Auswirkungen der Benutzung von zentralen und dezentralen Versionsverwaltungssystemen In Open Source Projekten

Masterarbeit an der Freien Universität Berlin für den Abschluss zum Master of Computer Science

Jana von dem Berge

16. April 2009

Fachbereich Mathematik und Informatik
Arbeitsgruppe Software Engineering
Betreuer: Christopher Oezbek

Selbstständigkeitserklärung:

Hiermit erkläre ich, dass ich diese Arbeit vollkommen selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Berlin, 16. April 2009

Jana von dem Berge

Inhalt

1 Einleitung

1.1 Open Source Projekte

Open Source-Projekte sind Software-Projekte, deren Quellcode frei verfügbar und allen Interessenten über das Internet zugänglich ist. Sie sind nicht-kommerziell. Jeder Interessent kann sich an der Weiterentwicklung dieser Projekte beteiligen – d. h. er kann den Code nach seinen Vorstellungen verändern, und ggf. seine Änderungen an die Projekte weiterleiten. Es gibt Beteiligte auf verschiedenen Ebenen, die mitunter an unterschiedlichen Orten der Welt arbeiten. Ihre Hauptkommunikationsplattform sind so genannte Mailinglisten.

Der Quellcode von Software-Projekten wird von Versionsverwaltungssystemen verwaltet, deren Ansatz entweder zentral oder dezentral ist, was Auswirkungen unterschiedlicher Art auf die Arbeitsprozesse in der Entwicklung der betroffenen Projekte hat.

Diese Arbeit befasst sich damit mit einem Teilbereich eines größeren Forschungsvorhabens (Doktorarbeit von Christopher Oezbek am Institut für Informatik der FU Berlin, Arbeitsgruppe Sofware Engineering, „Einführung von Innovationen in Open Source Projekte“). Im Sinne dieser Arbeit ist eine Innovation “something that is intended to change any kind of process in a project” [1].

1.2 Aufgabenstellung

Untersucht werden sollten zentrale und dezentrale Versionsverwaltungssysteme hinsichtlich ihrer unterschiedlichen Auswirkungen auf die Arbeitsprozesse speziell in Open Source Projekten. Laut Aufgabenstellung sollte dabei auf Einführungsprozesse bei einer Migration des Werkzeuges, Adoption- sowie Lernprozesse nach der Einführung und die Machtverhältnisse der Projektmitglieder, die dabei eine Rolle spielten

eingegangen werden. Erforscht werden sollten außerdem Unterschiede in der Verwaltung und der Administration.

Mit Hilfe einer qualitativen Forschungsmethode, d.h. „ohne statistische Verfahren oder andere Arten der Quantifizierung“ [2] wurden offene, d. h. allgemein zugängliche, Mailinglisten untersucht. Mailinglisten von Open Source-Projekten erlauben einen großen Einblick in den Entwicklungsprozess. Die untersuchten Mailinglisten stammen von Gmane [3] einem Online-Mailinglisten-Archiv, welches hauptsächlich für Mailinglisten mit dem Schwerpunkt Open Source Software eingerichtet wurde. Die Untersuchung beschränkt sich dabei auf die Emails des Jahres 2007.
Die zugehörigen Projekte waren unterschiedlich umfangreich – in den Mailinglisten befanden sich zwischen 500 und 9000 E-Mails aus dem besagten Zeitraum, die teilweise schon im Vorfeld durch Christopher Oezbek nach Art der Grounded Theory kodiert wurden.

Ein wesentliches Problem bei der Untersuchung der E-Mails lag darin, dass sicher keine Einblicke in alle Arbeitsprozesse und Probleme der betreffenden Projekte gewährt werden, da es außerhalb der Mailinglisten noch andere Kommunikationsplattformen gibt wie etwa Chats oder persönliche Gespräche.

Am Ende kam diese Arbeit deshalb auch leider nicht in allen Punkten der Aufgabenstellung zu einem erwünschten Ergebnis.

1.3 Versionsverwaltungssysteme

1.3.1 Einführung

Versionskontrollsysteme (englisch Source Control Management, kurz SCM) dienen der Verwaltung von und der Zugriffsregelung auf Dateien sowie der Protokollierung von Änderungen an den Dateien. Je mehr Personen an gleichen Dateien arbeiten, desto häufiger kann es vorkommen, dass es bei Änderungen der Dateien zu Konflikten kommt. Diesen Konflikten versucht ein SCM entweder selbst aus dem Wege zu gehen oder die Benutzer auf mögliche Konflikte hinzuweisen.

Alle zu verwaltenden Dateien sowie deren Enwicklungsgeschichte lagert das SCM im sogenannten Repository. Sobald ein Benutzer Änderungen an einer oder mehreren Dateien vornimmt und per Commit in das Repository überträgt, legt das SCM eine neue Version - im SCM Revision genannt - an. Jede Revision hat innerhalb eines Repositories eine eindeutige Kennzeichnung, über die auf sie zugegriffen werden kann. Zu einer Revision gehören zudem Metadaten, wie der Name des Autors, ein Zeitstempel sowie die Commit Message, in der der Autor seine Änderungen kurz beschreiben kann. Ein Benutzer hat so die Möglichkeit sehr einfach über die Commit Messages den Entwicklungsverlauf nachvollziehen zu können, vorausgesetzt, jeder Benutzer kommentiert seine Änderungen sorgfältig.

Die meisten SCMs unterstützen Branches. Ein Branch ist ein Fork von Dateien aus dem Repository zu einem bestimmten Zeitpunkt. An diesen Dateien kann dann parallel und unabhängig von den Dateien der Hauptentwicklungslinie, dem Trunk, aus denen sie hervorgegangen sind, gearbeitet werden. Ein Branch ist nützlich, wenn ein Benutzer gravierende Änderungen an verschiedenen Dateien über einen längeren Zeitraum machen möchte, und sicher gehen will, dass kein anderer Benutzer dazwischen einen unerwarteten Commit macht. Auch wenn man etwas ausprobieren möchte und sich noch nicht sicher ist, ob es überhaupt hinterher in den Trunk mit einfließen soll, verwendet man dafür üblicherweise einen Branch.

Ist die Arbeit an einem Branch abgeschlossen, dann kann er wieder mit dem Trunk zusammengeführt werden (Merge). Die Verschiedenen SCMs verwenden dafür unterschiedliche Mergealgorithmen, welche diese Zusammenführung übernehmen, jedoch sollte dieser Vorgang stets noch einmal von einem Benutzer überprüft und ggf. modifiziert werden, auch wenn das SCM keinen Konflikt meldet. Merges zwischen Trunk und Branch werden aber oft auch zwischendurch gemacht, wenn zum Beispiel ein Fehler im Branch behoben wurde, der auch für den Trunk von Bedeutung ist.
Um eine Revision besonders zu kennzeichnen (z.B. wenn man einen Entwicklungsstand veröffentlichen möchte), versieht man sie mit einem Etikett (Tag). Das ermöglicht es, der Revision anstatt der im Repository üblichen Identifikation einen sinnvollen Namen zu geben, z.B. „Release 2.1“.

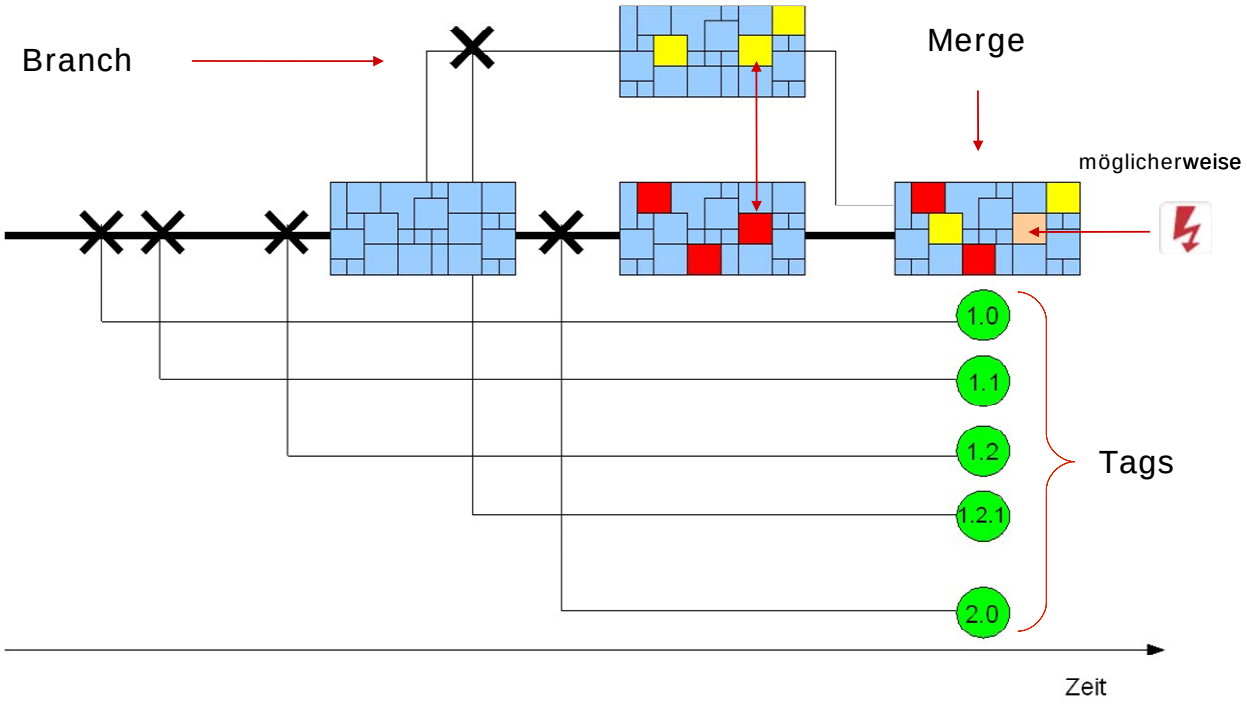

Abbildung 1: Branching, Merging, Tagging

Repositoryzugriff

Es gibt verschiedene Ansätze, wie auf die Dateien eines Repositorys zugegriffen werden kann.

Bei Open Source-Projekten wird üblicherweise ein lesender Zugriff gewährt, in anderen Projekten wird der Zugriff so weit wie notwendig eingeschränkt.

Beim schreibenden Zugriff gibt es grundsätzlich zwei Verfahrensweisen. Hat der Benutzer Schreibrechte für das Repository, in das er seine Änderungen übertragen möchte, kann er dies mit einem Commit direkt tun *(Commit-based Collaboration)*. Andernfalls, muss er seine Änderungen in einem Patch zusammenfassen und an eine Person, die die benötigten Schreibrechte hat, z.B. per Email, weitergeben *(Patch-based Collaboration)*. Diese kann dann entscheiden, ob sie das Patch ins Repository per Commit an das Repository weitergibt oder ablehnt. Hierzu mehr in den späteren Kapiteln.

Es gibt sehr viele unterschiedliche SCMs, die sich teilweise nur durch Kleinigkeiten unterscheiden. Auch lassen sich SCMs durch den Einsatz von zahlreichen zusätzlichen Tools erweitern und an die jeweiligen Bedürfnisse ihrer Benutzer anpassen. Grundsätzlich lassen sich SCMs aber in zwei Kategorien einordnen, in zentrale und dezentrale SCMs.

1.3.2 Zentrale Versionsverwaltungssysteme

Verwendet ein Projekt ein zentrales SCM, existiert genau ein Repository auf einem zentralen Server. (Natürlich kann ein Projekt auch in mehrere Teile unterteilt sein, die auf mehrere Repositorys aufgeteilt sind, das spielt hier aber zur Klärung des Begriffs keine Rolle).

Üblicherweise hat jeder Entwickler in seinem lokalen Arbeitsverzeichnis eine Kopie aller Dateien des Repositorys. Diese erhält er, indem er ein Checkout auf das Repository ausführt. Das Arbeitsverzeichnis ist dann ein Snapshot des Repositorys, zu

dem Zeitpunkt des Checkouts. Die Dateien in dem Arbeitsverzeichnis nennt man Working Copys.

Möchte der Entwickler Änderungen an seinen Working Copys ins Repository übertragen und hat auch die entsprechenden Rechte dafür, dann führt er ein Commit aus. Das SCM prüft zunächst, anhand der Revisionsnummer, die jede Working Copy als Attribut mit sich führt, ob diese mit der aktuellen Revisionsnummer im Repository übereinstimmt. Ist dies der Fall, dann hat der Entwickler eine aktuelle Version gehabt und seine Änderungen werden ins Repository übernommen, das SCM erstellt dafür eine neue Revision (üblicherweise wird die Revisionsnummer einfach um 1 erhöht). Hat das SCM festgestellt, dass zwischenzeitlich schon andere Entwickler Änderungen an einer gleichen Datei vorgenommen haben, dann gibt es zwei Möglichkeiten, wie das SCM verfährt. Entweder es versucht mit Hilfe eines Merge-Algorithmus die verschiedenen Änderungen selber miteinander zu vereinen, was der Entwickler dann noch auf Richtigkeit überprüfen muss, oder es weist den Entwickler daraufhin, dass seine Working Copy zu alt ist. Der Entwickler muss dann ein Update ausführen. Dieses bringt dann sein Arbeitsverzeichnis auf den neuesten Stand des Repositorys. Nun muss der Entwickler selber versuchen, seine Änderungen mit denen der anderen in Einklang zu bringen, und dann ins Repository übertragen. Damit so etwas nicht so oft passiert, sollte jeder Entwickler sein Arbeitsverzeichnis möglichst oft auf den neuesten Stand bringen. Die verschiedenen Arbeitsschritte (Checkout, Update, Commit) können bei den meisten CSCMs sowohl mit einzelnen Dateien als auch mit allen Dateien des Repositorys / Arbeitsverzeichnisses durchgeführt werden.

Hat ein Entwickler keine Commit-Rechte, kann er sich nur mit Hilfe von Patches an der Entwicklung beteiligen. Dann werden seine Änderungen aber unter dem Namen desjenigen in das Repository eingespeist, der sie dann ins Repository committet.

Ein großer Nachteil eines CSCM ist die Abhängigkeit von einem zentralen Server. Fällt dieser aus, hat niemand mehr die Möglichkeit, Änderungen in das Repository zu übertragen bzw. ein Update auszuführen. Wenn die Entwickler nämlich öfter nach kleineren Änderungen ein Commit machen und andere dies durch ein Update mitbekommen, dann ist die Gefahr möglicher Konflikte kleiner, als wenn über einen länge-

ren Zeitraum ohne Commit große Änderungen an den Dateien vorgenommen werden. Bei kleineren Commits lassen sich die Änderungen Schritt für Schritt nachvollziehen und ggf. mergen, bei einem großen Commit ist der Aufwand dementsprechend größer.

Schlecht ist es natürlich, wenn zum Beispiel durch einen Festplattenschaden das Repository auf dem Server ganz zerstört wird. Zwar haben dann die verschiedenen Entwickler noch ihre Working Copys in ihrem Arbeitsverzeichnis, die Entwicklungsgeschichte des Projektes geht jedoch vollständig verloren. Deshalb machen die Projekte regelmäßige Backups.

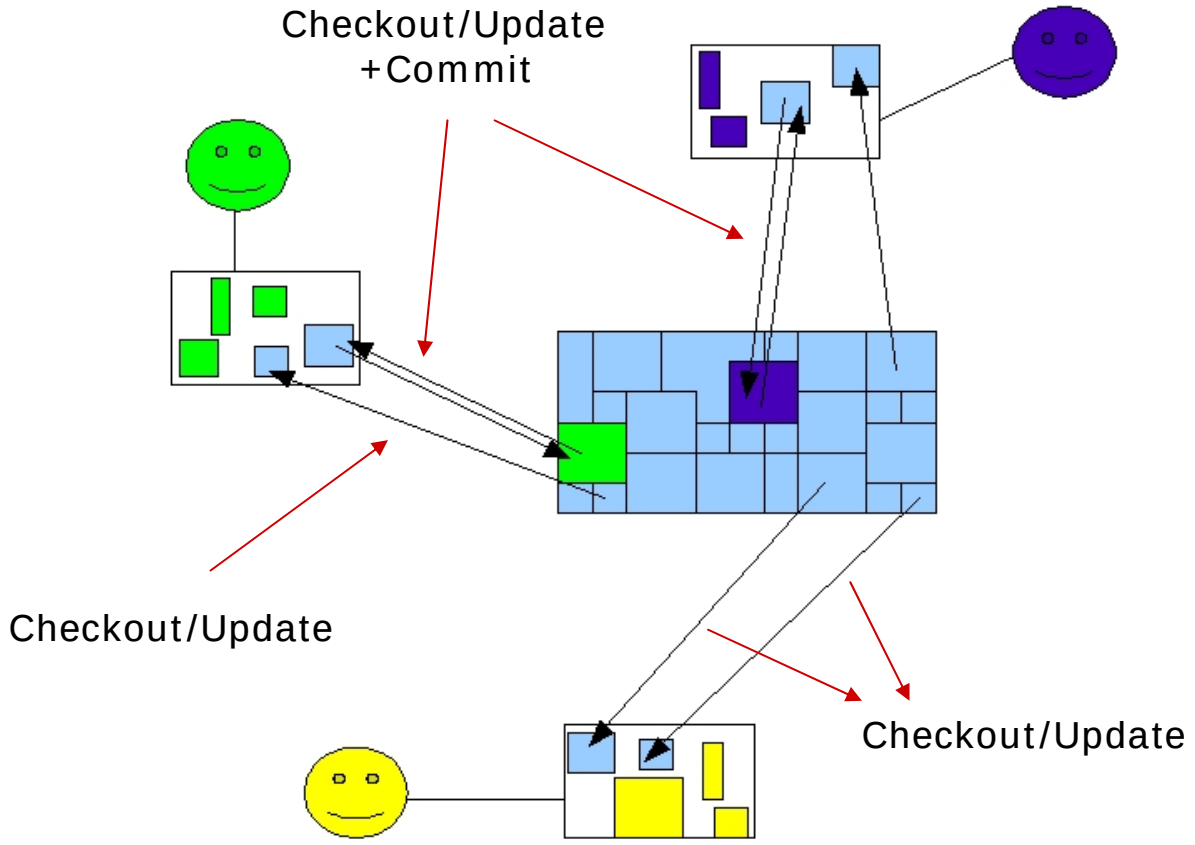

Abbildung 2: Zentrale Versionsverwaltung

1.3.3 Dezentrale Versionsverwaltungssysteme

Bei dezentralen SCMs unterhält jeder Benutzer ein eigenes Repository. Das dezentrale SCM verwaltet dieses Repository in der gleichen Weise wie ein zentrales SCM. Auch hier checkt sich der Benutzer eine Working Copy einer oder mehrerer Dateien aus seinem persönlichen Repository aus, um an ihnen zu arbeiten und die Änderungen dann per Commit ins selbige zu übertragen.

