Masterarbeit

Ein empirischer Vergleich von semistrukturierter und unstrukturierter Konfliktbehandlung in Versionsverwaltungssystemen

Verfasser:

Benjamin Brandl

19. April 2011

Betreuer:

Dr.–Ing. Sven Apel

Erstgutachter:

Prof. Christian Lengauer, Ph.D.

Zweitgutachter:

Prof. Dr. Harald Kosch

Universität Passau
Fakultät für Informatik und Mathematik
94030 Passau

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Passau, 19. April 2011

Inhaltsverzeichnis

Kapitel 1

Einleitung

Zum Handwerkzeug in der Softwareentwicklung gehören Versionsverwaltungssysteme. Sie stellen aktuelle Versionsstände zur Verfügung und sorgen gleichzeitig für die Archivierung der Projektdaten. Zugleich bieten sie Unterstützung bei Wartung und Funktionserweiterung. Über das Konzept der Verzweigungen können verschiedene Entwicklungslinien geführt werden. Es erlaubt individuelle Änderungen an verschiedenen Ausprägungen der Software. Aus dieser Möglichkeit der nebenläufigen Entwicklung ergibt sich das Bedürfnis zum Zusammenführen der verschiedenen Zweige oder Ausschnitten daraus. So möchte man beispielsweise eine fertiggestellte Programmfunktion aus einem experimentellen Zweig in den Hauptentwicklungszweig überführen. Dieser Vorgang wird auch Verschmelzen oder engl. Merge genannt.

Aktuelle Versionsverwaltungssysteme verwenden zum Zusammenführen ein generisches textuelles Verfahren. Es behandelt die Ausgangsdateien unabhängig von der verwendeten Programmiersprache als Text und führt die Verschmelzung zeilenbasiert durch. Dabei werden die beteiligten Programmvarianten auf textuelle Gemeinsamkeiten und Unterschiede untersucht. Die Zusammenführung erfolgt nach allgemeinen Mergeregeln. Liegen Unterschiede in zwei Varianten an der gleichen Stelle im Programmtext vor, so kann nicht entschieden werden, welcher Ausschnitt in das Ergebnis übernommen werden soll. Dies stellt eine Konfliktsituation dar. Sie muss am Ende des Verschmelzungsvorgangs vom Benutzer gelöst werden. Auf Grund der Generalität des dieser Technik können alle Formen an textueller Eingabe verarbeitet werden. Ebenso können Konflikte an beliebigen Stellen entstehen. Da Spracheigenschaften keine Beachtung finden, spricht man von unstrukturierter Konfliktbehandlung.

Den Gegenpol dazu bildet strukturierte Konfliktbehandlung. Hierbei werden die an einer Verschmelzung involvierten Programmvarianten zunächst in ihre grammatikalischen Elemente zerlegt und in eine passende Datenstruktur überführt. Auf Basis dieser Aufgliederung und mit dem Hintergrundwissen um syntaktische und semantische

Bedeutung der einzelnen Elemente, erfolgt eine Zusammenführung der Datenstrukturen. Abschließend wird das Resultat in Quelltext überführt. Durch diese Kenntnis um die Eigenschaften der Sprache sind viele Konflikte automatisch lösbar. Als nachteilig erweist sich, dass eine neue Konfliktklasse der Verhaltenskonflikte eingeführt wird. Deren Effekte äußern sich erst zur Laufzeit des Programms. Weiter sind diese Verfahren fest mit einer spezifischen Programmiersprache und einer speziellen Versionsverwaltung verankert, so dass sie nicht bei Projekten im Open Source Bereich Verwendung finden.

Methoden beider Ansätze vereint semistrukturierte Konfliktbehandlung. Wie bei dem strukturierten Verfahren werden die zu verschmelzenden Programmvarianten in eine Baumstruktur überführt. Die Aufgliederung ist gröber, aber fein genug um Programmelemente zu erkennen, die sich im Quelltext frei anordnen lassen. Anhand der allgemeinen Mergeregeln wird bestimmt, welche dieser Elemente in die zusammengeführte Version übernommen werden. Sie werden anschließend sortiert und in den Programmtext eingefügt. Eine Schablone zum Erkennen der Programmelemente liefert eine annotierte Grammatik. Sie beschreibt zusätzlich mit welchen Verfahren das jeweilige Element verschmolzen wird. Dabei wird für Reihenfolge unabhängige Elemente eine spezielle Konfliktbehandlung verwendet, während Reihenfolge abhängige Elemente mit dem zeilenbasierten Verfahren verschmolzen werden.

Im Kontext der Programmiersprache Java spielt beispielsweise die Anordnung von Attributen und Methoden in Klassen keine Rolle. Tritt zwischen dieser Art von Elementen ein Konflikt auf, so spricht man von einem semantischen Konflikt. Als Reihenfolge abhängig präsentieren sich Methodenrümpfe. Sie bestehen aus einer Abfolge an Anweisungen, die im Konfliktfall nicht automatisch vereint werden können. Die Konfliktbehandlung erfolgt mit dem unstrukturierten Verfahren. Da die Abfolge der Anweisungen Ursache für den Konflikt ist, spricht man von einem Ordnungskonflikt[1].

Wie das strukturierte Verfahren muss semistrukturierte Konfliktbehandlung an die jeweilige Programmiersprache angepasst sein. Die Schnittstelle hierfür ist wesentlich kleiner. Die Integration einer neuen Programmiersprache beschränkt sich auf die Formulierung der Grammatik und Festlegung der Konfliktbehandlung. Alle weiteren Bestandteile des Verfahrens sind sprachunabhängig. Somit kann das Verfahren mit geringem Aufwand um eine neue Programmiersprache erweitert werden. Trotz möglicher Spezialisierung bleibt es allgemein anwendbar. Ist für eine Sprache keine Grammatik vorhanden, erfolgt die Verschmelzung nach dem unstrukturierten Verfahren.

[1] Bei unstrukturierter Konfliktbehandlung ist jeder Konflikt ein Ordnungskonflikt.

Diese Arbeit vergleicht unstrukturierte und semistrukturierte Konfliktbehandlung anhand verschiedener Revisionskonstellationen ausgewählter Softwareprojekte. Es wird ermittelt in wie weit das Kontextwissen sich auf das Konfliktverhalten bei Zweigverschmelzungen in Versionsverwaltungssystemen auswirkt und ob sich Vorteile ergeben. Merges werden zwischen verschiedenen Programmrevisionen durchgeführt. Dabei sind drei Revisionen, das sog. Mergetripel, beteiligt: eine Basisrevision und zwei abgeleitete Revisionen. Bei einer Verschmelzung werden die Änderungen einer Version in die andere unter Berücksichtigung der gemeinsamen Basis übertragen. Dieses Vorgehen wird Drei-Wege-Verschmelzung (engl. Three-Way-Merge) genannt und in Versionsverwaltungen angewandt.

Für den Vergleich beider Verfahren werden über die Projekthistorie Mergetripel identifiziert und zu Mergeszenarien zusammengefasst. Zur Analyse werden Mergetripel verwendet, die Revisionen eines tatsächlich durchgeführten Merges kapseln. Da in manchen Projekten Zusammenführungen nur selten auftreten und außerdem reale Merges nicht immer erkennbar sind, werden zusätzlich Konstellationen betrachtet, in denen ein Merge denkbar gewesen wäre. Der Abschluss eines Entwicklungszweig und die Integration in den Hauptentwicklungszweig stellen zum Beispiel realistische Szenarien dar.

Die an den Szenarien beteiligten Revisionen werden über die Versionsverwaltung extrahiert und nach einem Vorverarbeitungsschritt mit den beiden Verfahren verschmolzen. Aus beiden Ergebnissen werden Kennzahlen bestimmt, anhand derer der Vergleich durchgeführt wird. Diese Maßzahlen erfassen Zahl und Umfang der Konflikte. Gemessen werden pro Szenario:

Konfliktzahl Anzahl der Ordnungskonflikte

Konfliktzeilen Anzahl der Quelltextzeilen, die die Ordnungskonflikte umfassen

Konfliktdateien Anzahl der Dateien, in denen Konflikte auftreten

Semantische Konflikte Anzahl der semantischen Konflikte bei semistrukturierter Konfliktbehandlung

Diese Zielsetzung und Vorgehensweise schafft gewisse Anforderungen an zu untersuchende Softwareprojekte. So muss das Projekt lesenden Zugriff auf die verwendete Versionsverwaltung gewähren. Um Mergeszenarien zu finden, müssen Verzweigungen vorhanden sein. Ebenfalls muss das Projekte einen angemessenen Umfang an Quelltext vorweisen. Somit bieten sich zur Analyse Open-Source-Projekte an. Sie erlauben einen realistischen Einblick in Entwicklungs- und Verschmelzungstrends. Durch die große Auswahl an Projekten und Mergeszenarien für den Vergleich lassen sich die Erkenntnisse festigen.

Mit semistrukturierter Konfliktbehandlung können Konstellationen behandelt werden, die bei unstrukturiertem Vorgehen stets zum Konflikt führen. Daher erwartet man bei semistrukturierter Behandlung bessere Ergebnisse, die sich durch niedrigere Kennzahlen zeigen. Dies trifft für die Mehrzahl der Mergeszenarien zu. Allerdings zeigten sich in den übrigen Fällen deutliche Abweichungen. Zurückführen ließ sich dieses Phänomen auf ein Umbenennungsproblem. Beide Verfahren reagieren gegensätzlich, wenn das Mergetripel nicht vollständig ist.

Die Arbeit gliedert sich in folgende Kapitel auf:

Kapitel 2 Dieses Kapitel bietet einen Überblick über Versionsverwaltungen. Dabei werden Aufgaben, Strategien und Architekturen verschiedener Systeme betrachtet. Zudem wird die Notwendigkeit von Entwicklungszweigen und deren Zusammenführung aufgezeigt. In diesem Zusammenhang werden unstrukturierte, strukturierte und semistrukturierte Mergeverfahren näher erörtert.

Kapitel 3 Dieser Abschnitt schildert die Rahmenbedingungen des Vergleichs. Er geht auf Zielsetzung und Methodik ein. Weiter beschreibt er die Auswahl der Kandidaten.

Kapitel 4 In diesem Kapitel wird die Durchführung des Vergleichs ausgeführt. Es konkretisiert die Überlegungen aus dem vorgehenden Abschnitt und schildert insbesondere die Bestimmung der Mergeszenarien.

Kapitel 5 Dieses Kapitel wertet die Ergebnisse des Experiments aus. Der erste Teil vergleicht die beiden Verfahren anhand der ermittelten Kennzahl. Der zweite Teil stellt typische Konfliktsituationen nach und wertet diese aus.

Kapitel 6 In diesem Abschnitt werden die gewonnen Erkenntnisse zusammengefasst. Verbleibende Probleme und Lösungsmöglichkeiten werden in einem Ausblick aufgezeigt.

Kapitel 2

Versionsverwaltung

Die Entwicklung von Software ist ein vielschichtiger Prozess. Auf Grund wachsender Anforderungen und damit wachsender Komplexität können Softwareprojekte nicht von einzelnen Personen bewältigt werden. Damit mehrere Entwickler an einem Projekt arbeiten können, müssen sie Zugang zu den Quelltexten haben und ihre Änderungen an diesen untereinander austauschen können.

Diese Verwaltungsaufgaben werden im Allgemeinen von einem Versionsverwaltungssystem übernommen. Es archiviert alle Versionen von Quelltextdateien und und protokolliert damit den Verlauf der Entwicklung. Ebenso wird bei verteilten Dateischreibzugriffen sichergestellt, dass die Nutzer ihre Änderungen nicht aus Versehen gegenseitig auslöschen. Eine weitere Funktion von Versionsverwaltungen ist die Aufspaltung der Hauptentwicklungslinie und getrennte Nebenentwicklungslinien. In diesen nebenläufigen, isolierten, Zweigen (engl. *Branches*) können neue Programmfunktionen implementiert oder Fehlerbehebungen getestet werden. Ist die Entwicklung abgeschlossen, kann man die neuen Programmteile in den Hauptentwicklungszweig integrieren. Dieser Prozess wird Zusammenführen oder *Merge* genannt.

Das Zusammenführen der Zweige ist ein komplexer, fehleranfälliger und zeitaufwändiger Vorgang. Wenn in den zu vereinigenden Zweigen gemeinsame Dateien geändert wurden, müssen diese von Entwicklerhand zu einer neuen Version zusammengeführt werden. Zur Unterstützung dieser Aufgabe wird im Allgemeinen ein textbasiertes Verfahren eingesetzt, welches die textuellen Unterschiede an den beteiligten Dateien anzeigt.

In diesem Kapitel werden weitere Aufgaben und unterschiedlichen Typen von Versionsverwaltungssystemen vorgestellt und verschiedene Zusammenführungskonzepte betrachtet.

2.1 Überblick

Wie bereits erwähnt dient ein Versionsverwaltungssystem der Verwaltung von Quelltextdateien eines Softwareprojekts. Im Allgemein gruppieren sich die Aufgaben um Archivierung von Versionen und Zugriffskontrolle beliebiger Dateien. Ein Einsatz bietet sich an, wenn mehrere Personen eine gemeinsame Menge an Dateien bearbeiten und/oder Versionsstände abrufbar sein sollen. Wegen fehlender textueller Vergleichbarkeit steht bei Binärdateien nicht der volle Funktionsumfang zur Verfügung.

Um die Funktionen einer Versionsverwaltung erläutern zu können, werden zunächst häufig verwendete Begriffe eingeführt.

2.1.1 Aufgaben und Begriffe

Der zentrale Datenspeicher eines Versionsverwaltungssystem ist das *Repository*. Es bildet das Projektarchiv und beinhaltet die versionierten Dateien und deren Metadaten, die häufig in einer Datenbank abgelegt werden. Organisiert ist das Archiv zu meist in Dateien und Verzeichnisse. Das Verzeichnis, das die aktuelle Entwicklungslinie beinhaltet, wird Hauptentwicklungszweig (engl. *Trunk*) genannt. Kopien von Trunk oder einem seiner Unterverzeichnisse bilden eine Nebenentwicklungslinie (engl. *Branch*). Die Arbeitskopie (engl. *Working Set*) ist eine lokale Kopie der Repository oder eines Entwicklungszweigs, an der Dateien geändert werden können. Ein *Tag* markiert den Versionsstand eines Zweiges. Die kann per Konvention durch Kopie oder Annotation in Metadaten erfolgen.

Jede Datei im Repository liegt in allen ihren Versionen vor. Eine *Version* ist die Momentaufnahme einer Datei zu einem bestimmten Zeitpunkt. Die jüngste Version heißt *Head*. *Revision* wird in dieser Arbeit im Sinne von *überarbeitete Fassung* und damit als Synonym zu Version verwendet. Als *Variante* bezeichnet man eine Datei, die in unterschiedlichen Versionen in mehreren Zweigen vorkommt. Jeder neuen Version wird das Änderungsdatum, der Benutzer und eine Änderungsnachricht (engl. *Commit Message*) zugeordnet. Mit dem Kommando *Add* wird eine Datei dem Versionsverwaltungssystem übergeben. Der Befehl *Update* synchronisiert die Arbeitskopie mit dem Repository. Das Abspeichern einer neuen Revision einer Datei im Repository nennt man *Check in* oder auch *Commit*, das herunterladen einer bestimmten Revision *Check out*.

Das Verschmelzen, das Übertragen von Änderungen von einer Datei auf eine andere, von zwei Varianten wird zusammenführen (engl. *Merging*) genannt. Überlappen sich Änderungen beiden Varianten, so liegt ein *Konflikt* vor.

Versionsverwaltungssysteme erfüllen folgende zentrale Aufgaben.

Archivierung und Wiederherstellung Eine Funktion von Versionsverwaltungssystemen ist die Sicherung und Wiederherstellung von Dateien. Neue Dateien werden dem Repository hinzugefügt, bearbeitete Dateien in einer neuen Revision abgelegt. Bei jedem Commit einer Datei wird dem jeweiligen Versionsstand eine eindeutige Nummer, die sog. Revisionsnummer zugeordnet. Gleichzeitig kann jeder eingecheckte Versionsstand einer Datei wieder abgerufen werden.

Kurzfristige Wiederherstellung Änderungen geschehen an der Arbeitskopie zunächst differentiell, so ist es jederzeit möglich lokal geänderte Dateien wieder auf den Stand des letzten Checkouts zu bringen. Dies bietet Sicherheit und erlaubt eine gewisse Experimentierfreudikeit beim Bearbeiten der Dateien.

Langfristige Wiederherstellung Da sämtliche Revisionen einer Datei im Repository liegen, kann auf ältere Revisionen einer Datei weiterhin zugegriffen werden. Ebenfalls verbleiben Revisionen von zum aktuellen Zeitpunkt gelöschten Dateien verfügbar.

