Änderungen (F) ausgelassen wurden. Da sich bei E der Parent-Commit geändert hat, ändert sich eine Prüfsumme und ein neuer Commit entsteht. Dies betrifft sukzessiv auch G.
Der dritte Graph zeigt das Basis-Repository, in dem seit der Projekterzeugung bereits neue Commits (J und K) stattfanden. Daneben wird gezeigt, wie das Repository aussieht, nachdem Pull & Merge durchgeführt wurde. Die beiden zurückzumergenden Commits E' und G' wurden importiert und über den neuen Mergecommit L in den Graph integriert. Damit ist das eigentliche Synchronisieren abgeschlossen. Der letzte Graph zeigt, was geschieht, wenn man vom Projekt aus ein Basis-Update durchführen würde, mit dem Ziel, die neuen Commits J und K in das Projekt einzuspielen. Dabei ist deutlich zu sehen, dass neben den eigentlich gewollten Commits

auch noch die zurückgemergten (E' und G') sowie ein Merge (L) in das Projekt übernommen werden.

Aus Sicht des Projekts sind $\{E', G', L\}$ überflüssiger Ballast, da sie keine neuen Inhalte mitbringen. Die Commits müssen allerdings mit importiert werden, damit zu einem späteren Zeitpunkt die weitere Entwicklung des Basis-Systems (Commits, die auf L folgen werden) übertragen werden können (um die Children von L in ein Projekt zu übernehmen, müssen alle Vorgänger übernommen werden, womit E' und G' ins Projekt gelangen). Es lohnt sich daher nicht, K beispielsweise direkt in das Projekt zu mergen, um den Ballast zu vermeiden, da er später ins Projekt gelangen muss.

Für AddOns gilt die gleiche Vorgehensweise mit den gleichen Konsequenzen für Quell- und Zielrepository. Aus Gründen der Übersichtlichkeit ist dieses Vorgehen daher nicht zu empfehlen. Der Overhead durch die Konvertierungen und entstehenden Merges bietet keinen Mehrwert an Informationen und ist im Endeffekt auch nicht effizienter als das Extrahieren und Anwenden von Patches.

Trotzdem kann die Konvertierung helfen, das manuelle Nachbearbeiten von Patches zu vermeiden. Zu diesem Zweck können aus den konvertierten Projekt-Repositories (in Abbildung 5.4 auf der vorherigen Seite der zweite Graph) Patches erstellt werden, die dann auf die jeweiligen Quell-Repositories angewandt werden. Beim Anwenden von Patches werden die neuen Commits an die bestehende History des Repositories angefügt, sodass keine Merges notwendig sind. Dieses Vorgehen ist (bis auf die Gefahr von inhaltlichen Konflikten) automatisierbar.
Der große Vorteil dieser Herangehensweise ist, dass Commits nicht mehr strikt nach Komponente getrennt erfolgen müssen, da sie durch die Konvertierung (die auch hier wieder mehr einem Filter entspricht) aufgetrennt werden. So würde für jedes AddOn und die Basis ein einzelner Konvertierprozess durchgeführt werden. Gleichzeitig können bei AddOns die Pfade korrigiert werden, indem der zur Erstellung des Projekts `rename`-Befehl verwendet wird. Damit wird die Umbenennung aus Abschnitt 4.5.3 auf Seite 44 rückgängig gemacht.

```
exclude .
include sally/include/addons/test
rename sally/include/addons/test .
```

Je nach Menge der betroffenen Commits (siehe vorangegangener Abschnitt) sollte spontan entschieden werden, welche Technik zum Einsatz kommt und ob es sich lohnt, die Konvertierung zu verwenden, nur um ein eventuelles Suchen und Ersetzen in Patches zu vermeiden.

Manuelle Übertragung

Die manuelle Übertragung ist die Notlösung, die nur verwendet werden muss, wenn sich effektiv keine Patches extrahieren lassen oder der Aufwand, diese nachträglich zu bereinigen, größer wäre, als die Änderung per Copy & Paste zu übertragen. Dies ist nur selten der Fall und betrifft Commits, in denen selbst nach dem Filtern noch projektspezifische und -neutrale Änderungen gemischt sind. Auf diese Möglichkeit soll daher nur der Vollständigkeit wegen eingegangen werden.

Hierbei ist insbesondere zu beachten, dass beim Anlegen der neuen Commits in den Basis-Repositories der Autor und die originale Commitzeit verloren gehen. Über den Parameter `-u` (oder über die erweitere Ansicht im Commit-View von TortoiseHg) kann der Autor zwar entsprechend angepasst werden, die vom Unternehmen gemachte Erfahrung zeigt jedoch, dass dies eher selten der Fall ist.
Je nachdem, wie intensiv die Revisionsgeschichte genutzt wird, können so Verwechslungen bei der Frage der Verantwortung für eine Änderung entstehen. Die gleiche Gefahr besteht allerdings auch für Patches, die nicht konfliktfrei anzuwenden waren und durch einen manuellen Commit abgeschlossen wurden.

5.2.3 Ergebnis

Die einfachste und schnellste Art, Änderungen zu übertragen, stellen Patches dar. Ihr einfaches Format macht sich leicht in der Handhabung und dennoch ausdrucksstark genug, um alle Wünsche abzudecken. Über einfache Scripts können sie automatisiert erzeugt werden.

Die Geschwindigkeit und Einfachheit stellen kritische Eigenschaften dar, da sie direkt beeinflussen, wie häufig Mitarbeiter die Arbeit des Zurückmergens ausführen und wie viel Zeit dabei aufgewandt werden muss.

5.3 AddOns veröffentlichen

Die im Unternehmen entwickelten AddOns stehen vollständig als Quellcode zur Verfügung und werden unter MIT-Lizenz[2] veröffentlicht. Um Dritten den Zugriff auf aktuelle Entwicklungen zu ermöglichen, sollen die AddOn-Repositories automatisch und regelmäßig im Internet zur Verfügung gestellt werden. Der Einsatz von Mercurial vereinfacht diesen Prozess im Vergleich zu Subversion deutlich, da es von Anfang an als dezentrales System entwickelt wurde.

Im Folgenden sollen mehrere Möglichkeiten beschrieben werden, Repositories zu veröffentlichen. Die Verfahren unterscheiden sich nur in der Art und Weise, wie und wann der eigentliche Push (also die Übertragung der Repositories zu einer externen Plattform) angestoßen wird.

Plattformen

Die Repositories können bei den Verfahren entweder von einem eigenen Server aus bereitgestellt werden, oder von auf das Hosting spezialisierte Drittanbietern. Diese stellen neben dem eigentlichen Repository oft noch weitere Funktionen wie Wikis, Bugtracker und andere Kollaborationswerkzeuge zur Verfügung.

Für Mercurial gibt es eine Reihe von Anbietern, darunter[3]

- Bitbucket (https://bitbucket.org/)
- Google Code (http://code.google.com/hosting/)
- CodePlex (http://www.codeplex.com/)

Soll ein eigener Server verwendet werden, so kann das in Abschnitt 4.3 auf Seite 38 vorgestellte Setup mit leichten Anpassungen bereits dazu verwendet werden.

[2] http://www.opensource.org/licenses/MIT
[3] http://mercurial.selenic.com/wiki/MercurialHosting

5.3.1 Repositories veröffentlichen

Um externe Entwickler zu ermutigen, Änderungen beizusteuern, sollten die veröffentlichten Repositories stets aktuell sein. Andernfalls würden Entwickler Gefahr laufen, Code zu entwickeln, der nicht konfliktfrei gemergt werden kann oder schlichtweg obsolet ist, wenn es zu einem Merge kommt.

Ständig aktuelle Repositories können erreicht werden, indem man selbst zum Ziel-Server pusht, einen Hook einrichtet oder einen Cronjob verwendet. Beim manuellen Push muss nur die Ziel-URL angegeben werden:

```
$ hg push https://bitbucket.org/SallyCMS/trunk/
```

Hooks

Ein Hook sollte ebenso wie beim Erzeugen des Update-Repositories auf das `changegroup`-Ereignis hören. Listing A.17 auf Seite 93 zeigt eine beispielhafte Konfiguration, bei der ein Repository sowohl zu Bitbucket als auch zu Google Code übertragen wird. Die dafür notwendigen Zugangsdaten können in der systemweiten Konfiguration abgelegt werden.

Nachteilig dabei ist, dass die einzelnes Pushes der Mitarbeiter durch diese Verbindung mit externen Diensten unter Umständen wesentlich verzögert werden können. Ist ein Anbieter nicht oder nur sehr langsam zu erreichen, muss der Mitarbeiter lange auf den Abschluss der Operation warten. Um diese Abhängigkeit zu entfernen, können Cronjobs eingesetzt werden.