Möchte sich nun ein Benutzer A an der Entwicklung von B beteiligen, muss er zunächst das Repository von B klonen. Dafür muss B sein Repository zunächst öffentlich verfügbar machen. Dafür gibt es Hoster, wie Sourceforge und ähnliche, die einen solchen Service anbieten. Nach dem Klonen besitzt A dann ein eigenes Repository, welches vollständig von seinem lokalen dezentralen SCM kontrolliert wird und unabhängig von dem Repository existiert, von dem es geklont wurde.
Möchte er auf dem Stand des Repositorys von B bleiben, so kann er ein regelmäßiges Pull von dessen Repository ausführen (Remote Update). Möchte er selber seine Änderungen in das Repository von B übertragen, dann gibt es im dezentralen Ansatz 3 Möglichkeiten. Entweder, er veröffentlicht ein eigenes Repository und bittet ihn von seinem Repository zu pullen, oder er hat die entsprechenden Rechte von B eingeräumt bekommen, direkt in dessen Repository zu pushen (Remote Commit). Die dritte Möglichkeit, die immer funktioniert, ist die Versendung eines Patches per Email.

Ein großer Vorteil eines dezentralen SCM liegt darin, dass ein Entwickler zu jedem Zeitpunkt vollständig auf sein Repository zugreifen und Änderungen vornehmen kann, ohne eine Verbindung zu einem Server herstellen zu müssen. Jede Änderung steht unter vollständiger Kontrolle, und wenn man Änderungen von verschiedenen externen Repositorys übernimmt, kann jederzeit nachvollzogen werden, von welchem Autor zu welchem Zeitpunkt welche Änderungen vorgenommen wurden. Denn wenn Benutzer B im obigen Beispiel A keine Schreibrechte für sein Repository erteilen möchte, wo in diesem Fall im zentralen Ansatz nur ein Patch-Betrieb möglich wäre, dann kann er sich die Änderungen von A aus dessen Repository pullen, und dann erscheint auch dessen Name bei den entsprechenden Commits im Repository von B.

Anders als im zentralen Ansatz können die verschiedenen Revisionen nicht anhand von fortlaufenden Nummern identifiziert werden. Da jedes Mitglied eines Projektes sein eigenes Repository unterhält, würden diese Nummern nicht mehr global eindeutig sein. Für dieses Problem gibt es verschiedene Lösungen, bei Git werden Revisionen, wie auch alle anderen Objekte anhand von SHA-1-Prüfsummen identifiziert.

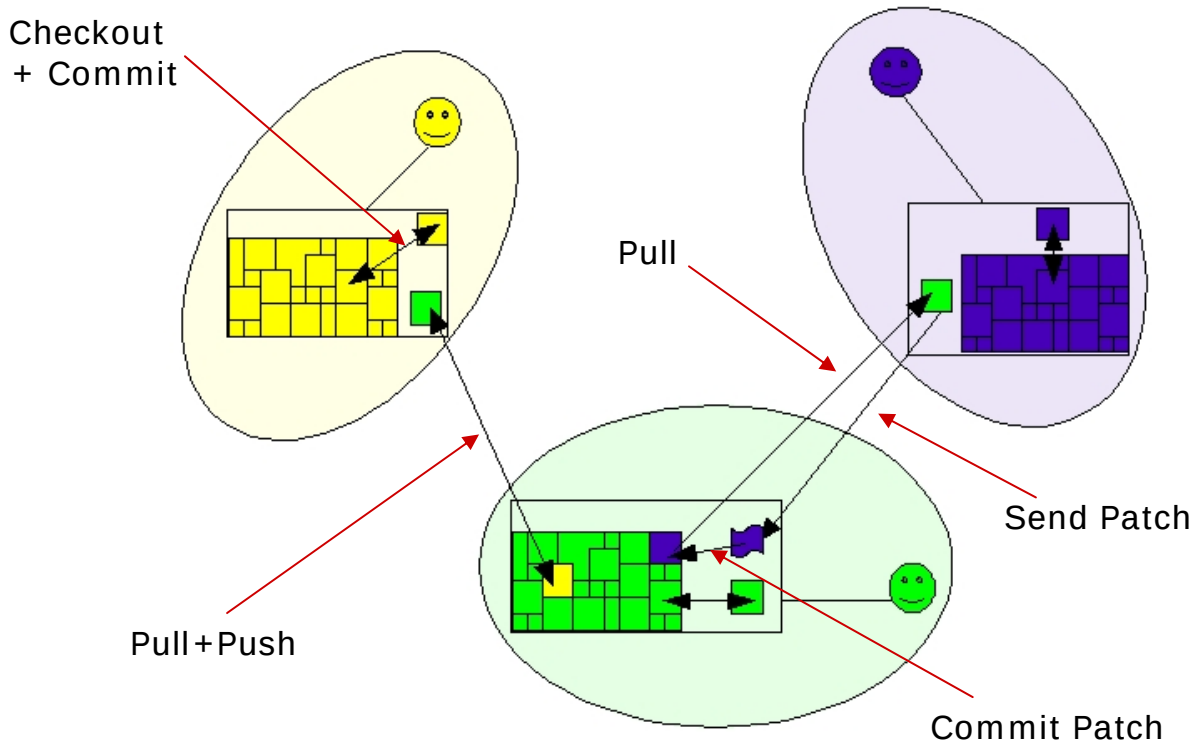

Abbildung 3: Dezentrale Versionsverwaltung

2 Forschungsmethode

2.1 Grounded Theory

Die Grounded Theory ist eine ursprünglich von Strauss und Glaser in den 60ern entwickelte Forschungsmethode, „die eine systematische Reihe von Verfahren benutzt, um eine induktiv abgeleitete, gegenstandsverankerte Theorie über ein Phänomen zu entwickeln“.[1]

2.1.1 Grundlagen

Der Forscher entwickelt eine Theorie schrittweise aus empirisch erhobenen Daten. Dabei beginnt die Theoriebildung schon während der Erhebung der Daten, die weiterführende Datenerhebung kann also den bereits entwickelten Hypothesen angepasst werden – zum Beispiel können im Verlauf einer Reihe von Interviews gewonnene Erfahrungen genutzt werden, um die Formulierung der Fragen zu optimieren.

Die Grounded Theory wird insbesondere in den Bereichen angewendet, in denen eine Theorie nicht allein durch quantitative Analyse von Daten generiert bzw. verifiziert werden kann. Es geht vielmehr um die Erforschung von „sprachvermittelte[n] Handlungs- und Sinnzusammenhänge[n]“ [2]. Bei der Grounded Theory handelt es sich also um eine qualitative Datenanalyse.

Untersucht wird nach Strauss und Corbin stets eine Handlung bzw. Interaktion, „die auf ein Phänomen gerichtet ist, auf den Umgang mit ihm und seine Bewältigung, die Ausführung oder die Reaktion darauf“ [2] oder die zu dem Phänomen in Beziehung stehen. Dabei gibt es sowohl strategische als auch Routinehandlungen. Der Forscher will verstehen, warum oder wie sich Handlungen verändern, gleich bleiben oder zurückentwickeln. Mehrere Handlungs- und Interaktionssequenzen, denen Handlungs- und interaktionale Strategien unterlegt sind, verknüpfen sich zu einem Prozess.

2.1.2 Vorgehensweisen

2.1.2.1 Datenerhebung

Dem Forscher ist es laut Strauss und Corbin selbst überlassen, in welcher Weise er seine Daten auswählt und erhebt. Wichtig ist aber, dass Datenerhebung und –analyse parallel praktiziert werden. Der Forscher formuliert schon in den ersten Analysephasen verschiedene analytische Fragen und Hypothesen, welche er dann bei der weiteren Erhebung von Daten mit einbezieht. Er analysiert in den ersten Phasen seine Daten vollständig; je weiter sich aber seine Theorie entwickelt, desto mehr geht er nur auf Datenteile ein, die mit der Theorie in Zusammenhang stehen.

Bei der weiteren Erhebung von Daten geht es darum „Ereignisse, Vorkommnisse usw. auszuwählen, die Indikatoren für Kategorien, ihre Eigenschaften und Dimensionen sind“, welche wiederum „Indikatoren für theoretisch relevante Konzepte sind“ [2]. Das Heranziehen von Beispielen dieser Art nennt man Theoretical Sampling. Diese Art von Datenerhebung ist nicht planbar, sie orientiert sich an der während der Forschung entstehenden Theorie.

Während des Analyseprozesses können sich neue Einsichten/Fragen ergeben. Dann muss der Forscher schon analysierte Daten nochmals anhand des neuen Wissens durchgehen. Er kehrt während des gesamten Forschungsprozesses immer wieder zu ursprünglichen Daten zurück.
Wenn auch durch Heranziehen neuer Dateien keine neuen Erkenntnisse erwartet werden können, ist die Untersuchung abgeschlossen, der Zustand der theoretischen Sättigung ist erreicht.

Die Ergebnisse seiner Analyse hält ein Forscher während des gesamten Analyseprozesses in Memos und Diagrammen fest. Memos enthalten theoretische Fragen, Hypothesen, Memos stellen also die Ergebnisse in immer aktualisierterer Form dar. Sie helfen, die Theorie zu integrieren.

2.1.2.2 Kodierverfahren

Nach Strauss und Corbin stellt Kodieren „Verfahrensweisen dar, durch die die Daten aufgebrochen, konzeptualisiert und auf neue Art zusammengesetzt werden. Es ist der zentrale Prozess, durch den aus den Daten Theorien entwickelt werden.“ [2]

Die Grounded Theory bietet drei Ebenen der Kodierung: Das offene Kodieren, das axiale Kodieren und das selektive Kodieren.

Beim *offenen Kodieren* werden zu Beginn des Forschungsprozesses Daten im Bezug auf die interessierenden Phänomene „befragt“, der Forscher will wissen, worum es geht, wer beteiligt ist, warum, wie, und womit gehandelt und argumentiert wird. Die Antworten auf diese Fragen ergeben dann die so genannten Konzepte, die Phänomene mit einem Namen benennen. Dabei werden beim weiteren Vorgehen Vergleiche mit vorher benannten Konzepten angestellt, damit „ähnliche Phänomene denselben Namen bekommen können“. [2] Kategorien entstehen durch die Gruppierung von Konzepten, die sich auf ein ähnliches Phänomen beziehen. Das offene Kodieren ist also „der Prozess des Aufbrechens, Untersuchens, Vergleichens, Konzeptualisierens und Kategorisierens von Daten“. [2]

Axiales Kodieren ist das Bilden eines Beziehungsnetzes um eine zentrale Kategorie. Dabei geht es um die Beziehungen zwischen dieser Kategorie und der mit ihr in Beziehung stehenden Kategorien, ihren Subkategorien, die aus dem offenen Kodieren hervorgegangen sind. Die Subkategorien spezifizieren die untersuchte Kategorie (das zentrale Phänomen) und „verleihen ihr Präzision“. [2]
Mit Hilfe des Paradigmatischen Modells versucht der Forscher, die Subkategorien mit einer Kategorie in Verbindung zu setzen, indem er die Subkategorien versucht als das Phänomen verursachende Bedingung, „Kontext, in den das Phänomen eingebettet ist“, „Handlungs- und interaktionale Strategien, durch die es bewältigt, mit ihm umgegangen oder durch die es ausgeführt wird“ oder als Konsequenz dieser Strategien identifiziert. [2]

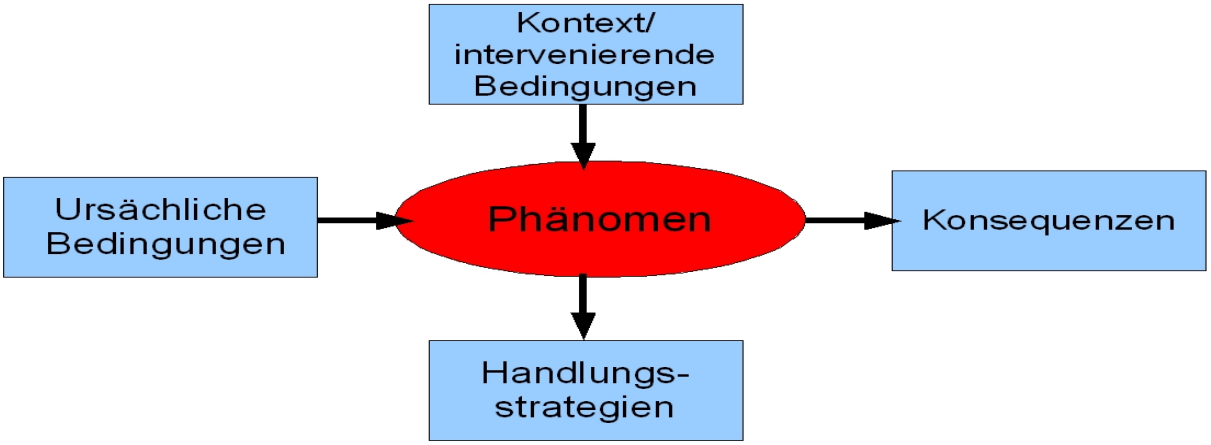

Abbildung 4: Das paradigmatische Modell

Das Modell ermöglicht es dem Forscher „systematisch über Daten nachzudenken und sie in komplexer Form miteinander in Beziehung zu stellen.
Schon beim Offenen Kodieren wird der Forscher oft schon einige Kategorien einem Teil dieses Modells automatisch zuordnen können und entsprechend benennen. Aber auch aus dem axialen Kodieren heraus wird er neue Konzepte entwickeln. Der Forscher pendelt also während der Analyse zwischen offenem und axialem Kodieren hin und her.

Am Ende folgt das *selektive Kodieren*. Dabei wird das Beziehungsnetz aus dem axialen Kodieren sortiert und verfeinert. Die Kategorien werden systematisch gruppiert, und eine Kernkategorie wird bestimmt. Dies ergibt sich durch vielfältige Relationen zu allen anderen Kategorien, sie besitzt eine zentrale Stellung. Diese Kernkategorie „stellt den entscheidenden Kitt beim Zusammenfügen – und beim ordentlichen Zusammenhalten – aller Komponenten der Theorie dar.“ [2]

2.2 Werkzeug GmanDA

Gmanda ist eine von Christopher Oezbek an der Freien Universität in Berlin entwickelte Open Source Software zur qualitativen Analyse von Textdokumenten. Es wurde ursprünglich entwickelt, um Mailinglisten von Gmane.org zu analysieren. Mittlerweile können mit dieser Software aber auch beliebige andere mbox- und Textdateien importiert werden. GmanDA unterstützt die Arbeitsweisen der Grounded Theory.

Die Dokumente werden in einem Fenster in Baumstruktur angezeigt, benannt nach der ID und des jeweiligen Betreffs der Email, geordnet nach Threads. Diese Sicht lässt sich jedoch auch zum Beispiel in die Sicht einer zeitlichen Reihenfolge ändern. Um sich ein bestimmtes Dokument anzeigen zu lassen, scrollt man entweder im Fenster entsprechend herunter, oder kann mit Hilfe einer Menüfunktion unter Angabe der Email-ID sowie des Projektnamens auch direkt zu dem gewollten Dokument gelangen.

2.2.1 Kodieren

Kodiert werden in GmanDA stets ganze Dokumente durch eine Menge von durch Kommata getrennten Codes. Jeder Code hat hierbei folgende Struktur:

Code-1.Code-2.Code-3…Code-X

Mit einer solchen Kodierung wird primär ausgedrückt, dass das kodierte Dokument ein Exemplar oder eine Äußerung zu einem Exemplar der Klasse mit Namen „Code-X“ enthält.

Würde also beispielsweise ein Dokument mit „innovation.scm.Subversion“ kodiert, so kann der Leser primär daraus den Schluss ziehen, dass dieses Dokument eine Aussage bzgl. Subversion enthält. Darüber hinaus dient die Kodierung dazu die Klasse der Dinge mit Namen „Code-X“ in eine Klassenhierarchie einzubetten. Hier soll die

Beziehung „Code-1.Code-2“ gelesen werden als „jedes Exemplar der Klasse Code-2 ist auch ein Exemplar der Klasse Code-1“, d.h. eine „ist ein“-Beziehung. Im vorangehenden Beispiel würde man also aus der Kodierung des Dokumentes mit „innovation.scm.Subversion“ schlussfolgern können, dass dieses a) eine Äußerung zu einem Exemplar von Subversion enthält, b) jede Äußerung zum Thema „Subversion“ auch eine Äußerung zum Thema „Versionsverwaltung“ und c) jede Äußerung zum Thema „Versionsverwaltung“ oder „Subversion“ auch eine Äußerung zum Thema „Innovation“ ist.

Es gibt die Möglichkeit jeden Code mit Hilfe einer Definition zu beschreiben. Das hilft dem Benutzer sich zu einem späteren Zeitpunkt darüber klar zu werden, für welche Verwendung er diesen Code vorgesehen hatte, um ihn nicht möglicherweise an einer anderen Stelle falsch zu vergeben. Auch können andere Benutzer vorkodierte Dokumente besser verstehen und die Vergabe von Codes besser nachvollziehen.

2.2.2 Filtern

GmanDA bietet verschiedene Möglichkeiten, Emails nach vergebenen Codes oder Suchwörtern zu filtern. Die Emails können nach einem oder mehreren Codes gefiltert werden, Emails mit den vergebenen Codes können ausgeschlossen werden, oder es können auch Emails gesucht werden, die mehrere Suchwörter gleichzeitig enthalten müssen.

GmanDA ermöglicht zusätzlich auch eine visuelle Darstellung von Codes, in der das Auftreten bestimmter ausgewählter Codes auf einer Zeitachse dargestellt wird. GmanDA befindet sich aktiv in der Entwicklung, den aktuellen Stand kann man unter [5] finden.

Die folgende Darstellung zeigt die normale, zunächst voreingestellte Ansicht, mit den Emails in Baumstruktur (Document Tree), allen bereits vergebenen Codes (Codes), Dokumentansicht (Document View) sowie für das aktuelle Dokument vergebenen Codes (Coding View).