Zugriffsfunktion Das Repository kann mehreren Benutzern zur Verfügung gestellt werden. Dabei kann entschieden werden, welche Rechte der einzelne Anwender hat. D.h. ob er nur lesend oder auch schreibend auf die Inhalte zugreifen darf.

Dokumentationsfunktion Bei einem Commit werden im Versionsverwaltungssystem Metadaten zur Änderung abgelegt. Diese Daten umfassen die bereits erwähnte Revisionsnummer, den Zeitpunkt der Änderung, die betroffenen Dateien, den Namen des Benutzers, der die Änderung durchführt und einen frei wählbaren Kommentar. Somit ist jeder Änderung eindeutig zurechenbar.

Verzweigungen Alle Benutzer teilen eine gemeinsame Menge an Dateien. Die Entwicklung findet dabei im Hauptentwicklungszweig statt. Wie in Abbildung 2.1 dargestellt, kann dieser zu jedem Zeitpunkt in Nebenentwicklungszweige aufgespalten werden. Eine solche Verzweigung ist hilfreich, um neue Programmfunktionen oder Fehlerbehebungen isoliert zu implementieren und zu testen. In der Abbildung wird Versionsstand `T1` zu Versionsstand `B1` kopiert. Die Entwicklung läuft an beiden Zweigen parallel und unabhängig von einander ab. In `B3` ist die Entwicklung des neuen Programmteils abgeschlossen. Der Nebenentwicklungszweig wird in den Hauptentwicklungszweig integriert. Aus den aktuellsten Versionen der Zweige `T4` und `B3` wird die gemeinsamen Version `TB1` gebildet.

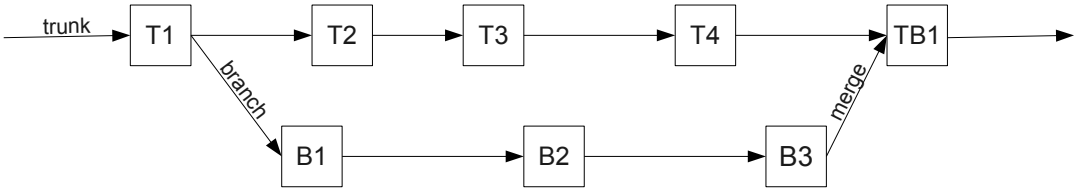


Abbildung 2.1: Lebenszyklus eines Nebenentwicklungszweigs

Archivierung von Meilensteinen Obwohl die Versionsgeschichte stets verfolgbar ist, sollte die Möglichkeit bestehen einen bestimmten Versionsstand als Meilenstein zu markieren und verfügbar zu machen. Insbesondere bei bestimmten Projektständen wie Meilensteinen und Versionssprüngen bietet sich diese Funktion an.

2.1.2 Strategien

Eine Versionsverwaltung muss mehreren Benutzern paralleles Arbeiten am Repository erlauben und gleichzeitig sicherstellen, dass Änderungen nicht versehentlich überschrieben werden. Dies kann geschehen, wenn wie in Abbildung 2.2 zu sehen ist, zwei Benutzer A und B zeitgleich die selbe Datei C bearbeiten. Ohne Sicherungsmechanismen würde der Benutzer B, der die Datei C″ als letztes eincheckt die Änderungen C′ des anderen Benutzers überschrieben.

Im Allgemeinen spricht man von einem Konflikt, wenn eine lokal geänderte Version einer Datei ins Repository aufgenommen werden soll, sich aber die lokal nicht geänderte Version dieser Datei nicht mit der aktuellen Version im Repository deckt. Die Datei wurde zwischenzeitlich von einem Dritten bearbeitet.

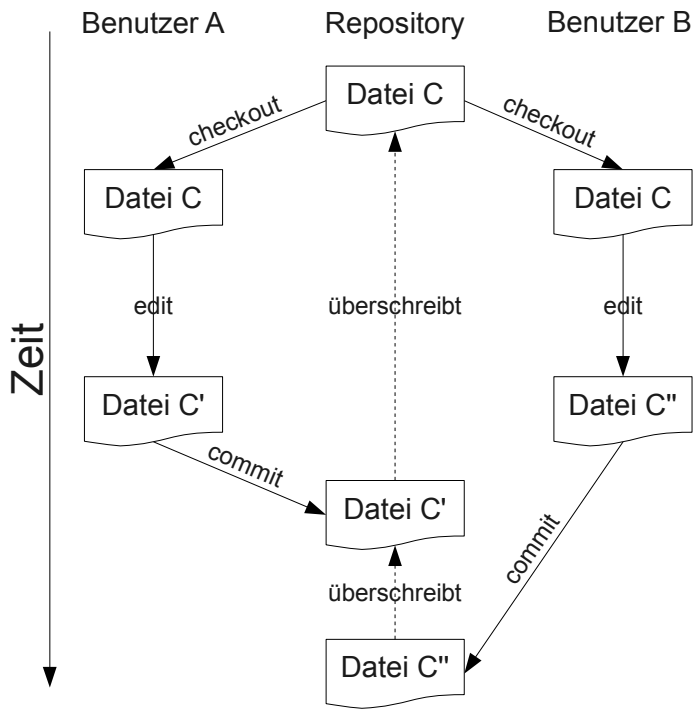


Abbildung 2.2: Versehentliches Überschreiben (nach [CSFP10])

Um einen reibungslosen Mehrbenutzerbereich gewährleisten zu können, muss ein Versionsverwaltungssystem die Schreibzugriffe mehrerer Benutzer koordinieren und Konflikte entweder vermeiden, lösen oder zumindest anzeigen.

Obiges Konflikt-Szenario ähnelt dem *Lost Update Problem* aus dem Datenbankbereich. Hier löschen sich bei unglücklicher Transaktionsreihenfolge Schreiboperationen gegenseitig aus. Die Änderungen sind bei Datenbanken endgültig verloren, während man bei Versionsverwaltungssystemen die überschrieben Änderungen durch die Archivierungsfunktion wiederherstellen könnte.

Pessimistische Versionsverwaltung

Ein konservative Lösung des Lost Update Problems stellt das Zwei-Phasen-Sperrprotokoll dar (siehe auch [EK06]). Dort wird bei jedem Lesevorgang eines Datums eine Sperre auf dieses gesetzt, so dass es nicht von einer anderen, möglicherweise parallel laufenden, Transaktion gelesen werden kann. Erst mit Abschluss der ursprünglichen Transaktion wird die Sperre aufgehoben. Damit ist sichergestellt, dass das Datum nur exklusiv bearbeitet werden kann und Konflikte sind ausgeschlossen.

In Versionsverwaltungssystemen kommt das Sperren-Ändern-Entsperren-Modell zum Einsatz, das ähnlich wie das des Zwei-Phasen-Sperrprotokolls funktioniert. Will ein

Benutzer eine Datei modifizieren, so holt er die aktuelle Version aus dem Repository und setzt eine Sperre auf die Datei. Leseversuche anderer Benutzer auf diese Datei werden von nun an blockiert. Ist die Modifikation abgeschlossen, wird die neue Version in das Repository aufgenommen und die Sperre entfernt. Wird dieses Modell verwendet, spricht man von einer *pessimistischen Versionsverwaltung*. Abbildung 2.3 zeigt den Ablauf bei konkurrierenden Dateimodifikationen.

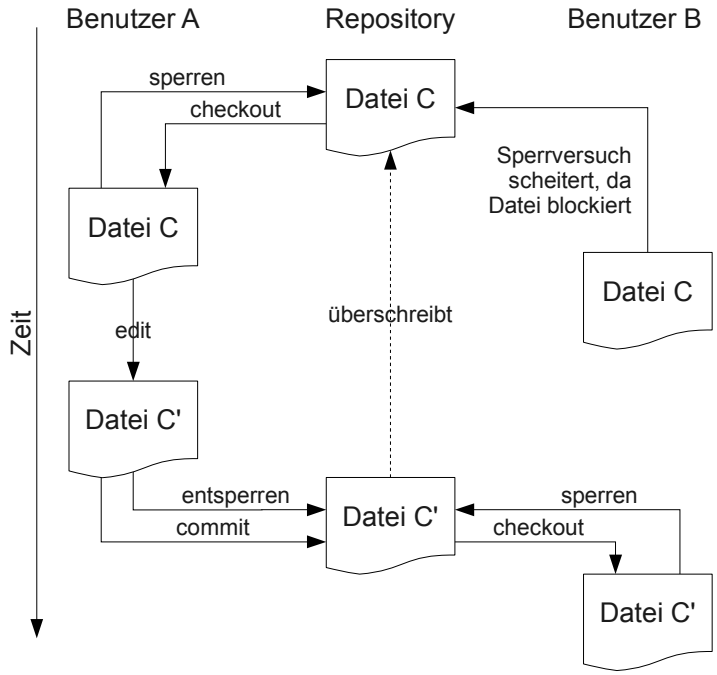


Abbildung 2.3: Sperren-Ändern-Entsperren-Modell (nach [CSFP10])

Wie in [CSFP10] erörtert führt ein dieses Modell zu mehreren Problemen. So muss vom Benutzer sichergestellt werden, dass die Sperre wieder freigeben wird. Ebenfalls wird so paralleles Arbeiten an einer Datei unmöglich, selbst wenn disjunkte Teile geändert werden sollen. Weiterhin können sich Benutzer durch eine unglückliche Sperrreihenfolge blockieren.

Frühe Versionsverwaltungen wie SCCS [All80] und RCS [Tic85] verwendeten dieses Modell. In aktuellen Versionsverwaltungssystem wird es nur in Ausnahmefällen angewandt. So können bei Subversion bei Bedarf einzelne Dateien gesperrt werden. Ein Sperre bietet sich für Dateien an, die die jeweilige Zusammenführungsroutine nicht bearbeiten kann. Die Daten sind unvergleichbar und bei einem Konflikt müsste man sich zwischen einer der beiden Varianten entscheiden. Ein Beispiel für typischerweise unvergleichbare Daten sind Binärdateien.

Optimistische Versionsverwaltung

Da obiges Modell Mehrbenutzerbetrieb stark einschränkt, kommt in aktuellen Versionsverwaltungssystemen das Kopieren-Ändern-Zusammenfassen-Modell zum Einsatz. Hier erhält jeder Benutzer eine Arbeitskopie des Repository, auf dessen Datenbestand er arbeitet. Eine stetige Verbindung zum Repository ist nicht nötig. Die Nutzer können isoliert arbeiten. Änderungen werden von der Arbeitskopie in das Repository übertragen.

Der Preis den Mehrbenutzerbetrieb sind Konflikte. Tritt ein Konflikt auf, so wird der Benutzer aufgefordert aus seinen lokalen Änderungen und der Version des Repository eine neue Version zusammenführen. In Abbildung 2.4 wird der Ablauf eines einfachen Konfliktszenarios dargestellt.

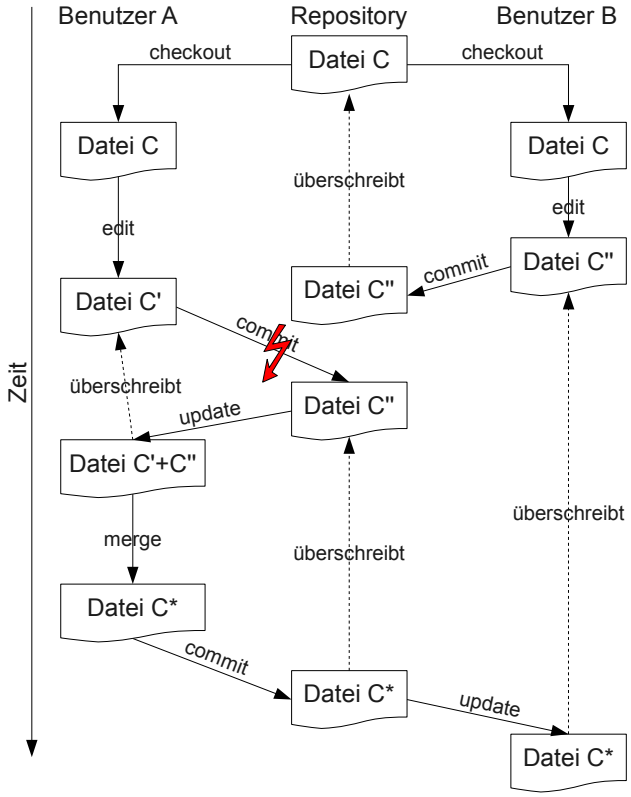


Abbildung 2.4: Kopieren-Ändern-Zusammenfassen-Modell nach [CSFP10]

Die Benutzer `A` und `B` checken parallel `Datei C` aus dem Repository aus. Beide modifizieren die Datei und `B` überträgt seine Version `C''` in das Repository. Benutzer `A` scheitert bei seinem Commit von `C'`, da eine neuere Version im Projektarchiv liegt. Ein Konflikt zwischen `C'` und `C''` tritt auf. Um diesen zu lösen muss `A` die Version `C''` des Repository mit seiner bearbeiteten Version zusammenführen. Datei `C*` ist das Ergebnis

des Zusammenführungsvorgangs. Diese Version kann nun dem Repository übergeben werden.

2.1.3 Architekturen

Versionsverwaltungssysteme unterscheiden sich bei der Frage, wo der Datenspeicher gehalten wird und wie Zusammenführungen ablaufen.

Lokale Versionsverwaltung

Eine lokale Versionsverwaltung ist nicht netzwerkfähig. Das Repository liegt auf dem Rechner des Benutzers. Sie eignet sich zur Archivierung von Änderungen an einzelnen Dateien und findet immer noch Anwendung im Bürobereich. In begrenztem Umfang führt die Entwicklungsumgebung Eclipse[1] die Versionen von lokal geänderten Dateien mit.

Bekanntester Vertreter von lokalen Versionsverwaltungen ist das 1985 entwickelte RCS ([Tic85]). Es nutzt die pessimistische Strategie und erlaubt die Versionierung einzelner Dateien. RCS führte das Konzept des Verzweigens und Zusammenführens ein. Als Toolunterstützung beim Zusammenführen kam `rcsmerge` zur Einsatz. In seiner weiterentwickelten Form in den *GNU diffutils* ist es der Standard für textuelles Zusammenführen (siehe 2.2.3 auf Seite 19).

Name	URL
Source Code Control System (SCCS)	`http://sccs.berlios.de`
Revision Control System (RCS)	`http://www.cs.purdue.edu/homes/trinkle/RCS`

Tabelle 2.1: Populäre lokale Versionsverwaltungssysteme

[1] `http://www.eclipse.org`

Zentrale Versionsverwaltung

Zentrale Versionsverwaltungen verwenden ebenfalls ein einziges Repository im Gegensatz zur lokalen Verwaltung kann sie auch über ein Netzwerk genutzt werden. Dabei bietet sich die Aufspaltung des Repository in Projektarchiv auf einen Server und eine lokale Arbeitskopie an.

Kommt die optimistische Strategie zu Einsatz, so gestalten sich die Abläufe wie in Abbildung 2.4 und 2.5 dargestellt. Die Benutzer arbeiten lokal und gleichen ihre Änderungen mit dem zentralen Archiv ab. Ggf. ist beim Abgleich eine Konfliktbehandlung notwendig. Da die Projekthistorie auf dem Server verbleibt, ist die Arbeitskopie wesentlich kleiner als das Repository und jeder Benutzer hat stets Zugriff auf den gesamten Revisionsverlauf aller Dateien. Zudem ist der Verlauf der Revisionsnummern linear.

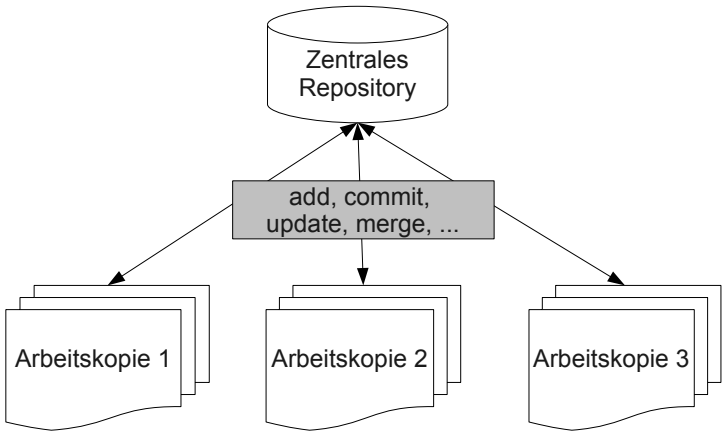


Abbildung 2.5: Zentrale Versionsverwaltung

Da Branches als Kopie eines Verzeichnisse im Repository angelegt werden, sind sie von allen Benutzern sichtbar. Ebenso sind neu eingecheckte Versionen sofort für die anderen Benutzer abrufbar.