Cronjobs

Cronjobs ermöglichen es, in regelmäßigen Intervallen Programme ausführen zu lassen. Dies kann dazu genutzt werden, um Repositories beispielsweise jede Nacht um 1 Uhr automatisch zu veröffentlichen. Die Konfiguration der Zugangsdaten kann dabei wie beim Einsatz von Hooks aus dem vorherigen Abschnitt erfolgen. In der `crontab`-Datei müssen dann nur noch die entsprechenden Anweisungen zum Push notiert werden:

```
0 1 * * * cd /path/to/addons/feeds; hg push bb
5 1 * * * cd /path/to/addons/usermngt; hg push bb
10 1 * * * cd /path/to/addons/shop; hg push bb
15 1 * * * cd /path/to/addons/errorhandler; hg push bb
```

Listing 5.1: Konfiguration von regelmäßigen Pushs als Cronjobs

Obige Konfiguration würde täglich zwischen 01:00 und 01:15 Uhr die vier notierten AddOns zu Bitbucket pushen. Je nach Menge der Repositories kann hier auch ein Script zum Einsatz kommen, das nur einen Eintrag im crontab erfordert und beim Aufruf alle Repositories in einem Verzeichnis pusht.

5.3.2 Änderungen einholen

Die zur Verfügung gestellten Repositories können nun von Dritten geklont werden, um Änderungen vorzunehmen. Es ist dann die Aufgabe eines Mitarbeiters, die gemachten Änderungen zu sichten und in die Original-Repositories im Unternehmen einzuspielen. Das Importieren fremder Änderungen erfolgt dabei über einen regulären Pull, bei dem als Quelle die URL des Klons des Drittentwicklers angegeben wird. Alternativ kann dieser auch Patches, zum Beispiel per E-Mail, einsenden.

Verschiedene Hosting-Plattformen bieten rund um diesen Arbeitsablauf mehr oder weniger ausgereifte Werkzeuge an, um Änderungen vorher zu prüfen oder überhaupt erst über deren Existenz benachrichtigt zu werden. Auf Bitbucket kann ein Mitglied dazu einem anderen einen sog. „Pull-Request“ senden, indem er darum bittet, die eigenen Änderungen zu übernehmen. Im Zweifelsfall reicht zur Benachrichtigung auch eine einfache E-Mail aus.

5.3.3 Zusammenfassung

Um Repositories im Internet bereitzustellen gibt es eine Reihe von Möglichkeiten. Die Einrichtung automatisierter Abläufe ist einfach und kann pro Repository erfolgen. Dritten wird so die Möglichkeit gegeben, ohne spezielle Zugriffsrechte die Repositories zu klonen und mit den Inhalten zu arbeiten (und dabei sämtliche Funktionen von Mercurial zu nutzen).

Die von Dritten getätigten Änderungen können über Pull-Operationen in die eigenen Repositories übernommen oder via Patches eingespielt werden. Beide Verfahren werden von Mercurial direkt unterstützt und können teilweise (zum Beispiel bei Bitbucket) sogar im Browser erledigt werden (wobei Bitbucket den Pull und den Merge übernimmt). Mercurial eignet sich daher hervorragend, um Open Source-Projekte zu entwickeln und Dritte in die eigene Entwicklung einzubinden.

6. Erfahrungen und Ausblick

Nachdem in den vorangegangenen Kapiteln die Gründe für einen Wechsel von Subversion nach Mercurial und die Durchführung einer konkreten Migration beschrieben wurden, sollen die Erfahrungen in diesem Kapitel zusammengefasst werden.

Der folgende Abschnitt quantifiziert die bisher getroffenen Aussagen über die Verbesserungen am Commit-Verhalten der Entwickler. Dabei wird insbesondere auf die Häufigkeit und Größe der Commits eingegangen. Im Anschluss an die statische Auswertung wird ein Ausblick auf weitere Möglichkeiten, Mercurial einzusetzen, gegeben. Dazu werden Erweiterungen und aktuelle Entwicklungen besprochen. Zum Schluss werden die gesammelten Erfahrungen noch einmal zusammengefasst.

6.1 Statistik

Der Einsatz von TortoiseHg hat sich deutlich positiv auf die Entwicklung von Projekten ausgewirkt. Commits wurden häufiger getätigt, enthielten weniger querschneidende Belange (d. h. Änderungen, die verschiedene Funktionen oder Arbeitspakete betreffen; vgl. [CK10]) und führten zu einer besseren Nutzung der zur Verfügung stehenden Tools.

Die folgenden Daten in Tabelle 6.1 auf der nächsten Seite wurden aus einer zufälligen Menge von internen Projekten erhoben und sollen zeigen, wie sich das Commit-Verhalten durch den Wechsel von Subversion auf Mercurial verändert hat. Gemessen wurde dabei nur die projektspezifische Entwicklung, also diejenigen Commits, die im Verlauf eines Projekts entstanden (sodass AddOn-Commits sowie die Commits der Basis nicht immer wieder mitgezählt wurden).

Die Tabelle zeigt, dass im Schnitt doppelt so viele Commits pro Tag in den Projekten durchgeführt wurden. Insgesamt wurden mehr als doppelt so viele Commits (wenn normalisiert auf die gleiche Anzahl von Projekten) getätigt. Da sich der Umfang der Projekte nicht unterscheidet, kann darauf geschlossen werden, dass die Commits im Schnitt weniger Änderungen enthalten.

Kriterium	Subversion	Mercurial
Projekte	14	10
Commits	850	1601
Commits pro Tag	4,35	8,18
leere Commit-Nachricht[1]	676 (79,5 %)	5 (0,3 %)
quasi leere Commit-Nachricht[2]	789 (92,8 %)	5 (0,3 %)
geänderte Dateien[3]	5,71	4,15

Tabelle 6.1: Auswertung des Commit-Verhaltens vor und nach der Migration

Abgesehen von der Menge an Commits hat sich auch die Beschreibung verbessert: Die Anzahl von Commits ohne Commit-Nachricht([1]) hat sich drastisch reduziert (von 79,5 % auf 0,3 %). Dies ist dem Umstand geschuldet, dass TortoiseHg standardmäßig keine leeren Nachrichten zulässt. Außerdem fällt es bei kleineren Änderungen naturgemäß einfacher, diese kurz beim Commit zusammenzufassen. Betrachtet man diejenigen Commits, deren Nachricht automatisch von Eclipse generiert wurde (z. B. „ASSIGNED: TASK 123 Feature X“), ebenfalls als leer (weil wenig aussagekräftig), wird deutlich, wie wenig Commits unter Verwendung von Subversion und Eclipse nur kommentiert wurden (7,2 %).

Der letzte Teil der Tabelle zeigt, dass durchschnittlich weniger Dateien pro Commit geändert wurden. Dies lässt auf kleinere Iterationen während der Entwicklung schließen, bei der in kürzeren Abständen die gemachten Änderungen committet wurden.

Diese Ergebnisse lassen sich natürlich nicht ohne Weiteres verallgemeinern. Schon das Testprojekt aus Abschnitt 3.2.4 auf Seite 19 (FeatureIDE mit Subversion) weist völlig andere Eigenschaften auf: Hier wurden nur in 10,6 % der Fälle (92x) leere Commit-Nachrichten verwendet. Die bessere Beschreibung der Commits durch den Wechsel zu Mercurial ist daher keine generelle Eigenschaft des neuen Systems, ebenso wie fehlende Beschreibungen kein prinzipielles Problem von Subversion darstellen.

Dennoch sprechen die Zahlen eine eindeutige Botschaft. Insbesondere dass pro Tag fast doppelt so viele Commits wie noch unter Subversion getätigt wurden, zeigt, wie wertvoll diese sind und dass dieser Wert von den Entwicklern auch realisiert wurde.

6.2 Ausblick

Die bisher behandelten Themen sind natürlich nur ein kleiner Ausschnitt von dem, was mit Mercurial möglich ist. Viele weitere Funktionen können über Erweiterungen, *Extensions* genannt, nachgerüstet werden. Dazu zählen auch die beiden schon genannten Erweiterungen mq (zur Verwaltung von Commits als Serie von Patches) und convert (zur Konvertierung von Repositories).

Im Folgenden sollen weitere nützliche Erweiterungen besprochen und ihr Nutzen für Unternehmen diskutiert werden. Sie stellen für die Arbeitsabläufe kein Muss dar, sondern vereinfachen in den meisten Fällen nur die Arbeit mit Mercurial. Im

[3] arithmetisches Mittel

Anschluss wird auf aktuelle Entwicklungen des Mercurial-Projekts eingegangen, so zum Beispiel die stabilere Unterstützung für Subrepositories und Neuerungen in TortoiseHg.