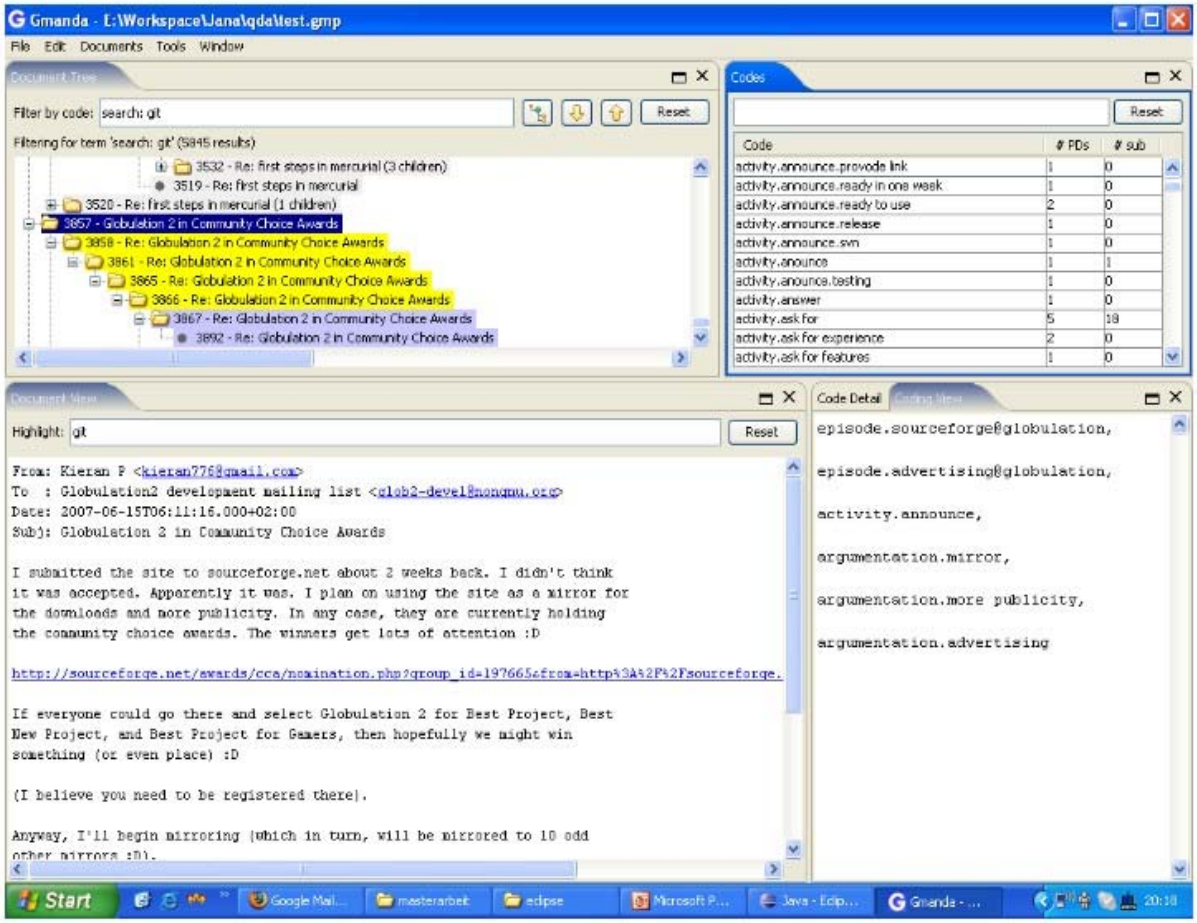

Abbildung 5: Screenshot: GmanDA in einer Version von November 2008

2.3 Datenbasis

Diese Arbeit beruht auf der Analyse von 16 Mailinglisten aus verschiedenen Open Source Projekten. Im Rahmen dieser Arbeit wurde hiervon eine Mailingliste komplett selber durchgearbeitet und kodiert und eine zweite nur mit Hilfe von Suchanfragen beschränkt auf das Thema SCM kodiert. Die weiteren 14 Mailinglisten waren bereits von Christopher Oezbek im Rahmen seiner Doktorarbeit vorkodiert. Es stellte sich heraus, dass eine solche Vorkodierung sehr nützlich ist um mittels Filteranfragen auf Episoden und Emails zu stoßen, welche für diese Arbeit relevante Inhalte enthalten. Trotz einer existierenden Kodierung durch eine andere Person muss jedoch immer ein großer Zeitaufwand für das Verständnis der Email und der Kodierung eingeplant werden. In [4] werden verschiedene Strategien zur Analyse von Prozessdaten vorgestellt und miteinander verglichen.

Von jeder Mailingliste wurden alle Emails des Jahres 2007 betrachtet. Des Weiteren wurden teilweise auch externe Daten aus den Homepages der Projekte oder Artikel, auf die in den Emails hingewiesen wurde, in die Untersuchung mit einbezogen, wenn die Emails nicht genügend Informationen enthielten.

Die Projekte sind sehr unterschiedlicher Art, bezogen auf Projektgröße, Mitgliederanzahl, Anwendungsdomäne und Leitungsstrukturen. Unter den Projekten finden sich auch ein akademisch betriebenes Open Source Projekt und zwei Projekte die von einer Firma geleitet und betrieben werden.
Deswegen werden im Folgenden nur die Projekte kurz vorgestellt, aus denen für die folgenden Resultate Erkenntnisse gewonnen werden konnten.

2.3.1 Globulation 2

Globulation 2 ist ein Echtzeit-Strategiespiel. An der Kernentwicklung sind nur relativ wenige Personen beteiligt. Geleitet wird das Projekt von zwei Maintainern, wobei der eine sich mehr um die Programmierung kümmert, der andere auch administrative Aufgaben übernimmt. Das Projekt hat im Zeitraum der Untersuchung das Versionsverwaltungssystem gewechselt und ist von einem zentralen System auf ein dezentrales umgestiegen. [5]

2.3.2 ROX

ROX (RISC OS on X) ist eine zu KDE oder Gnome vergleichbare Desktopumgebung, die der traditionellen Unix-Sicht Vorrang einräumt, alles als eine Datei anzusehen. Also werden zum Beispiel Programme nicht als Verweise aus einem Startmenü verlinkt, sondern durch können per Drag-and-Drop installiert und gelöscht werden. ROX ist entsprechend dieser Philosophie aus vielen unabhängigen Modulen aufgebaut. Der Kern des Projektes ist der Dateimanager ROX-Filer von dem Maintainer des Gesamtprojekts federführrend entwickelt wird. Im Betrachtungszeitraum 2007 wurde die Versionsverwaltung von ROX-Filer auf ein dezentrales System umgestellt, währrend viele der restlichen Module im existierenden zentralen System weiterverwaltet wurden. [7]

2.3.3 Grub

GRUB ist ein Bootloader für x86 PCs, der in den Versionen GRUB legacy und GRUB2 entwickelt wird. Das Projekt wird geleitet von einem Maintainer, der sich jedoch an der Kommunikation in den Mailinglisten nur begrenzt beteiligt und erst bei wichtigen Entscheidungen eingreift. Im Beobachtungszeitraum wurde die Migration weg vom existierenden Versionsverwaltungssystems CVS diskutiert, scheiterte jedoch am Widerstand des Maintainers. [8]

2.3.4 Flyspray

Flyspray ist ein web-basierter Bug-Tracker ähnlich dem bekannteren Bugzilla. Das Projekt wird geleitet durch zwei Hauptentwickler und über ein zentrales Versionsverwaltungssystem koordiniert. Da Flyspray eine Software mit der Zielgruppe Softwareentwickler und Administratoren von Softwareentwicklungsprojekten ist, treten diese Nutzer häufig auch auf der Entwicklermailingliste dieses Projektes auf um Fehler zu melden und Änderungswünsche zu äußern, können diese Äußerungen aber häufig auch mit Patches und detaillierten internen Fehlermeldungen untermauern. Dementsprechend verwischen in Flyspray die Grenzen zwischen Entwickler- und Nutzerkommunikation auf der Mailingliste. [9]

2.3.5 Das U-Boot

U-Boot ist ein Bootloader, der sich auf eingebettete Systeme spezialisiert hat und eine große Menge an Architekturen und Motherboards unterstützt. Das Projekt wird betrieben durch eine Firma, die sich auf eingebettete Systeme spezialisiert hat und vom Gründer der Firma als Maintainer geleitet. In 2007 wurde die Umstellung des Versionsverwaltungssystems von einem zentralen auf ein dezentrales System durch den Maintainer durchgeführt. Nach dieser Umstellung wurde auch der gesamte Entwicklungsprozess neu strukturiert. Ein sehr großer Teil der Kommunikation dieses Projektes läuft über die Mailingliste. Diese bietet daher einen tiefen Einblick in die Arbeitsprozesse, und war sehr hilfreich für diese Arbeit. [10]

2.3.6 KVM

KVM (Kernel based virtual machine) ist eine Virtualisierungslösung für Linux auf x86 Hardware. KVM ermöglicht es damit mehrere Betriebssysteminstanzen auf einer einzelnen Linuxinstallation laufen zu lassen. Das Projekt KVM wird durch ein israelisches Start-Up betrieben und durch einen Mitarbeiter von KVM geleitet. Im Beobach-

tungszeitraum wurde die existierende Versionsverwaltungssoftware von Subversion auf Git umgestellt. [11]

2.3.7 Xfce

Xfce (usprünglich für XForms Common Environment) ist eine Desktopumgebung vergleichbar mit KDE und Gnome, die versucht besonders klein und schnell zu sein. Das Projekt besteht aus einem Windowmanager, einer Desktopverwaltung, Schaltflächen für das starten von Anwendungen, dem Dateimanager Thunar und vielen weiteren Komponenten. [12]

2.3.8 MonetDB

MonetDB ist eine Open Source Datenbank für den Einsatz in Anwendungen mit besonderen Anforderungen bzgl. der Leistungsfähigkeit der Datenbank. MonetDB enthält neben der Unterstützung für SQL auch viele Erweiterungen zur Verwendung von XML im Backend und als Abfragesprache. Das Projekt ist ein akademisches Open Source Projekt der Arbeitsgruppe Datenbanken am Centrum Wiskunde & Informatica (CWI) in den Niederlanden. [13]

2.4 Vorgehen

Zunächst wurde das bereits von Christopher Oezbek vorkodierte Material begutachtet, um verstehen zu können, wie dieser die Emails kodiert hat, und die gewonnenen Erkenntnisse dann in die eigene Kodierung mit einzubringen.
Bei der Kodierung einer neuen Mailingliste stellte es sich als sehr problematisch heraus, dass versucht wurde, genau die gleichen Konzeptnamen zu verwenden, wie Christopher Oezbek, um eine einheitliche Konzeptsammlung zu erhalten. Dies hat den Fortschritt des Kodierens sehr gebremst, da immer wieder in den schon vorhandenen Konzepten nachgesehen werden musste, ob diese auf ein bestimmtes in den Emails auftretendes Phänomen passen könnte.
Im Nachhinein wäre es vielleicht besser gewesen, direkt eigene Konzepte zu benennen, da so zunächst mehr darauf geachtet wurde, welche vorhandenen Konzepte passen könnten, als andere interessante Konzepte in den Daten zu finden.

Gleichzeitig wurde im Internet nach Literatur über verschiedene SCMs gesucht, um sich in das Thema hereinzufinden, denn es waren keinerlei Vorkenntnisse, weder im praktischen noch im theoretischen vorhanden, was am Anfang der Arbeit ein großes Problem darstellte.

Im nächsten Schritt folgte die Filterung der Emails anhand von vergebenen Codes mit Hilfe der Filterfunktionen von GmanDA. So wurde nach allen vergebenen Codes, die in irgendeiner Art mit SCMs zu tun haben gefiltert, und alle Emails durchgearbeitet. Interessante Emails wurden mit entsprechenden Memos gekennzeichnet.
Während der ganzen Zeit wurde ein Stichwortkatalog geführt und immer mehr erweitert. In diesen Katalog wurden alle Stichwörter aufgenommen, die in irgendeiner Weise auf die Verwendung von SCMs hindeuten. Dies waren zunächst einfach Schlagwörter wie „commit“, „merge“ oder „patch“, später kamen dann aber noch viele verschiedene Wörter dazu, die auch auf spezielle SCMs hinwiesen. Nachdem dann alle Emails, die Anhand der Filterung über Codes gefunden werden konnten durchgearbeitet worden waren, hatte man schon ein Gefühl dafür entwickelt, welche Art von Emails es sich lohnt zu lesen, und welche nicht. Denn mitunter wurde versucht, Ar-

beitsprozesse innerhalb des Projektes nachzuvollziehen, die im Hinblick dieser Arbeit unwichtig sind, was sehr viel Zeit gekostet hat.

Im Folgenden wurden dann alle Emails gefiltert, die mindestens ein Wort aus dem Stichwortkatalog enthielten. Auch diese Emails wurden dann auf interessante Aspekte hin untersucht. Natürlich waren unter ihnen aber sehr viele Emails, die schon vorher betrachtet wurden. Dadurch wurden aber teilweise vergessene Dinge wieder entdeckt und konnten mit Hilfe eines Memos festgehalten werden.

Alle geschriebenen Memos wurden dann gesammelt und es wurde versucht, Beziehungen zwischen einzelnen Aspekten herzustellen, und diese in Form von Diagrammen nach dem Paradigmatischen Modell darzustellen. Das stellte sich aber als schwierig heraus, und so konnten zunächst nur die Emails aus verschiedenen Projekten gruppiert werden, die Unterschiede in gleichen Bereichen der Verwendung von SCMs aufwiesen um diese miteinander zu vergleichen.

Nur, im Bereich Migration konnte das paradigmatische Modell teilweise angewendet werden, denn in den Diskussionen um einen Wechsel kamen mehrere Aspekte zum tragen, die man miteinander in Beziehung setzen konnte.

Während der Beschreibung der Ergebnisse wurde sehr darauf geachtet, dass alle Aussagen anhand von Emails belegt wurden. Leider waren die Memos teilweise sehr schlecht geschrieben, so dass lange nach Emails gesucht werden musste um auch zu belegen, was man im Kopf hatte, aber in keiner ordentlichen Form in Memos oder Codes abgelegt hatte, um hinterher wieder schnell auf entsprechende Emails zugreifen zu können.

Leider wurde bei manchen Emails erst im Nachhinein klar, dass diese im späteren Verlauf noch wichtig werden würden. Ein vorheriges Kennzeichnen dieser Emails hätte eine große Zeiteinsparung bedeutet.

3 Forschungsergebnisse

Dadurch, dass die Projektmitglieder nicht ausschließlich über die öffentliche Mailingliste kommuniziert haben, waren viele Informationen nicht zugänglich. Auch wurde natürlich nicht über das verwendete SCM gesprochen, wenn es keine Probleme damit gab. Zudem werden die administrativen Aufgaben zum Beispiel eines Maintainers normalerweise nicht in einer öffentlichen Mailingliste besprochen. Interessant wurde es nur, wenn es darum ging, dass Projektmitglieder mit dem verwendeten SCM unzufrieden waren, es also wechseln wollten. Es kam auch vor, dass neue Projektmitglieder kurz in die Verwendung des SCM eingeführt wurden, aber insgesamt waren die Informationen, die zu Fragen der Nutzung und Administration von Versionsverwaltungswerkzeugen zur Verfügung standen sehr spärlich.

3.1 Migration – Wechsel des SCM

Im Jahr 2007 wurde in vielen Projekten ein Wechsel des SCM vorgeschlagen. Nicht wenige Projekte sind dann auch tatsächlich zu einem anderen System migriert. Ausschlaggebend für die Wahl des neuen SCM waren verschiedene Gründe. Wichtig waren unter anderem die Position beziehungsweise die Macht des Innovators (desjenigen, der den Wechsel vorschlägt und damit die Diskussion einleitet) innerhalb des Projektes, die Meinung des Projekt-Maintainers sowie sein Ansehen bei den Projektmitgliedern, die Erfahrung des Innovators sowie der Projektmitglieder mit verschiedenen SCMs, die Annahme des gewählten SCMs durch die Mitglieder, die Bereitschaft zu einem durch die Migration erzwungenen Hostwechsels, die Projektgröße sowie der Aufwand des Wechsels.

Die meisten Projekte, die 2007 das SCM gewechselt haben, sind von einem zentralen System wie CVS oder Subversion zu einem dezentralen wie Git oder Mercurial migriert. Eine wesentliche Rolle für die mehrfachen Migrationen von zentralen zu de-

zentralen SCMs im Jahr 2007 scheint dabei die fortschreitende Verbreitung von dezentralen SCMs zu sein. Die Entwickler eines Open Source Projektes sind oft in mehreren Projekten involviert, deshalb haben sie auch Erfahrungen mit unterschiedlichen SCMs. Wenige aber sind auch von CVS auf Subversion umgestiegen, haben also ein zentrales System beibehalten. Dies war zum Beispiel im Projekt Freedos der Fall (2824@freedos ff**).** Es kam bei den von mir untersuchten Projekten aber nicht vor, dass von einem dezentralen zu einem anderen dezentralen SCM oder von einem dezentralen zu einem zentralen System umgestiegen wurde.

Eine Migration des SCM wurde aus unterschiedlichen Gründen vorgeschlagen. Entweder, weil das verwendete SCM Probleme machte, weil andere Projekte auch zu einem anderen System migriert sind, oder aber die den Wechsel vorschlagende Person hat in anderen Projekten gute Erfahrung mit einem bestimmten SCM gemacht, welches sie dann dem Projekt vorschlägt.

Der Diskussionsverlauf war sehr unterschiedlich. Es gab Projekte, die innerhalb von einem Tag das SCM gewechselt haben, bei manchen dauerte es mehrere Monate, während es aber auch Projekte gab, die lange über die unterschiedlichsten SCMs diskutierten, aber am Ende dann doch nicht migrierten.

Die wichtigste Voraussetzung für eine erfolgreiche Migration ist aber natürlich, dass es überhaupt jemanden gibt, einen Innovator, der das verwendete SCM nicht mag oder aber aus irgendeinem anderen Grund für einen Wechsel plädiert und dies auch den anderen Projektmitgliedern unterbreitet.

3.1.1 Rolle des Innovators und des Maintainers

Die Rolle des Innovators und seine Position innerhalb des Projektes kann eine entscheidende Rolle bei der Migration des SCM spielen.

ROX

Speziell bei einem Projekt, nämlich bei ROX, war die erste Mail des Innovators schon sehr ausschlaggebend für die spätere erfolgreiche Migration. Ein wesentlicher Punkt

dabei war nicht nur, dass es der Maintainer selbst war, der den Wechsel vorschlug, sondern auch die Gestaltung der entsprechenden Email dazu. Diese war so gut strukturiert und durchdacht wie bei keinem anderen Projekt (9368@rox). In seiner Email machte er deutlich, dass es für alle Beteiligten das Beste wäre, das SCM zu wechseln, und hatte auch eine Lösung parat. In dieser Email erwähnte er auch, dass es ohne einen Wechsel schnell dazu kommen könne, dass Arbeiten der Entwickler möglicherweise vergessen und nicht ins Repository aufgenommen würden, da nur er als Maintainer Patches in das Repository committen könne. Er führte den Projektmitgliedern so seine Macht vor Augen und gab ihnen das Gefühl, durch eine Migration nicht mehr so eingeschränkt und von ihm abhängig zu sein. Er hätte auch einfach in einer Email die Migration direkt bekannt geben können. Wahrscheinlich hatte er die Email so formuliert, dass es wie ein Vorschlag für einen Wechsel aussah, um den Projektmitgliedern das Gefühl eines Mitbestimmungsrechtes zu geben und sie so nicht vor den Kopf zu stoßen. Insgeheim war er sich seiner Sache aber wahrscheinlich schon ziemlich sicher und führte die Migration auch sehr schnell durch, ohne lange auf etwaige Einwände seitens anderer zu warten. Schon nach kurzer Zeit hatte er aber auch Rückenstärkung von einzelnen Projektmitgliedern, die sich positiv zu seinem Vorschlag äußerten. Der Maintainer konnte sich auch deswegen ohne weiteres durchsetzen, weil er bereit war, die Migration selber durchzuführen, er war dabei nicht unbedingt auf fremde Hilfe angewiesen.