Das Erstellen von Branches erfolgt im Repository. Dort wird das gewünschte Verzeichnis in einen neuen Entwicklungszweig kopiert. Dies geschieht bei Subversion beispielsweise durch Anlegen einer symbolischen Verknüpfung zwischen Ursprungsverzeichnis und neuem Branch. Das Erstellen ist serverseitig eine einfache Operation. Da die Übertragung von einem Branch zu einem Benutzer meist über ein Netzwerk mit beschränkter Bandbreite abläuft, dauert dieser Vorgang ab einer gewissen Projektgröße lange. Das Auschecken eines Branches ist ein zeitlich teurer Vorgang. Dadurch werden Branches vor allem für längerfristige Aufgaben wie Wartung eingesetzt.

Name	URL
Concurrent Versions System (CVS)	http://www.nongnu.org/cvs
Apache Subversion (SVN)	http://subversion.apache.org

Tabelle 2.2: Populäre zentrale Versionsverwaltungssysteme

Dezentrale Versionsverwaltung

Dezentrale Versionsverwaltungen lösen sich von dieser Struktur. Es ist kein zentraler Server notwendig, eine Verwendung ist optional. Jeder Benutzer hält selbst das Projektarchiv und die Arbeitskopie. Damit ist jede Änderung an beiden Datensätzen zunächst nur am Rechner des Benutzers verfügbar. Diese Lokalität bringt wegen fehlender Netzwerkkommunikation einen deutlichen Geschwindigkeitsvorteil gegenüber der zentralen Variante. Somit ist das Erstellen und Auschecken von Verzweigungen praktisch ohne Zeitverlust möglich. Branches werden so »billig« und kurzlebiger, was ebenfalls die Anzahl potentieller Konflikte verringert. Eine dezentrale Versionsverwaltung ist immer optimistisch, es können keine Dateien gesperrt werden. Auf Grund der Dezentralität ist jeder Commit ein Branch für sich.

Der Datenaustausch unter den Entwicklern läuft nach einen Peer-To-Peer-Schema auf Repository-Ebene ab. Dazu müssen die Begriffe aus 2.1.1 erweitert werden. Um an einem Projekt teilzunehmen, muss ein Entwickler zu Beginn das Repository eines anderen kopieren. Diese Operation wird *Clone* genannt. Untereinander tauschen die Entwickler ihre Änderungen über sog. *Pull*-Operationen aus. Dazu muss der Empfänger, der »Pullende«, den Speicherort der neuen Daten kennen. Ist dies der Fall läuft ein Pull wie ein Update-Kommando ab. Das *Push*-Kommando verhält sich wie ein Checkin. Damit können Daten in ein fremdes Repository übertragen werden. Dies findet üblicherweise nur Verwendung, wenn mit einem zentralen Repository das Verhalten einer zentralen Versionsverwaltung nachgestellt werden soll.

Mangels einer zentralen Datenstelle ergibt sich ein freies Entwicklungsmodell. Alle Clients sind gleichwertig und unabhängig. Dennoch ist es für ein Projekt hilfreich einen Verwalter zu führen. Dieser hält ein Repository mit einem aktuellen Stand und bietet eine Anlaufstelle für neue Entwickler. Möchte ein Entwickler seine Änderungen mit anderen teilen, so stellt er einen *Pull Request* an den Verwalter, der den Speicherort und die Inhalt der Änderung bekannt gibt. Nach einem Code Review wird die Änderung integriert. Mittels Pull-Operation können andere Benutzer nun die aktualisierten Quellen von der bekannten Stelle abrufen. Verwendet das Projekt ein zentrales Repository, sind Abläufe wie bei einer zentralen Versionsverwaltung möglich. Abbildung 2.1.3 illustriert das Zusammenspiel.

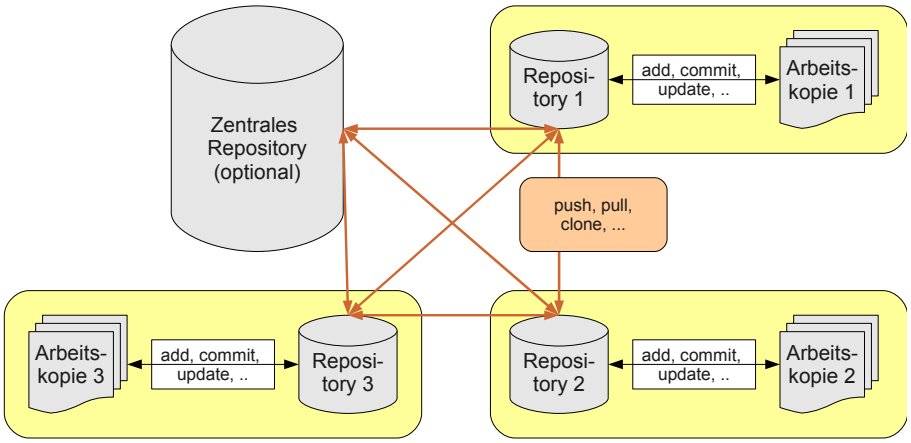


Abbildung 2.6: Dezentrale Versionsverwaltung

Die Erstellung eines Branch erfolgt zunächst in der Arbeitskopie, danach wird der Zweig in das Repository übertragen. Da diese beiden Operationen lokal auf dem Rechner des Benutzers ablaufen, gehen diese schnell von statten. Die Zweige sind somit für kurzfristige lokale Entwicklungen und Tests geeignet. Für längerfristige Entwicklungen bestehen die gleichen Einschränkungen wie bei zentralen Systemen ([O'S09a], [Cha10]).

Anmerkung zur Terminologie Der englischsprachige Begriff für dezentrale Versionsverwaltung ist *distributed version control system*. In deutscher Literatur findet man häufig die Übersetzung *verteiltes Versionsverwaltungssystem*. Nach [TvS07] ist ein verteiltes System *»ein Zusammenschluss unabhängiger Computer, der sich für den Benutzer als ein einzelnes System präsentiert«*. Diese Definition trifft auf die Implementierung von zentralen und dezentralen Versionsverwaltungssystemen zu, so dass man streng genommen von *verteilten zentralen* und *verteilten dezentralen* Versionsverwaltungen sprechen müsste. Folglich ist der Begriff *verteilte Versionsverwaltung* nicht geeignet, um Versionsverwaltungssysteme mit unabhängiger Arbeitskopie und Projektarchiv zu beschreiben [Pod09].

Die dezentrale Architektur erweist sich an manchen Stellen als nachteilig. So können auf Grund der Verteilung der Repository keine linearen Versionsnummern vergeben werden. Stattdessen werden Hashwerte zur Versionsidentifikation verwendet, was den Zugriff auf ältere Revisionen erschwert. Die freie Architektur gestattet jedem Benutzer durch sein eigenes Repository einen eigenen Entwicklungszweig. Diese Vielzahl der Varianten zu einer gemeinsamen Version zusammenzuführen benötigt einen großen organisatorischen Aufwand.

Name	URL
Darcs	http://darcs.net
Git	http://git-scm.com
Mercurial	http://mercurial.selenic.com
Monotone	http://monotone.ca

Tabelle 2.3: Populäre dezentrale Versionsverwaltungssysteme

Risiken beim Zusammenführen

Bei größeren Softwareprojekten kann die Entwicklung in mehrere Zweige aufgeteilt werden. Dies hat den Vorteil, dass die Entwicklung eines Moduls in einer isolierten Umgebung abläuft und so nicht von Fehlern in andern Modulen beeinträchtigt werden kann. Hier besteht allerdings die Gefahr, dass Änderungen an gemeinsamen Dateien vorgenommen werden. Beim Zusammenführen der Zweige können Probleme wie in [O'S09a] erläutert entstehen.

Wenn gemeinsame Dateien in mehreren Branches geändert wurden, muss manuell entschieden werden, welche Teile welcher Versionen in das Resultat einfließt. Komplexer zeigt sich die Konfliktauflösung, falls die gemeinsamen Änderungen Funktionalitätsanpassungen geschuldet sind. Es schleichen sich Fehler bei der Zusammenführung ein. Das Projekt wird in einem inkonsistenten Zustand hinterlassen, so dass die Programmtexte nicht mehr lauffähig übersetzt werden können.

Obwohl Subversion zu den optimistischen Versionsverwaltungen zählt, kann es beim Merge trotzdem zu dem in Abschnitt 2.1.2 beschriebenen Lost Update Problem kommen. In einem Branch verändern zwei Benutzer eine gemeinsame Datei. Der erste Benutzer checkt seine Änderungen in das Repository ein. Nun erhält der zweite Benutzer beim Commit eine Konfliktmeldung und er wird aufgefordert die betroffenen Versionen zusammenzuführen. Schlägt dieser Merge fehl sind die Änderungen des Benutzers verloren, da er seine Änderungen nicht ins Repository einpflegen konnte. Die Versionsverwaltung erlaubt ihm nur auf den letzten eingecheckten Stand zurück zu kehren.

Einige dezentrale Versionsverwaltungssysteme wie *Git* trennen das Abrufen von Änderungen und die Pflicht Konflikte sofort zu lösen. Das Kommando Pull besteht aus zwei Komponenten, *fetch* und *merge* . Fetch holt die Änderungen aus einem Repository und speichert diese in einem temporären Zweig. Danach wird dieser mit den lokalen Quellen zusammengeführt (merge). Durch den Befehl `git fetch` wird nur der temporäre Branch erstellt. Ein Verschmelzen kann später vom Benutzer mittels `git`

`merge` durchführt werden. Das Auslöschungsproblem tritt bei der Verwendung von `fetch` also nicht auf.

2.2 Mergeverfahren

Moderne Versionsverwaltungen verwenden das in Abschnitt 2.1.2 auf Seite 11 vorgestellte Kopieren-Ändern-Zusammenfassen-Modell. Aus der möglichen Nebenläufigkeit von Änderungen ergibt sich die Notwendigkeit verschiedene Versionen einer Datei wieder zu einer zu verschmelzen.

Mergeverfahren kommen ebenfalls bei Softwareproduktlinien zum Einsatz. Dabei werden verschiedene Varianten oder Komponenten eines Programms zu einer speziell auf ein bestimmtes Aufgabengebiet angepassten Variante vereint.

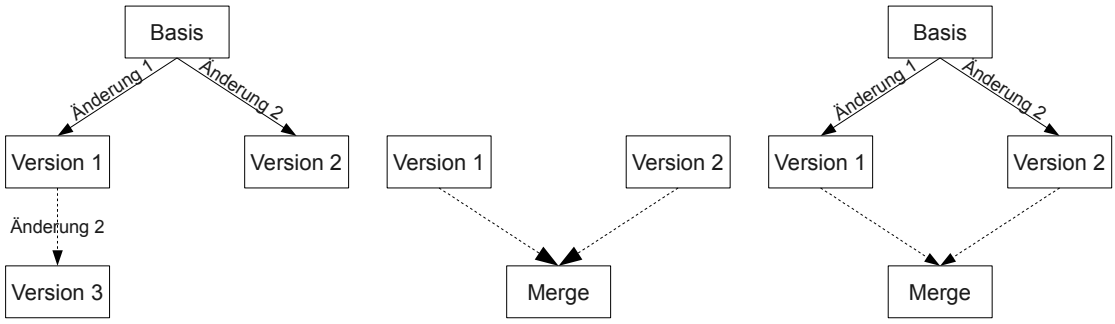
Das Zusammenführen von Dateien wird anhand eines Mergeschemas und mit einem bestimmten Hintergrundwissen um die Beschaffenheit der Quelltexte durchgeführt ([CW98]). Dabei beziehen manche Verfahren die Versionsgeschichte mit ein. Dieser Abschnitt stellt die verschiedenen Mergeschemata vor und betrachtet Zusammenführungsverfahren unterschiedlicher Granularität.

2.2.1 Mergeschemata

In Mergeverfahren werden verschiedene Versionen oder Änderungen kombiniert. Dabei wird unterschieden, welche Dateien beteiligt sind und welche Änderungen in das Endergebnis mit einfließen. Die Schemata lassen sich wie folgt klassifizieren.

Raw Merging

Beim *Raw Merging* (siehe Abb. 2.7.1) werden durch verschiedene Änderungen aus einer Basisdatei zwei Versionen erzeugt. Eine Änderung ist hierbei ein Folge an Operationen. Die Folge, die von der Basisversion zur ersten Version führte, wird nun auf die zweite Version angewandt und eine dritte, beide Änderungen umfassende, Version entsteht. Es treten keine Konflikte auf, da überlappende Änderungen überschrieben werden. Falls die Anwendung einer Änderung (oder einer Operation aus der Sequenz) fehlschlägt, wird die Datei in einem inkonsistenten Zustand hinterlassen [CW98].



2.7.1: Raw Merging **2.7.2:** Two-way Merging **2.7.3:** Three-way Merging

Abbildung 2.7: Verschiedene Mergeschemata (nach [CW98])

Two-way Merging

Im Two-way Merging (zweiseitigen Vergleich, siehe Abb. 2.7.2) werden Version 1 und Version 2 miteinander verglichen und einer gemeinsamen zusammengeführt (Merge). In dieser Datei werden dem Nutzer die Unterschiede präsentiert und er wählt jeweils eine Alternative aus. Dieses Verfahren erlaubt keine automatische Integration der Änderungen, ebenso kann nicht entschieden werden, wie die einzelnen Unterschiede aus einer Vorgängerversion entstanden sind. Es bleibt unklar, ob diese aus Hinzufügen, Änderungen oder Entfernen von Zeilen an einer oder an beiden Versionen hervorgegangen sind ([Men02]).

Three-way Merging

Der Three-way Merge (dreiseitiger Vergleich, siehe Abb. 2.7.3) bildet unter Berücksichtigung einer Vorgängerversion (Basis) aus den beiden Versionen 1 und 2 einen Nachfolger (Merge). Im Gegensatz zum Two-way Merging lassen sich mit diesem Verfahren Änderungen zum Teil automatisch in die gemeinsame Version übernehmen. Sind die Unterschiede der Dateien disjunkt, so werden diese ohne Rückfrage in den Nachfolger kombiniert. Finden sich nicht-disjunkte Unterschiede, so entspricht dies einer Konfliktsituation (vgl. 2.1.2), die durch weitere Verarbeitung gelöst werden muss.

Tabelle 2.4[2] zeigt alle möglichen Konstellationen, die bei einem Drei-Wege-Merge auftreten können.

[2]Revctrl Wiki: `http://revctrl.org`

Konstellation	Basis	Version 1	Version 2	Ergebnis
Keine Änderung	A	A	A	A
Änderung in Version 1	A	B	A	B
Änderung in Version 2	A	A	B	B
Änderung beiden Versionen	A	B	C	**Konflikt**

Tabelle 2.4: Mergekonstellationen nach Revctrl Wiki

In Versionsverwaltungssystemen werden Three-way Merges angewandt. Das System hat Zugriff auf die Dateihistorie und kann zu zwei Dateien stets den gemeinsamen Vorfahren finden. .

2.2.2 Versionshistorie

Je nach Verfahren wird beim Zusammenführen die Versionsgeschichte einer Datei betrachtet. Wird ein Merge anhand von Revisionen ohne Kenntnis des Versionsverlaufs durchgeführt, so handelt es sich um ein status-basiertes (engl. *state-based*) Verfahren. Betrachtet man zusätzlich zu den Revisionen die Entstehungsgeschichte, d.h. die Abfolge an Operationen, die zu der jeweiligen Revision führt, spricht man von einem änderungs-basierten (engl. *change-based*) Verfahren.

2.2.3 Granularität

Die Mergeschemata können auf unterschiedliche semantische Ebenen angewandt werden. Als semantische Ebene wird nach [CW98] das Wissen des jeweiligen Mergeprogramms um seine Einsatzumgebung, Versionsverwaltung und Programmiersprache, bezeichnet.

Unstrukturierter Merge

Verfahren, die nicht die Struktur einer Programmiersprache beim Verschmelzen zu Rate ziehen, werden als *unstrukturiert* bezeichnet.

Ausschneiden-und-Einfügen Merging Bei diesem Verfahren werden keine Mergeprogramme eingesetzt. Ein Programmierer selektiert hier händisch die Programmteile zweier Dateiversionen und fügt diese nach eigenem Ermessen zu einer gemeinsamen Version zusammen ([Buf95]). Damit könnte man das Verfahren auch als manuellen Two-way Merge bezeichnen.

Text-Orientiertes Merging Bei *zeilenbasierten* text-orientierten Merging wird angenommen, dass jede Dateiversion eine Menge an Zeilen umfasst. Dabei ist eine Zeile die kleinste, unteilbare Einheit. Zwei Zeilen sind identisch oder sie sind verschieden. Es werden nur ganze Zeilen ersetzt, verschiedene Zeilen werden nie zu einer einzigen Zeile vereint. In diesem Zusammenhang ist ein Konflikt, wenn bei einem Three-way Merge in allen drei Revisionen die gleiche Zeile bearbeitet wurde. Ein Konflikt ist automatisch auflösbar, wenn die Änderungen an verschiedenen Stellen in der Datei vorgenommen und wenn die Änderungen nicht ineinander verschachtelt sind.