6.2.1 Erweiterungen

Mercurial liefert eine Reihe von Erweiterungen bereits bei der Installation mit. Diese sind daher mit der aktuellen Version getestet und kompatibel. Zusätzlich gibt es eine umfangreiche Menge an von Dritten beigesteuerten Erweiterungen. Das Mercurial-Wiki[4] bietet eine umfassende Übersicht dazu.

Erweiterungen werden als Python-Module implementiert und können so auf das gesamte API (Programmierschnittstelle) von Mercurial zugreifen. Um sie zu aktivieren, müssen sie in der Konfigurationsdatei jeweils mit Namen und Pfad zur Python-Datei angegeben werden, wobei bei mitgelieferten Erweiterungen der Pfad weggelassen werden kann. Listing 6.1 zeigt, wie eine mitgelieferte und eine externe Erweiterung aktiviert werden.

```
[extensions]
mq =
myext = /full/path/to/myext.py
```

Listing 6.1: Konfiguration von Erweiterungen

Mercurial Queues (mitgeliefert)

mq (Mercurial Queues) wurde bereits in Abschnitt 4.5.4 auf Seite 44 angesprochen und dient dazu, Commits als Serie veränderlicher Patches zu verwalten. Es ermöglicht es, Patches nachträglich zu verändern, ihre Reihenfolge zu ändern oder mehrere Patches zu einem einzelnen zu vereinen.

mq eignet sich besonders, um noch nicht übertragene Commits nachträglich zu korrigieren, da es dem Entwickler ermöglicht, nicht nur den letzten Commit rückgängig zu machen. Die Erweiterung ist durchgehend in Mercurial integriert (viele Befehle können statt auf regulären Commits auch auf Patch-Serien operieren) und wird ebenfalls von TortoiseHg unterstützt.

Weiterhin stellt mq den Befehl `strip` bereit, mit dem es möglich ist, Commits endgültig aus dem Repository zu entfernen. Dabei wird, wenn Commit X entfernt werden soll, alle Children ebenso entfernt (da diese auf ihm basieren). Dies ist besonders praktisch, wenn mehr als ein Commit entfernt werden soll, da der in Mercurial integrierte Befehl `rollback` immer nur die jeweils letzte Transaktion (d. h. in diesem Fall den letzten Commit) rückgängig machen kann.

Fetch (mitgeliefert)

Um ein Repository zu aktualisieren, sind zumeist zwei bis drei Befehle nötig: `pull` überträgt die Änderungen des Quell-Repositories in das eigene Repository, lässt

[4] http://mercurial.selenic.com/wiki/UsingExtensions

dabei aber die Arbeitskopie unangetastet. `update` aktualisiert im Anschluss die Arbeitskopie. Sollten durch den Pull neue Heads entstanden sein, sind vorher ein `merge` sowie ein `commit` nötig, um die durch den Pull importierten Commits in die eigene Projektgeschichte zu mergen.

Die fetch-Erweiterung[5] automatisiert diesen Vorgang und führt die benötigten Schritte automatisch in der richtigen Reihenfolge aus. Da TortoiseHg `fetch` als mögliche Operation beim Abrufen von Änderungen unterstützt, wurde die Verwendung dieser einfachen Erweiterung in jedem Fall empfohlen den Mitarbeitern des Unternehmens empfohlen.

Bei der Arbeit an Repositories, die öffentlich gemacht werden sollen, ist zu beachten, dass `fetch` beim Durchführen eines Merges die vollständige URL des Quell-Repositories als Commit-Nachricht verwendet. Wird intern mit privaten Servern gearbeitet, ist die Veröffentlichung ihrer URLs ggf. nicht gewünscht und sollte vorher bereits unterbunden werden. Zu diesem Zweck kann der `fetch`-Befehl mit einem Alias überschrieben werden, der immer eine feste Commit-Nachricht vergibt. Listing 6.2 zeigt den dafür nötigen Eintrag in der Konfigurationsdatei.

```
[alias]
fetch = fetch -m "Automated Merge"
```

Listing 6.2: Alias für den fetch-Befehl

Rebase (mitgeliefert)

Beim Ausführen des `rebase`-Befehls wird ebenso wie bei der mq-Erweiterung die bestehende Projektgeschichte im Repository verändert. Dabei wird der Parent eines Commits geändert, sodass sich der Revisionsgraph ändert. Daraus folgt auch der Name der Erweiterung, da sich die „Basis“ ändert.[6]

Rebase lässt sich am einfachsten an einem konkreten Beispiel erklären. Abbildung 6.1 auf der nächsten Seite zeigt drei Revisionsgraphen, wobei die beiden rechten durch ein Rebase entstanden sind. Im oberen wurde die Revision F auf E umgesetzt, während beim unteren Graph D auf H umgesetzt wurde. Dadurch, dass sich die Prüfsummen der Parents jeweils geändert haben, ändern sich (gemäß der schon in Abschnitt 3.2.2 auf Seite 17 besprochenen Regeln) die Prüfsummen der umgesetzten Commits sowie aller Children (G', H' und E').

Deutlich zu sehen ist, dass ein Rebase die Zahl der Heads verringert. So wurde der Mergecommit, der Revision E und H verbinden müsste, um die Änderungen beider Branches in der Arbeitskopie widerzuspiegeln, eingespart. Aus diesem Grund wird Rebase meist verwendet, wenn ein Pull durchgeführt wird: So werden im Anschluss an den Pull die eigenen Änderungen an die gerade importierten angefügt, anstatt mit ihnen gemergt. Die resultierende Arbeitskopie ist dabei identisch und Konflikte, die bei einem Merge auftreten würden, müssen auch beim Rebase entsprechend behandelt werden.

[5] http://mercurial.selenic.com/wiki/FetchExtension
[6] http://mercurial.selenic.com/wiki/RebaseExtension

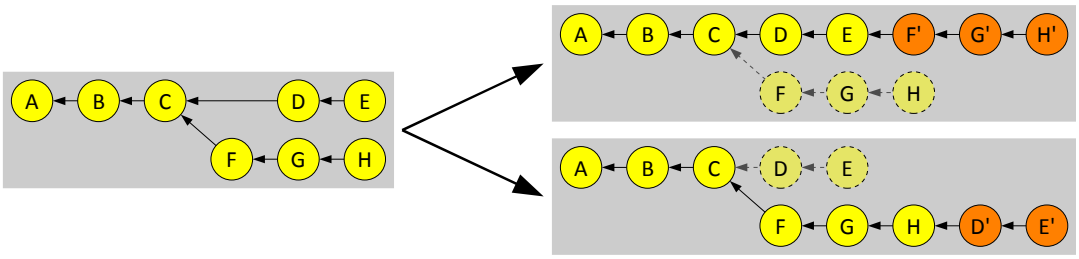

Abbildung 6.1: Revisionsgraph mit zwei Rebase-Varianten

In den meisten Fällen ist es während der Projektentwicklung nicht zwingend notwendig, Rebase zu verwenden. Es kann jedoch eingesetzt werden, um weniger Merges zu erzeugen und so eine lineare Projektgeschichte zu erhalten. Eine lineare Projektgeschichte ist wiederum beim Einsatz von mq von Vorteil, da mq keine Merges durch Patches ausdrücken kann.

RebaseIf (Drittanbieter)

RebaseIf stellt eine Erweiterung der eben besprochenen rebase-Erweiterung dar.[7] Sie ist in der Lage, vor dem eigentlichen Rebase zu prüfen, ob Konflikte auftreten würden (die sowohl beim Mergen als auch beim Rebase manuell gelöst werden müssten). Falls Konflikte vorliegen, wird kein Rebase, sondern ein Merge durchgeführt.

Der Grund für dieses Vorgehen ist, dass bei einem konfliktbehafteten Rebase die Änderungen, die zur Konfliktlösung durchgeführt werden, in den Commits verschwinden und nicht mehr explizit in einem neuen Commit festgehalten werden. Dies macht es unmöglich, die Konfliktbereinigung später gesondert zu betrachten.

Die Erweiterung bietet damit einen guten Mittelweg zwischen Rebases und Merges und sollte, wenn Rebase verwendet werden soll, immer verwendet werden.

Graphlog (mitgeliefert)

Die Graphlog-Erweiterung stellt einen neuen Befehl zur Verfügung, der eine grafische Repräsentation des Revisionsgraphen ausgibt.[8] Dabei wird ein Graph mittels ASCII-Zeichen erzeugt, um auch auf der Kommandozeile ein Gefühl für die Struktur des Repositories zu erhalten.

Der neue Befehl `glog` unterstützt einen Teil der Parameter des integrierten `log`-Befehls und wird den Graph links neben den jeweiligen Commits erzeugen. Ein Beispiel für die Darstellung eines Logs findet sich im Listing A.11 auf Seite 87.

Bei der Verwendung von grafischen Oberflächen wie TortoiseHg ist diese Erweiterung nicht nötig, bei der Arbeit mit Repositories auf der Kommandozeile (zum Beispiel über eine SSH-Verbindung mit einem externen Server) kann sie jedoch gute Dienste leisten.