KVM

Im Projekt KVM gab es keinen ersichtlichen Innovator, da es auch keine öffentliche Diskussion zum Wechsel des SCMs gab. Der Maintainer informierte die Projektmitglieder in einer Email, sagte ihnen, wie sie jetzt an das Repository gelangen könnten, und erklärte kurz, aus welchen Gründen der Wechsel eines Teils von KVM von Subversion zu Git durchgeführt wurde. (1399@kvm) Auf diese Email wurd nicht geantwortet, die Entscheidung wurde stillschweigend akzeptiert. Die Meinung der Projektmitglieder war dem Maintainer in diesem Projekt aber grundsätzlich nicht gleichgültig, denn als er überlegte, die Repositorystruktur zu ändern, wendete er sich fragend an die anderen Projektmitglieder (1839@kvm).

Globulation 2

Auch in dem Projekt Globulation 2 war es der Maintainer, der als Innovator einen Wechsel des SCMs vorschlug (2606@globulation). Jedoch folgte hier ein langes Hin und Her, bis es schließlich zu einem Wechsel kam. Das lag zum einen daran, dass die Machtverhältnisse in diesem Projekt nicht so stark ausgeprägt sind, zum anderen schlug der Innovator auch nur einen Wechsel, nicht aber eine entsprechende Lösung vor. Bei Globulation 2 wird jeder, der sich an der Diskussion beteiligt, sehr ernst genommen, was auch daran liegt, dass man niemanden verlieren möchte, da das Projekt unter ständigem Entwicklermangel leidet. Wichtig scheint aber, dass auch der zweite Maintainer zustimmt, auch wenn seine Zustimmung erst eingeholt wurde, als der Wechsel von CVS zu Mercurial (dezentral) praktisch schon beschlossen und alle Aufgaben verteilt waren (3265@globulation). Man kann auch davon ausgehen, dass sich dieser Maintainer, hätte er einen Einwand gehabt, schon eher mit in die Diskussion eingebracht hätte. Dieser ist in dem Projekt auch fast ausschließlich für die Entwicklung und für keine administrativen Aufgaben zuständig.

Grub

Bei einem anderen Projekt, Grub, gab es im Jahr gleich zwei Anläufe, das SCM zu wechseln (3022@grub ff, 4057@grub ff). Dabei war der Innovator jeweils nicht der Maintainer, die Migration wurde aber von dem Maintainer verhindert. Er tat das nicht deswegen, weil er eine Migration grundsätzlich ausschließt, sondern weil es seiner Meinung nach noch keine passende Lösung gab. Er griff auch erst im letzten Moment ein, als ein Mitglied die Migration in Gang setzen wollte (4116@grub), in der ersten Diskussion meldete er sich gar nicht zu Wort.

Dadurch, dass er eine Migration nicht grundsätzlich ausschloss, sondern erst ein passendes System gefunden werden musste, entfachte er die Diskussion um die Wahl des SCMs erneut. Er fand immer neue Argumente gegen Git, und die Diskussion endete dann irgendwann, weil wahrscheinlich keiner mehr glaubte, den Maintainer von Git überzeugen zu können.

Man könnte annehmen, dass er sich von vornherein nicht für die Diskussion interessierte und deswegen keine Zeit in sie investierte, weil es für ihn klar war, dass er als Maintainer sowieso die Macht hat, eine Migration zu verhindern. Obwohl er in einer Email direkt angesprochen und nach seiner Meinung gefragt wurde, reagierte er nicht.

In diesem Projekt wäre der Maintainer als Innovator wahrscheinlich sehr schnell erfolgreich. Seine offensichtliche Macht und der grundsätzliche Wille zum Wechsel des SCM der Projektmitglieder sprechen dafür.

Ist der Innovator gleichzeitig der Maintainer des Projektes, so hat er es auf Grund seiner Position innerhalb des Projektes wesentlich einfacher, die Migration des SCMs zu einem anderen System durchzusetzen. Auch ist der Maintainer das Projektmitglied, das auf jeden Fall überzeugt werden muss, denn ansonsten hat er es in der Hand, die Migration zu verhindern.

Interessant wäre es, herauszufinden, ob sich der Diskussionsverlauf ändert, wenn der Innovator zunächst in einer privaten Email den Maintainer von einer Migration zu einem neuen System überzeugen würde und dieser sich dann selber als Innovator an die öffentliche Mailingliste wendet. Wenn der Maintainer den Vorschlag jedoch ablehnt, wird der Initiator (der mögliche Innovator) sich wahrscheinlich auch selber nicht mehr an die Mailingliste wenden, wo er vielleicht Zuspruch von anderen Projektmitgliedern gefunden hätte, mit denen er dann gemeinsam den Maintainer hätte überzeugen können.

3.1.2 Rolle anderer Projekte

Während der Diskussionen über eine Migration werden oft andere Projekte erwähnt, die das jeweils befürwortete System erfolgreich verwenden, um den Wechsel voranzutreiben. Wenn andere, besonders auch bekannte wichtige Projekte ein SCM erfolgreich verwenden, dann verlassen sich viele Mitglieder anderer Projekte darauf, dass es für ihr eigenes Projekt auch gut sein müsste. Das fällt zum Beispiel bei der Diskussion in dem Projekt Grub auf. In (4131@grub) erwähnt Projektmitglied O, dass in

fast allen Projekten, in denen er mitwirkt, Git verwendet wird. P wiederum befürwortet Git, da es seiner Meinung nach gut sein muss, wenn es so ein großes Projekt wie Linux verwendet (4134@grub). W weist auf Sun hin, die Mercurial verwenden, ohne jedoch weiter darauf einzugehen und Mercurial zu bewerten (4129@grub). Auch bei Rox zählt der Maintainer in seiner Email, in der er Git vorschlägt, mehrere Projekte auf, die auch Git einsetzen, um seinen Vorschlag zu untermauern. Bei dem Projekt KVM könnte man sich vorstellen, dass die Migration zu Git auch deswegen durchgeführt wurde, da Linus, der Git-Entwickler, regelmäßig von KVM pullt.
In dem Projekt Globulation 2 fiel auf, dass nur sehr selten auf externe Projekte hingewiesen wurde. Vielmehr tauchten in der Diskussion häufig Links auf (2541@globulation), die auf Artikel im Internet verweisen, welche verschiedene SCMs vergleichen.

3.1.3 Migrationserfahrungen

Wenn Mitglieder eines Projektes Erfahrungen mit SCM-Migrationen aus anderen Projekten haben, kann sich das sowohl positiv als auch negativ auf die Entscheidung zu einer Migration auswirken. Im Projekt Grub beendete der Maintainer auch deshalb die Diskussion über eine Migration und wendete damit einen Wechsel der SCM ab, weil er bereits Erfahrungen mit Migrationen gemacht hatte und deswegen wusste, auf was sich das Projekt einlassen würde. In (4132@grub) erklärte er, was für Probleme eine Migration mit sich bringen könne und dass er deswegen erst bereit sei zu wechseln, wenn ein SCM gefunden würde, dass allen seinen Anforderungen genüge. Er sei der Meinung, dass das alte System CVS die letzten 10 Jahre funktioniert habe und deswegen auch noch solange beibehalten werden könne, bis das richtige System gefunden werde, um CVS zu ersetzen.

3.1.4 Mitgliedergewinnung

Die Mitgliedergewinnung, bzw. ein möglicher Mitgliederverlust kann bei der Wahl eines neuen SCM eine große Rolle spielen. Die Mitglieder eines Projektes sind oft nicht bereit, die Arbeit mit einem SCM neu zu erlernen, wenn es sich dabei um ein unpopuläres System handelt. So bemerkt etwa ein Entwickler in der Diskussion über ein neues SCM, dass es sich nicht lohnen würde, SCMs wie Darcs oder Monotone zu erlernen, da sie anders als Git und Mercurial in wenigen Projekten verwendet würden (4134@grub). Der Maintainer meint sogar, dass es ihm wichtig sei, ein SCM schon benutzen zu können, an einem Projekt, welches Darcs verwenden würde nie mitarbeiten (4156@grub).

Besonders für Projekte, die an Mitgliedermangel leisten kann es negative Auswirkungen haben, wenn es zukünftige Mitglieder wegen des SCM schwer hätten, etwas zu dem Projekt beizutragen. So wurde zum Beispiel bei der der Umstellung des SCM im Projekt Globulation 2 sogar ein anonymer Commit ermöglicht, damit möglichst vielen Personen den Zugriff auf das Repository zu ermöglichen.

In den Projekten, in denen die Mitgliedergewinnung keine so große Rolle spielt, wird neuen Mitwirkenden solange der schreibende Zugriff auf das Repository gewährt, bis dieser sich mit mehreren entsprechend guten Beiträgen dieses Recht erarbeitet hat.

Das ist zum Beispiel im Projekt Xfce der Fall gewesen (13770@xfce). Ein neuer Entwickler hatte bereits zuvor ein Patch gesendet, welches aber nicht beachtet wurde. Nun wollte er Commit-Rechte beantragen, was er tat, indem er ein entsprechendes Formular des Projektes ausfüllte. Darauf erhielt er aber keine Antwort und wandte sich fragend an die Mailingliste. Ihm wurde erklärt, dass er nicht so einfach Commit-Rechte bekommen würde, und, dass es ein paar Wochen dauern könnte, bis sein Patch beachtet werden würde. Man könnte sich gut vorstellen, dass dieser Entwickler erst einmal abgeschreckt wird, denn er weiß auch erst in ein paar Wochen, ob sein Patch überhaupt akzeptiert werden würde.

3.1.5 Migration - Beispiele

3.1.5.1 Lösungsfindung - Grub

Wenn eine Diskussion um eine Migration des SCMs stattfindet, welche dann aber erfolgslos endet, kann das unterschiedlichen Gründe haben. Zum einen kommt es darauf an, wie viele Personen sich an der Diskussion beteiligen, zum andern, welche Personen sich an der Diskussion beteiligen. An der ersten Diskussion bei Grub (3022@grub ff) nahmen in insgesamt sechs Emails auch sechs unterschiedliche Personen teil. Und alle sechs Personen stimmten für einen Wechsel von CVS zu Git. Die Diskussion hatte nach sechs Tagen ein offenes Ende, denn direkt abgelehnt wurde der Vorschlag auch nicht. Selbst als fast einen Monat später ein Projektmitglied ein Skript zur Verfügung stellte, mit dem man CVS inklusive der gesamten Entwicklungsgeschichte in Git importieren kann, wurde darauf einfach nicht mehr reagiert. Es lag also nicht an mangelndem Zuspruch oder Wissen, wie die Migration durchgeführt werden könnte. Auch die Voraussetzungen wie z. B. ein Host, der Git unterstützt, waren erfüllt. Eine große Rolle spielte wahrscheinlich auch, dass sich der Maintainer nicht zu Wort gemeldet hatte, auch wenn er in einer Email direkt gefragt wurde, was er von einem Wechsel zu Git hält.

Am wichtigsten scheint aber zu sein, dass eine weitere Arbeit mit CVS keine großen Probleme erwarten lässt. Meistens wird ein Wechsel vorgeschlagen, weil man mit dem verwendeten SCM aus irgendwelchen Gründen unzufrieden ist. Auch bei Grub wurde in der diskussionseinleitenden Email erklärt, dass Git für die Arbeit in diesem Projekt gut geeignet sei. Es sei ein gutes Werkzeug, wenn es darum gehe, dass ein neues Feature längere Zeit brauche, bis es gut genug sei, um gemerget werden zu können. Git stelle eine gute Möglichkeit dar, die Entwickler bei dieser Arbeit zu unterstützen. Dabei ging der Innovator allerdings nicht näher darauf ein, wie genau diese Unterstützung aussehen würde.
Der Innovator der zweiten Diskussion bei Grub ein halbes Jahr später ging jedoch näher darauf ein (4067@grub) und erreichte dadurch auch, dass sich mehrere Projektmitglieder für das Thema interessierten, so dass an dieser Diskussion fast doppelt so viele Projektmitglieder teilnahmen. Die Diskussion wurde fast einen Monat aufrechterhalten, trotzdem scheiterte dann aber die Umstellung am Maintainer.

Im ersten Anlauf war die Migration nicht durchgeführt worden, weil sich nicht genügend Personen an der Diskussion beteiligt hatten und der Innovator nicht deutlich gemacht hatte, welche Vorteile ein neues SCM für das Projekt Grub hätte. Im zweiten Anlauf war es der Maintainer, der eine Migration verhinderte.

Fazit

Möchte ein Innovator also erfolgreich sein, sollte er bereits in der ersten Email klar hervorheben, welche Vorteile ein Wechsel des SCMs für das Projekt haben würde. Ist jedoch eine mächtige Person wie der Maintainer dennoch gegen einen Vorschlag und setzt sich durch, so kann auch eine solche Email nicht zu einer erfolgreichen Migration führen.

3.1.5.2 Lösungsfindung – Globulation 2

In dem Projekt Globulation 2, welches letztendlich erfolgreich von CVS nach Mercurial, einem dezentralen SCM, migrierte, verlief die Diskussion ganz anders als bei Grub.

Der Ausgangspunkt war, dass ein Teil des Codes neu geschrieben werden sollte und einfach aus dem Repository zunächst gelöscht werden sollte. Der Vorschlag, das SCM zu wechseln, wurde damit begründet, dass man mit einem anderen SCM als CVS besser mit Branches umgehen könnte (2518@globulation), die Neuschreibung des Codes also innerhalb eines Branches stattfinden und somit der Trunk mit dem alten Code bis zur Fertigstellung weiterhin bestehen könnte. Die Projektmitglieder waren sich einig, dass überhaupt gewechselt wurde, konnten sich aber lange nicht für ein SCM entscheiden. Es dauerte etwa 3 Monate, bis die Migration dann in Gang gesetzt wurde. Die Diskussion war ziemlich durcheinander und unstrukturiert und kann in zwei Teile unterteilt werden.

Im ersten Teil der Diskussion wurde über mehrere SCMs gesprochen, manche wurden aber nur kurz mit Namen erwähnt. Die Projektmitglieder hatten nicht besonders

viele Erfahrungen mit verschiedenen SCMs und konnten sich daher zum größten Teil nur über Artikel im Internet darüber informieren, welches SCM in Frage kommen könnte. Nur zwei Mitglieder von Globulation 2 hatten zu diesem Zeitpunkt längere Erfahrung mit Mercurial. Das war auch der Grund, wieso sich der Großteil der Diskussionsmitglieder relativ schnell auf Mercurial fixiert hat.

Das Problem war, dass der Host, auf dem das CVS-Repository von Globulation 2 lag, zu diesem Zeitpunkt kein Mercurial unterstützte, dies aber auf dessen TODO-Liste stand. Deshalb wurde darüber nachgedacht, vielleicht eine temporäre Lösung, wie zum Beispiel Subversion, zu finden, um dann, wenn der Host Mercurial unterstützen würde, endgültig zu diesem SCM zu migrieren. Die Diskussion endete abrupt mit einer Email, in der aufgelistet wurde, welche Möglichkeiten es gebe: eine temporäre Lösung, ein Hostwechsel, ein eigener Host oder aber beim alten SCM zu bleiben und sich damit herum zu schlagen (2663@globulation). Diese Email kam von dem Projektmitglied, welches sich mit Abstand am meisten an der Diskussion beteiligte. Das Mitglied gab aber an, nicht genügend Zeit zu haben, um sich weiter um eine Lösung zu kümmern.

Die Mitglieder dieses Projektes scheinen generell oft keine Zeit zu haben bzw. zu investieren, und oft tragen sie über einen längeren Zeitraum nichts bei, widmen sich dem Projekt dann aber wieder sehr intensiv.

Auch das Projektmitglied, welches die Diskussion zwei Monate später erneut entfachte, hatte längere Zeit nichts mehr zum Projekt beigetragen.
Es war die erste Email des Jahres 2007 (April) von diesem Innovator2. Wahrscheinlich las er sich während seiner inaktiven Zeit auch nicht alle Emails durch.
Es kann deshalb sein, dass er von der vorherigen Diskussion nichts mitbekommen hatte. Er wusste aber, dass eine Neustrukturierung des Codes stattfinden sollte. In dieser Email (3029@globulation) machte er eine Reihe von Vorschlägen, wie sich die Arbeit in dem Projekt verändern sollte. Unter anderem schlug er einen Hostwechsel vor. Er hatte bereits einen Account auf einem neuen Host eingerichtet, der neben der Unterstützung von Subversion auch ein Wiki sowie einen Bugtracker bietet. Gleichzeitig sollte das alte CVS-Repository beibehalten werden, dort sollten aber keine

neuen Features, sondern nur Fehlerbehebungen stattfinden, während man sich dann in dem Subversion-Repository auf die Neuschreibung verschiedener Teile des Codes konzentrieren könnte.

Der Innovator2 sah die Vorteile darin, alles zusammen auf einem Host zu haben, während es den anderen egal war, ob zum Beispiel der Bugtracker oder das Wiki woanders liegen als das Repository. Man sollte auch nicht die alten Systeme verwerfen, in die viel Zeit inverstiert wurde.

Innovator2 versuchte, seine Lösung durchzusetzen, indem er unbeirrt weiter an ihr arbeitete und Teile der Dokumentation bereits auf den neuen Host übertrug, auch wenn noch niemand anderes aus dem Projekt der Umstellung zugestimmt hatte. Er war der Meinung, dass die Hauptsache sei, ein anderes SCM als CVS zu wählen, er war außerdem der Meinung, dass Subversion gut sei, um das Problem mit CVS und den schlechten Merges zu lösen. Er kenne viele Projekte, die zu Subversion gewechselt seien (3036@globulation) und damit sehr zufrieden seien. Außerdem wäre die Umstellung auf Subversion für alle Beteiligten am einfachsten, da sich die Befehle kaum unterschieden.

Auch die beiden anderen Projektmitglieder, die sich an der Diskussion hauptsächlich beteiligten, gaben zu, dass Subversion eine Verbesserung zu CVS darstellen würde, sagten aber, dass, wenn man schon den Aufwand einer Migration auf sich nehme, man herausfinden solle, ob man nicht eine noch bessere Lösung als Subversion finden könne (3041@globulation). Andere Projekte seien nur zu Subversion gewechselt, weil sie sich gar nicht erst mit anderen SCMs auseinandergesetzt hätten.

Erst als sich ein Diskussionsteilnehmer mit direkten klaren Worten darüber äußerte, dass die Vorgehensweise des Innovators, alle Teile des Projektes auf einen Host übertragen zu wollen, für ein neu startendes Projekt eine gute Lösung sei, aber nicht für ein bereits bestehendes, und dass Subversion als reine Erweiterung von CVS genauso schlecht sei wie CVS, lenkte der Innovator ein. Er überließ den anderen die Entscheidung bei der Wahl eines neuen SCMs, bot aber weiterhin seine Hilfe bei der Umsetzung an (3057@globulation).

Wie auch in der ersten Diskussion lagen die Prioritäten bei Mercurial. Zunächst wurde aber kein Host gefunden, der dieses SCM unterstützt. Als man sich schon fast darauf geeinigt hatte, Git auf dem Host zu verwenden, auf dem auch das CVS-Repository liegt, meldete sich ein Projektmitglied, welches Probleme mit Windows bei der Verwendung von Git sieht. Er bot an, ein Mercurial-Repository auf seinem eigenen Server zu hosten. Diese Email beendete die Diksussion, die Entscheidung für Mercurial war damit gefallen.