Diese Verfahren zeigen in der Praxis gute Resultate. Nach [Men02] lassen sich 90% der Änderungen automatisch mit einem zeilenbasierten textuellen Merge lösen. Weiter sind sie unabhängig von der verwendeten Programmiersprache und können mit allen Textdateien umgehen. Daher ist diese Art weit verbreitet und in alle gängigen Versionsverwaltungssysteme integriert. Entsprechende Three-way Mergeprogramme waren bereits in die populären Versionsverwaltungen RCS (2.1.3), CVS ([BIB90]) und Subversion ([CSFP10]) eingebaut und dadurch eine breite Nutzerbasis gefunden.

In RCS wurde das Tool `rcsmerge` und dessen standalone Version `GNU Merge` für Three-way Merges verwendet [Her99]. Die aktuelle Version von `GNU Merge` verwendet zum zeilenweisen Vergleich die Tools `diff3` und `diff` aus den `GNU diffutils`[Tic95]. `diff` gibt die Unterschiede zweier Dateien als Liste minimaler Zeilenänderung, um die eine Datei in die andere zu überführen[3][HM76]. `diff3` leistet analoges für drei Dateien.

Um den in Abbildung 2.7.3 dargestellten Drei-Wege-Merge durchzuführen, wird GNU merge wie folgt aufgerufen:

```
merge Version1 Basis Version2
```

Dabei werden die Änderungen die von `Basis` zu `Version1` geführt haben, `Version2` hinzugefügt. Treten Konflikte auf, so werden diese wie in Listing 2.1 in der zusammenführten Datei `Version2` dargestellt. Der Benutzer muss diese Textstellen editieren, um den Konflikt zu lösen.

[3] Das zugrunde liegende Problem ist das Finden der größten gemeinsamen Teilsequenz zweier Dateien.

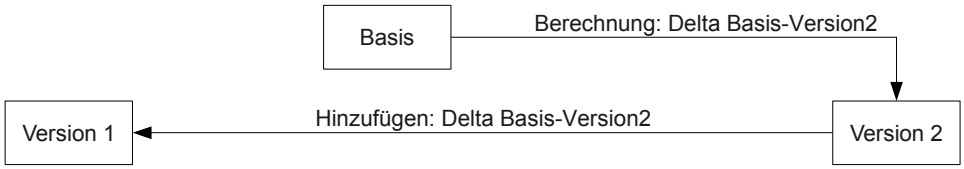


Abbildung 2.8: Zusammenführung mit GNU Merge

```
1 <<<<<<< Version1
2 Überlappende Zeilen aus Datei Version1
3 =======
4 Überlappende Zeilen aus Datei Version2
5 >>>>>>> Version2
```

Listing 2.1: Konfliktdarstellung in GNU Merge

Zur Zusammenführung ruft `merge diff3` auf. Dieses verwendet zunächst `diff`, welches maximale Überdeckungen zwischen `Basis` und `Version1` und zwischen `Basis` und `Version2` findet. Anschließend werden die Bereiche, in denen sich `Basis` von `Version1` oder `Version2` unterscheidet bei Überlappung zusammengefasst und gegenübergestellt. Schließlich werden nach den in Tabelle 2.4 auf Seite 19 dargestellten Regeln die 3-Wege-Unterschiede berechnet [KKP07], [HM76]. `Merge` greift diese Unterschiede auf und schreibt das Ergebnis in `Version1`.

[KKP07] zeigt einige Schwächen des `diff3`-Tools, so können Konflikte bei nicht überlappenden Änderungen konstruiert werden. Zudem nimmt `diff` für eine bessere Laufzeit, $O(N^{1.5} \log N)$ statt $O(NM)$, nicht optimale Ergebnisse für große Eingaben mit vielen Unterschieden in Kauf [S+10]. Im folgenden wird ein Beispiel für einen unnötigen Konflikt dargestellt. Tabelle 2.5 illustriert die Ausgangssituation. Datei `Basis` besteht aus einer $n+1$-stelligen Folge von 1 und 2 ($(1,2)^{n+1}$: aufgeteilt in Mitteil $(1,2)^n$ und einen Endteil 1, 2). In `Version 1` wird `Basis` zu einer $n+2$-stelligen Folge erweitert, in dem vorne 1, 2 eingefügt wird. In `Version 2` wird der Endteil zu 3 abgewandelt.

Version 1	1, 2	$(1,2)^n$	1, 2
Basis		$(1,2)^n$	1, 2
Version 2		$(1,2)^n$	3

Tabelle 2.5: Ausgangssituation für unnötigen Diff3-Konflikt

Diese drei Versionen werden nun mit `diff3` auf Überdeckungen untersucht. Im Ergebnis wird ein Konflikt am Endteil angezeigt.

Version 1	$(1,2)^n$	1,2,1,2
Basis	$(1,2)^n$	1,2
Version 2	$(1,2)^n$	3
		Konflikt

Tabelle 2.6: Mergeergebnis von `diff3`

Der Konflikt, dargestellt in 2.6, entsteht, obwohl die Dateiversionen an jeweils disjunkten Stellen bearbeitet wurden. Nach den in Tabelle 2.4 auf Seite 19 aufgestellten Regeln, hätten die Dateien konfliktfrei verschmolzen werden können.

Version1	1,2	$(1,2)^n$	1,2
Basis		$(1,2)^n$	1,2
Version2		$(1,2)^n$	3

Tabelle 2.7: Konfliktfreier Merge

Durch den zeilenbasierten Dateivergleich können nur physikalische Konflikte entdeckt werden. Logische Konflikte bleiben unentdeckt. So kann beispielsweise die Umbenennung einer Variable zu Konflikten führen, obwohl die Namensänderung keinerlei Verhaltensänderung des Programms verursacht. Ebenfalls kann die automatische Kombination von disjunkt geänderten Programmteilen ein nicht lauffähiges Ergebnis erzeugen.

Neben zeilenorientierten Verfahren erwähnt [Buf95] *kontextorientierte* Verfahren. Varianten werden hier als Menge von Zeichen betrachtet. Die kleinste, unteilbare Einheit ist somit das einzelne Zeichen. Im Verfahren zeichnet man den Verlauf von Modifikationen an Dateien in einer Änderungsdatei auf. Zu jeder Änderung werden der ursprüngliche Text und der eingefügte Text gespeichert. Zur Durchführung eines Merge werden die Änderungsdateien zweier Versionen nacheinander auf die Ursprungsdatei angewendet. Ein Konflikt tritt auf, wenn eine Änderung (z. B. wegen bereits gelöschter Textstellen) nicht angewendet werden kann. Änderungsdateien kapseln Änderungen an bestimmten Textstellen. Diese Stellen werden durch textuellen Vergleich gefunden. Daher ist dieser Ansatz toleranter gegenüber Änderungen an der Originaldatei als der zeilenbasierte Three-way Merge. Das kontextorientierte Verfahren ist eine Anwendung des Raw Merging Schemas (siehe 2.2.1).

Strukturierter Merge

Beim strukturierten Merge werden die beteiligten Programmvarianten anhand ihrer strukturellen Elemente analysiert und verschmolzen. Die syntaktische Ebene kann dabei eine Basis bilden. Dabei wird jede an einem Merge beteiligte Variante geparst und in syntaktische Regionen aufgeteilt. Diese Regionen der einzelnen Varianten werden anschließend miteinander verschmolzen.

Zusammenführung auf Deklarationsebene In [Buf95] ist ein speziell für die Programmiersprache C implementierter Ansatz beschrieben. Der Quelltext eines Programms wird im ersten Schritt in einzelne Teile, sog. *name declarations* aufgeteilt. Ein Teil umfasst dabei eine Variablen- oder eine Funktion mit Deklaration und Rumpf. Diese Deklarationen werden in einer externen Tabelle festgehalten. Zusätzlich speichert diese Namen, Typ , Inhalt und die Zeilenposition der Deklaration im Quelltext.

Beim Verschmelzen von drei Varianten, wie in 2.7.3 auf Seite 18 dargestellt, werden zunächst die Quelltexte in oben genannte externe Tabellen aufgeteilt. Anschließend untersucht man die Namensmenge der Tabellen, um hinzugefügte und entfernte Deklarationen zu erkennen. Wurde eine Deklaration in nur einer Variante hinzugefügt oder entfernt, so wird diese in das Endergebnis übernommen. Kommt ein Deklarationsname in mindestens zwei Tabellen vor, vergleicht man die Deklarationen textuell. Bei Übereinstimmung fließt die Deklaration ins Ergebnis ein, bei textuellen Unterschieden werden beide Varianten und die Basis mit dem in Abschnitt 2.2.3 beschriebenen `GNU Merge`-Tool verschmolzen.

Analyse von Programmevolution anhand ihrer Abstrakten Syntaxbäume Einen auf Funktionsebene arbeitenden Ansatz beschreibt [NFH05]. Hier wird Programmevolution anhand von Abstrakten Syntaxbäumen nachvollzogen. Ein Abstrakter Syntaxbaum entsteht als Zwischenschritt bei der Programmübersetzung.

Die Übersetzung von Quelltext zu einem lauffähigen Programm wird von einem Compiler ausgeführt. Dabei teilt sich der Prozess in mehrere Schritte auf. Zunächst wird der Programmtext, der als Aneinanderreihung von Zeichen vorliegt, von einem Lexer (oder Scanner) untersucht und in Wörter (Token) eingeteilt. Zusätzlich wird sichergestellt, dass die erkannten Wörter im Wortschatz der Programmiersprache vorkommen bzw. gültig sind. Im nächsten Schritt setzt ein Parser diese Wörter zu Sätzen zusammen und überprüft anhand der Grammatik der Programmsprache, ob diese zulässig sind. Dazu wird aus dem Satz durch Anwendung der Grammatikregeln ein Parsebaum (oder konkreter Syntaxbaum) aufgebaut. Dies gelingt nur, wenn der Quelltext syntaktisch

korrekt ist. Da ein Parsebaum viele grammatikalische Informationen ohne semantischen Gehalt trägt, die für die weiteren Compilierschritte nicht von Nöten sind, wird dieser zu einem Abstrakten Syntaxbaum zusammengefasst. Aus diesem Baum wird durch Traversieren Zwischencode erzeugt, der nach Optimierung in ein lauffähiges Programm umgewandelt wird [ALSU06].

Der Unterschied zweier Programmvarianten kann mit Hilfe des Tools `diff` angezeigt werden. Da dieses nur textuelle Vergleiche anstellt, kann es zu feingranular sein. Daher vergleicht [NFH05] zwei Programmvarianten anhand der Abstrakten Syntaxbäume der einzelnen Funktionen. Damit ist es möglich Programmentwicklungen von textuellen Änderungen zu abstrahieren und einem aussagekräftigeren, semantischen Kontext zu sehen. Ein Vergleich der Bäume auf Funktionsebene ist ausreichend, da Änderungen häufig »lokal« sind. Funktionsnamen ändern sich in der Regel nicht, die Rümpfe hingegen häufig. Funktionsaddition ist häufig und Funktionen werden selten gelöscht. Somit werden zwei Funktionen mit einander verglichen, falls ihre Namen übereinstimmen. Zwei Variablen werden als gleich erachtet, wenn sie an der selben syntaktischen Position in beiden Varianten steht. Dabei wird zwischen lokalen und globalen Variablen unterschieden. Für alle Variablen wird versucht eine Zuordnung zwischen beiden Varianten zu finden. Eine fehlende Zuordnung deutet auf deine Funktionalitätsänderung hin. Diese wird so dann angezeigt.

Merge auf Parsebaumebene Ein Syntaxbaum bildet ein Programm genau ab. Daher bietet er sich als Ausgangspunkt für Zusammenführungsalgorithmen an. In [Yan94] wird ein Ansatz beschrieben, der zwei Programmvarianten anhand ihrer Parsebäume zusammenführt. Ein Three-way-Merge kann indirekt durch drei Vergleiche erreicht werden.

Um zwei Programmvarianten zu verschmelzen, werden die Quelltexte in Parsebäume überführt. Anschließend vereint ein Tree-Matching Algorithmus die Knoten der Bäume. Zwei Knoten werden genau dann zusammengeführt, wenn sie identische oder ähnliche grammatikalische Elemente, extra-grammatikalische Elemente oder nicht-terminale Symbole enthalten. Der Algorithmus versucht eine maximale Anzahl von Knoten zu kombinieren, so dass die Eltern-Kind-Knoten Beziehung und die Ordnung unter den Blättern erhalten bleibt. Dabei können nur gleiche Ebenen der Bäume matchen und es können nur Einfügungen und Löschungen von Knoten oder Unterbäumen entdeckt werden. Verschiebung von Knoten oder Unterbäumen wird nicht entdeckt. Aus dem Ergebnis des Matching-Algorithmus entsteht eine Ausgabedatei, die der Benutzer weiter bearbeiten muss.

Da das Überführen von Quelltext in einen Parsebaum abhängig von der verwendeten Programmiersprache ist, erreicht man größere Freiheiten, in dem man die sprachabhängige Analyse (Lexer und Parser) und die sprachunabhängige Zusammenführung (Tree-Match-Algorithmus und Ergebniskonstruktion) von einander trennt.

Durch eine Verschmelzung von Programmvarianten anhand ihrer Syntaxbäume können wieder syntaktisch korrekte Programme erzeugt werden. Dennoch ist nicht garantiert, dass sich das Ergebnis wiederum ein lauffähiges Programm übersetzen lässt. So können beispielsweise undeklarierte Variablen oder undefinierte Funktionsaufrufe eingeführt werden. Diese Art von Konflikten nennt man *statische semantische Konflikte*, die erst in einem späteren Kompilierschritt entdeckt werden können. In Syntaxbäumen verbergen sie sich, da dort keine Zuordnung von Funktionsdeklaration und Funktionsaufruf besteht [Men02].

Merge mit Abstrakten Syntaxbäumen und Versionshistorie Um diese Art der Konflikte auszuräumen, verfolgt [Wes91] Änderungen in einer speziellen Entwicklungsumgebung mit einem integrierten Versionsverwaltungssystem und reichert Abstrakte Syntaxbäume mit kontext-sensitiven Informationen an. Dabei werden Variablen mit ihrer Deklaration verknüpft. Der Ansatz ist generisch, da die programmiersprachenspezifische Komponente, eine normalisierte EBNF-Grammatik, zum Aufbau der Abstrakten Syntaxbäume, austauschbar und erweiterbar ist. Der Mergealgorithmus arbeitet schließlich auf diesen Bäumen.

Da der Algorithmus durch die Versionsverwaltung änderungsbasiert funktioniert, kann die Entwicklung von einer Basisversion zu ihren Varianten jederzeit nachvollzogen werden. Somit stehen diese Informationen auch beim Zusammenführen von zwei Varianten zur Verfügung und so wird stets ein Three-way-merge durchgeführt. Der Merge operiert auf Ebene der Abstrakten Syntaxbäume und beachtet die in Tabelle 2.4 aufgestellten Regeln. Durch die gesammelten kontext-sensitiven Informationen ist das Ergebnis syntaktisch korrekt und lässt sich in ein lauffähiges Programm übersetzen.

Trotz der Behandlung von statischen semantischen Konflikten können im Mergeergebnis noch Verhaltenskonflikte auftreten. Diese Konflikte können nicht zur Übersetzungszeit, sondern erst zu Laufzeit entdeckt werden. Ein Verhaltenskonflikt tritt beispielsweise auf, wenn eine Unterklasse die Methode einer Oberklasse verwendet, diese jedoch unerwartet bearbeitet wird und von nun an andere Ergebnisse liefert.

Erkennung von Verhaltenskonflikten Um derartige Konflikte zu erkennen muss das Laufzeitverhalten des gemergten Programms analysiert und verglichen werden. Da eine derartige Analyse für ein komplettes sehr komplex und aufwändig ist, führen [JL94] eine semantische Analyse auf Funktionsebene durch. Dabei werden die Funktionen von zwei Versionen eines Programms anhand ihres Eingabe-Ausgabe-Verhaltens analysiert. Verhalten sich zwei Funktionen unterschiedlich wird dies gemeldet, da dann ein potentieller Verhaltenskonflikt vorliegt.

Semistrukturierter Merge

Vergleicht man die vorgestellten Verfahren zu unstrukturierten und strukturierten Merges, so zeigen sich die textuellen Methoden als sehr allgemein und sprachunabhängig. Dafür können sie keine Konflikte lösen, die Kenntnis von der konkret verwendeten Programmiersprache fordern. Hingegen sind strukturierte Methoden sehr speziell und obwohl sie Schnittstellen für Erweiterungen bieten, wurden Prototypen nur für jeweils eine Programmiersprache implementiert.