[7] http://mercurial.selenic.com/wiki/RebaseIfExtension
[8] http://mercurial.selenic.com/wiki/GraphlogExtension

6.2.2 Aktuelle Entwicklungen

Sowohl Mercurial als auch TortoiseHg und auch Subversion werden aktiv weiterentwickelt. Im Folgenden sollen aktuelle Entwicklungen der Systeme angerissen werden, um die besprochenen Techniken in einen besseren Kontext zu rücken.

Mercurial

Die zum Zeitpunkt der Arbeit aktuelle Version von Mercurial ist 1.9, die am 1. Juli 2011 veröffentlicht wurde.[9] Sie bringt eine Reihe von Neuerungen mit, wie beispielsweise eine funktionale Sprache zur Auswahl von Dateien sowie einen Befehlsserver, der die Integration in Drittanwendungen erleichtern soll.

Über die funktionale Sprache können Dateien über Attribute wie Größe, Dateiname oder Status selektiert werden. So wird es möglich, über einen einzelnen Befehl beispielsweise alle geänderten Binärdateien im Projekt zurückzusetzen, indem `hg revert "set:modified() and binary()"` ausgeführt wird.

Der integrierte Befehlsserver ermöglicht die Kommunikation mit Mercurial, ohne dass für jeden Befehl ein neuer Prozess gestartet werden muss. Er erlaubt es somit, den in dieser Arbeit beschriebenen Prozess der Projekterzeugung über einen permanent laufenden Hintergrunddienst zu steuern und so den Overhead der einzelnen Prozesse zu reduzieren.

Weiterhin wird die Unterstützung für Subrepositories (Repositories innerhalb der Arbeitskopie anderer Repositories) als stabil betrachtet. Die Methode, die einzelnen Repositories bei der Projekterzeugung zu mergen wurde gewählt, da zum damaligen Zeitpunkt (Ende 2009) die Unterstützung noch in einem frühen, unausgereiften Status war. Mit der jetzt aktuellen Version sollte der Weg über Subrepositories neu evaluiert werden, bevor eine endgültige Entscheidung über den Aufbau von Projekten getroffen wird.

Mit Mercurial 1.9 erzeugte oder bearbeitete Repositories können nicht von früheren Versionen verarbeitet werden, wenn der Zugriff direkt über das Dateisystem erfolgt. Dies kommt daher, dass Version 1.9 ein neues Format zum Speichern der Revisionen verwendet. Es ist jedoch problemlos möglich, mit früheren Versionen die erzeugten Repositories über HTTP oder HTTPS zu klonen, da die Netzwerkschicht von Mercurial mit allen Versionen kompatibel ist.

TortoiseHg

TortoiseHg ist mit Version 2.0 vom 1. März 2011 von GTK+ auf Qt als verwendete GUI-Bibliothek gewechselt.[10] Mit dieser Änderung einher geht ein Umbau des Programms, bei dem nun eine Anwendung („Workbench“) mehrere Repositories verwalten soll. Gleichzeitig sollen sich in dieser Anwendung auch das Committen und Synchronisieren abspielen. Abbildung 6.2 auf der nächsten Seite zeigt ein Beispiel, in dem eine Reihe von Repositories in zwei Gruppen mit der Workbench verwaltet werden.

[9] http://mercurial.selenic.com/wiki/WhatsNew
[10] https://bitbucket.org/tortoisehg/thg/wiki/ReleaseNotes

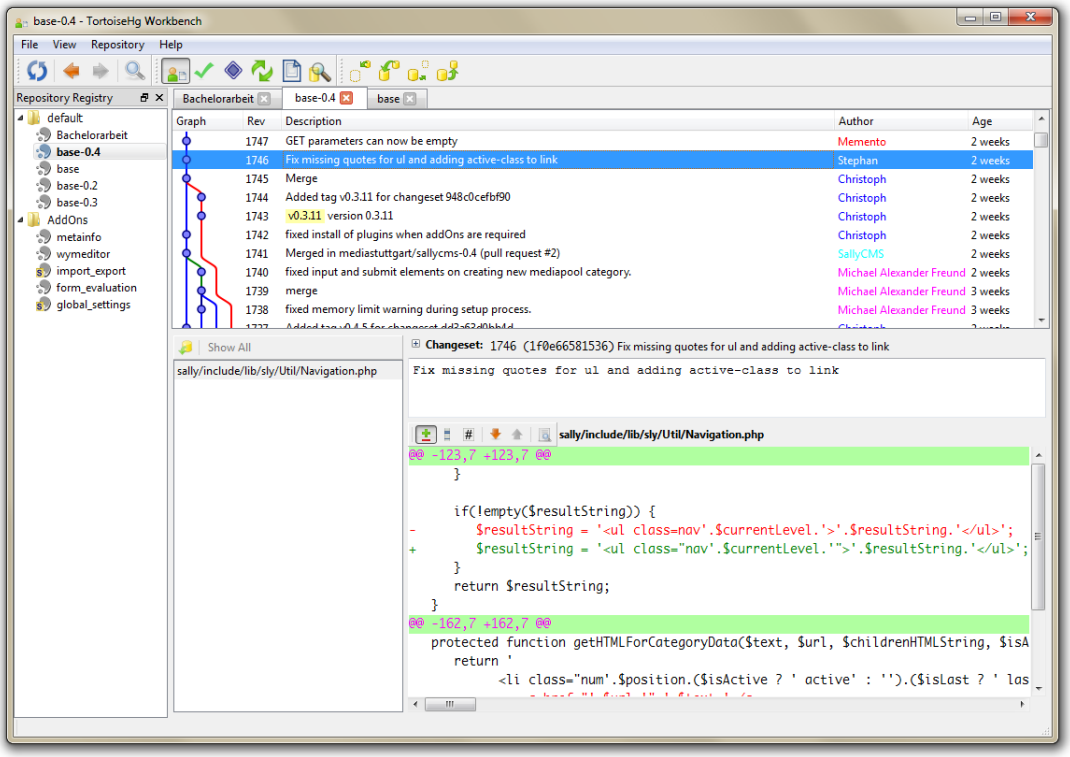

Abbildung 6.2: Workbench von TortoiseHg 2.1 mit mehreren Repositories

Die teilweise deutlich geänderte Menüstruktur und Aufbau der einzelnen Dialoge erfordert eine erneute Schulung der Mitarbeiter. So wurde beispielsweise der Merge-Dialog vollständig umgestaltet und der Commit-Dialog neu strukturiert. Aus diesem Grund wird die Einführung der Version 2 bisher noch zurückgehalten (auch da beispielsweise in Version 2.1 Probleme auftraten, die eine Arbeit mit TortoiseHg unmöglich machten).

Da der Branch für die Entwicklung der Version 1.x nicht mehr gepflegt wird, ist ein Umstieg jedoch zu einem späteren Zeitpunkt zwingend notwendig, damit Entwickler die Vorteile neuerer Mercurial-Versionen nutzen können. Die letzte 1.x Version von TortoiseHg, 1.10, wird mit Mercurial 1.8.1 ausgeliefert. Beide Versionen können nicht parallel installiert werden, sodass zwischen der alten GUI und der Unterstützung neuer Funktionen von Mercurial abgewogen werden muss.

Subversion

Subversion entwickelt sich ebenso stetig weiter; am 10. Juni 2011 erschien die erste Alpha-Version der kommenden Version 1.7.[11] Sie bringt eine Reihe von Neuerungen mit, die teilweise die in Abschnitt 3.2.1 auf Seite 16 getroffenen Aussagen revidieren. So verwendet Subversion 1.7 nicht mehr ein Metadaten-Verzeichnis pro Verzeichnis der Arbeitskopie, sondern ebenfalls (wie Mercurial oder Git) nur noch ein einzelnes Verzeichnis im Root der Arbeitskopie.

Die fundamentalen, vom zentralen Konzept geerbten Probleme, wie sie bereits in Abschnitt 3.1.1 auf Seite 12 angesprochen wurden, bleiben dabei natürlich weiterhin

[11] http://subversion.apache.org/docs/release-notes/1.7.html

bestehen. Die Veränderungen beim Speichern der Metadaten sowie beim Mergen stellen jedoch spürbare Verbesserungen von Subversion dar und machen es zu einem noch besseren zentralen Versionskontrollsystem.

7. Zusammenfassung

In den vorangegangenen Kapiteln wurde argumentiert, warum dezentrale Systeme eine bessere Wahl für die Entwicklung von Software sind als zentrale Systeme. Die erhöhte Flexibilität wirkte sich positiv auf die Qualität der entwickelten Projekte aus. Entwickler versionierten ihre Änderungen häufiger und legten mehr Sorgfalt bei der Dokumentation der Commits an den Tag, wodurch Änderungen an Projekten und AddOns leichter nachzuvollziehen wurden. Die Möglichkeiten eines Versionskontrollsystems wurden häufiger wahrgenommen und besser in den Arbeitsalltag integriert.