Fazit

Die Diskussion bei diesem Projekt war sehr merkwürdig. Denn obwohl nur drei Personen maßgeblich daran teilgenommen hatten, war die Diskussion ziemlich durcheinander. Es wurde auf Emails reagiert, die trotz insgesamt kurzer Diskussionszeit (2 Tage) aufgrund der Masse von Emails schon veraltet waren. Deswegen kam man immer wieder auf Argumente zurück, die schon vorher abgelehnt bzw. entkräftet worden waren, aber dennoch wieder aufgegriffen wurden. Ein Diskussionsmitglied der zweiten Diskussion war auch dasjenige, welches sich bereits in der ersten am meisten eingebracht hatte. Wahrscheinlich freute es sich, dass endlich jemand mit Zeit und Engagement aufgetaucht war. Deswegen versuchte es auf ruhige Art und Weise, den anderen Diskussionsteilnehmer davon zu überzeugen, dass seine Lösung nicht die beste sei, um ihn nicht ganz abzuschrecken, denn dann hätte er nicht mehr auf seine Unterstützung bei der Umsetzung einer anderen Lösung zählen können. Bereits in der ersten Diskussion hatte das o. g. Diskussionsmitglied die Erfahrung gemacht, dass es an Projektmitgliedern fehlte, die Zeit oder Willen haben, zur Umstrukturierung des Projektes beizutragen. Was wahrscheinlich aber diese Diskussion maßgeblich beschleunigt hätte, wäre, wenn der Diskutant explizit nach einem Projektmitglied gefragt hätte, das bereit wäre oder die Möglichkeit hätte, ein Mercurial-Repository zu hosten. Denn es wurde den anderen Projektmitgliedern nicht klar, dass er sein bevorzugtes SCM Mercurial nicht mehr in die Diskussion mit einbrachte, weil er keinen entsprechenden nicht-kommerziellen Hoster dafür fand. Als dann von sich aus ein Mitglied seinen Server zur Verfügung stellen wollte, war die Entscheidung perfekt.

3.1.5.3 Umsetzung – Globulation 2

Bei Globulation 2 entschied man sich trotz des gewählten SCM ein Hauptrepository aufzusetzen. Zunächst überlegte man, wie man die bestehenden Accounts des Wikis übernehmen könnte, um den schreibenden Zugriff auf das Repository zu regeln. Es kam auch zwischendurch die Idee auf, jemanden zu bestimmen, der sich alleine um das Repository kümmert, der dann aus den Repositorys der Entwickler pullen sollten, denn das Projektmitglied, welches den Server zur Verfügung stellte, wollte aus Sicherheitsgründen nicht so vielen Personen den SSH-Zugriff auf seinen Server gewähren (3337@globulation). Aber es fand sich kein Projektmitglied, welches die Zeit dafür hätte aufbringen können, und außerdem wäre es nicht allen Mitgliedern möglich gewesen, ein eigenes Repository zum Pullen (3318@globulation) zu veröffentlichen.

Letztendlich entschied man sich dafür, jedem Kernentwickler einen Zugriff über ein globales Passwort zu gestatten, während andere Personen über einen anonymen Zugriff schreibend auf das Repository zugreifen können (3340@globulation). Die anonymen Commits könnten dann, wenn jemand von den Kernentwicklern Zeit findet, überprüft und gegebenenfalls rückgängig gemacht werden.
Zunächst wurde ungefähr zwei Wochen, nachdem das zweite Mal die Diskussion über einen Wechsel begann, zu Testzwecken ein read-only-Repository aufgesetzt, am gleichen Tag dann aber schon eines, in das auch committed werden konnte. Jeder, der Schreibrechte haben wollte, sollte sich an den Besitzer des Servers wenden.

Auch wenn sie am Ende vollständig im neuen Repository enthalten war, war es den meisten Globulation 2-Mitgliedern nicht besonders wichtig, die Entwicklungsgeschichte vollständig bei der Migration zu erhalten (3385@globulation). Der Code sollte sowieso größtenteils neu geschrieben werden und man hätte sonst das alte CVS-Repository in einem *Readonly*-Modus erhalten, wodurch man sich alte Revisionen bei Bedarf weiterhin hätte auschecken können.
Fazit

Die Umstellung von CVS nach Mercurial ging recht schnell vonstatten. Dadurch, dass es recht wenige Entwickler in diesem Projekt gibt, war es relativ einfach, den Schreibzugriff auf das Repository einzurichten. Wenn das Projekt größer gewesen wäre, dann hätte man sich wohl mehr Gedanken über die Administration des Servers machen müssen.

3.1.5.4 Adoption – Globulation 2

Fast alle Entwickler haben das neue SCM gut angenommen. Bis auf ein paar kleine Schwierigkeiten, bei denen sie schnell über die Mailingliste Hilfe bekamen (3628@globulation, 3676@globulation), gab es keine weiteren Probleme. Für sie hatte sich im Prinzip ja auch nicht sehr viel geändert. Sie hatten wie vorher Schreibrechte für das Repository und haben so weiter gearbeitet, wie vor der Migration auch.

Derjenige, der allerdings zunächst nichts in das Hauptrepository committed hatte, war der Entwickler, der den Kerncode neu schreiben wollte, weswegen man ursprünglich das SCM wechseln wollte. Zunächst bekam er gar nicht mit, dass er sich an den Besitzer des Servers wenden musste, um Schreibrechte zu bekommen. Dieser Entwickler verfolgt nicht so aktiv die Emails der Mailingliste wie die anderen Projektmitglieder: „Unfortently I have been deeply confused recently. Too much activity on the ML for my mind to follow. Twenty new threads?! “(3387@globulation)
Erst, als er in einer Email über den Fortlauf seiner Entwicklung berichtete, wurde er nochmals darauf hingewiesen. Zu diesem Zeitpunkt war er sich ziemlich unsicher, wie Mercurial funktioniert, obwohl er vorher zugestimmt hatte, als er das letzte Wort bei der Entscheidung zum Wechsel hatte (3265@globulation,3293@globulation), er hatte es im Prinzip aber doch verstanden. Ihm wurde nahegelegt, sich in Zukunft öfter online am Projekt zu beteiligen (3548@globulation), um das neue System ausnutzen zu können. Er war der Meinung, dass es für ihn aufgrund seiner langsamen Internetverbindung sehr lange dauern würde, das Mercurial-Repository herunterzuladen (3571@globulation), wurde aber dann eines besseren belehrt und verwendete später wenigstens lokal Mercurial (3671@globulation). Dadurch würde ein späterer Push, wenn er fertig sein würde, in mehrere Changesets unterteilt sein, den anderen Entwicklern also die einzelnen Entwicklungsphasen verdeutlichen und so nachvoll-

ziehbarer machen. Er wollte nicht mitten während der Arbeit ins Hauptrepository mergen, hatte aber nicht darüber nachgedacht, regelmäßig aus dem Hauptrepository zu pullen, um dann bei sich lokal zu mergen, um mögliche Schwierigkeiten möglichst früh zu entdecken. Nachdem er aber auch darauf hingewiesen wurde, tat er dies (3756@globulation).

Dass dieser Kernentwickler teilweise so wenig mitbekam, lag wahrscheinlich daran, dass er sich stark auf die Neuentwicklung des Kerncodes konzentrierte und nicht so oft wie die anderen an den Diskussionen in der Mailingliste teilnahm. So wunderte er sich zum Beispiel in 3387@globulation, wie viele neue Threads mittlerweile schon wieder entstanden waren.

Zu einem späteren Zeitpunkt, ließ er sich von einem anderen Entwickler dazu bringen, seine Änderungen doch ins Hauptrepository zu pushen, denn dieser wollte sich die Entwicklung ansehen. Damit sorgte er für Ärger, denn er hatte keinen neuen Branch eröffnet sondern in den Trunk gepusht, der stabil sein sollte (3759@globulation). Er hatte sich vorher nicht damit auseinandergesetzt, dass er hätte einen Branch eröffnen können, hatte dies weder in den Emails nachgelesen noch sich in einer Dokumentation darüber informiert. Wahrscheinlich wusste er nicht, dass dies möglich gewesen wäre. Die Entwickler lösten das Problem, indem ein weiterer erneut in den Trunk committet und somit erreicht, dass dieser Commit den Head darstellt, also bei einem einfachen Pull gewählt werden würde (3762@globulation). In einem zweiten Anlauf gelang es dann später noch, den Code in einem neuen Branch zu pushen (3780@globulation) und zu verstehen, wie Mercurial mit Branches umgeht (3772@globulation).

Fazit

Es hatte kein Mitglied des Projektes große Probleme, das neue SCM anzunehmen. Das lag auch daran, dass es ein Projektmitglied gab, das bei jeder Frage zur Benutzung schnell und ausführlich antwortete. Gerade aber der Entwickler, der die Entwicklung mit den meisten Auswirkungen in der Hand hatte, ging unvorsichtig mit dem neuen SCM um.

Das Beste wäre wahrscheinlich gewesen, wenn ein anderer Entwickler speziell vorab für ihn einen Branch angelegt hätte, ihm eine genaue Anleitung zur Nutzung gegeben hätte und ihn erst dann animiert hätte, das neue SCM zu verwenden. Auf der anderen Seite war es vielleicht auch gut, dass so ein Fehler passierte, denn so wissen nun alle, was in einem solchen Fall gemacht werden sollte und wie man es richtig macht.

3.1.5.5 Umsetzung – ROX

In diesem Projekt begann der Maintainer zunächst alleine, die Migration der ROX-Komponente ROX-Filer durchzuführen. Bereits weniger als 24 Stunden, nachdem er den Vorschlag zum Wechsel von Subversion zu Git gemacht hatte, bot er es den anderen über die Mailingliste zum Pullen und Ausprobieren an. Weniger als 48 Stunden später wurde das Subversion-Repository vom alten Host gelöscht. Dort wurde nur noch auf das Git-Repository verwiesen [14]. Der Maintainer hielt die Projektmitglieder über die Mailingliste über den Fortschritt der Migration und die auftretenden Probleme auf dem Laufenden (9373@rox). Es kam zu kleineren Problemen, die unter anderem auch noch von der ursprünglichen CVS-Migration nach Subversion herrührten, die aber gelöst wurden.

Ein zweites Projektmitglied schaltete sich ein, als es erfuhr, dass der Maintainer aus Zeitgründen zunächst keine Pläne hatte, auch andere Teile von ROX nach Git zu migrieren.
Der Maintainer war der Meinung, dass das die Struktur des Subversion-Repositorys es nicht zuliesse, für jeden ROX-Teil ein eigenes Git-Repository aufzusetzen. Der Maintainer hatte jedoch schon vorher darauf hingewiesen, wieso er nicht alle Teile in ein gemeinsames Git-Repository migrieren wollte:
Es sei einfach, Teile aus einem Subversion-Repository herauszufiltern, um sie in ein eigenes Repository zu tun, es sei jedoch schwer, sie in Subversion wieder in ein gemeinsames Repository zusammenzuführen, denn die Entwicklungsgeschichte einer der beiden zwischenzeitlich getrennt weiterentwickelten Teile würde verloren gehen. In Git allerdings sei es sehr einfach, zwei verschiedene Repositorys zu vereinigen, sie müssten nur einen gemeinsamen Ursprung in der Vergangenheit haben

(9373@rox). Der Maintainer möchte sich also vielleicht die guten Eigenschaften von Subversion zu Nutze machen, um möglicherweise später auf diejenigen von Git zugreifen zu können, denn wenn man zunächst in mehrere Teile splittet, kann man sie mit Git hinterher immer noch gut zusammenführen.

Das zweite Mitglied fing nun auch an, eigene Git-Repositorys aus dem Subversion-Repository zu erstellen. Genauso wie der Maintainer beschrieb er in der Mailingliste sein Vorgehen (9399@rox). Es stellte öffentliche Repositorys von ROX-Pager und ROX-Lib2 auf einem persönlichen Server zur Verfügung, welche zum jetzigen Zeitpunkt aber nicht mehr existieren [15]. ROX-Lib wurde inklusive ROX-Lib2 erst Anfang 2008 vom Maintainer nach Git migriert und auf dem gleichen Server (repo.or.cz) wie ROX-Filer bereitgestellt (9647@rox). ROX-Pager allerdings wird immer noch auf dem ursprünglichen über Subversion entwickelt.

Fazit

Bei ROX hat also nur eine partielle Migration stattgefunden. Zunächst hatte sich der Maintainer aus zeitlichen Gründen dafür entschieden, nur einen Teil des Projektes nach Git zu migrieren. Die anderen Komponenten schienen ihm wahrscheinlich nicht so wichtig, denn seine Gründe für die Migration waren, dass in einem dezentralen System keine Patches verloren gehen, wenn jeder sein eigenes Repository aufsetzt. Mittlerweile sind auch andere Teile von ROX nach Git migriert bzw. neue Teile sind direkt in einem neuen Git-Repository entstanden.

Die Komponenten, die noch unter Subversion entwickelt werden, sind nicht so groß und umfangreich, Beispiel ROX-Filer. Dort wird nicht soviel entwickelt, so dass diejenigen, die schreibend auf das Repository zugreifen können, keine Angst haben, es könnte etwas in einem Patch verloren gehen.

Ein dezentrales System macht also dann Sinn, wenn es einerseits viele Entwickler gibt, die etwas zu einem Projekt beitragen wollen, andererseits der Maintainer aber nicht möchte, dass zu viele Entwickler schreibenden Zugriff auf sein Repository haben.

Anders als zum Beispiel bei Globulation 2 musste man sich keine großen Gedanken über die Einrichtung von Accounts machen, um den schreibenden Zugriff auf ein - trotz gewähltem dezentralen SCM - zentrales verwendetes Repository zu regeln.

3.1.5.6 Adoption – ROX

In diesem Projekt wird Git als dezentrales System auch dezentral angewandt. Die Mitglieder nahmen das neue System sehr gut an und setzten relativ schnell ihre eigenen Repositorys auf, auf die sie hinweisen, wenn sie etwas zum Pullen anbieten (z.B.:9519@rox). Die Mitglieder sind teilweise schon deswegen froh darüber, dass auf Git umgestiegen wurde, da sie nun gezwungen wurden, Git zu erlernen, was sie sowieso schon vorhatten, aber bisher noch nicht in Anlauf genommen hatten (9369@rox). Der Maintainer sorgte aber mit einigen Hilfestellungen und Anleitungen, wo und wie man sich ein eigenes Repository aufsetzen kann, auch dafür, dass es keine Probleme dabei gibt (9384@rox). Selbst ein Entwickler, der noch nicht einmal von Subversion zu CVS gewechselt war, konnte Problemlos innerhalb einer halben Stunde ein Repository zum Pullen bereitstellen (9424@rox). Die Adoption geht sogar soweit, dass ein Entwickler sogar auf ein Repository hinweist (9499@rox), welches ein Applet enthält, das für einen ROX-Teil entwickelt wurde, der selber noch unter Subversion entwickelt wird.

Fazit

Die Adoption hat bei ROX keine großen Probleme bereitet. Das könnte daran liegen, dass die Entwickler erfahrener sind als die Entwickler von Globulation 2. Viele von ihnen hatten bereits Erfahrung mit Git. Außerdem konnten keine Probleme wie bei Globulation 2 entstehen, z. B. dass unstabiler Code in den Trunk gepusht wird, weil jeder Entwickler für sein eigenes Repository verantwortlich ist und kein anderer dort hereinpushen kann. In diesem Projekt herrscht – wenigstens für die Teile, die nach Git migriert sind – ein reiner Pullbetrieb.

3.1.6 Nebeneffekt: Umstellung des Entwicklungsprozesses

Speziell in dem Projekt U-Boot konnte man sehen, was eine Migration zu einem dezentralen System für weitere Auswirkungen haben kann. Dieses Projekt hatte schon vorher vor, den Entwicklungsprozess zu verändern, und nahm nun die Migration zu einem dezentralen SCM zum Anlass, die Entwicklung nach dem Vorbild von Linux zu verändern.
Diese Struktur sieht es vor, für verschiedene Teile des Projektes Betreuer, so genannte Custodians zu bestimmen. Für jedes dieser Projektmodule wird ein eigenes Repository aufgesetzt, welches der Custodian selber administriert, und für das er allein verantwortlich ist. Dadurch soll die Arbeit aufgeteilt werden, die durch den PatchBetrieb entsteht. Jeder Custodian ist dafür verantwortlich, die Patches, die für den von ihm betreuten Teil des Projektes bestimmt sind zu begutachten und dann entweder aufzunehmen oder abzulehnen. Der Maintainer von U-Boot, der die Umstellung des Entwicklungsprozesses vorschlägt möchte aber nur Projektmitglieder zu Custodians ernennen, die entsprechend Zeit für diese Aufgaben aufwenden können. Er schreibt in (25853@uboot): „[…] it is unfortunately necessary, that you are able to commit to a certain amount of time and willingness to discuss topics on the mailing list for this maintenance “ Aus den Repositorys der Custodians soll dann vom Maintainer in das „offizielle“ U-Boot-Repository gepullt werden.

In 25853@uboot erklärt der Maintainer des Projektes sein Vorhaben, und macht auch schon Vorschläge für Custodians von verschiedenen Modulen. Gleichzeitig bietet er auch eine Host als Lösung an, falls die Custodians selber nicht in der Lage sind ihr Repository selber zu hosten. Die vorgeschlagenen Personen fühlen sich geehrt, mit einer solchen Aufgabe bedacht zu worden, und für alle anderen Teile werden auch freiwillige Custodians gefunden (26247@uboot), die sich freiwillig melden (25979@uboot). Man einigt sich darauf, dass alle Repositorys der Custodians auf dem gleichen Server liegen sollen. Denn so können alle jederzeit aus allen Repositorys pullen, falls die Entwicklungen aus den Repositorys der Custodians mal verspätet von dem Maintainer in das offizielle Repository gepullt werden würden (25990@uboot).

Es gab keine Projektmitglieder, die dieses Vorhaben ablehnten. „The reaction was all positiv.“ (26247@uboot) Aber ein Projektmitglied hatte zum Beispiel Angst, dass die Custodians möglicherweise ihre Macht ausnutzen könnten, und eigene Patches in ihr Repository pushen könnten, ohne die anderen Projektmitglieder darüber in der Mailingliste zu informieren. (25995@uboot) Der Maintainer klärt ihn daraufhin darüber auf, dass es auch schon vorher vorkam, dass Commits stattfanden, die nicht über die Mailingliste bekanntgegeben wurden, und, dass das wohl auch nicht ganz verhindert werden könnte. Der angestrebte Ablauf sollte aber sein, dass jedes Patch auch an die Mailingliste gesendet wird.