Die strukturierten Mergeverfahren arbeiten mit Bäumen als Datenstrukturen. Konkret genannt sind dies Parsebäume und Abstrakte Syntaxbäume. Sie bilden die Struktur der beteiligten Programme ab und eignen sich zur weiteren Verarbeitung im Mergeprozess. Beim Verschmelzen dieser Bäume können semantische und Verhaltenskonflikte in das resultierende Programm eingeführt werden. Diese Fehler können nicht durch die Mergealgorithmen, sondern erst durch den Compiler in einem späteren Übersetzungsschritt oder durch den Benutzer zur Laufzeit entdeckt werden. Eine automatische Erkennung dieser Konfliktarten erfordert zusätzliche Informationen in den Bäumen sowie den Einsatz externer Tools. Dies macht die Verfahren kompliziert. Weiterhin muss letztendlich der Benutzer diese Konflikte bewerten und lösen.

Die Lücke zwischen unstrukturierter und strukturierter Konfliktbehandlung wird durch den semistrukturierten Ansatz gefüllt. Dabei überführt ein programmiersprachenabhängiger Parser ein Programm zunächst in eine Baumdatenstruktur. Diese Aufgliederung ist so feingranular, dass Programmelemente, die sich frei im Quelltext anordnen lassen, erkannt werden. Anschließend behandelt der programmiersprachenunabhängige Mergealgorithmus auftretende Ordnungskonflikte und nutzt in ausgewählten Fällen das textuelle Verfahren. Dieser Ansatz wird semistrukturiert genannt, da er ausgewählte strukturelle Aspekte beim Mergevorgang berücksichtigt und auf unstrukturierte Mergestrategien zurückgreifen kann ([ALL+10]).

Program Structure Trees Dreh- und Angelpunkt des semistrukturierten Merge sind *Program Structure Trees*(PSTs) oder *Feature Structure Trees*(FSTs), die die beteiligten Programmvarianten darstellen. Diese Bäume entstammen dem feature-orientierten Programmierparadigma.

Ein Feature ist »eine Struktur, die die Struktur eines gegebenen Programms erweitert oder modifiziert, um eine Anforderung zu erfüllen, eine Entwurfsentscheidung zu implementieren und um eine Konfigurationsoption anzubieten.« Damit kann ein Feature ein ausführbares Programm oder ein Funktionszuwachs eines Programms sein. In letzterem Fall werden Features zusammengesetzt, um ein Programm zu bilden [ALMK08].

Ein aus *Basis-Features* zusammengesetztes Programm wird in einem FST dargestellt. Dieser geordnete Wurzelbaum bildet dabei die Struktur des Programms hierarchisch ab. Dabei besteht ein Basisfeature aus zwei Komponenten, einen Typ, der die strukturelle Ebene widerspiegelt und einem Namen.

Über *Modifikationen* können Basis-Features verändert werden. Ein *Full-Feature* kapselt beide beteiligten Komponenten in einem Tupel. Ein solches Tupel wird *Quark* genannt. Beim semistrukturierten Merge finden Full-Features keine Anwendung. Im weiteren Verlauf werden daher die Begriffe Basis-Feature und Feature synonym verwendet.

Ein FST wird aus zwei Arten von Knoten zusammengesetzt: *Nicht-terminale Knoten* bilden die inneren Knoten eines Baumes. Sie enthalten ein Liste ihrer Kindknoten und bilden somit die Struktur eines Features. Ein nicht-terminaler Knoten hat einen Namen und einen Typ. Die Blätter des Baumes sind *terminale Knoten*. Deren Unterstruktur bzw. deren Inhalt wird nicht im FST dargestellt. Ein solcher Knoten hat einen Namen, einen Typ und einen Inhalt.

Ein Beispiel soll den Aufbau und die Zuordnung von Quellcode zu FST verdeutlichen. Listing 2.2 zeigt eine Implementierung eines Stacks in der Programmiersprache Java. Die Graphik 2.9 zeigt den korrespondierenden FST. Nichtterminale Knoten des Baumes werden als runde innere Knoten dargestellt, terminale Knoten als eckige Blattknoten. Die inneren Knoten geben die hierarchische Struktur des Programms wieder und stellen in Java `packages`, Klassen und Interfaces dar. Die Blattknoten repräsentieren Methoden, Felder und Listen von `extends`, `implements` und `throw` Klauseln.

Im Gegensatz zu einem Abstrakten Syntaxbaum enthält ein Feature Structure Tree nur Elemente, die für eine Weiterverarbeitung notwendig sind. So werden Java-Methoden als terminale Knoten repräsentiert. Diese Knoten identifizieren sich über die Signatur

```java
package datastructure;

import java.util.LinkedList;

public class Stack<T> {

  private LinkedList<T> content = new
      LinkedList<T>();

  public void push(T item) {
    items.addFirst(item);
  }

  public T pop() {
    return content.removeFirst();
  }

}
```

Listing 2.2: Javaimplementierung eines Stack

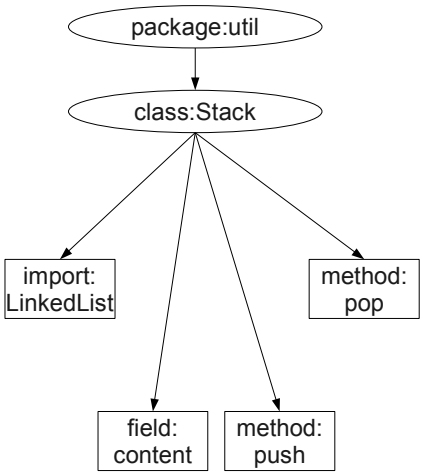


Abbildung 2.9: FST des Stack

der Methode. Der Rumpf der Methode, bestehend aus Reihenfolge abhängigen Anweisungen, wird nicht weiter aufgegliedert. Er ist als Text im Inhalt des Methodenknotens gespeichert.

Superimposition von Program Structure Trees Die Kombination zweier Feature Structure Trees wird nach [AL08] *Superimposition* genannt. Zwei Bäume werden mit einander zu einem neuen Baum kombiniert, in dem man rekursiv beginnend bei der Wurzel die Knoten ebenenweise vereint. Zwei Knoten werden miteinander kombiniert, falls sie in Name und Typ übereinstimmen und falls ihre Elternknoten miteinander kombiniert wurden. Der Ergebnisknoten erhält Namen und Typ der Ausgangsknoten. Falls ein Knoten keinen Gegenpart in dem anderen Baum hat, so wird dieser Knoten als Kindknoten an den zusammengesetzten Elternknoten angefügt.

Da nicht-terminale Knoten nur Name und Typ beinhalten, ist ihre Kombination problemlos möglich. Terminale Knoten enthalten neben Namen und Typ zusätzlichen Inhalt. Um diesen zu vereinen werden spezifische Regeln benötigt oder die Komposition muss abgebrochen werden.

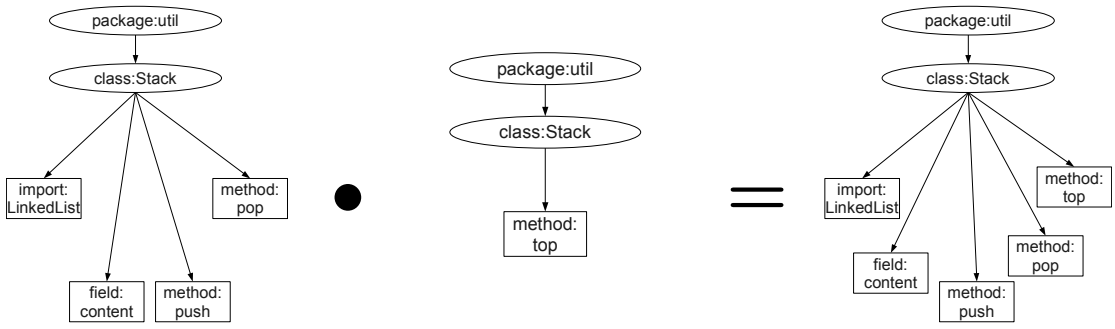
Für die Programmiersprache Java gelten folgende Regeln für die Kombination von terminalen Knoten:

- Zwei Methoden werden kombiniert, falls ihre Rümpfe kombiniert werden können.

- Zwei Klassenattribute werden kombiniert, falls ihre Werte ausgetauscht werden sollen oder eines einen Wert zugewiesen hat und eines nicht.
- Zwei `extends`-Anweisungen werden kombiniert, falls ihre Namen übereinstimmen, sonst Fehler
- Zwei Sichtbarkeits-Modifier werden kombiniert, dass die Sichtbarkeit erhöht wird. `public` ersetzt bspw. `private`.
- Zwei `import`-Anweisungen werden kombiniert, in dem die Einträge aneinandergereiht und doppelte Einträge entfernt werden.
- Zwei `throws`-Anweisungen werden kombiniert, in dem die Einträge aneinandergereiht und doppelte Einträge entfernt werden.
- Zwei `implements`-Listen werden kombiniert, in dem die Einträge aneinandergereiht und doppelte Einträge entfernt werden.

Sprachunabhängige Regeln zur Kombination von FSTs bzw. zur Kombination von Features führt [ALMK08] ein. Die *Feature Algebra* beschreibt auf formaler Ebene Komposition und Modifikation von Features. Dabei erfüllt Komposition von Features die algebraischen Eigenschaften der Assoziativität (Klammerung hat keine Auswirkung auf das Endergebnis), Identität (Komposition mit einem leeren Baum ist neutral), Nicht-Kommutativität (im Allgemeinen spielt die Reihenfolge der Blattknoten eine Rolle) und Idempotenz (ein Feature wird nur einmal in das Ergebnis übernommen).

So kann mit Hilfe dieser Regeln der in Abbildung 2.9 einfach um ein Feature `top` erweitert werden. Dabei kapselt das Feature die entsprechende Methode, die das oberste Element des Stacks liefert, ohne es zu entfernen. Die Komposition zweier Features notiert der Operator •.



Basis-Stack Top-Erweiterung Top-Stack

Abbildung 2.10: Superimposition

Merge mit Program Structure Trees Superimposition kann auch zum Verschmelzen verschiedener Programmvarianten verwendet werden, indem die Operation sukzessive auf die beteiligten Varianten angewendet wird. Abbildung 2.11 auf Seite 31 illustriert dies anhand eines Beispiels. Ausgangsversion des Beispiels ist das bereits bekannte Programm `Basisstack`. Dieses wird verzweigt und zu den beiden Varianten `Top-Stack` mit der zusätzlichen Methode `top()` und `Size-Stack` mit der zusätzlichen Methode `size()` erweitert. Mittels der Regeln der Feature Algebra können die drei Varianten konfliktfrei zum Programm `Merged-Stack` vereinigt werden.

Superimposition ist ein Verfahren zur Komposition von Programmelementen. Bei einem Merge können aber nicht immer alle Baumknoten in das Ergebnis übernommen werden. Vielmehr müssen zusätzlich die Mergekonstellationen, wie in Tabelle 2.4 auf Seite 19 dargestellt, auf Ebene der Knoten berücksichtigt werden. So kann die Löschung von Elementen behandelt werden. Wird beispielsweise ein Blattknoten in einer abgeleiteten Variante entfernt und bleibt in der anderen Variante unverändert, so muss er gemäß den Mergeregeln aus dem Ergebnis entfernt werden.

Das Vorgehen beim semistrukturierten Merge erfolgt also nach dem gleichen Vorgehen wie beim unstrukturierten Merge. Der Unterschied ist, dass beim Abgleich nicht auf Zeilenebene, sondern auf Knotenebene gearbeitet wird. Zusätzlich kann zu jedem Knotentyp eine spezielle Konfliktbehandlungsmethode hinterlegt werden. Tritt zwischen Knoten ein Konflikt auf und es ist zu dem Knotentyp nicht das zeilenbasierte Verfahren als Konfliktbehandlungsmethode hinterlegt, so spricht man von einem *semantischen Konflikt*. Über Konfliktbehandlungsmethoden auf weitere Spracheigenschaften eingegangen werden. So lässt sich beispielsweise die Aufzählungseigenschaft bei der Verschmelzung von `implements`–Listen nutzen.

Weiter können Reihenfolge abhängige Sprachelemente verarbeitet werden, wenn der entsprechenden Knotentyp mit dem zeilenbasierten Verfahren verknüpft wird. Für die Behandlung von Java-Methoden bedeutet dies, dass die Knoten über die Signatur in den Bäumen identifiziert werden. Ist die Methode in mehreren Bäumen vorhanden, so erfolgt die Verschmelzung der Rümpfe textuell. Erzeugt der Methodenrumpfmerge einen Konflikt, so spricht man von einem *Ordnungskonflikt*, da dieser durch die Abfolge der Anweisungen verursacht wird.

Durch die Behandlung von Reihenfolge abhängigen Programmelementen mit dem unstrukturierten Verfahren, können Probleme strukturierter Konfliktbehandlung wie Verhaltenskonflikte vermieden werden ([ALL+10]).

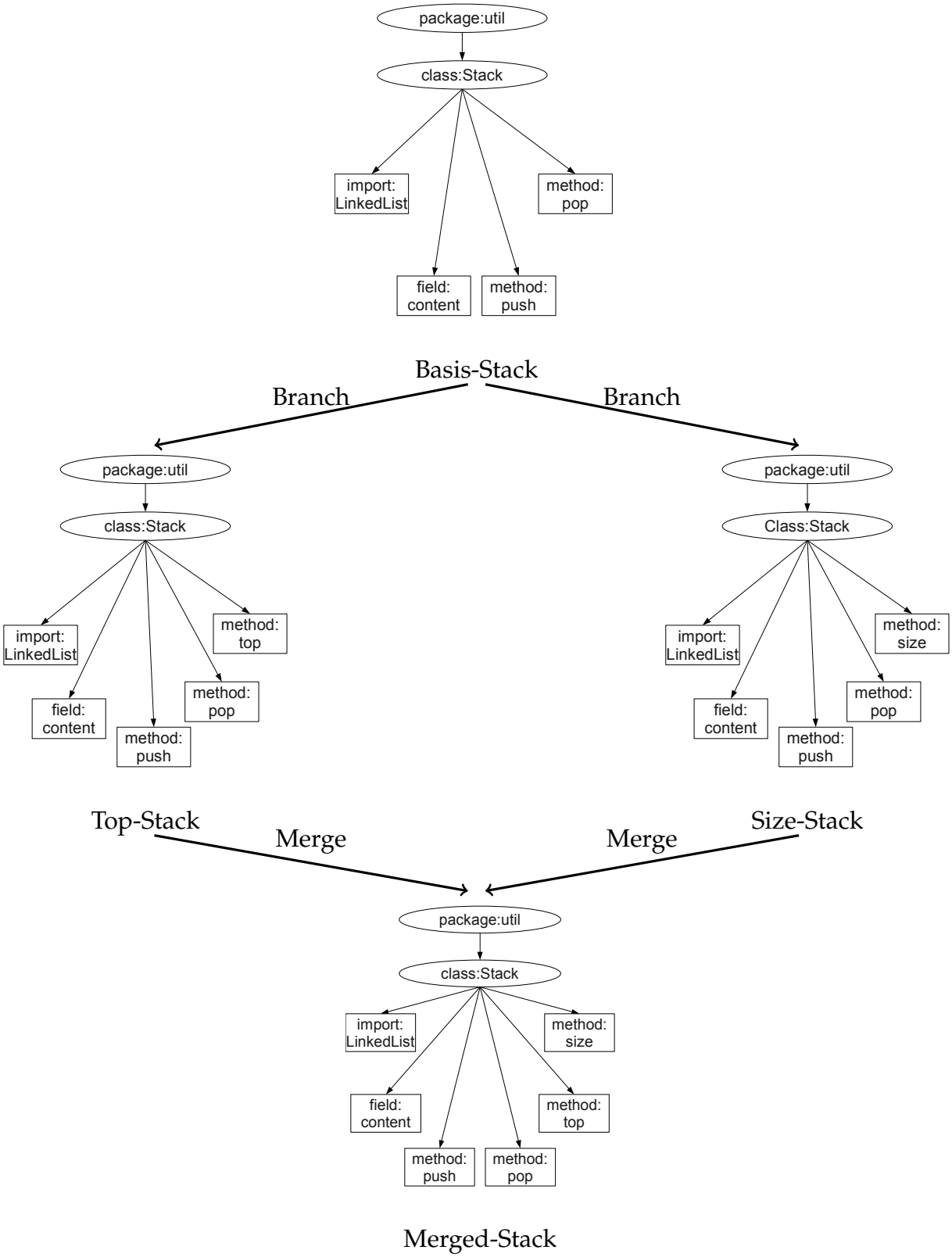


Abbildung 2.11: Merge durch Superimposition

Überführung von Quelltext zu Program Structure Trees und Festlegung des Konfliktbehandlungsverfahrens Um aus Quelltext einen Program Structure Tree zu generieren, wird auf eine annotierte Grammatik in *FeatureBNF*-Form der jeweiligen Programmiersprache zurückgegriffen. In dieser Grammatik ist beschrieben, welchen Sprachelementen welcher Knotentyp, terminal oder nicht-terminaler Knoten, zugeordnet wird. Dabei ist die Grammatik so genau, dass durch die Aufteilung Ordnungskonflikte behandelt werden können.