Die Verwendung von Mercurial förderte ebenfalls die Integration Dritter während der Entwicklung. Durch automatisiertes Pushen zu Bitbucket steht ständig aktueller Code für Fremdentwickler zur Verfügung, die ohne Beschränkungen die Repositories und dazugehörigen Werkzeuge von Mercurial verwenden können. Bereits nach kurzer Zeit wurden auf diese Weise erste beigesteuerte Änderungen generiert, die wie zu erwarten problemlos in die eigene Entwicklung übernommen werden konnten. So zeigt Abbildung 6.2 auf Seite 75 drei Commits (1738, 1739 und 1740), die von externen Entwicklern beigesteuert wurden.

Der Umstieg auf Mercurial erforderte eine gründliche Auseinandersetzung mit den verfügbaren Techniken. Insbesondere die Unteilbarkeit von Repositories und fehlende Unterstützung für Operationen auf einem zentralen Server mussten dabei berücksichtigt werden. Auch die Integration in bereits vorhandene Entwicklungsumgebungen spielt eine gewichtige Rolle, wenn nicht auf TortoiseHg oder eine andere GUI für Mercurial umgestiegen werden soll. Die vielfältigen Möglichkeiten, die Arbeitsabläufe eines Teams zu koordinieren (vgl. [Cha09] und [Ott09]), sind dabei ebenso zu bedenken: Ein bisher zentral ausgelegtes Modell muss nicht zwangsweise zentral bleiben, sondern kann beliebig komplex erweitert werden, um die Workflows abzubilden.

Betrachtet man alle Faktoren, die Voraussetzungen, Planungen und die notwendigen Implementierungen, hat sich die Migration durchgängig als positiv erwiesen. Die deutlichen Vorteile in der Entwicklung machen die minimalen Nachteile wett und bereiten das Unternehmen auf Wachstum und Expansion auf mehrere Standorte vor.

A. Anhang

Im Folgenden sind die einzelnen Listings gegeben, die im Verlauf der Arbeit referenziert wurden. Die jeweiligen Quellcodes stehen unter keiner speziellen Lizenz und können ohne Einschränkung weiterverwendet werden.

Konfiguration des Apache-Webservers

```
<VirtualHost *:10109>
  ServerName hg.webvariants.de

  DocumentRoot /srv/hg/
  SSLEngine on
  # (hier weitere Konfiguration der SSL-Unterstützung)

  <Location />
    AuthType Basic
    AuthName "Restricted"
    AuthUserFile /srv/hg/htpasswd
    AuthGroupFile /srv/hg/htgroups
    Require group wv

    ProxyPass http://localhost:8001/
  </Location>
</VirtualHost>
```

Listing A.1: Konfiguration von Apache

Obiger Code demonstriert, wie der Zugriff auf Mercurial Repositories über einen Apache-Webserver konfiguriert werden kann. Der Webserver übernimmt dabei die Authentifizierung der Benutzer.

Konfiguration von hgwebdir

```
[web]
allow_push = *
push_ssl = false

[paths]
/ = /srv/hg/**
```

Listing A.2: Konfiguration von hgwebdir

Listing A.2 zeigt die zu Listing A.1 auf der vorherigen Seite gehörende Konfiguration von hgwebdir, bei der Push-Operationen von jedem authentifizierten Benutzer erlaubt werden. In `paths` wird angegeben, dass rekursiv alle Repositories in `/srv/hg` zur Verfügung gestellt werden sollen.

PHP-Funktion zum Ausführen von Mercurial-Kommandos

```php
/**
 * führt einen hg-Befehl aus
 *
 * @param  string $command      der auszuführende Befehl ohne "hg "
 * @param  int    $returnValue  Variable, die den Exit-Code enthalten wird
 * @return mixed                false im Falle eines Fehlers, sonst die Ausgabe
 *                              als String
 */
function hg($command, &$returnValue = -1) {
  // Variable für die Ausgabe, pro Zeile ein Element im Array
  $output = array();

  // Befehl ausführen, dabei Kodierung angeben, um Problemen unter Windows
  // vorzubeugen; eventuelle Fehler werden nach stdout umgeleitet.
  exec('hg --encoding=ISO-8859-1 '.$command.' 2>&1', $output, $returnValue);

  // wenn fehlerhaft, false zurückgeben, sonst Ausgabe als langen UTF-8-String
  return $returnValue !== 0 ? false : utf8_encode(implode("\n", $output));
}
```

Listing A.3: PHP-Funktion zum Aufruf von Mercurial

Obiges Listing definiert eine PHP-Funktion, die einen Mercurial-Prozess startet und die generierte Ausgabe zurückgibt, wenn der Befehl erfolgreich ausgeführt werden konnte. Im Fehlerfall wird `false` zurückgegeben. Die Funktion stellt damit die Basis für alle weiteren Implementierungen dar.

PHP-Code zum Erzeugen eines Repositories

```php
// Daten importieren
$directory   = isset($_POST['directory']) ? $_POST['directory'] : '';
$repository  = isset($_POST['repo']) ? trim($_POST['repo']) : '';
$description = isset($_POST['description']) ? $_POST['description'] : '';
$contact     = isset($_POST['contact']) ? $_POST['contact'] : '';

// Daten validieren (ungültige Zeichen entfernen)
$repository = preg_replace('#[^a-z0-9_-]#i', '', $repository);

// (weitere Prüfung auf leere/ungültige Werte)

// Repository anlegen
$repo   = '/srv/hg/'.$directory.'/'.$repository;
$output = hg('init "'.$repo.'"');

// Metadaten ablegen
if ($output !== false) {
  $hgrc     = $repo.'/.hg/hgrc';
  $contents = "; This files has been generated by Dexter.\n\n".
    "[web]\nname = $directory / $repository";

  if (!empty($description)) {
    $contents .= "\ndescription = $description";
  }

  if (!empty($contact)) {
    $contents .= "\ncontact = $contact";
  }

  file_put_contents($hgrc, $contents."\n");
}
```

Listing A.4: PHP-Code zum Erzeugen eines Repositories

Der obige Code demonstriert, wie basierend auf Formulardaten ein Repository erzeugt werden kann. Dabei werden Name, Beschreibung und Kontaktperson entgegengenommen und entsprechend in der `.hg/hgrc`-Datei des neuen Repositories hinterlegt. Damit tauchen die Informationen im Webfrontend von hgwebdir auf.

Der Code stellt nur einen Ausschnitt aus dem vollständigen Controller dar und sollte ohne weitere Prüfung der Formulardaten nicht direkt verwendet werden.

PHP-Funktion zum Umbenennen von Tags

```php
/**
 * Tags mit Präfix versehen
 *
 * @param string $filename  voller Pfad zur .hgtags
 * @param string $prefix    das zu vergebende Präfix
 */
function prependTagNames($filename, $prefix) {
  // Sollte die .hgtags mehrere Megabyte umfassen, sollte sie zeilenweise
  // eingelesen und verarbeitet werden. Für die die meisten Versionen sollte
  // dieser Code jedoch ausreichend performant sein.

  if (file_exists($filename)) {
    $tagsContent = file_get_contents($filename);
    $tagsContent = preg_replace(
      '#^([a-f0-9]{40})\s+(.*?)$#im',
      '$1 '.$prefix.'$2',
      $tagsContent
    );

    file_put_contents($filename, $tagsContent);
  }
}
```

Listing A.5: PHP-Funktion zum Umbenennen von Tags

Der Code in Listing A.5 definiert eine PHP-Funktion, die in einer `.hgtags`-Datei jedem Tag ein gemeinsames Präfix gibt. Dabei wird der einfache zeilenweise Aufbau der Datei ausgenutzt. Die Funktion wird benötigt, wenn verschiedene Repositories gemergt werden, um sicherzustellen, dass Tags nicht kollidieren und im Ergebnis-Repository eindeutig sind.

Shellscript zur Ad-Hoc-Umbennung von Tags

```
#!/bin/sh
# Aufruf via ./updatetags.sh [addon]

cd $1-converted
hg up
sed -e "s/^\([0-9a-f]\+\) \(.\+\)$/\1 $1 \2/" .hgtags > .hgtags.new
rm .hgtags
mv .hgtags.new
hg remove sally/include/addons/$1/.hgtags
hg commit -m "update tags"
```

Listing A.6: Shellscript zur Ad-Hoc-Umbennung von Tags

Das obige Script wechselt in das Repository, aktualisiert die Arbeitskopie und erweitert dann die `.hgtags` um den String `test`, bevor es die Änderung als neuen Commit ablegt. Gleichzeitig wird die überflüssige Version der Tagdatei gelöscht.