Der Maintainer übernahm die gesamte administrative Arbeit, in dem er die Repositorys für die verschiedenen Custodians aufsetzte und mit Hilfe der ihm zugesandten öffentlichen SSH-Schlüsseln den entsprechenden Projektmitgliedern Schreibrechte auf deren Repositorys erteilte. Jedes Repository stellte ein Fork des offiziellen Repositorys dar.
Der Maintainer setzt auch ein zusätzliches „u-boot-testing“-Repository auf, in dem alle Patches automatisch landen, die von keinem Custodian aufgenommen wurden, zum Beispiel, aus dem Grund, weil sie keinem Bereich eines Custodian klar zugeordnet werden konnten (27021@uboot). Dieses Repository wird später auch verwendet für Änderungen, die ein Modul betreffen, für das es noch kein offizielles eigenes Repository gibt. (27841@uboot)
In das offizielle Repository möchte der Maintainer zunächst noch manuell pullen, da er erst herausfinden will, wie die Zusammenarbeit mit den Custodians funktioniert: „As for the custodian trees, I think I'm going to do the merging manually for some time. I need to get a feeling first how good the resonance between me and the respective custodian is ...“ (27021@uboot)

Die Custodians nehmen ihre Arbeit auf, und werden dabei von dem Maintainer unterstützt, bei Laune gehalten (26841@uboot) aber auch instruiert, falls sie etwas falsch gemacht haben (26911@uboot). Die kommunikation zwischen den Custodians und dem Maintainer verläuft zunächst nicht nach seinen Wünschen, er vermisst Pull-Anfragen ihrerseits. Deswegen kommt es auch einmal dazu, dass jemand anderes als der Custodian den Maintainer zum Pullen aus einem Repository auffordert, der

das entsprechende Repository getestet hat, und der Meinung ist, dass es mit dem offiziellen Repository zusammengeführt werden könnte (27765@uboot). Der Maintainer kommt dem Custodian somit dem Pullen zuvor, der eigentlich um einen Aufschub gebeten hatte (2773@uboot).

Die Repositorys der Custodians sollten laut Maintainer zunächst sehr einfach und ohne Branches gehalten werden, da der es selbst unter den Custodians Mitglieder gibt, die sich noch nicht so gut mit Git auskennen (27564@uboot). Außerdem möchte der Maintainer die Endnutzer nicht überfordern, denn sie müssten auch erstmal mit der neuen Situation zurechtkommen. Er selber scheint aber auch Angst zu haben, möglicherweise von einem falschen Branch eines Custodian in das offizielle Repository zu pullen, er möchte immer nur von dem Trunk der Custodians pullen (27584@uboot).
Die Custodians sind aber der Meinung, dass es angebracht wäre, wenn der Trunk die aktuellen Entwicklungen enthalten würden, und für den Maintainer ein Branch eingerichtet werden sollte, aus dem er fertig gestellten stabilen Code in sein offizielles Repository pullen könnte. Das würde es einfacher machen, Tester zu erreichen, die aus dem Trunk pullen könnten. Denn die Tester kennen sich nicht unbedingt mit Branchs aus, dem Maintainer würde es wesentlich einfacher fallen, aus einem solchen zu pullen. (27679@uboot).
Dennoch gilt auch jetzt noch die Konvention, dass ein Custodian den Trunk für das Pullen vom Maintainer bereithält, dieser also stabil gehalten wird. [10]

Fazit

Die Umstellung des Entwicklungsprozesses in der Art, wie sie in dem Projekt Uboot geschehen ist, ist eine auf der einen Seite eine sehr große Entlastung für den Maintainer, wenn dieser vorher alleine dafür verantwortlich war, zu entscheiden, welche Patches aufgenommen werden sollten. Auf der anderen Seite müssen erstmal Personen gefunden werden, denen der Maintainer vertrauen kann. Bei Uboot ging das relativ einfach, denn er selber hat die meisten Custodians vorgeschlagen, sich also Projektmitglieder herausgesucht, die sich bereits stark an der Entwicklung des ent-

sprechenden Moduls beteiligt hatten. Er kannte ihre Arbeit bereits, eine Vertrauensbasis war schon geschaffen. Dennoch wollte er aber nichts riskieren und automatisch aus den Repositorys der Custodians pullen um sich ein gewisses Maß an Sicherheit zu bewahren.

Wenn man den Entwicklungsprozess derart strukturiert wie in diesem Projekt, ist aber darauf zu achten, dass bestimmte Konventionen eingehalten werden. In (33785@uboot ff) forderte zum Beispiel ein Entwickler zum Pullen aus seinem persönlichen Repository auf. Er verwendete im Betreff der Email „Please Pull“, so dass der Maintainer in dem Glauben war, es handele sich um einen Custodian. Wenn das „Merge-Window“ des Maintainers zu diesem Zeitpunkt offen gewesen wäre, hätte dieser wahrscheinlich ohne darüber nachzudenken von diesem Entwickler zu pullen.

Es sind also klare Regeln aufzustellen, die die Custodians aber auch andere Projektmitglieder einhalten müssen, damit das System funktioniert. Der Maintainer muss genau wissen, welcher Branch im Repository des Custodians für ihn bestimmt ist, der Custodian muss den Maintainer außerdem in Kenntnis setzen, sobald dieser etwas Neues pullen kann. Die anderen Entwickler sollten sich aus dieser Prozedur heraushalten, um nicht für Verwirrung zu sorgen. Um diesen aus dem Weg zu gehen sollte der Maintainer jederzeit wissen, welche Personen Custodians sind. Es ist auch sinnvoll, dass die Custodians wie in [10] beschrieben, dass jeder Custodian in einem Branch die Entwicklung des offiziellen Repositorys mit verfolgt, und regelmäßig lokal mit seinem eigenen Trunk zusammenführt, um Fehler schon im Vorfeld zu beheben, die entstehen würden, wenn der Maintainer von ihnen pullen würde.

Eine Entwicklungsstruktur in dieser Art ist natürlich nur ohne weiteres möglich, wenn sich das Projekt in verschiedene Module unterteilen lässt.

3.2 Repository – Hosting

Das Hosting von Repositorys stellt normalerweise kein großes Problem in Open Source Projekten dar. Es gibt mittlerweile mehrere öffentliche Hoster, wie Sourceforge [16], deren Service man in Anspruch nehmen kann und die auch die Verwendungen von unterschiedlichen SCMs unterstützen.
Am einfachsten ist es wohl, wenn man ein dezentrales SCM im Pullbetrieb verwendet, da dann keine großen administrativen Aufgaben, wie die Einrichtung von Accounts für ein zentrales Repository anfallen. Auch aus diesem Grund ist der Umstieg von ROX-Filer auf ein dezentrales System so glatt gelaufen.

Auch das Projekt Globulation 2 ist auf ein dezentrales System umgestiegen. Das gewünschte SCM wurde aber zu dem damaligen Zeitpunkt von keinem öffentlichen Hosting Service unterstützt und nicht alle Entwickler hatten die Ressourcen privat ein Repository zum Pullen zu veröffentlichen. Deshalb war es in diesem Projekt nur möglich, das SCM umzustellen, weil ein Projektmitglied seinen eigenen Server zur Verfügung stellte.

Wenn aber Open Source Projekte ihre Repositorys oder andere Projektteile wie Bug-Tracker auf verschiedenen Hosts lagern, so kann das die Validität von Studien über Open Source Projekte beeinflussen so wird [17] beschrieben:“It is very difficult, without doing detailed manual examination of each project, to know exactly how each project is using its repository tools.“ Denn Open Source Projekte hosten ihre Repositorys zum Teil zum Beispiel außerhalb Sourceforge, sind aber dort dennoch registriert, um zum Beispiel die Mailingliste zu verwenden. Unter Umständen findet man in dieser Registration dann aber nur veraltete Daten. Das würde die Validität von Studien beeinflussen, die auf solchen Daten beruhen.
Das Projekt Freedos zum Beispiel hat ein Subversion-Repository bei Sourceforge, ohne zu dem Zeitpunkt verwendet zu werden: „If we do decide to migrate to Subversion, we should use the SVN repository on SourceForge (it's not activated yet, but it's there to be used.) “ (4826@freedos)

3.3 Repository – Strukturen

In den verschiedenen Projekten sind unterschiedliche Repository-Strukturen aufgefallen. Das hängt natürlich zum einen von dem verwendeten SCM, aber auch von der Projektgröße und anderen Faktoren ab.

3.3.1 Einfache Repositorys

Die einfachste Repositorystruktur ergibt sich, wenn ein zentrales SCM verwendet wird, mit einem einzigen Branch, wie es bei Globulation 2 bis zur Migration vom zentralen SCM CVS zum verteilten Mercurial war. Bei Verwendung dieser Struktur muss allerdings klar geregelt werden, wie man sich bei einem bevorstehenden Release verhält. Bei Globulation 2 wurde kurz vor der Migration noch ein Release gemacht, und alle Entwickler waren angehalten, nur noch Bugfixes zu comitten. Diese Art der Repositoryhaltung ist nur möglich, wenn das Projekt nicht besonders groß ist und auch nicht besonders schnell weiter entwickelt wird. Deswegen war diese Struktur auch bei Globulation 2 nur so lange möglich, bis es darum ging, Kernteile des Quellcodes von Grund auf zu erneuern. Wäre das Projekt bei der alten Arbeitsweise geblieben, dann hätten die Entwickler, die nicht an der Grunderneuerung beteiligt gewesen wären, nichts mehr beitragen können. Andernfalls hätte die Grunderneuerung nicht unter Revisionskontrolle gestanden, und es hätte große Probleme gegeben, wenn die gesamte Erneuerung innerhalb eines einzigen Patches ins Repository committed worden wäre.

Die meisten Projekte unterhalten aber normalerweise verschiedene Branches, in denen unterschiedliche Teile des Projektes bearbeitet werden, die erst mit dem Trunk zusammengeführt werden, wenn sie fertig entwickelt wurden, also stabil funktionieren.

3.3.2 Modulstruktur

Viele Projekte, die aus Modulen bestehen, die unabhängig voneinander entwickelt werden können, lagern diese in Branches. Jedes Modul bekommt einen eigenen Branch zugeordnet. Und wenn ein neues Modul hinzukommt, wird dafür ein neuer Branch geöffnet (13099@xfce). Diese Repository-Struktur lässt sich gut betreuen, da für jeden Branch ein eigener Maintainer bestimmt werden kann, der für diesen verantwortlich ist. Ein Maintainer des Gesamtprojektes kann somit Verantwortung abgeben. Indem er diesem die Rechte für andere Branches verwehrt, würde er dabei trotzdem ein hohes Maß an Sicherheit bewahren. Die Entwickler des Projektes können sich dann direkt an den Maintainer des Moduls wenden, zu dem sie beitragen wollen, und Patches gehen nicht so leicht unter. (Vgl. Kap. 3.1.6)

3.3.3 Externe Repositorys

Es gibt allerdings auch Projekte, die ein zentrales SCM verwenden, die aber zusätzlich für ihr Projekt noch Bibliotheken verwenden, die in entfernten Reporitorys liegen. Dies hat zum Beispiel bei Flyspray für Schwierigkeiten gesorgt. Ein Entwickler hatte versucht sich ein Update aus dem Repository auf seine Working Copy zu machen. Dies endete mit einem Fehler, da bei einem Update aus Flyspray automatisch auch ein Update aus den Repositorys der entsprechenden Bibliotheken gemacht wurde. Diese schienen aber zu diesem Zeitpunkt nicht erreichbar. Wenn man nur ein Update für den Code eines auf diese Weise auf mehreren Repositorys verteilten Projektes machen möchte, kann man das Problem umgehen (bei Subversion z. B. mit dem Flag „ignore-externals“). Wenn man allerdings den Code das erste Mal auscheckt, also noch keine Working Copy hat, dann muss man warten, bis die Bibliotheken wieder erreichbar sind. Hat das Projekt selber keinen Einfluss darauf, da es sich um Bibliotheken von Fremdprojekten handelt, kann dies natürlich möglicherweise zu größeren Problemen führen.

3.3.4 Gemischte Repositorys

Findet wie bei ROX eine partielle Migration von Teilen des Projektes statt, dann findet man hinterher eine Repository-Struktur wieder, die sich aus mehreren Repositorys zusammensetzt, welche sich unter der Kontrolle von verschiedenen SCMs befinden. Bei ROX ist das bis zu dem jetzigen Zeitpunkt zusätzlich ein Gemisch aus zentralen und verteilten SCMs.
Diese Struktur kann verschiedene Konsequenzen haben. Die Entwickler müssen sich mit unterschiedlichen SCMs auseinandersetzen, wenn sie an verschiedenen Teilen aus solchen unterschiedlichen Repositorys arbeiten. Für die ROX-Entwickler waren innerhalb der Mailingliste keine ersichtlichen Probleme zu finden. Man könnte sich jedoch vorstellen, dass Benutzer, die kein Release verwenden wollen, sondern den aktuellen Entwicklungsstand testen möchten, Probleme bei dem Zusammensuchen der einzelnen Teile aus den verschiedenen Repositorys haben könnten.

3.4 Betriebsmodi

Ein Betriebsmodus ist die Art und Weise, wie ein SCM in einem Projekt eingesetzt wird. Je nachdem, ob das Projekt ein zentrales oder ein verteiltes SCM verwendet, gibt es unterschiedliche Möglichkeiten.

3.4.1 Patch-Betrieb

Wird ein zentrales Repository verwendet, dann gibt es die Möglichkeit, dass es einen Maintainer gibt, der als einziger schreibenden Zugriff auf das Repository hat, wie es dem Repository ROX-Filer vor der Migration zu einem verteilten SCM der Fall war. Möchten nun andere Mitglieder etwas zu diesem Repository beisteuern, bleibt ihnen keine andere Wahl, als ihre Änderungen in einem Patch zusammenzufassen und es per Email an die Mailingliste oder direkt an den Maintainer zu senden und zu hoffen, dass dieser es akzeptiert und es dann ins Repository committet. Der Maintainer muss das Patch zunächst testen, es möglicherweise modifizieren, um es dann zusammen mit einer Log-Message ins Repository zu übertragen. Diesen Arbeitszyclus beschreibt der Maintainer anschaulich in seiner Email, in der er vorschlägt, ROX-Filer nach Git zu migrieren (9368@rox). Wenn der Maintainer nun keine Zeit hat, kann es unter Umständen lange dauern, bis das Patch ins Repository gelangt.

Das Projekt Xfce ist in mehrere Unterprojekte unterteilt, welche jeweils über einen Maintainer betreut werden. Auch in diesem Projekt sind die Maintainer sehr beschäftigt. Ein Maintainer gibt an, er werde „in ein paar Wochen“ nach einem Patch schauen, dessen Autor sich zuvor beschwert hatte, dass niemand auf ein Patch reagiert hatte und deswegen Commit-Rechte beantragt hatte, die ihm allerdings erst einmal verwehrt worden sind (13777@xfce).

Der Maintainer versucht das Problem, dass Patches in einer Menge von Emails untergehen könnten, so zu lösen, dass sein Email-Client die Patches mit Betreff „[Patch]“, immer ganz oben auflistet. Das würde das zeitliche Problem allerdings auch nicht lösen (888@kvm).

Bei einem Patch-Betrieb ist außerdem zu beachten, dass immer nur der als Autor in dem Repository angegeben wird, der das Patch am Ende committet hat. der eigentliche Patch-Autor kommt hierdurch höchstens namentlich in der Commitnachricht zur Geltung aber nicht in den Metadaten des Commits. Dies ist unter anderem dann problematisch, wenn diese Metadaten für computergestützte Analysen der Teilnahme in dem Projekt genutzt werden und somit die interne Validität der Studie beeinträchtigen können. So merkt Stefan Koch in [18] bezogen auf die Konstruktvalidität an: „In some projects, people could be contributing code without relevant account, which sometimes is only granted to longtime participants by sending it to one of those persons who then does the actual commit.“.

Ein Patch-Betrieb wäre durchaus auch denkbar bei Projekten, die ein verteiltes SCM verwenden. Denn viele Projekte, wie zum Beispiel Globulation 2, wechseln auch aus dem Grund zu einem verteilten SCM, weil mit ihnen das Mergen meist besser gelingt als mit einem zentralen System (3056@globulation).

Einen Patch-Betrieb betreiben Projekte, die sehr auf Sicherheit achten. Das Sicherheitsrisiko wird natürlich umso größer, je mehr Entwickler schreibenden Zugriff auf das Projekt-Repository haben. Jedes Projekt muss dabei selber abwägen, ob es wichtiger ist, auf die Sicherheit zu achten oder aber Gefahr zu laufen, dass vielleicht wichtige Patches in einer Masse von gesendeten Patches untergehen.

3.4.2 Commit-Betrieb

In einem Open source Projekt findet selten ein reiner Commit-Betrieb statt. Denn das würde bedeuten, dass alle Mitwirkenden einen schreibenden Zugriff auf das Projekt-Repository haben. Oft tragen aber auch Personen etwas zum Projekt bei, die eigentlich nur normale Nutzer sind und Fehler gefunden haben, die sie selber beheben konnten. Diesen Personen, die nur sporadisch etwas zu einem Projekt beitragen, erst Commit-Rechte geben zu müssen, wäre äußerst umständlich und macht auch keinen Sinn, da eigentlich jedes Open Source-Projekt eine öffentliche Mailingliste hat, an die man sich sehr einfach mit einem Patch wenden kann.

Der Commit-Betrieb bedeutet außerdem, dass die Projektleitung die Kontrolle über die Commits ein Stück weit aus der Hand gibt. Jeder der Rechte hat, kann committen und nur mittels einer Rückgängigmachenstrategie kann dann das Projekt manches Fehlverhalten verhindern. So weist der Maintainer des Projektes MonetDb die Entwickler darauf hin, dass ab sofort Bug-Fixes nur noch in den Release-Branch, neue Features in den Trunk committet werden sollten und dass „Any violation of these rules might be „punished“ by a forced undo of the respective changes“ (54@monetdb).

Wird im Commit-Betrieb entwickelt, dann kann es sehr von Vorteil sein, wenn eine bei einem Commit automatisch generierte Email mit Informationen zu den Änderungen an die Mailingliste gesendet wird. Dies geschieht zum Beispiel in dem Projekt Flyspray. Die Email i den Autor, das Datum, die Revisionsnummer und die Commit-Nachricht des Autors auf. Außerdem wird noch darauf hingewiesen, mit welchem Subversion-Befehl man das entsprechende Changeset erhält (4366@flyspray).

In Projekten, die diese automatisch generierten Emails nicht verwenden, kann es vorkommen, dass Projektmitglieder nicht mitbekommen, wenn etwas ins Repository committet wurde. Im Projekt Globulation ist ein Projektmitglied der Überzeugung, dass es (zu dem damaligen Zeitpunkt) nur einen aktiven Entwickler gab, woraufhin ihn jemand auf die Commit-Logs verweist, um sich davon zu überzeugen, dass dem nicht so ist: „You can check the commit logs to convince yourself“ (3761@globulation). In dieser Diskussion ging es darum, dass versehentlich massive Änderungen in den Trunk committet wurden anstatt in einen Branch. Das kann zum Beispiel auf andere Commits, die in der Zwischenzeit gemacht wurden, Auswirkung haben, wenn der falsche Commit rückgängig gemacht werden soll. Die Entwickler können zwar jederzeit im Repository nachsehen, und sich über neue Commits informieren, was jedoch aufwändiger ist, als wenn man alles in Emails nachlesen kann.