Zusätzlich kann zu den Programmelementen eine spezifisches Konfliktbehandlungsverfahren hinterlegt werden. Ohne dessen Angabe werden die Elemente in einem Drei-Wege-Merge verarbeitet. Ist hingegen bekannt, dass eine Element Bestandteil einer Aufzählung sein kann, kann dieses Element über einen Listenhandler verschmolzen werden. Dieser erhält als Eingabe die Auszählungselemente der beteiligten Bäume und versucht diese aneinander zu reihen.

Durch die Verwendung von verschiedenen Konfliktbehandlungsverfahren kann auf individuelle Besonderheiten der jeweiligen Programmiersprache eingegangen werden. Damit nutzt semistrukturierte Konfliktbehandlung Methoden unstrukturierter und strukturierter Verschmelzungstechniken.

Liegt für die zu behandelnde Sprache keine Grammatik vor, so wird zur Verschmelzung das unstrukturierte Verfahren verwendet. In diesem Fall verzichtet das Verfahren auf die Erstellung der Baumstruktur.

Überführung von Program Structure Trees zu Quelltext Das Ergebnis eines Drei-Wege-Merge auf PSTs ist wiederum ein PST. Dieser besteht aus den zusammengeführten Elementen. Ein sog. Pretty Printer traversiert über den Baum generiert daraus den resultierenden Quelltext. Dabei reiht er die Baumknoten an syntaktisch zulässiger Stelle an. Dabei hinterlegt er die aufgetreten Konflikte an entsprechender Stelle.

FeatureHouse Diese Ansätze und Schritte zur Softwarekomposition ([AKL09]) und zur Verschmelzung von Varianten mittels Feature Structure Trees ([ALL+10]) wurde in *FeatureHouse* implementiert. Abbildung 2.12 stellt dessen Architektur dar.

Die annotierte Grammatik beschreibt die Elemente der jeweiligen Programmiersprache. Zusätzlich kann für jedes dieser Bestandteile eine spezielle Mergevorgehensweise hinterlegt werden. Diese Grammatik ist dabei der einzige sprachabhängige Bestandteil des Verfahrens. Auf dieser Basis erzeugt *FSTGenerator* einen Parser für die Sprache. Dieser Parser überführt die am Merge beteiligten Programmvarianten in die entsprechenden PSTs. Anschließend werden die Bestandteile der Bäume mit der hinterlegen Strategie

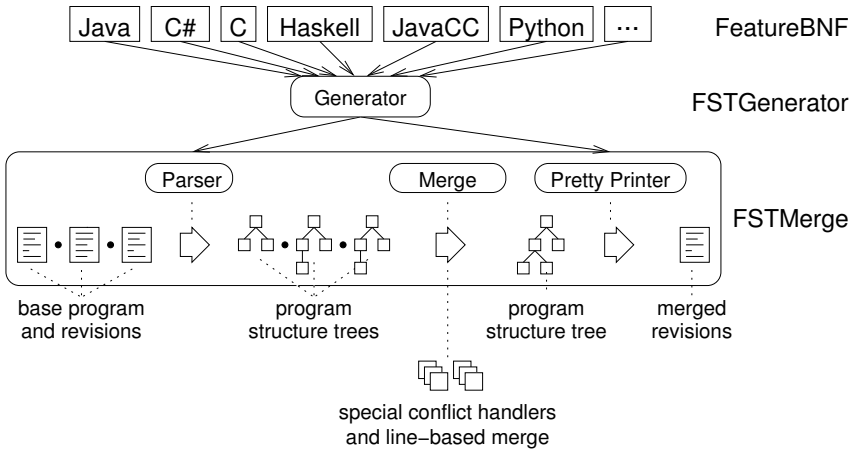


Abbildung 2.12: FeatureHouse-Architektur (nach [ALL+10])

zu einem gemeinsamen PST verschmolzen. Dieses Resultat transformiert ein *Pretty Printer* aus der Baumstruktur in gemergten Quelltext. Eventuell verbleibende Konflikte müssen anschließend von Benutzerseite behoben werden.

Kapitel 3

Experimentplanung

Das vorhergehende Kapitel stellte verschiedene Versionsverwaltungen sowie Mergeverfahren vor. Im Zuge dieser Arbeit werden nun die semistrukturierte und die unstrukturierte Konfliktbehandlung miteinander verglichen. Dabei befasst sich dieses Kapitel mit der Schaffung einer geeigneten Testumgebung.

3.1 Zielsetzung

Zielsetzung dieser Arbeit ist der Vergleich zwischen unstrukturierter und semistrukturierter Konfliktbehandlung in Versionsverwaltungssystemen. Dazu werden aus der Historie verschiedener Softwareprojekte Szenarien extrahiert, anhand derer beide Verfahren verglichen werden. Als Vergleichsmaßstab werden die im Folgenden gelisteten Kennzahlen verwendet. Diese Kennzahlen werden aus den Zusammenführungsergebnissen beider Verfahren pro Szenario bestimmt.

Eine Maßzahl ist die Anzahl der Ordnungskonflikte. Da das semistrukturierte Verfahren ist auf die Lösung dieser Konfliktart spezialisiert ist, kann durch die Kennzahl der Grad der Verbesserung gemessen werden. Weiter wird die Anzahl der Zeilen bestimmt, die in einem Szenario an Konflikten beteiligt sind. Diese Kennzahl gibt den Umfang der erzeugten Konflikte wider und wird abkürzend Konfliktzeilen genannt. Die dritte Maßzahl gibt die Anzahl der Dateien, die Konflikte beinhalten an. Zusätzlich wird bei semistrukturierter Behandlung die Anzahl der semantischen Konflikte ermittelt.

Die Wahl dieser Zahlen ist praxisnah, schließlich muss jeder Konflikt von einem Benutzer manuell gelöst werden. Dabei bestimmen Anzahl und Umfang der Konflikte auch die Dauer die Lösung. Zudem lassen sich beide Verfahren anhand dieser Werte absolut vergleichen. Das bessere Verfahren liefert somit auch niedrigere Kennzahlen.

3.2 Methodik

Um die Vergleichswerte zu ermitteln werden verschiedene Softwareprojekte betrachtet. Über die verwendete Versionsverwaltungssoftware des Projekts wird die Historie analysiert. Dadurch lassen sich Zeitpunkte bzw. Versionsstände ausmachen, für die ein Merge durchgeführt wurde bzw. für die Merge realistisch wäre. Da Merges immer zwischen zwei Entwicklungszweigen stattfinden, sind drei Versionsstände beteiligt: die beiden Revisionen in den Zweigen und ihr gemeinsamer Vorgänger als Basisrevision. Diese Konstellation wird *Mergetripel* genannt.

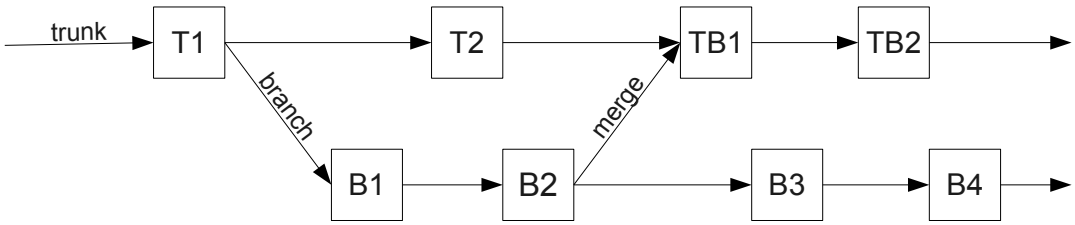


Abbildung 3.1: Projekthistoriengraph

Der Projektverlauf einer Versionsverwaltung stellt einen gerichteten azyklischen Graphen dar. Dabei bilden die Revisionen die Knoten. Geht eine Revision in eine andere über, so wird zwischen beiden eine Kante gezogen. Abbildung 3.1 skizziert einen Projekthistoriengraph. Es lässt sich zu jedem Versionsstand, außer dem ersten, ein Vorgänger finden. Ebenso haben zwei Revisionen aus verschiedenen Zweigen einen gemeinsamen Vorgänger. Dieser Knoten stellt den Zeitpunkt des Entstehens eines Zweiges dar. Eine Revision, die aus der Zusammenführung zweier Revisionen entsteht, wird keine Basisrevision in einem Mergetripel. Die Gründe hierfür werden in Abschnitt 4.3.2 erörtert.

Open Source Programme bieten sich zur Analyse an. Software dieser Art unterliegt einer stetigen Entwicklung und Anpassung durch mehrere Programmierer, so dass häufig neben-läufige Entwicklung in verschiedenen Zweigen stattfindet. Da viele Projekte ihre Versionsverwaltung und die darin enthaltenen Daten meist öffentlich lesbar machen, kann die Historie nach geeigneten Mergetripel durchsucht werden.

Für ein gefundenes Mergetripel werden die beteiligten Revisionen ausgecheckt. Anschließend werden diese in einem Drei-Wege-Merge mit einem unstrukturierten und einmal mit dem semistrukturierten Mergeverfahren verschmolzen. Beide Ergebnisse werden hinsichtlich der Kennzahlen Konflikte und Konfliktquelltextzeilen ausgewertet und schließlich gegenüber gestellt.

3.3 Systemauswahl

Dieser Abschnitt beschreibt die verwendete Software.

3.3.1 Mergetools

Als Mergetool kommt für das semistrukturierte Verfahren `FSTMerge` als FeatureHouse-Plugin zum Einsatz. Für das unstrukturierte Verfahren wird wegen seiner großen Verbreitung und Anwendung in den gängigen Versionsverwaltungssystemen `GNU merge` verwendet.

3.3.2 Versionsverwaltung

Weiter stellt sich die Frage, welche Versionsverwaltungen für die Analyse verwendet werden. Dabei muss zunächst eine Entscheidung hinsichtlich der Architektur getroffen werden.

Dezentrale Versionsverwaltung

Bei dezentralen Versionsverwaltungen erhält jeder Entwickler das gesamte Repository der geklonten Quelle. Er hat lokal Zugriff auf die komplette Projekthistorie. Man könnte hier also nach dem Checkout lokal nach Mergetripel suchen. Verzweigungen sind ein essentieller Bestandteil des Entwicklungskonzepts. Sie können schnell und kostengünstig erstellt werden. Da diese Experimentierfreudigkeit weckt, könnte man eine hohe Anzahl an Branches erwarten. Streng genommen ist sogar jeder Checkout ein eigener Branch für sich. Es zeigt sich aber, dass lokale Branches kurzlebig sind und auf Grund eines geringen Änderungsumfangs wenig bis keine Konflikte zeigen.

Dadurch, dass jeder Entwickler sein eigenes Repository hat, ist man bei vollständig dezentralen Versionsverwaltungen nie im Besitz der vollständigen Historie. Selbst bei Einsatz eines zentralen Repository, dem von den Hauptentwicklern verwalteten »Blessed Repository«, ist dies nicht gewährleistet. Es müssten nämlich stets alle sich im Umlauf befindenden Repositories synchronisiert werden. Weiter erhalten Versionsstände keine fortlaufenden Nummern, sondern Hashwerte, so dass Entwicklungsverläufe nur über das Änderungsdatum nachvollzogen werden können.

Zusätzlich ist es möglich die Historie nachträglich zu verändern. Bei Git kann der letzte Commit mit der Funktion `amend` überschrieben werden. Weiter können mit der Funktion `rebase` die Ursprungsversionen ganzer Branches verschoben werden. Als Folge dieser Operationen können anschließende Merges zu vorhersehbaren Ergebnissen führen ([Cha10]). Die `rebase`–Funktion kann für Mercurial mittels eines Plugins nachgerüstet werden. Zusätzlich dazu erlaubt Mercurial das Zusammenfassen von Commits ([O'S09b]). Damit werden mögliche Einsprungpunkte für die Analyse genommen.

Zentrale Versionsverwaltung

Im Open Source Bereich ist bei zentralen Versionsverwaltungen ist das Repository öffentlich zugänglich. So kann jeder lesend auf die Daten zugreifen. Schreibende Operationen sind nur einem ausgewählten Entwicklerkreis erlaubt. Dies hat zur Folge, dass das Repository »sauberer« ist als bei der dezentralen Variante. Verzweigungen erfordern einen höheren Aufwand beim Erstellen, dadurch werden sie vornehmlich für längerfristige Entwicklungen verwendet und bestehen länger. Für die Analyse ist dieser Umstand viel versprechend.

Die Nummerierung der Revisionshistorie erfolgt bei Subversion inkrementell numerisch. So kennzeichnet eine höhere Nummer eine aktuellere Version. Einzelne Versionsstände lassen sich so einfach einordnen. Verzweigungen arbeiten auf Verzeichnis- und Dateiebene. Per Konvention werden die Hauptverzeichnisse des Projekts nach ihrer Funktion benannt. Das Verzeichnis des Hauptentwicklungszweig heißt `trunk` und das für Nebenentwicklungszweige heißt `branches`. Zusätzlich gibt es ein Verzeichnis für feste Versionsstände `tags`, in dem keine Änderungen vorgenommen werden sollten. Dadurch lassen sich aktive Zweige gut erkennen.

Eine Manipulation der Projekthistorie ist nur sehr umständlich möglich. Man müsste dabei das Repository in eine Datei sichern, in dieser die Historie bearbeiten und aus der bearbeiteten Datei ein neues Repository erstellen. Dies schließt Manipulation aus. Damit ist stets der gesamte Projektverlauf unverändert enthalten und abrufbar ([CSFP10]).

CVS ist weiterer verbreiteter Vertreter zentraler Versionsverwaltungen. Im direkten Vergleich zeigt CVS Schwächen, die bei Subversion nicht vorhanden sind ([Fis]). So sind bei CVS Commits nicht atomar, d.h. dass beispielsweise die Unterbrechung einer Schreiboperation das Repository in einem inkonsistenten Zustand hinterlassen kann.

Ebenso können Dateien nicht durch die Versionsverwaltung umbenannt oder verschoben werden. Diese Operationen müssen manuell durchgeführt werden, in dem die betroffene Datei in ihr neues Verzeichnis kopiert und aus dem alten gelöscht wird. Dadurch teilt sich die Dateihistorie auf mehrere für die Versionsverwaltung nicht zusammenhängende Glieder auf. Bei Kopie eines Verzeichnisses oder einer Datei wird der Verlauf ebenfalls nicht übernommen. Diese Einschränkungen behindern die Analyse, da der Projektverlauf schwer zu verfolgen ist. Gerade um die Programmevolution nachvollziehen und Basisrevisionen der Mergetripel finden zu können, muss der Ursprung einer Datei klar ersichtlich sein.

Das Löschen von Dateien ist in CVS unter Umständen endgültig. Zwar können gelöschte Dateien, und ältere Versionsstände, wiederhergestellt werden, allerdings nur solange das Ursprungsverzeichnis noch existiert. Wird ein Verzeichnis gelöscht, so verschwindet auch die Historie der Dateien, die es beinhaltet hat. Aus diesem Grund erlaubt CVS nur das Löschen bereits leerer Verzeichnisse als Sicherungsmaßnahme. In Subversion hingegen werden Dateien und Verzeichnisse nicht tatsächlich gelöscht, sondern als gelöscht markiert. So werden Dateien nur abgerufen, falls sie nicht als gelöscht markiert wurden oder die angeforderte Revisionsnummer kleiner ist als die Revisionsnummer der Löschmarkierung. CVS würde damit nur die Betrachtung aktiver Branches erlauben.

Da bei der Analyse von Projekthistorien häufig beobachtet werden konnte, dass fertige, in den Trunk integrierte, Featurebranches gelöscht wurden, disqualifiziert sich CVS.

Fazit

Für die Analyse werden vorrangig Open Source Projekte betrachtet, die zur Verwaltung ihres Quellcodes das zentrale Versionsverwaltungssystem Subversion verwenden. Denn damit ist sichergestellt, dass die Historie vollständig erhalten ist.

3.3.3 Vorbereitung und Auswertung

Zur Vorbereitung des Quellcodes für die Analyse, kamen verschiedene Tools zur Entfernung von Quelltextkommentaren und zur Anpassung des Zeichensatzes zum Einsatz. Die Auswertung wurde von Shell- und Pythonskripten bei der Ermittlung der Kennzahlen unterstützt und mit der Statistiksoftware R[1] durchgeführt.