Shellscript zur In-Place-Umbenennung von Tags

```
#!/bin/sh
# Aufruf via ./updatetags.sh [addon]

cd $1-converted
hg up
hg qimport -r tip
sed -e "s/^\([0-9a-f]\+\) \(.\+\)$/\1 $1 \2/" .hgtags > .hgtags.new
rm .hgtags
mv .hgtags.new
hg remove sally/include/addons/$1/.hgtags
hg qrefresh
hg qfinish tip
```

Listing A.7: Shellscript zur In-Place-Umbenennung von Tags

Das in Listing A.7 gegebene Script stellt eine Erweiterung des Scripts aus Listing A.6 dar und verwendet mq (Mercurial Queues), um die Änderung an den Tags ohne neuen Commit zu versionieren. Dabei wird die Datei auf die gleiche Weise wie im davor gegebenen Script geändert, diese Änderung dann aber in den letzten Commit („update tags“, der durch die Konvertierung des Repositories entstanden ist) integriert.

PHP-Code zum automatischen Mergen zweier Tagdateien

```
// Eingabe prüfen
if (count($argv) < 4) {
  die('Usage: php '.basename(__FILE__).' local other output'.PHP_EOL);
}

// Variablen anlegen
list ($program, $local, $other, $output) = $argv;

// Dateien prüfen
if (!file_exists($local)) die('Local file does not exist.'.PHP_EOL);
if (!file_exists($other)) die('Other file does not exist.'.PHP_EOL);

// Dictionary {tag: node, tag: node}
$lines = array();

// Tags einlesen
$lines = readTags($lines, $local);
$lines = readTags($lines, $other);

// Tags sortieren
uksort($lines, 'strnatcasecmp');

// Tags schreiben
$out = fopen($output, 'wb');
```

```
foreach ($lines as $tag => $hash) {
  fwrite($out, "$hash $tag\n");
}

fclose($out);

/**
 * Tags einlesen
 *
 * Liest die Tagdatei zeilenweise ein und beachtet dabei
 * die spezielle Sematik der .hgtags-Datei
 *
 * @param  array  $dictionary  aktuelles Dictionary
 * @param  string $filename    Dateiname
 * @return array               neues Dictionary
 */
function readTags(array $dictionary, $filename) {
  $lines = file($filename); // Datei zeilenweise einlesen

  foreach ($lines as $line) {
    $line = trim($line);

    // nur bei nicht-leeren Zeilen agieren
    if (strlen($line) > 0) {
      // in max. zwei Teil auftrennen
      list ($hash, $tag) = explode(' ', $line, 2);

      // ist der Hash 000..000, entferne das Tag
      if (preg_match('#^0+$#', $hash)) {
        unset($dictionary[$tag]);
      }

      // Tag hinzufügen / neu setzen
      else {
        $dictionary[$tag] = $hash;
      }
    }
  }

  return $dictionary;
}
```

Listing A.8: PHP-Code zum automatischen Mergen zweier Tagdateien

Listing A.8 enthält ein Script, das als Merge-Programm direkt in Mercurial konfiguriert werden kann, um Konflikte zwischen zwei `.hgtags`-Dateien automatisch aufzulösen. Dabei werden die beiden Dateien nacheinander eingelesen, wobei ein Dictionary von definiertem Tag auf die dazugehörige Prüfsumme erstellt wird. Das Dictionary wird dann im Anschluss in die Ausgabedatei geschrieben.

Der Vorgang löst nicht nur doppelte Tags auf, sondern sortiert die Tags auch noch alphabetisch. Da beim Neusetzen von Tags durch Mercurial mehrere Zeilen in den Dateien erzeugt werden, ist die resultierende Datei in der Regel kleiner, da der Vorgang Dopplungen entfernt.

C#-Code zum automatischen Mergen zweier Tagdateien

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text.RegularExpressions;

namespace mergetags {
  class Program {
    static void Main(string[] args) {
      if (args.Length < 3) {
        System.Console.Error.WriteLine("Usage: app.exe local other output");
        return;
      }

      String local = args[0];
      String other = args[1];
      String output = args[2];

      if (!File.Exists(local)) {
        System.Console.Error.WriteLine("Local file does not exist.");
        return;
      }

      if (!File.Exists(other)) {
        System.Console.Error.WriteLine("Other file does not exist.");
        return;
      }

      // {tag: node, tag: node}
      SortedDictionary<string, string> lines = new SortedDictionary<string, string>();

      // read all tags

      ReadFile(lines, local);
      ReadFile(lines, other);

      // write the new tags file

      StreamWriter w = new StreamWriter(output);

      foreach (string key in lines.Keys) {
        w.Write(lines[key]+" "+key+"\n"); // manually ensure UNIX newlines
      }

      w.Close();
```

```
    }

    static void ReadFile(SortedDictionary<string, string> lines, String filename) {
      StreamReader r = new StreamReader(filename);
      String line = null;

      while ((line = r.ReadLine()) != null) {
        line = line.Trim();

        if (line.Length > 0) {
          string[] parts = line.Split(new char[] { ' ' }, 2);

          string hash = parts[0];
          string tag  = parts[1];

          // if the hash is 000...000 only, the tag has been deleted

          if (Regex.Match(hash, "^0+$").Success) {
            lines.Remove(tag);
          }
          else if (lines.ContainsKey(tag)) {
            lines[tag] = hash;
          }
          else {
            lines.Add(tag, hash);
          }
        }
      }

      r.Close();
    }
  }
}
```

Listing A.9: C#-Code zum automatischen Mergen zweier Tagdateien

Listing A.9 enthält das schon in A.8 auf Seite 83 gegebene Programm, jedoch implementiert in C#. Dies macht es möglich, aus dem Code eine eigenständig lauffähige Programmdatei zu erstellen, die auf Entwicklerrechnern einfacher zu konfigurieren ist als ein PHP-Skript. Die beiden Codes sind algorithmisch identisch.

Konfiguration des Mergetools für .hgtags

```
[merge-patterns]
.hgtags = append

[merge-tools]
append.executable = php
append.args = /pfad/zur/mergetags.php $local $other $output
```

Listing A.10: Konfiguration des Mergetools für .hgtags

Listing A.10 auf der vorherigen Seite zeigt, wie das Mergetool für `.hgtags`-Dateien in Mercurial konfiguriert werden kann. Dargestellt wird dabei die Integration des PHP-Skripts, das in `/pfad/zur/mergetags.php` abgelegt wurde. Die PHP-Programmdatei `php` muss dabei systemweit erreichbar sein (beispielsweise in der Umgebungsvariable `PATH` von Windows notiert sein).

Script einer beispielhaften Projekterzeugung

```
$ ls -1
base-project
convert.sh
feeds
mergetags.php
updatetags.sh
usermngt

$ ./convert.sh feeds
initializing destination feeds-converted repository
scanning source...
sorting...
converting...
1 initial
0 Added tag beta1 for changeset 9f4a80544562
updating tags

$ ./convert.sh usermngt
initializing destination usermngt-converted repository
scanning source...
sorting...
converting...
1 initial
0 Added tag beta1 for changeset 0e5d3dd09a85
updating tags

$ ./updatetags.sh feeds
3 files updated, 0 files merged, 0 files removed, 0 files unresolved

$ ./updatetags.sh usermngt
3 files updated, 0 files merged, 0 files removed, 0 files unresolved

$ cd base-project
$ hg tags
tip                                1:8a01124e57c6
v1.0.0                             0:373c7816274d

$ hg pull -f ../feeds-converted
pulling from ../feeds-converted
searching for changes
warning: repository is unrelated
adding changesets
adding manifests
adding file changes
added 3 changesets with 3 changes to 3 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)

$ hg merge
```

```
merging .hgtags
1 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)

$ hg commit -m "merged feeds addOn"
$ hg tags
tip                                5:19ff87abfb5e
feeds beta1                        2:49127a741dce
v1.0.0                             0:373c7816274d

$ hg pull -f ../usermngt-converted
pulling from ../usermngt-converted
searching for changes
warning: repository is unrelated
adding changesets
adding manifests
adding file changes
added 3 changesets with 3 changes to 3 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)

$ hg merge
merging .hgtags
1 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)