3.4.3 Pull-Betrieb

Ein Pullbetrieb kann verwendet werden, wenn ein dezentrales SCM verwendet wird. Ein reiner Pullbetrieb sähe so aus, dass es kein zentrales Repository gibt, sondern jedes Projektmitglied ein eigenes Repository unterhält und sich nur die Änderungen aus dem entfernten Repository eines anderen Projektmitgliedes holt, die es für gut

hält. So kann jeder Besitzer eines Repositorys die Entwicklung innerhalb dessen selber steuern und entscheiden, in welche Richtung sie verläuft. Der Maintainer merkt im Projekt ROX zum Beispiel an, dass die Entwickler sich bei der Verwendung eines dezentralen SCM aussuchen können, von welchem Repository sie pullen möchten: „My Copy of the repository isn't spezial, so if you think I'm taking too long, just use someone else's copy as the master instead“ (9368@rox).

Möchte ein Entwickler, dass andere aus seinem Repository pullen, dann sendet er üblicherweise eine Pull-Aufforderung an die Mailingliste. Dort beschreibt er, welche Änderungen er gemacht hat unter welcher URL man sein Repository finden kann. Wichtig ist hierbei aber, dass man auch den entsprechenden Branch mit angibt, solange das nicht der Trunk ist, damit ein reibungsloser Ablauf garantiert werden kann. In dem Projekt KVM wird Linus regelmäßig aufgefordert, aus dem Repository des Maintainers zu pullen. Linus möchte die Aufforderung in einer bestimmten Form bekommen, damit er ohne sich Gedanken zu machen pullen kann. So beschwert er sich in einer Email: „*please* put the branch-name after the git repo, so that I can cut-and-paste without noticing only afterwards that the diffstat doesn't match what it was supposed to, and I got the wrong commits, and have to undo and re-do the pull..“ (2414@kvm). Bei der Verwendung eines Pull-Betriebs muss man sich also auf bestimmte Konventionen einigen, denn ein Entwickler kann nicht immer wissen, wie das entfernte Repository aufgebaut ist, aus dem er pullt.

Für einen reinen Pull-Betrieb ist es notwendig, dass auch jeder die Möglichkeit hat, ein Repository öffentlich zum Pullen zur Verfügung zu stellen. Hierfür gibt es aber öffentliche Hoster, deren Service man dafür in Anspruch nehmen kann. Je nach gewünschtem SCM muss man sich einen entsprechenden Host suchen. Für Git zum Beispiel gibt kann man repo.or.cz wählen. Dieser wird zum Beispiel auch von dem Maintainer von ROX verwendet, als er sein erstes Git-Repository erstellt hat (9371@rox).

3.4.4 Gemischter Betrieb

In der Realität sieht es so aus, dass sich die Betriebsmodi meist mischen. Verwendet ein Projekt ein zentrales SCM, dann gibt es mindestens eine Person, meistens mehrere, die Commit-Rechte besitzen Andere wiederum müssen Patches senden, um sich an einem Projekt beteiligen zu können. Es kommt auf das Projekt an, wie leicht man zu einem Commit-Recht kommen kann.

Verwendet ein Projekt ein dezentrales System, kommt es vor, dass es dennoch ein zentrales Repository verwendet. Dieses ist zum Beispiel das Repository, aus dem auch ein Release hervorgeht oder auf das auf der Projekthomepage hingewiesen wird. Es gibt also mindestens ein Mitglied, welches in dieses Repository pushen darf, meistens der Maintainer des Projektes, der bestimmt, wessen Änderungen in dieses Repository einfließen sollen.

3.5 Adaptersysteme

Während der Untersuchung ist aufgefallen, dass mehrere Entwickler andere SCMs verwenden, als das Projekt, zu dem sie beitragen. Hierfür verwenden sie Adapter-Systeme, um das Repository des Projektes in das Repository der SCM ihrer Wahl zu importieren. Gerade, wenn Entwickler in mehreren Projekten mitwirken wollen, die unterschiedliche SCMs verwenden, bieten diese Adapter die Möglichkeit, lokal ein einheitliches System zu verwenden.
Wahrscheinlich gibt es sehr viele Nutzer dieser Adaptersysteme. In den Mailinglisten wurden diese aber meist nur in den Diskussionen über eine mögliche Migration des SCM erwähnt. Diese treten aber meist nur als Befürworter ihres persönlich verwendeten SCM in den Diskussionen auf, nicht aber als Innovator. Das kann daran liegen, dass diese Nutzer problemlos ihr bevorzugtes SCM verwenden und somit keinen Grund sehen, in die Verwaltung des Projektes einzugreifen.

Fast alle neueren SCMs verfügen über die Möglichkeit, Repostitorys aus den bekanntesten SCMs zu importieren. Auch werden diese Tools noch weiterentwickelt, um Projekten die Migration zu erleichtern und das eigene SCM populärer zu machen (9401@rox).

Einen Adapter könnte man auch einsetzen, wenn eine Migration stattfindet, die sehr lange dauert. Die Mitglieder hätten während der Umstellung schon die Möglichkeit, das neue SCM auszuprobieren, falls sie noch keine Erfahrung damit gemacht haben. Diese Idee hatte auch ein Entwickler bei ROX, der eine entsprechende Möglichkeit während der Migration zur Verfügung stellen wollte (9385@rox).

In einem Fall hat die Verwendung eines Adapters sogar dazu geführt, dass ein Fehler in einem Repository entdeckt wurde. In diesem Fall wurde eine Commit-Nachricht Auslöser dafür, dass der Adapter Fehler angezeigte. Denn in dieser Nachricht gab es einen Teil, der nicht UTF-8-kodiert war, was für Commit-Nachrichten in Subversion eigentlich gelten sollte, damit sie für verschiedene Zwecke in XML exportiert werden können (997@kvm).

3.6 Living on the Bleeding Edge

Der Ausdruck „Living on the Bleeding Edge“ im Bereich Softwareentwicklung bedeutet, die Version eines Projektes zu verwenden, an der aktuell entwickelt wird, die aber noch nicht als öffentliches, stabiles Release herausgegeben wurde. Meistens wird die aktuelle Version des Trunks genommen, es kann sich aber auch um einen speziellen Branch handeln, in dem ein Feature entwickelt wird, welches der Nutzer verwenden oder einfach nur testen möchte. Jeder Nutzer der Bleeding Edge-Version muss davon ausgehen, dass diese Version nicht ausreichend getestet und damit auch nicht immer voll funktionsfähig ist. Im Folgenden wird diese Version abgekürzt nur noch BE genannt.

Man kann grundsätzlich zwischen drei Arten von Personen, die BE verwenden, unterscheiden: Entwickler, Tester und Endnutzer.

Die aktiven Entwickler eines Projektes brauchen selbstverständlich immer die BE-Version, um sich über den Entwicklungsstand zu informieren und etwas beitragen zu können, es sei denn, sie beschäftigen sich ausschließlich mit der Fehlerbeseitigung in älteren Versionen. Wichtig ist das für sie auch, um sich nicht um Dinge zu kümmern zu müssen, die möglicherweise schon ein anderer Entwickler übernommen hat.

Die Verfolgung der BE-Version ist in zentralen SCMs einfacher zu handhaben als in dezentralen. Bei der Verwendung eines dezentralen SCM kann es eine große Anzahl von Repositorys geben, von deren Existenz ein Entwickler nicht immer etwas wissen muss. Wenn ein Entwickler zum Beispiel davon ausgeht, dass die von ihm in seinem Repository gemachten Änderungen für andere unwichtig sind, er sie für nicht professionell genug hält oder schlicht keine Möglichkeit hat, sein Repository öffentlich zum Pullen zur Verfügung zu stellen, kann niemand seine BE-Version begutachten und seine Änderungen für die eigene Entwicklung verwenden.

Der Begriff „Bleeding Edge“ bezieht sich aber auch meistens auf das Haupt-Repository eines Projektes. Das Hauptrepository ist bei der Verwendung eines de-

zentralen SCM dasjenige Repository, welches zum anfänglichen Klonen für neue Nutzer zur Verfügung steht, auf welches auf der Projekt-Hompage hingewiesen wird. Dieses beinhaltet normalerweise keine aktuellen Entwicklungen, denn hier fließen erst dann die Entwicklungen der Projektmitglieder ein, wenn diese, bzw. der Maintainer des Projektes der Meinung sind, dass sie ausreichend gestestet wurden. Es sollte also eine stabile Version enthalten. Im Folgenden wird deshalb nicht mehr auf die Verwendung eines dezentralen SCM eingegangen.

Die Entwickler des Projektes Flyspray haben den Ehrgeiz, die BE-Version so stabil wie möglich zu halten. In diesem Projekt werden z.B. selten Releases herausgegeben, alle, auch die normalen Nutzer, werden angehalten, die BE-Version aus dem Repository zu verwenden (5359@flyspray). Sie verweisen immer auf die BE-Version, sichern sich aber gleichzeitig vor möglichen Beschwerden seitens der Nutzer ab, indem sie darauf hinweisen, Entwicklungsversionen nicht für den produktiven Einsatz empfohlen werden (4923@flyspray), und weisen daraufhin, dass es in Open Source Projekten eine normale Verfahrensweise sei, dass „…bleeding-edge code is not supposed to work fine or to work at all.“ (5362@flyspray).

In diesem Projekt versucht man aber auch deshalb die Nutzer dazu zu bringen die BE-Version zu verwenden, um sie von möglichst vielen Nutzern testen zu lassen, um sich so selber Arbeit zu ersparen. Man versucht auch dadurch Tester zu erreichen, dass ein Online-Demo-System in der BE-Version zusätzlich zu dem bestehenden stabilen Online-Demo-System aufgesetzt wird, da nicht genügend Tester vorhanden sind, die bereit sind, SVN zu verwenden (zum Beispiel, weil sie nicht mit Kommandozeilenprogrammen umgehen können und auch nicht bereit sind, das zu lernen (4462@flyspray)). Das Projekt überlegt, einen so genannten Commit-Hook einzurichten, der dafür sorgt, dass bei jedem Commit das Demo-System automatisch aktualisiert wird. Ob das tatsächlich umgesetzt wurde, ist aus der Mailingliste aber nicht ersichtlich.

Um Tester zu erreichen, erzeugen viele Projekte auch automatisch regelmäßige Snapshots (6926@kvm). Die gepackten Dateien können von Testern heruntergeladen und kompiliert werden, ohne dafür ein SCM verwenden zu müssen. Das dient

dazu, auch Tester zu erreichen, die nicht bereit sind, sich mit der Verwendung eines SCM auseinandersetzen zu müssen. Hierfür könnten die Projekte aber auch Checkout- und Update-Scripts zur Verfügung stellen, um dieses zu erleichtern, wie es das Projekt Xfce zum Beispiel tut (13027@xfce).

Laut Harvey und Dawei ist es ein Markenzeichen von Open Source Entwicklungen, dass die Endutzer eine große Rolle in der Entwicklung einnehmen: „This is also encouraged by early and frequent releases of the source code, usually with a clear indication that this is not the recommended stable release. In this way, adventurous and technically adept users can use the bleeding edge of development and, since they have access to the source, can characterise bugs for developers to fix, or can even supply a fix themselves.” [19]

Für den Endnutzer eines Open Source Projekts, der auf einen fehlerfreien Betrieb der Software angewiesen ist, stellt sich die Frage nach Verwendung der BE-Version auf eine etwas andere Weise. Auf der einen Seite möchte der Endnutzer von den neuen Features und den möglicherweise reparierten Fehlern profitieren, riskiert aber auf der anderen Seite, dass die Bleeding Edge Version nicht hinreichend für einen produktiven Einsatz ausgereift ist. So sagt Michael Turnlund in [20] über den Einsatz von entwicklungsunterstützenden Werkzeugen die aus Open Source-Projekten hervorgehen: „The conundrum of staying close to the bleeding edge requires more judgment, however, and specific tooling knowledge is required—there may be some benefits in a latest release you want, but many have paid the price for developing with tools not-yet-ready for prime time. “.
Dieses Spannungsfeld wird insbesondere dann immer größer, wenn längere Zeit keine Veröffentlichung der Software als stabile Version erfolgt und die Anzahl der nur in der Bleeding Edge Version verfügbaren Fehlerreparaturen und neuen Features ansteigt. Ein Endnutzer im Projekt Flyspray, der das Bucktrackingsystem neu installieren musste, da seine Festplatte kaputt gegangen war sagte noch in (4422@flyspray): „Yes, I've read all the "but it's easy to get the SVN up and running" mails... I rather wait for the installer “. Dennoch entschied er sich schon ein paar Tage später, doch die SVN-Version zu verwenden, da er wohl ahnte, dass es lange dauern würde, bis ein neues Release herausgegeben worden wäre (4523@flyspray). Im Gegensatz zu

Entwicklern und Testern, die die Software nicht produktiv einsetzen, hat der Endnutzer bei der Verwendung der Bleeding Edge Version nicht nur das bereits erwähnte Risiko durch den Einsatz einer möglicherweise fehlerbehafteten Version zu tragen, sondern muss auch den Aufwand für die Installation der neuen Version in der Produktivumgebung mit einberechnen. Dies wird z.B. dann zusätzlich erschwert, wenn Datenbestände übernommen werden müssen wie im Falle des Bugtrackingsystems Flyspray. Das Projekt bietet seinen Nutzern hierfür einen Upgrader, der die Datenbanktabellen für die Nutzung in einer neuen veröffentlichten Version anpasst. Da eine Anpassung des Upgraders für jede Bleeding Edge Version zu aufwendig wäre, hat ein Nutzer einer Bleeding Edge Version den zusätzlichen Aufwand die Konvertierung möglicherweise von Hand durchführen zu müssen (4507@flyspray).

Endnutzer haben üblicherweise nicht die Entwickler-Mailingliste abonniert und werden nur über den Fortschritt der Software informiert, wenn sie die Projekthomepage besuchen, auf der meist nur über neue Releases informiert wird. Erst, wenn sie einen Fehler finden oder sich explizit ein Feature wünschen, wenden sie sich möglicherweise an die Mailingliste (4835@flyspray), was in dem Projekt Flyspray durchaus Sinn macht, denn dort wird versucht, Wünschen so schnell wie möglich gerecht zu werden (4880@flyspray).

Bei Flyspray gibt es sogar Endnutzer, die nicht nur die aktuelle Trunkversion verwenden, sondern die aus einem Branch, da sie ein Feature verwenden wollten, das derzeit nicht im Trunk enthalten ist (5330@flyspray).

Auch Flyspray selber fungiert als Endnutzer eines Projektes, indem es dessen Repository als External einbindet. In 5489@flyspray wird deshalb vorgeschlagen, nicht mehr deren BE-Version zu verwenden, sondern eine feste Revision zu verwenden, von der man weiß, dass diese funktioniert. Der Autor dieser Email selber verwendet immer die BE-Version von Flyspray, aber Vertrauen zu einem entfernten Projekt zu finden geht dann noch ein Stück weiter. Ein Maintainer des Projektes aber beruhigt ihn, denn dieser hat Schreibrechte für das eingebundene Repository, und könnte jederzeit eingreifen, wenn dort etwas kaputt geht (5501@flyspray). Schon zu einem früheren Zeitpunkt hatte es in diesem Projekt Probleme mit eingebundenen Reposito-

rys gegeben. In 5282@flyspray erklärt der gleiche Maintainer, wie man mit Hilfe des flags „ignore-externals“ in Subversion verhindern kann, dass ein Update fehlschlägt, wenn die eingebundenen Repositorys mal nicht erreichbar sind.
Die Verwendung der BE-Version eingebundenen externen Repositorys erhöht also eigentlich noch das Risiko für die Endnutzer. Sie müssen den Entwicklern zum einen in der Hinsicht vertrauen, dass diese sich um ihr eigenes Projekt kümmern, aber zusätzlich müssen sie ihnen vertrauen, dass sie auch das externe Projekt im Auge behalten, und rechtzeitig eingreifen, wenn sich dort etwas tut, was zu Beeinträchtigungen in ihrem eigenen Projekt führen könnte.

Natürlich kann man Entwickler, Tester und Endnutzer nicht streng voneinander trennen. Entwickler sind im Grunde auch oft Tester. Wird ein zentrales System verwendet, werden sie praktisch dazu gezwungen, auch Änderungen anderer Entwickler zu testen. Im dezentralen Bereich können sie sich dagegen aussuchen, welche Features anderer Entwickler sie in ihr Repository übernehmen wollen, und somit auch testen. Auch kommt es gerade in Open Source-Projekten sehr oft vor, dass Entwickler auch Endnutzer sind, denn gerade wegen des Interesses an einer bestimmten Software und dem Bedarf an einem bestimmten Feature bringen sich neue Entwickler in einem Projekt ein. In einer Studie kam heraus: “The development of Open Source/Free Software is not at all a matter of leisure "work" at home. 95% of the sample claim that they use OS/FS at work, school, or university. Thus, the professional background seems to be a very motivating factor for developing OS/FS.” [21]. Auch Raymond ist der Meinung dass “Users are wonderful things to have, and not just because they demonstrate that you're serving a need, that you've done something right. Properly cultivated, they can become co-developers.” [22].

Entwickler, die auch als Endnutzer fungieren, verwenden daher 2 Versionen, eine stabile, wenn auch Kunden involviert sind, und die BE-Version, um an der aktuellen Entwicklung teilnehmen zu können.

4 Zusammenfassung

4.1 Fazit zentrale SCM

Die Auswirkungen der Benutzung von zentralen und dezentralen SCM in Open Source Projekten sind äußerst unterschiedlich, da sie sich aus einer Kombination von verschiedensten Faktoren ergeben.

Die Wahl eines zentralen SCM führt dazu, dass Entwickler von mindestens einem Projektmitglied abhängig sind um sich am Projekt beteiligen zu können. Denn um einen schreibenden Zugriff auf das Repository zu erreichen, muss man unter Umständen zunächst sehr viel Arbeit investieren um das Vertrauen der Projektleitung zu erlangen. Wenn aber die Projektleitung das Einräumen dieser Rechte von vornherein ausschließt, dann kann dies einen potentiellen neuen Entwickler abschrecken, wenn dieser das Gefühl bekommt, dass seine Patches, die er ohne Commit-Rechte gezwungenermaßen senden muss, gar nicht, oder erst sehr spät beachtet werden.

Räumen die Projektleiter allerdings vorschnell einem Entwickler Schreibrechte auf ihr Repository ein, dann müssen sie damit rechnen, dass sie unter Umständen viel Arbeit investieren müssen, um ungewollte Commits rückgängig zu machen, und andere wieder einzupflegen, die nach einem ungewollten Commit gemacht wurden. Ein zentrales System ist durchaus für die Verwendung in einem kleinen Projekt geeignet, in dem kein großer Betrieb herrscht, und in dem es Projektmitglieder mit Schreibrechten gibt, die sich regelmäßig um möglicherweise eintreffenden Patches kümmern.

Die Projekte aber, bei denen sehr viele Patches eingehen, müssen damit rechnen, dass es schnell vorkommen kann, dass von diesen einige untergehen, und somit nicht ins Repository gelangen. Das bedeutet möglicherweise Mehraufwand, wenn es sich um wichtige Änderungen wie Bugfixes handelt, die dann ein anderer Entwickler noch einmal durchführen muss.