[1] http://www.r-project.org

3.4 Kandidatenauswahl

Dieser Abschnitt beschäftigt sich mit der Kandidatenauswahl. Dabei werden verschiedene Plattformen betrachtet, die Infrastruktur für Open Source Projekte bereitstellen. Weiter werden Kriterien aufgestellt nach denen Programme und Revisionen ausgewählt werden.

3.4.1 Plattform

Eine Plattform stellt die Infrastruktur für ein Softwareprojekt zur Verfügung. Diese umfasst neben Speicherplatz für das Repository, eine Versionsverwaltung und eine Webrepräsentation. Abhängig vom Anbieter können weitere Features wie Datenbanken und Content-Management-Systeme bereitgestellt werden. Für Open Source Projekte werden diese Dienste teilweise kostenlos angeboten, wenn der Quellcode ohne Zugriffsbeschränkung öffentlich abrufbar gemacht wird.

3.4.2 Programmkriterien

Um analysiert zu werden, muss ein Programm gewisse Voraussetzungen erfüllen. Als Programmiersprache muss Java, C# oder Python verwendet werden. Der Quelltext sollte mindestens 10000 Zeilen umfassen. Weiterhin muss ein Projekt mindestens zwei aktive Entwicklungszweige pflegen. Zusätzlich sollten mehrere Entwickler beteiligt sein. Das Projekt muss außerdem aktiv sein, d.h. der letzte Commit darf nicht zu weit zurückliegen. Des weiteren sollte sich eine angemessene Anzahl an Mergetripel finden lassen, die mindestens einen Konflikt verursachen.

3.4.3 Mergeszenarien

Mergeszenarien ergeben sich aus der Analyse der Projekthistorie und werden durch ein Mergetripel beschrieben. Ein Mergetripel besteht aus einer Basisrevision und zwei abgeleiteten Revisionen aus verschiedenen Branches. Beim durchgeführten Verschmelzen werden die drei Revisionen in einem Three-Way-Merge zu einer Version zusammengeführt.

Die Szenarien können in zwei Typen unterteilt werden. Zum einen in Merges, die tatsächlich durchgeführt wurden und zum anderen in Merges, die aufgrund der Entwicklung der Projekthistorie hätten durchgeführt werden können.

Merges, die tatsächlich durchgeführt wurden, sind meist, siehe 4.1, aus der Projekthistorie ersichtlich. Identifiziert man einen solchen Merge fasst man die Basisrevision und die beiden Zweigrevisionen in einem Mergetripel zusammen. In Abbildung 3.1 auf Seite 36 würde (`T1`,`T2`,`B2`) ein solches Tripel darstellen. Revision `TB1` wurde aus `B2` und `T2` zusammengeführt, `T1` ist der gemeinsame Vorgänger von `T2` und `B2`.

Da Merges nicht in allen Projekten in ausreichender Zahl vorkommen, werden auch Szenarios betrachtet, die aufgrund der Projekthistorie hätten durchgeführt werden können. Einen Hinweis auf nebenläufige Programmentwicklung gibt die Commitreihenfolge. Wird in verschiedenen Branches abwechselnd commitet und dabei eine gemeinsame Menge an Dateien bearbeitet, bietet dies einen erfolgversprechenden Einsprungpunkt für ein Szenario. Nebenläufige Entwicklung verursacht bei einem Merge häufig Konflikte.

Ein weiteres Szenario simuliert die Reintegration eines Zweiges in den Hauptentwicklungzweig. Nach einer Serie von Commits in einem Branch, kann man ein Mergetripel aus dem letzten Branch-Commit, Trunk-Commit und der Basisrevision bilden. Umfasst die Serie zu viele Commit, kann zu einem früheren Zeitpunkt ein Mergetripel gebildet werden. So kann die Entwicklung der Konfliktanzahl dargestellt werden. Endet ein Branch, d.h. es folgen keine weiteren Commits oder der Branch wird gelöscht, kann ein Mergetripel gebildet werden um das letzte Zusammenführen zu simulieren.

Zusätzlich wird darauf geachtet, dass die Zweigrevisionen der jeweiligen Szenarien in etwa gleich weit von einander entfernt sind. Dabei kann der zeitliche Abstand oder die Anzahl der dazwischen liegenden Commits betrachtet werden. Dieser Aspekt wird berücksichtigt, spielt aber eine untergeordnete Rolle. Vordergründig werden Mergetripel bzw. Revisionen gesucht, in denen signifikante Funktionalitätserweiterungen am Programm erfolgen.

Kapitel 4

Experimentdurchführung

In diesem Kapitel werden die in Kapitel 3 gesteckten Rahmenbedingungen konkretisiert. Die Wahl der verwendeten Versionsverwaltung, Programme und betrachteter Mergeszenarien wird getroffen. Ebenso erfolgt die Vorstellung eines Algorithmus, der bei der Bestimmung der Szenarien behilflich ist.

4.1 Versionsverwaltung und Plattform

Im Zuge dieser Arbeit wird auf Programme zurückgegriffen, die auf der Plattform `SourceForge`[1] gehostet werden. Sie bietet ein reichhaltiges Verzeichnis an Java, C# und Python Programmen. SourceForge stellt den Projekten die Wahl des Versionsverwaltungssystem frei und bietet SVN, Git, Mercurial, Bazaar und CVS an. Da die Repository-Server mit großer Bandbreite angebunden sind, können an einem Mergeszenario beteiligte Revisionen zügig abgerufen werden.

Als weitere Plattformen wurden u.a. Berlios[2], CodePlex[3] und Launchpad[4] betrachtet. Keiner der Anbieter konnte aber eine ähnliche Programm- und Optionsfülle wie SourceForge bieten.

Bereits in Abschnitt 3.3.2 erfolgte die Diskussion und Festlegung auf Subversion als Versionsverwaltung. Es werden nur Projekte betrachtet, die dieses System verwenden. Nur damit ist sichergestellt, das die Projekthistorie vollständig erhalten und abrufbar ist.

[1] `http://sourceforge.net`
[2] `http://developer.berlios.de`
[3] `http://www.codeplex.com`
[4] `https://launchpad.net`

Betrachtet man verschiedene SourceForge-Projekte hinsichtlich des verwendeten Versionskontrollsystems, so erkennt man, dass trotz des breiten Angebots, Subversion die am häufigsten verwendete Versionsverwaltung ist. Dezentrale Systeme werden selten verwendet.

4.2 Programmauswahl

Im Folgenden werden die ausgewählten Programme aufgelistet. Die in Abschnitt 3.4.2 aufgestellten Kriterien bzgl. Programmiersprache, Umfang und Aktualität sind bei der Selektion beachtet worden. Dabei konnte mit nur einer Ausnahme die Forderung nach einem Umfang von mindestens 10000 Quelltextzeilen erfüllt werden. Die Anzahl der maximal betrachteten Mergetripel pro Programm wird auf 10 beschränkt. Bietet ein Programm weniger Ansatzpunkte für Mergeszenarien, so wird die größte mögliche Anzahl betrachtet.

In den Tabellen 4.1, 4.2 und 4.3 sind die Eckdaten der Programme aufgeführt. *LOC* ist die Abkürzung für *Lines of code* und bezeichnet den Umfang eines Programms in Anzahl der Zeilen des Quelltexts. Der Wert wird aus dem aktuellen Hauptentwicklungszweig bestimmt. Kommentare und Leerzeilen wurden bei der Berechnung nicht berücksichtigt. Die Spalte *MS* gibt die Anzahl der gefundenen Mergeszenarien pro Projekt an.

Java-Programme

Projekt	**Anwendungsgebiet**	**LOC**	**MS**
DrJava	Entwicklungsumgebung	89 K	10
FreeCol	Strategiespiel	86 K	10
GenealogyJ	Stammbaumeditor	56 K	10
iText	PDF Bibliothek	71 K	10
JabRef	BibTeX-Verwaltung	75 K	10
jEdit	Texteditor	107 K	10
JFreeChart	Diagramm Bibliothek	149 K	10
Jmol	Molekülbetrachter	135 K	10
PMD	Statische Codeanalyse	71 K	10
SQuirrelSQL	SQL Client	218 K	10

Tabelle 4.1: Analysierte Java-Programme

C#-Programme

Projekt	Anwendungsgebiet	LOC	MS
AutoWikiBrowser	Wikipedia-Editor	63 K	9
CruiseControl.NET	Integrationsserver	148 K	9
Eraser	Sichere Datenlöschung	18 K	6
FireIRC	IRC Client	17 K	1
iFolder	Ordnersynchronisation	68 K	4
NASA WorldWind	Virtueller Globus	193 K	8
Process Hacker	Prozess Manager	99 K	1
RSS Bandit	News Feeds Reader	116 K	4

Tabelle 4.2: Analysierte C#-Programme

Python-Programme

Projekt	Anwendungsgebiet	LOC	MS
BitPim	Handysynchronisation	180 K	7
emesene	Instant Messaging Client	29 K	1
eXe	eLearning XHTML Editor	98 K	9
matplotlib	Bibliothek für math. Funktionen	86 K	10
SpamBayes	Spamfilter	47 K	6
Wicd	Verbindungsmanager	5 K	5

Tabelle 4.3: Analysierte Python-Programme

4.3 Mergeszenarien

Aus dem Projektarchiv der ausgewählten Programme wurden für den Vergleich der beiden Zusammenführungsverfahren Mergeszenarien bestimmt. Der folgende Abschnitt beschreibt das Vorgehen zur Identifikation der Szenarien und der beteiligten Revisionen.

4.3.1 Aufbau der Mergeszenarien

Ein Mergeszenario besteht aus einer Basisrevision und zwei Zweigrevisionen. Eine Basisrevision ist der gemeinsame Vorgänger der beiden Zweigrevisionen. Er stellt dabei den Entstehungszeitpunkt eines Nebenentwicklungszweiges dar und markiert damit die Revision an der ein Entwicklungszweig zu einem neuen Entwicklungszweig verzweigt (kopiert) wurde.

Eine Revision qualifiziert sich als Zweigrevision, falls in ihr ein nach Abschnitt 3.4.3 relevantes Ereignis auftritt. Die zeitlich entsprechende Revision im gegenüber liegenden Zweig komplettiert das Mergetripel. Beide Zweige werden mit der selben Revisionsnummer ausgecheckt. Damit ist sichergestellt, dass der jeweils letzte Stand zur Revisionsnummer vorhanden ist.

Die Vergabe der Revisionsnummern erfolgt bei Subversion global inkrementell, beginnend bei 1. Bei jedem Commit wird die maximale Nummer um eins erhöht und den bearbeiteten Dateien zugewiesen. Im Projekthistoriengraph spiegeln die Nummern den zeitlichen Bearbeitungsverlauf wider. Abbildung 4.1 zeigt Subversion-Historiengraph.

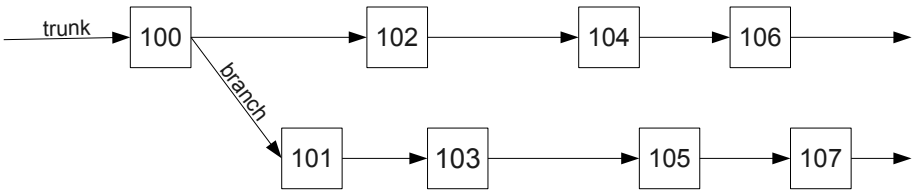


Abbildung 4.1: Subversion-Projekthistoriengraph

Findet in Revision 107 ein relevantes Ereignis statt, wird `branch` in Revision 107 ausgecheckt. Als zweite Zweigrevision wird `trunk` mit selbiger Nummer ausgecheckt. Da in diesem Zweig diese Revisionsnummer nicht existiert, wählt Subversion die nächst kleinere Revisionsnummer zu der ein Commit in diesem Verzeichnis vorliegt, 106. Beide Zweige sind nun auf dem selben zeitlichen Entwicklungsstand. Die Basisrevision in diesem Beispiel wäre `trunk` mit Revision 100. Ein Mergetripel lässt sich dadurch wie folgt abkürzen: `rev<Basisrevision>-<Zweigrevision>`. Im Beispiel wäre das Mergetripel `rev100-107`.

4.3.2 Basisrevision

Die Basisrevision in einem Mergetripel ist ein gemeinsamer Vorgänger der beiden Zweigrevisionen. Dabei stellt die Basisrevision den Zeitpunkt der Entstehung des Branches, der eine Zweigrevision beinhaltet dar.

Diese konservative Lösung wurde gewählt, da Merges einseitig sind. Es werden die Änderungen von einem Zweig ein einen anderen übertragen. Ein gegenseitiger Austausch findet nicht statt. Dies bedeutet, dass eine Revision, die durch die Zusammenführung zweier Revisionen entstanden ist, keine Basisrevision werden kann. Sie stellt keinen Vorgänger für Revisionen des anderen Branches dar.

Würde man eine zusammengeführte Revision als Basis wählen, würde es bei der Durchführung des nächsten Merges zu ungewollter Auslöschung kommen. Die Basisrevision enthält Elemente von beiden Branches, ebenso die nachfolgende Zweigrevision. Die Zweigrevision des anderen Branches umfasst nur eigene Elemente. Wird dieses Mergetripel zusammengeführt, so werden Elemente, die in der Basisrevision und ihrer nachfolgenden Zweigrevision unverändert blieben und nicht in der zweiten Zweigrevision enthalten sind gemäß den Regeln in Tabelle 2.4 auf Seite 19 entfernt. Wählte man dieses Ergebnis wiederum als Basis für das nächste Szenario würden die betroffenen Elemente wieder eingefügt werden. Da Einfüge- und Löschoperationen häufig Auslöser für Konflikte sind, würde dieses Vorgehen das Ergebnis zu stark beeinflussen.

Bleibt die Basisrevision hingegen für alle Mergeszenarien beider beteiligter Branches gleich, so sind die Ergebnisse von tatsächlichen durchgeführten Merges und hypothetischen Merges direkt miteinander vergleichbar. Die Entwicklung von Konflikten bei Quellcodeevolution wird erkennbar und messbar. Bei ständigem Basiswechsel würden man keine derartigen Einblicke erhalten.

4.3.3 Bestimmung der Mergeszenarien

Subversion liegt aktuell in Version 1.6 vor. Bis Version 1.4 führte Subversion keine Einträge über durchgeführte Merges. Seit Version 1.5, die am 19. Juni 2008 veröffentlicht wurde, werden Zusammenführungen verfolgt. Dieses Feature wird *Merge-Tracking* genannt. Es speichert zu einer Datei, welche Revisionsbereiche zusammengeführt wurden. Die Speicherung erfolgt getrennt von den eigentlichen Logeinträgen in externen Dateieigenschaften. Für Entwickler ist dies eine Erleichterung, da beispielsweise bei der Reintegration eines Branches keine Revisionsnummern mehr angegeben werden

müssen. Für die Analyse sind die Daten nicht wertvoll, da nur Bereiche und keine spezifischen Revisionen ausgegeben werden.

Ist ein Repository mit einer älteren Subversionversion erstellt worden, muss Merge-Tracking erst nachträglich aktiviert werden. Dabei funktioniert das Tracking erst ab den Aktivierungszeitpunkt, so dass eine Nutzung für ältere Merges ausfällt. Zudem sind alle betrachteten Projekte bereits vor 2008 entstanden. Im Zeitraum seit Erscheinen von Merge-Tracking lassen sich außerdem nicht genug Mergetripel finden. Weiter zeigte sich, dass manche Projekte bis heute Merge-Tracking nicht aktiviert haben und sogar wieder entfernt haben.

Daher wurde für die Analyse ein generischer Ansatz verfolgt, der auf der Auswertung der Subversion-Logdateien basiert. Dabei werden die Szenarien sowohl manuell, als auch algorithmisch bestimmt.

Inhalt SVN-Logeinträge

Jeder Commit und jede dadurch entstehende Revision erhält eine eindeutige Revisionsnummer. Zu jeder Revision wird ein Logeintrag erstellt, der die Eigenschaften der Änderung zusammenfasst. Ein Eintrag umfasst die Revisionsnummer, den Verfasser und den Zeitpunkt. Zusätzlich werden alle Dateien gelistet, die in der Revision modifiziert wurden. Subversion unterscheidet verschiedene Arten von Dateiänderungen. So wird aufgeführt, ob eine Datei neu angelegt (A), ihr Inhalt oder ihre Eigenschaften modifiziert (M), sie gelöscht (D) oder ob sie durch eine andere Version ersetzt (R) wurde. Geht die Änderung aus einer Kopieroperation hervor, so wird zusätzlich der Ursprungsdateiname und die Ursprungsrevisionsnummer vermerkt. Das Kommentarfeld erlaubt Notizen zu hinterlassen, um andere Entwickler über die Intention des Commits zu informieren und die Änderungen zusammenzufassen [CSFP10].