$ hg commit -m "merged user addOn"
$ hg glog
@    changeset:   9:cbd681d1a17f
|\   tag:         tip
| |  parent:      5:19ff87abfb5e
| |  parent:      8:191fb0c96dc1
| |  summary:     merged user addOn
| |
| o  changeset:   8:191fb0c96dc1
| |  summary:     update tags
| |
| o  changeset:   7:9a60451902d8
| |  summary:     Added tag beta1 for changeset 0e5d3dd09a85
| |
| o  changeset:   6:32616871cb9b
|    tag:         usermngt beta1
|    parent:      -1:000000000000
|    summary:     initial
|
o    changeset:   5:19ff87abfb5e
|\   parent:      1:8a01124e57c6
| |  parent:      4:ee6055363321
| |  summary:     merged feeds addOn
| |
| o  changeset:   4:ee6055363321
| |  summary:     update tags
| |
| o  changeset:   3:3c545770d9c6
| |  summary:     Added tag beta1 for changeset 9f4a80544562
| |
| o  changeset:   2:49127a741dce
|    tag:         feeds beta1
```

```
|    parent:      -1:000000000000
|    summary:     initial
|
o  changeset:   1:8a01124e57c6
|  summary:     Added tag v1.0.0 for changeset 373c7816274d
|
o  changeset:   0:373c7816274d
 tag:         v1.0.0
 summary:     initial

$ cat .hgtags
49127a741dce9f1dc271565a87f9c3012241b4bb feeds beta1
32616871cb9b82af39667d22d3de26e818076697 usermngt beta1
373c7816274d5e31d7f7302245bd72fe7660055b v1.0.0

$ ls -aRI .hg -I . -I ..
.:
.hgtags  sally  test.txt

./sally:
include

./sally/include:
addons

./sally/include/addons:
feeds  usermngt

./sally/include/addons/feeds:
init.php

./sally/include/addons/usermngt:
init.php
```

Listing A.11: Script einer beispielhaften Projekterzeugung

Das vorangegangene Script zeigt den Ablauf einer Projekterzeugung, wenn sie manuell auf der Kommandozeile durchgeführt würde. Dabei werden dem Projekt `base-project` die beiden AddOns `feeds` und `usermngt` hinzugefügt. Diese werden zuerst konvertiert, um in das Pfadschema des Projekts zu passen. Im Anschluss werden die Tags korrigiert und die beiden Repositories dann über einen Pull in das Projekt importiert. `-f` erzwingt dabei den Pull, da Mercurial ihn sonst nicht durchführen würde, da die Repositories jeweils keine gemeinsamen Revisionen haben. Nach einem Pull wird jeweils gemergt, wobei das konfigurierte Mergetool zum Einsatz kommt.

Im Anschluss an die Projekterzeugung wird der entstandene Revisionsgraph mittels der Erweiterung Graphlog zu Demonstrationszwecken ausgegeben. Danach werden die resultierenden Tags aufgelistet und die Verzeichnisstruktur des Projekts angezeigt.

Die Befehle, die jeweils nur Struktur und Inhalte ausgeben, sind für die eigentliche Projekterzeugung nicht relevant und dienen nur der Demonstration.

Konfiguration eines changegroup-Hooks

```
[hooks]
changegroup.updaters = php /path/to/hook/scripts/updaters.php >&2
```

Listing A.12: Konfiguration eines changegroup-Hooks

Über `>&2` wird jegliche Ausgabe des Hooks auf `stderr` umgeleitet, wodurch sie zum Client übertragen wird. Dies ist notwendig, da eine Ausgabe auf `stdout` den eigentlichen Datentransfer von Mercurial stören würde (vgl. [Div11b], Abschnitt 4.21).

Für einen Server auf Basis von hgwebdir sollte diese Konfiguration in der für alle Repositories gültigen `config`-Datei erfolgen. Bei nur wenigen davon betroffenen Repositories kann dies auch in der jeweiligen `.hg/hgrc` angegeben werden.

Das Script übernimmt in diesem Fall die Aufgabe, zu prüfen, ob es sich um einen Push auf ein AddOn oder auf ein anderes Repository handelt. Im Anschluss kann die Konvertierung angestoßen werden.

Konfiguration eines changegroup-Hooks

```
// Dieses Script läuft als Mercurial-Hook immer im Root-Verzeichnis des
// Repositories, das gerade betroffen ist.
$cwd = getcwd();

if (strpos($cwd, '/srv/hg/Sally-Basis/AddOns/') !== 0) {
        exit('Kein Push auf ein AddOn erkannt.');
}

// Vorgang vorbereiten
$addonName = basename($cwd);
$targetDir = '/srv/hg/Sally-Basis/AddOns/Updaters/'.$addonName;

// entferne eventuell bestehendes Repository
if (is_dir($targetDir)) {
        exec('rm -rf "'.$targetDir.'"');
}

// erstelle Filemap
$filemap = tempnam(sys_get_temp_dir(), 'filemap');
file_put_contents($filemap, 'rename . sally/include/addons/'.$addonName);

// konvertiere Repository
exec('hg convert --filemap="'.$filemap.'" . "'.$targetDir.'"');

// aktualisiere Tags
// ...

// verhindere Pushes auf das Update-Repo
file_put_contents($targetDir.'/.hg/hgrc', "[web]\ndeny_push = *\n");
```

Listing A.13: Changegroup-Hook

Listing A.13 auf der vorherigen Seite zeigt ein PHP-Skript, das als Hook-Skript konfiguriert werden kann, um bei jedem Push auf ein Repository eine konvertierte Version zu erzeugen. Das Ergebnis kann als „Update-Repository“ bezeichnet werden, da von ihm aus von Entwicklern Pull-Operationen durchgeführt werden können, um die AddOns in einem Projekt zu aktualisieren.

Der letzte Schritt im Listing stellt eine Komfort-Funktion dar. Wird ein AddOn über TortoiseHg aktualisiert, so zeigte sich, dass Entwickler das gerade eingespielte Update in den meisten Fällen sofort in das Projekt-Repository auf dem Server pushen wollen. Da dabei oft vergessen wird, die URL zurückzusetzen, würden dann aus Versehen sämtliche Änderungen aus dem Projekt (die gesamte Basis inklusive aller anderen AddOns) in das Update-Repository gepusht werden.
Um dies zu vermeiden, werden Pushes auf diese Repositories generell nicht erlaubt.

PHP-Code zum Ermitteln des Start-Tags

```
$start  = '';
$starts = array('merged_back', 'project_created', '0');

foreach ($starts as $s) {
  $output = hg('identify -r '.$s);

  if ($output) {
    // Ausgabe ist "Prüfsumme Tag"
    $output = explode(' ', reset($output), 2);
    $start  = $output[0];

    break;
  }
}
```

Listing A.14: PHP-Code zum Ermitteln des Start-Tags

Listing A.14 zeigt beispielhaft, wie das ideale Start-Tag bei der Suche nach Änderungen ermittelt werden kann. Von diesem Tag aus werden alle Commits auf Änderungen hin untersucht, die von einem Projekt wieder zurück in die Basis-Repositories (entweder SallyCMS selbst oder ein AddOn) übertragen werden müssen. Zu diesem Zweck werden die Tags `merged_back` und `project_created` auf Existenz geprüft. Sollte keines der beidem Tags existieren, wird die Nullrevision verwendet, die in jedem Repository vorhanden ist.

Konfiguration von erweiterten Git-Diffs

```
[diff]
git = true

[mq]
git = true
```

Listing A.15: Konfiguration von erweiterten Git-Diffs

Listing A.15 auf der vorherigen Seite zeigt, wie die erweiterten Git-Diffs konfiguriert werden können. Diese enthalten auch Verschiebe/Kopier-Operationen und Diffs zwischen Binärdateien.

PHP-Skript zum Vorbereiten eines Repositories

```
// zu vergleichende Verzeichnisse
$src = '/srv/hg/Sally-Basis/AddOns/feeds';
$dst = '/srv/hg/Sally-Basis/AddOns/Updaters/feeds';

// falls das Ziel-Repository nicht existiert, abbrechen
if (!is_dir($dst)) return;

// ins Zielverzeichnis wechseln
chdir($dst);

// Die letzte Revision ("update tags") muss in jedem Fall entfernt werden.
$toStrip = hg('identify -r tip');

if (file_exists($dst.'/.hg/shamap')) {
  // Mapping einlesen und Reihenfolge umkehren
  $mapping = array_map('trim', file($dst.'/.hg/shamap'));
  $mapping = array_reverse($mapping);
  $mapping = array_values($mapping);

  // im Quellrepository die Revisionen suchen
  chdir($src);

  foreach ($mapping as $line) {
    list($old, $new) = explode(' ', $line);

    // prüfe, ob Quellrevision weiterhin existiert; falls nein, muss ihre
    // konvertierte Entsprechung entfernt werden
    if (hg('identify -r '.$old) === false) {
      $toStrip = $new;
    }
    else {
      break;
    }
  }
}

// Revisionen im Zielrepository entfernen
chdir($dst);
hg('strip --nobackup '.$toStrip);