4.2 Fazit dezentrale SCM

Projekte, die ein dezentrales System verwenden haben viel mehr Möglichkeiten in der Art, wie sie es einsetzen.
Sie können es entweder wie ein zentrales SCM einsetzen, aber trotzdem von den positiven Eigenschaften, die dezentrale SCM an sich haben profitieren. Denn wenn es Projektmitglieder gib, die unter Umständen große Entwicklungsarbeit leisten, aber dabei nicht immer Zugriff auf das zentrale Repository haben, so steht diese Entwicklung trotzdem unter der vollen Versionskontrolle, und kann dadurch besser von anderen Projektmitgliedern nachvollzogen werden.

Ein dezentrales SCM bietet aber auch den Vorteil, dass der Maintainer eines möglicherweise vorhandenen Hauptrepositorys stark entlastet wird. Dadurch, dass dezentrale SCMs die Aufnahme einer Reihe von Patches ermöglichen kann er die Änderungen von einem Entwickler über einen längeren Zeitraum insgesamt aufnehmen, und trotzdem Schritt für Schritt nachvollziehen. Bei einem zentralen System muss er jedes Patch einzeln prüfen.
Ein weiterer Vorteil dieser Funktionsweise liegt darin, dass auch die Entwicklung von potentiellen neuen Mitgliedern unter ständiger Versionskontrolle steht. Denn diese äußern sich manchmal erst, wenn sie ein Entwicklungsvorhaben vollständig zu ende gebracht haben

Die Verwendung eines dezentralen Systems verringert auch die Abhängigkeit von einem Maintainer in dem Sinne, dass die Entwickler untereinander ihre Änderungen austauschen und testen können, wenn diese noch nicht in das Hauptrepository übertragen wurden. Das kann ein klarer Vorteil sein, wenn der Maintainer mal längere Zeit abwesend ist.

Der größte Unterschied zu einem zentralen SCM liegt aber wohl in der Möglichkeit, die Struktur eines Projektes derart zu ändern, dass man die Arbeit auf mehrere Personen verteilen kann, ohne diesen aber Commit-Rechte für ein Hauptrepository erteilen zu müssen. Jeder Entwickler kann selber bestimmen, welchen anderen Entwick-

lern er vertraut und aus wessen Repositorys er pullen möchte. So kann jeder Entwickler nur aus einer begrenzten Anzahl von Repositorys regelmäßig pullen, erntet damit aber auch die Änderungen von weit entfernten Repositorys, die auf dem Weg bis zu ihm von mehreren Entwicklern getestet wurden.

Die folgende Abbildung zeigt eine Struktur, die sich dadurch ergeben könnte, welche aber natürlich noch viel komplizierter und verwobener aussehen könnte. Diese ist angelehnt an die Struktur im Projekt U-Boot die nach der Migration ist.

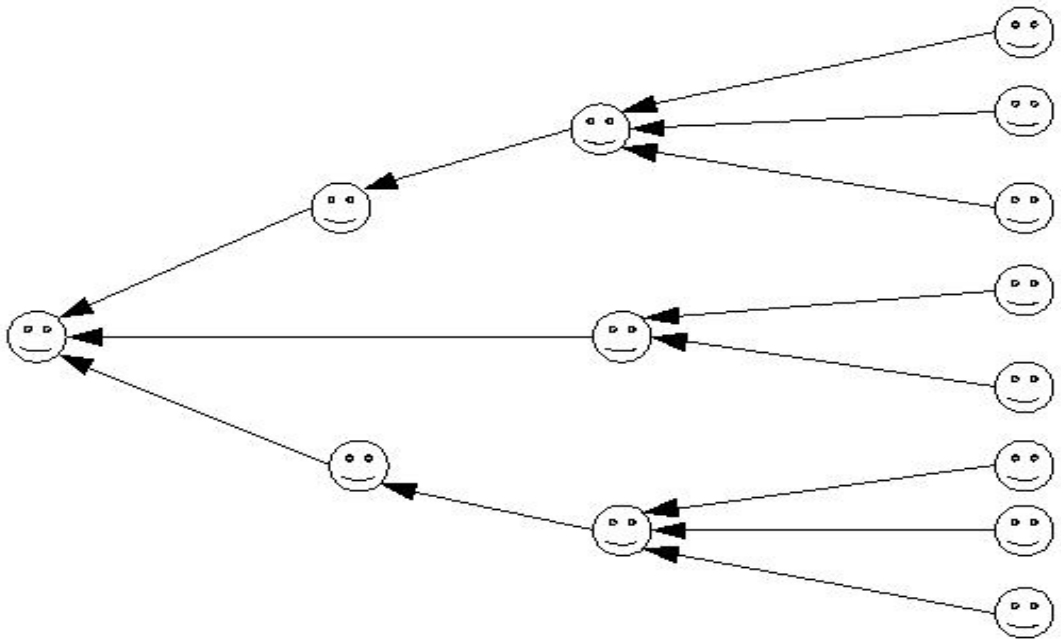

Abbildung 6: Repositorynetz - Beispiel

4.3 Zentrales Fazit

Insgesamt konnten leider viele Arbeitsprozesse anhand der Emails nicht erforscht werden, da über diese nicht in den Emails kommuniziert wird. So gab es etwa kaum Einblicke in administrative Prozesse wie Rechtevergaben oder Wartungsarbeiten. Diese Dinge werden wahrscheinlich in privaten Emails oder ähnlichem geklärt. Laut Aufgabenstellung sollten auch Möglichkeiten und Gefahren von Forking untersucht werden, aber auch in dieser Richtung war leider kein Material vorhanden.

5 Literaturverzeichnis

[1] Christopher Oezbek *Introduction of Innovation M1.* Technical report, Freie Universität Berlin, April 2008

[2] Strauss/Corbin *Grounded Theory: Grundlagen Qualitativer Sozialforschung* BELTZ, 1996

[3] http://gmane.org besucht am 15.4.2009

[4] Langley, Ann *Strategies for theorizing from process data* Academy of Management Review, 1999

[5] http://sourceforge.net/projects/gmanda/ besucht am 15.4.2009

[6] http://globulation2.org besucht am 15.4.2009

[7] http://roscidus.com/desktop besucht am 15.4.2009

[8] http://www.gnu.org/software/grub/ besucht am 15.4.2009

[9] http://flyspray.org/ besucht am 15.4.2009

[10] http://www.denx.de/wiki/view/U-Boot besucht am 15.4.2009

[11] http://www.linux-kvm.org besucht am 15.4.2009

[12] http://www.xfce.org/ besucht am 15.4.2009

[13] http://monetdb.cwi.nl/ besucht am 15.4.2009

[14] http://rox.svn.sourceforge.net/viewvc/rox/trunk/rox/README.Moved besucht am 15.4.2009

[15] http://git.die.net.au besucht am 15.4.2009

[16] http://sourceforge.net/ besucht am 15.4.2009

[17] Conklin, Howison, Crowston *Collaborating using OSSmole: a repository of FLOSS data and analyses* ACM SIGSOFT Software Engineering Notes, 2005

[18] Koch, Stefan — *Exploring the effects of SourceForge.net coordination and communication tools on the effiency of open source projects using data envelopment analysis* Journal: Empirical Software Engineering, August 2008

[19] Harvey, Hamish; Han, Dawei — *The relevance of Open Source to hydroinformatics* Jornal of Hydroinformatics, 2002

[20] Turnlund, Michael — *Distributed Development: Lessons Learned* Journal: Queue, 2004

[21] Ghosh, Rishab Aiyer and Glott, Ruediger and Krieger, Bernhard and Robles, Gregorio — *Free/Libre and Open Source Software: Survey and Study Partt 4: Survey of Developers* International Institute of Infonomics University of Maastricht Berlecon Research GmbH Berlin, 2002

[22] Raymond, Eric, S — *The Cathedral and the Bazaar* Journal: First Monday, 1998

[23] http://svnbook.red-bean.com/ besucht am 15.4.2009

[24] http://www.kernel.org/pub/software/scm/git-core/docs/everyday.html besucht am 15.4.2009

6 Abbildungsverzeichnis

7 Anhang

7.1 Zentrales SCM – Subversion

Subversion ist der Nachfolger von CVS, einem noch weit verbreiteten zentralen SCM. SVN wurde entwickelt, um CVS zu ersetzen, indem versucht wurde, dessen Schwachstellen auszumerzen. CVS war einmal das am weitesten verbreitete SCM, aus dem Grunde, weil es kein qualitativ vergleichbares anderes frei verfügbares SCM gab.

Mit Subversion kann man z. B. Änderungen ganzer Verzeichnisstrukturen verfolgen, bei CVS gilt das nur für einzelne Dateien. Somit werden auch Dateinamenänderungen und Verschiebungen archiviert.

Der Kommandozeilen-Client svn, auf den ich mich im Folgenden beziehe, ist für alle Plattformen erhältlich. Man kann Subversion aber auch über andere Programme verwenden. Es gibt z. B. ein Plugin für Eclipse (Subclipse) oder einen Windows Client, der in den Windows Explorer integriert wird (Tortoise SVN).

Bei Subversion werden Repositorys anhand von URLs *(ist doch überlall so?)* identifiziert. Wenn man auf ein Repository zugreifen möchte kann man das auf unterschiedliche Art und Weise tun:

`svn checkout file:///URL`	Zugriff auf der lokalen Festplatte
`svn checkout http://URL`	Zugriff über WebDAV auf Apache-Servern die SVN unterstützen
`svn checkout https://URL`	mit SSL-Verschlüsselung
`svn checkout svn://URL`	Zugriff auf einen svnserver-Server über SVN-eigenes Protokoll
`svn checkout svn+ssh://URL`	mit Tunnelung über SSH

Mit diesen Befehlen erzeugt man eine Kopie der Dateien aus der aktuellen Revision des Repositorys oder nur einer Datei aus dem Repository auf der lokalen Festplatte. Mit Hilfe der Option -r Versionsnummer kann man aber auf jede beliebige ältere Revision zugreifen. Änderungen an diesen Working Copys kann man mit dem Befehl `svn commit` in das Repository übertragen. Mit der Option -m kann man noch eine kurze Beschreibung der Änderungen mit angeben. Man kann eine Datei, aber auch eine ganze Verzeichnisstruktur commiten. Für jeden Commit legt SVN eine neue Revision an, deren Nummer um eins höher ist als die der vorherigen. Die neueste nennt sich HEAD. Mit dem Befehl `svn update URL` kann man die lokalen Arbeitskopien auf den neuesten Stand bringen. Subversion merkt sich für jede Arbeitskopie, aus welcher Revision sie stammt. Wenn man nun Änderungen an einer veralteten Datei, die schon vorher jemand anderes geändert hat, committen möchte, bricht Subversion dies mit dem Fehler *„out-of-date“* ab. Wenn man nun den Befehl `svn update` auf diese Datei anwendet, versucht SVN zunächst die beiden Versionen, die lokale Working Copy und die HEAD-Version aus dem Repository, selbständig zusammenzuführen (Merge). Schlägt dies fehl, muss der Entwickler sich selbst um die Konfliktbehebung kümmern.

Branches erzeugt man in Subversion mit Hilfe des Befehls `svn copy URLrepo URLrepozweig -m „Beschreibung.“`. Um Speicherplatz zu sparen, wird dabei nur ein Verzeichniseintrag erzeugt. Wirklich dupliziert werden nur die Dateien, an denen Änderungen vorgenommen werden. Aus Benutzersicht sieht die Verzweigung jedoch wie eine neue, zu Anfang noch identische Verzeichnisstruktur aus. Der Branch, in dem die Hauptentwicklung stattfindet liegt üblicherweise im Verzeichnis `trunk`, während die Verzweigungen im Verzeichnis `branches` liegen. Der Entwicklungszweig und der Trunk stellen nun 2 unterschiedliche Entwicklungslinien dar. Um diese wieder zusammenzuführen, verwendet man den Befehl `svn merge URLrepo`. Subversion versucht die beiden Zweige dann miteinander zu verschmelzen. Auftretende Konflikte müssen per Hand gelöst werden. Hat man die neue Version sorgfältig getestet, dann aktualisiert man mittels `svn commit` auch den Trunk. Zu diesem Zeitpunkt sind der Trunk und der Branch wieder identisch.

Eine Etikettierung einer Revision (Tag) erzeugt man auf dem gleichen Weg wie einen Branch mittels `svn copy`, nur wählt man als Ziel das Verzeichnis `tags`. Ein Tag verhält sich in Subversion auch genauso wie ein Branch, nur werden an einem Tag keine Änderungen mehr vorgenommen.

Die Dateien werden in einer Barkeley DB oder im Subversion-eigenen FSFS Dateisystem gespeichert. Man kann bisher zwischen einem Apache Server oder dem eigens zu Subversion-Zwecken entwickelten `svnserve`-Server wählen. Dabei ist ein Apache Server langsamer und aufwändiger in der Bedienung. Dadurch, dass Subversion protokolloffen ist, lässt es sich auch für andere implementieren. Zur Administration des Repositorys verwendet man das Programm `svnadmin`. Svnadmin kann nur direkt auf dem Server angewendet werden. Mit diesem Programm lässt sich ein Repository erstellen und verwalten. Der Administrator hat somit z.B. als einziger das Recht, Revisions-Log-Dateien zu ändern.

Eine Verbindung zu einem Apache-Server erhält man über WebDAV. Man kann aus allen bei Apache möglichen Authentifizierungsmethoden wählen und optional mit SSL verschlüsseln.
Der svnserve-Server verwendet ein eigenes Protokoll oder optional SSH (svnserve_ssh). Zur Authentifizierung stehen nur CRAM-MD5 und verschiedene SASL-Methoden zur Verfügung. Die Verschlüsselung kann man über SASL realisieren, bei svnserve_ssh ist sie Teil der Verbindung.

Die Rechtevergabe erfolgt jeweils auf Pfade, nur bei SSH aufs gesamte Repository, weshalb lokale Benutzerkonten erstellt werden müssen, die jeweils der Gruppe *„svn“* angehören müssen.

Der typische Arbeitszyklus sieht wie folgt aus: (direkt übernommen aus [22])

1. Aktualisieren Sie Ihre Arbeitskopie.
    - **svn update**
2. Nehmen Sie Änderungen vor.
    - **svn add**
    - **svn delete**
    - **svn copy**
    - **svn move**
3. Untersuchen Sie Ihre Änderungen.
    - **svn status**
    - **svn diff**
4. Nehmen Sie eventuell einige Änderungen zurück.
    - **svn revert**
5. Lösen Sie Konflikte auf (arbeiten Sie die Änderungen anderer ein).
    - **svn update**
    - **svn resolve**
6. Bringen Sie Ihre Änderungen ins Repository.
    - **svn commit**

7.2 Dezentrales SCM – Git

Git wurde 2005 von Linus Torvalds für die Entwicklung des Linux-Kernels entwickelt, da dieser kein neues geeignetes SCM fand. Zuvor wurde Bitkeeper verwendet, was aber durch eine Lizenzänderung nicht mehr möglich war.

Mittlerweile ist Git für alle Plattformen verfügbar, gilt aber noch nicht überall als fehlerfrei bzw. stabil. Git wurde und wird dafür entwickelt, möglichst schnell und effizient zu arbeiten und mit großen Codemassen umzugehen. Auch bei Git beziehe ich mich auf den Kommandozeilen-Client (git).

Auch bei Git kann man auf verschiedene Art auf ein Repository zugreifen:

git clone http://host.xz/path/to/repo.git/	Per HTTP
git clone https://host.xz/path/to/repo.git/	Mit SSL-Verschlüsselung
git clone rsync://host.xz/path/to/repo.git/	Nur noch nicht lokal gespeicherte Dateiteile
git clone git://host.xz/path/to/repo.git/ git clone git://host.xz/~user/path/to/repo.git/	git-eigenes Protokoll, baut auf TCP auf, optimale Bandbreitenausnutzung (Port 9418)
git clone ssh://[user@]host.xz[:port]/path/to/repo.git/ git clone ssh://[user@]host.xz/path/to/repo.git/ git clone ssh://[user@]host.xz/~user/path/to/repo.git/ git clone ssh://[user@]host.xz/~/path/to/repo.git	SSH-verschlüsselt

Nach erfolgreicher Ausführung einer der oben stehenden Befehle, hat man eine exakte Kopie des entsprechenden Repositories inklusive seiner kompletten Entwicklungsgeschichte. Alle Repository-Teile sind in 4 verschiedene Objekttypen unterteilt (Blops, Trees, Commits,

Tags), welche über SHA-1-Prüfsummen (über Objekttyp und Inhalt) angesprochen werden können. Dabei reichen normalerweise die ersten 6 Zeichen.

Aus dem lokalen Repository kann man sich nun per `git checkout` eine Working Copy auschecken. Wenn man nicht die aktuellste Version HEAD haben möchte, kann man über `git checkout revision` eine gewünschte ältere Version erhalten. Wenn man dann Änderungen an Dateien vorgenommen hat, kann man sie mittels `git commit –a –m „Beschreibung“` ins Repository übertragen. Einzelne Dateien übermittelt man über `git commit path/to/file`. Um die Änderungen dann in ein entferntes Repository zu übertragen, muss man den Befehl `git push` verwenden. Möchte man einen Commit rückgängig machen, dann verwendet man `git commit –amend`. Man kann auch `git reset –hard HEAD~x` verwenden. x steht dabei für die Anzahl von Revisionen, die man zurückgehen möchte. Dieser Befehl ist aber mit Vorsicht zu verwenden, da unter Umständen schon andere Entwickler mit einer nun gelöschten Version arbeiten. Die aktuelle Version des Master, also des Repositorys, welches man ursprünglich geklont hatte, erhält man über `git pull`.

Möchte man ein Patch erstellen und es per Email versenden, dann kann man den Befehl git format patchv verwenden. Dabei stehen viele verschiedene Optionen zur Verfügung, deren Erläuterung hier zu weit gehen würde. Das Patch wird dann lokal gespeichert. Mit `git-send-mail` kann man es dann als Email versenden, der Empfänger kann es dann mit `git aply patch` auf sein Repository anwenden. Dem zugrunde liegen die Linux-Kernel-Konventionen, bei denen Patches zu den jeweiligen Maintainern gesendet werden, welche es testen und erst dann auf das Stammrepository des Projektes anwenden. Einen Branch erstellt man per `git branch branchname`, einen Tag mittels `git tag –a name`. Zu einem anderen Branch wechselt man mit `git checkout branchname`. Das ist in Git deutlich einfacher gehalten als bei Subversion, wo man mit dem `copy`-Befehl arbeiten muss. Es gibt noch ein paar nützliche Befehle, wie man Branches auf unterschiedliche Art und Weise aneinander hängen kann, die ich hier nicht weiter erläutern möchte.

Um ein Repository für andere öffentlich zu machen, verwendet man `git-daemon`. Der Dämon wartet auf Port 9418 auf Service-Anfragen. Wenn man dem Git-Verzeichnis vorher noch die Datei `git-daemon-export-ok` hinzugefügt hat, können andere Entwickler per `git pull` auf das Repository zugreifen.

Mit git kann man wesentlich mehr machen als mit Subversion, die Grundbefehle sind aber ähnlich.

Quelle: [24]