// Nun kann die Konvertierung begonnen werden.
// hg convert ...
```

Listing A.16: PHP-Skript zum Vorbereiten eines Repositories

Listing A.16 auf der vorherigen Seite zeigt eine Implementierung des Verfahrens, das beim Erzeugen der „Update-Repositories“ alle Commits im Ziel-Repository entfernt, die keine Entsprechung im Quell-Repository haben. Dabei wird die von der convert-Erweiterung angelegte Datei `SHAMAP` verwendet, in der eine Zuordnung von Quell- zu Ziel-Commit abgelegt wird. Diese Datei wird rückwärts durchgegangen, wobei bei jedem Commit geprüft wird, ob dieser im Quell-Repository vorhanden ist. Ab dem ersten gefundenen Commit wird der Vorgang abgebrochen und alle nicht gefundenen Commits werden über `strip` aus dem Ziel-Repository entfernt.

Im Anschluss kann die convert-Erweiterung den Konvertierprozess inkrementell fortsetzen. Dabei werden nur die Commits konvertiert, die durch einen Push übertragen wurden. So bleibt der Aufwand für das Erstellen der Update-Repositories immer minimal.

Konfiguration von automatischen Push-Hooks

```
[paths]
bb = https://bitbucket.org/webvariants/sallycms/
gc = https://sallycms.googlecode.com/hg/

[hooks]
changegroup.push_bb = hg push bb >&2
changegroup.push_gc = hg push gc >&2
```

Listing A.17: Konfiguration von automatischen Push-Hooks

Das obige Listing zeigt, wie Hooks dazu verwendet werden können, bei jedem Push auf ein Repository die Änderungen direkt an weitere Repositories zu pushen. Im Beispiel ist zu sehen, dass bei Push-Operationen die Änderungen zu Bitbucket und Google Code weitergeleitet werden.

Literaturverzeichnis

[All] Allman, Eric: *An Introduction to the Source Code Control System*. http://sccs.berlios.de/man/sccs.me.html. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 4)

[BCS08] Ben Collins-Sussman, Brian W. Fitpatrick, C. Michael Pilatos: *Appendix B. Subversion for CVS Users*. http://svnbook.red-bean.com/en/1.5/svn.forcvs.html, 2008. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 8)

[Ben08] Ben Collins-Sussman, Brian W. Fitpatrick, C. Michael Pilatos: *hg*. http://svnbook.red-bean.com/en/1.5/svn.branchmerge.basicmerging.html, 2008. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 32 und 55)

[Bla11] Black Duck Software, Inc.: *Compare Repositories*. http://www.ohloh.net/repositories/compare, 2011. [online; abgerufen am 15. Mai 2011]. (zitiert auf Seite 20)

[Cha09] Chacon, Scott: *Pro Git*. Springer, 2009. (zitiert auf Seite 38 und 77)

[Cha11] Chacon, Scott: *Git Community Book*. http://book.git-scm.com/, 2011. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 4, 8 und 19)

[CK10] Chsitian Kästner, Sven Apel, Gunter Saake: *Erweiterte Programmierkonzepte für maßgeschneiderte Datenhaltung*. http://wwwiti.cs.uni-magdeburg.de/iti_db/lehre/epmd/2010/, 2010. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 69)

[Col09] CollabNet, Inc.: *Subversion Growth*. http://www.open.collab.net/community/subversion/articles/Subversion%20Growth.html, 2009. [online; abgerufen am 1. März 2011]. (zitiert auf Seite iii und 16)

[Col11] CollabNet, Inc.: *Become a committer*. http://sharpsvn.open.collab.net/servlets/ProjectProcess?documentContainer=c4_Submit%20a%20patch_Become%20a%20committer, 2011. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 13)

[Div11a] Diverse Autoren: *Convert extension*. http://mercurial.selenic.com/wiki/ConvertExtension, 2011. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 18 und 37)

[Div11b] Diverse Autoren: *Mercurial Frequently Asked Questions*. http://mercurial.selenic.com/wiki/FAQ, 2011. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 90)

[Div11c] Diverse Autoren: *Mercurial source control management*. http://mercurial.selenic.com/about/, 2011. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 19)

[Div11d] Diverse Autoren: *Nodeids*. http://mercurial.selenic.com/wiki/Nodeid, 2011. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 18)

[Fre08] Free Software Foundation: *The Free Software Definition*. http://www.gnu.org/philosophy/free-sw.de.html, 2008. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 2 und 3)

[Gav06] Gavin: *Subversion Standards*. http://framework.zend.com/wiki/display/ZFDEV/Subversion+Standards, 2006. [online; Version 21 vom 13. September 2006]. (zitiert auf Seite 13)

[Gru86] Grune, Dick: *Concurrent Versions System, A Method for Independent Cooperation*. Technischer Bericht, IR 113, Vrije Universiteit, 1986. (zitiert auf Seite 6)

[Lor05] Lord, Thomas: *[Gnu-arch-users] GNU Arch maintainership*. http://lists.gnu.org/archive/html/gnu-arch-users/2005-08/msg00030.html, 2005. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 8)

[Mac11a] Mackall, Matt: *hg*. http://www.selenic.com/mercurial/hg.1.html, 2011. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 18, 60 und 62)

[Mac11b] Mackall, Matt: *New Mercurial logo design*. http://www.selenic.com/hg-logo/, 2011. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 17)

[Mas06] Mason, Mike: *Pragmatic Version Control: Using Subversion (The Pragmatic Starter Kit Series) (2nd Edition)*. Pragmatic Bookshelf, 2006. (zitiert auf Seite 4, 5, 6, 7 und 8)

[Med11] MediaWiki.org: *Commit access*. http://www.mediawiki.org/wiki/Commit_access, 2011. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 13)

[Mic06] Microsoft Corporation: *Team Foundation Server Fundamentals: A Look at the Capabilities and Architecture*. http://msdn.microsoft.com/en-US/library/ms364062(v=VS.80).aspx, 2006. [online; abgerufen am 15. Mai 2011]. (zitiert auf Seite 3)

[Mic11] Microsoft Corporation: *Verwenden von Visual SourceSafe mit anderen Produkten*. http://msdn.microsoft.com/de-de/library/ms181081(v=vs.80).aspx, 2011. [online; abgerufen am 15. Mai 2011]. (zitiert auf Seite 3)

[Mon11] Monotone: *Changelog*. http://www.monotone.ca/NEWS, 2011. [online; abgerufen am 1. März 2011]. (zitiert auf Seite ix und 9)

[Ott09] Otte, Stefan: *Version Control Systems*. Technischer Bericht, Institute of Computer Science, 2009. (zitiert auf Seite 5, 17, 38 und 77)

[Pri05] Price, Derek Robert: *CVS – v1.12.12.1: Overview*. http://ximbiot.com/cvs/wiki/CVS--Concurrent%20Versions%20System%20v1.12.12.1%3A%20Overview, 2005. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 6)

[Pri06] Price, Derek Robert: *CVS – Concurrent Versions System*. http://www.nongnu.org/cvs/, 2006. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 8)

[Res09] Research, Forrester: *Global Developer Technographics Survey*, August 2009. (zitiert auf Seite 8)

[Roc75] Rochkind, Marc J.: *The Source Code Control System*. IEEE Transactions on Software Engineering, SE-1:364–370, dec 1975. (zitiert auf Seite 3)

[RS10] Randal Schwartz, Aaron Seigo: *FLOSS Weekly 122: Mercurial*. http://www.youtube.com/watch?v=1L1dlzwrCPU, 2010. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 21)

[Spa09] SpamAssassin: *BecomingCommitter*. http://wiki.apache.org/spamassassin/BecomingCommitter, 2009. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 13)

[Tai08a] Tai, Andy: *GNU arch*. http://www.gnu.org/software/gnu-arch/, 2008. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 8)

[Tai08b] Tai, Andy: *Re: [Gnu-arch-users] revc*. http://lists.gnu.org/archive/html/gnu-arch-users/2008-03/msg00003.html, 2008. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 8)

[The] The Chromium Authors: *Become a Committer*. http://www.chromium.org/getting-involved/become-a-committer. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 13)

[Tor05] Torvalds, Linus: *Git*. http://www.youtube.com/watch?v=4XpnKHJAok8, 2005. [online; abgerufen am 1. März 2011]. (zitiert auf Seite 14 und 17)

[TT85] Tichy, Walter F. und Walter F. Tichy: *RCS: a system for version control*. Software — Practice & Experience, 1985. (zitiert auf Seite 5)

[Wik11a] Wikipedia: *List of revision control software*. http://en.wikipedia.org/w/index.php?title=List_of_revision_control_software&oldid=428719876, 2011. [online; abgerufen am 15. Mai 2011]. (zitiert auf Seite 3)

[Wik11b] Wikipedia: *Symbolische Verknüpfung*. http://de.wikipedia.org/w/index.php?title=Symbolische_Verknoldid=88651798, 2011. [online; abgerufen am 20. Mai 2011]. (zitiert auf Seite xviii)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 1. August 2011