Die Ausgabe der Logeinträge kann über den Befehl `svn log --xml -v` *`Repository`* im XML-Format erfolgen. Dabei kapselt das Tag `logentry` die Eigenschaften eines Logeintrags. In ihrem Attribut `revision` ist die Revisionsnummer vermerkt. Verfasser und Datum speichern die Tags `author` und `date`. Das Tag `paths` kapselt die Liste der geänderten Dateien. Einen Eintrag für eine Datei oder ein Verzeichnis notiert das Tag `path`. Zu diesem Tag speichern Attribute weitere Eigenschaften. So vermerkt `action` die Art der Dateiänderung, `copyfrom-path` und `copyfrom-rev` ggf. den Ursprung einer Kopieroperation. Abschließend umfasst das Tag `msg` den optionalen Kommentar des Verfassers.

Listing 4.1 zeigt einen Logeintrag, der typischerweise bei der Erstellung eines Branches entsteht. Hier wird im Projekt DrJava in Revision 5268 der Zweig `drjava-compilers` generiert. Er geht aus einer Kopie des Hauptentwicklungszweigs der Revision 5266 hervor.

```
<logentry
    revision="5268">
<author>mgricken</author>
<date>2010-06-04T19:06:38.017200Z</date>
<paths>
<path
    kind=""
    copyfrom-path="/trunk"
    copyfrom-rev="5266"
    action="A">/branches/drjava-compilers</path>
</paths>
<msg>
Creating branch for refactored research compiler interface for Mint, HJ, NextGen.
</msg>
```

Listing 4.1: Logeintrag bei der Erstellung eines Branches

Das `path`-Attribut `kind` informiert, ob der Eintrag eine Datei oder ein Verzeichnis ist. Bei Abruf der Logeinträge wird dieses Attribut nicht gesetzt. Zur Bestimmung dieser Eigenschaft muss der Befehl `svn list -R -xml` *`Repository`* verwendet werden. Als Ausgabe erhält man hier Revisionsnummer, Datei und Typ der Datei, nicht aber Art der Änderung und den Kommentar. Da sich der Typ eines Eintrag aus dem Kontext ergibt, ist eine Bestimmung des Attributs für die Analyse nicht notwendig.

Manuelle Identifikation der Mergeszenarien

Anhand der Logeinträge eines Projekts können Mergeszenarien identifiziert werden. Da die Szenarien möglichst aktuell sein sollen, wird in der Historie zunächst nach dem jüngsten Entwicklungszweig gesucht. Über das in Listing 4.1 dargestellte Muster, lässt sich die Entstehungs- und die Ursprungsrevision ausmachen. Beide Verzeichnisse unterscheiden sich nur durch ihren Speicherort, sie sind inhaltlich identisch. Daher könnten beide Revisionen als Basis eines Mergetripels dienen.

Im nächsten Schritt betrachtet man die Historie der einzelnen Zweige, des ursprünglichen und des abgeleiteten Zweiges parallel, um auf Mergeszenarien zu stoßen. Hinweise auf tatsächlich durchgeführte Merges ergeben sich aus den Kommentaren der Entwickler. So sucht man speziell nach Commits in deren Bemerkungen auf die Verbreitung oder Abwicklung hingewiesen wird. Dabei kann gezielt nach dem Stichwort *merge* gefahndet werden. Ein weiteres Indiz liefern Revisionen, deren Kommentare von der Anpassung von Programmversionsnummern berichten.

Bei Merges kann sich eine spezielle Konstellation der beteiligten Dateien zeigen. Man kann hier beobachten, dass das `copyfrom-path`-Attribut häufiger gesetzt wurde und dass eine große Anzahl an Dateien eines Zweiges in einer Revision geändert werden.

Wurde eine Revision eines Merges erkannt, so wird den beide Zweige in der Vorgängerrevision ausgecheckt. Dadurch kann der Merge im Szenario mit beiden Verfahren simuliert werden.

Zweigenden bieten sich ebenfalls für Mergeszenarien an. Als Zweigende wird die letzte Revision eines Branches bezeichnet. Sie ist interessant, falls der Branch anschließend als gelöscht markiert oder über einen längeren Zeitraum keine Commits folgen. Es kann davon ausgegangen werden, dass die Entwicklung abgeschlossen oder ausgelaufen ist. Dadurch die Betrachtung der zugehörigen Logeinträge erkennt man, ob ein abschließender Merge getätigt wurde. Ist dies der Fall kann aus den beteiligten Revisionen ein Szenario geformt werden. Lässt sich kein Merge erkennen, so bietet sich die letzte Zweigrevision und den korrespondierenden Versionen des Hauptentwicklungszweigs als Mergetripel an.

Ebenfalls als vielsprechend erweisen sich Revisionen mit außergewöhnlich vielen Änderungen. Sie deutet auf einen Entwicklungssprung hin und lohnt sich für ein Mergeszenario, da bei Zusammenführung mit dem Hauptentwicklungszweig Konflikte zu erwarten sind.

Algorithmus zur Gruppierung von Revisionen

Durch die Gegenüberstellung der Logeinträge zweier Zweige lassen sich bereits einige Szenarien identifizieren. Allerdings ist dieses Vorgehen für das Auffinden von hypothetischen, konfliktreichen Mergetripel aufwändig und mühsam. Daher wurde ein Programm entworfen, das die Lösung dieser Aufgabe unterstützt.

Das Programm besteht aus zwei Komponenten. Ein Parser überführt die Logeinträge der Historie eines Softwareprojekts in eine interne Datenstruktur. Auf dieser Datenstruktur ermittelt ein Algorithmus potentielle Kandidaten für ein Szenario an.

Der Algorithmus arbeitet vor dem Hintergrund, dass Konflikte durch Dateien, die in zwei Zweigen bearbeitet wurden, verursacht werden. Er analysiert die Historie eines Softwareprojekts, gruppiert Revisionen und zeigt an wie viele Dateien parallel geändert wurden.

Die Umsetzung des Parsers ist vom Format der Logeinträge der jeweiligen Versionsverwaltung abhängig. Da Subversion in der Lage ist, die Änderungshistorie in

einem XML-Format auszugeben, kann die Transformation durch den Java-eigenen SAXParser[5] übernommen werden. Die Verarbeitung von Logeinträgen weiterer Versionsverwaltungssysteme könnte durch die Implementation eines entsprechenden Parsers bewerkstelligt werden.

Die interne Datenstruktur repräsentiert die Projekthistorie. Ihre Einträge sind feingranularer als das XML-Datenformat. Pro Eintrag wird der Name einer in einer Revision geänderten Datei oder Verzeichnis festgehalten. Zusätzlich speichert sie die zugehörige Revisionsnummer, die Art der Dateiänderung. Geht die Änderung aus einer Kopieroperation hervor, so werden ebenfalls Ursprungspfad und Ursprungsrevision erfasst.

Nach dem Aufbau der Datenstruktur wird diese, beginnend bei der kleinsten Revision traversiert. Stößt das Programm auf einen Eintrag, der durch ein Kopie entstanden ist, so markiert dieser Eintrag den Entstehungszeitpunkt eines Branches. In einer gesonderten Datenstruktur werden diese in sog. Mappings festgehalten. Sie speichern Revisionsnummer, so wie beide beteiligte Zweige.

In einem weiteren Durchlauf werden die Mappings gefüllt. Falls die Revisionsnummer einer Änderung größer als die Entstehungsrevisonsnummer des Mappings ist, wird geprüft, ob die geänderte Datei in einem Pfad des Mappings enthalten ist und ggf. festgehalten. Dieser Vorgang wird für alle Mappings wiederholt. Eine Revision einer Datei kann in mehreren Mappings auftreten.

Als Ergebnis enthält ein Mapping alle geänderten Dateien in beiden beinhalteten Branches, so wie deren Revisionsnummern. Folgen in einem Zweig mehrere Commits in fortlaufenden Revisionen nacheinander, so können diese für eine bessere Übersichtlichkeit gruppiert werden. Aus den Mappings lässt sich nun erkennen, wie viele Dateiänderungen an den Branches jeweils durchgeführt wurden. Ebenso wird aufgeschlüsselt, wie viele Dateien an den Modifkationen beteiligt werden. Weiterhin zeigt die Übersicht an, wie viele Dateien in jeweils beiden Branches innerhalb einer Revisonsgruppe bearbeitet wurden. Auch erkennbar ist der Verlauf der totalen Anzahl an gemeinsam geänderten Dateien.

Eine Beispielsausgabe stellt Listing 4.2 dar. Zeile 1 zeigt die Entstehungsrevision des Mappings, bzw. des Branches und die Anzahl der insgesamt gemeinsam geänderten Dateien. Die erste Spalte, `Rev`, stellt die zusammengefassten Revisionsgruppen dar.

In der zweiten, `LMods`, und dritten Spalte, `RMods`, wird die Art und Anzahl der jeweiligen Dateiänderungen angeführt. Dabei gruppiert `LMods` die Änderungen am linken

[5] http://download.oracle.com/javase/1.5.0/docs/api/javax/xml/parsers/SAXParser.html

Zweigen und `RMods` die Änderungen am rechten Zweig. Für jede Spalte wird die Anzahl der Änderungen wie folgt aufgeschlüsselt:

`A` Anzahl der hinzugefügten Dateien

`M` Anzahl der geänderten Dateien

`D` Anzahl der gelöschten Dateien

`Tot` Summe der Dateioperationen in der Revisionsgruppe

In den Spalten vier, `LModsSum`, und fünf, `RModsSum`, wird die Anzahl der Dateiänderungen, in den Spalten sechs, `LNoFiles`, und sieben, `RNoFiles`, die Anzahl der modifizierten Dateien im jeweiligen Branch aufsummiert. Die Spalte acht, `MututalFiles`, listet die Anzahl der gemeinsam geänderten Dateien von Revisionsgruppe zu Revisionsgruppe auf, `Max`, und zeigt an wie viele gemeinsame Dateien insgesamt bearbeitet wurden, `Tot`.

```
1  since 4990: /trunk/drjava <-169-> /branches/drjava-concjunit
2  Rev         LMods            RMods            LModsSum  RModsSum  LNoFiles  RNoFiles  MutualFiles
3              A/M/D/Tot        A/M/D/Tot                                                  Max/Tot
4  4990-4992   0/0/0/0          1/5/0/6          0         6         0         6         0/0
5  4995-4995   0/1/0/1          0/0/0/0          1         6         1         6         0/0
6  4996-4997   0/0/0/0          0/8/0/8          1         14        1         14        1/1
7  5000-5004   0/114/0/114      0/0/0/0          115       14        112       14        7/8
8  5005-5005   0/0/0/0          0/112/0/112      115       126       112       118       111/112
9  5007-5008   0/5/0/5          0/0/0/0          120       126       113       118       3/112
10 5012-5013   0/0/0/0          0/8/0/8          120       134       113       119       5/113
11 5014-5014   0/37/0/37        0/0/0/0          157       134       121       119       30/114
12 5017-5017   0/0/0/0          0/2/0/2          157       136       121       120       2/115
13 5018-5019   1/16/0/17        0/0/0/0          174       136       126       120       16/120
14 5020-5020   0/0/0/0          0/37/0/37        174       173       126       126       37/126
15 5021-5021   0/1/0/1          0/0/0/0          175       173       126       126       1/126
16 5022-5022   0/0/0/0          0/1/0/1          175       174       126       126       1/126
17 5024-5025   0/16/0/16        0/0/0/0          191       174       129       126       12/126
18 5026-5026   0/0/0/0          0/16/0/16        191       190       129       129       16/129
19 5027-5027   0/104/0/104      0/0/0/0          295       190       165       129       68/129
20 5031-5045   0/0/0/0          0/115/0/115      295       305       165       168       104/165
21 5047-5047   0/1/0/1          0/0/0/0          296       305       165       168       1/165
22 5048-5058   0/0/0/0          1/20/0/21        296       326       165       169       6/165
23 5059-5333   20/1264/4/1288   0/0/0/0          1584      326       602       169       169/169
```

Listing 4.2: Ausgabe des Programms FindRev

Durch die Gruppierung der Ergebnisse kann der Umfang einer Entwicklungsphase, erkannt werden. Für Mergeszenarien bieten sich beispielsweise Revisionen an, die am Ende einer Revisionsgruppe stehen, in der viele gemeinsame Dateien geändert wurden. Da am Ende einer Revisionsgruppe die Entwicklung in dem Schwesterzweig fortgesetzt wird, würde ein solches Mergeszenario die Verschmelzung am Ende eines Entwicklungsabschnitts simulieren.

Aus dieser Aufschlüsselung erkennt man nur, wie viele Änderungen vorgenommen wurden. Ob es sich bei der Änderung, um eine Quellcodemodifikation oder Anpassung in Kommentaren, wie Versionsnummern- oder Lizenztextänderungen handelt, bleibt verborgen.

Kombination der Methoden

Die Suche nach potentiellen Kandidaten für Mergeszenarien ist bei manueller Suche in den Logdateien, genau, aber geschuldet der Textlänge, unübersichtlich und langwierig. Bei Gruppierung des Entwicklungsverlaufs lassen sich zwar Änderungsumfänge erkennen, die Angabe einer bestimmten Revision ist allerdings nicht möglich.

Aus diesem Grund wurde für das Auffinden von simulierten Mergeszenarien wie folgt vorgegangen. Zunächst wurden die Logeinträge aus der Versionsverwaltung extrahiert. Anschließend gruppiert obiger Algorithmus Zweige, die direkt zueinander in Beziehung stehen und stellt den Entwicklungsverlauf anhand von Revisionsgruppen dar. Diese Untergliederung bildet den Einsprungpunkt für die folgenden Schritte.

Für Mergeszenarien werden Revisionsgruppen beginnend bei der Entstehung eines Zweiges betrachtet. Umfasst die Gruppe mehrere Revisionen und gleichzeitig im Vergleich zum direkten Umfeld zahlenmäßig viele Dateiänderungen, so stellt sie einen Entwicklungsabschnitt dar.

Enthält die nachfolgende Revisionsgruppe, welche nun einen Entwicklungsabschnitt im Schwesterzweig darstellt, keine oder nur wenige Änderungen, so wäre ein simulierter Merge wenig konfliktträchtig, da nur eine geringe Änderungsüberschneidung vorhanden ist. In diesem Fall beinhalten sie keinen Szenariokandidaten.

Weist die Revisionsgruppe hingegen ebenfalls viele Änderungen aus, so gilt es zu Ermitteln, welchen Ursprungs diese sind. Dazu werden die Logeinträge der umfassten Revisionen betrachtet. Durch Abgleich der Dateinamen der Modifikationen kann pro Revision entschieden werden, ob die Revisionen einen Merge oder eine eigenständige Entwicklung kapselt. Umfasst eine Revision eine überschaubare Anzahl an Modifikationen, so deutet auf eine Entwicklungsrevision hin. Umfasst eine Revision eine große Anzahl an Dateien, deren Namen sich mit denen aus der vorhergehenden Revisionsgruppe überschneiden, ist dies eine Mergerevision.

Findet der Merge am Anfang der Revisionsgruppe statt, so kann aus letzten Revision der Vorgängergruppe ein Mergeszenario erstellt werden. Erfolgt der Merge nach eigenständiger Entwicklung innerhalb der Gruppe, so wird aus der Revision vor dem Merge ein Szenario erstellt. Enthält die Gruppe keinen Merge, oder es ist keiner erkennbar, so bietet sich die letzte Revision der Gruppe als Szenario an. Eine Zusammenführen von zwei gegenläufigen Entwicklungsverläufen würde so simuliert.

Insgesamt kann aber nur bestimmt werden, dass Änderungen an den Dateien vorgenommen wurden. Der Inhalt der Änderung ergibt sich nicht aus Logeinträgen oder Revisionsgruppenverläufen. Diese Art der Analyse zeigt nur (gute) Kandidaten.

Dazu müsste man die Datei auschecken. Wenn also nur Änderungen in Quelltextkommentaren, Anpassung Lizenz, Autor oder häufig Versionsnummer, falsches Indiz, kein Konflikt.

Beispiel Die Ausgabe in Listing 4.2 zeigt den Entwicklungsverlauf des Hauptentwicklungszweigs, `/trunk/drjava`, und des Nebenentwicklungszweigs, `/branches/drjava-concjunit` des Programms DrJava. Man erkennt, dass der Nebenentwicklungszweig in Revision 4990 aus dem Hauptentwicklungszweig entstand und dass über den Verlauf Änderungen an insgesamt 169 gemeinsamen Dateien vorgenommen wurden.

Es ergeben sich insgesamt vier Revisionen, die für ein Mergeszenario in Frage kommen: 5004, 5019, 5044 und 5058.

Revision 5004 In der zugehörigen Revisionsgruppe 5000-5004 werden zahlreiche Dateiänderungen vorgenommen, die auch gemeinsame Dateien betreffen. Zu dem zeigt sich in der nachfolgenden Revision eine ähnliche Anzahl an Änderungen im Schwesterzweig. Mit der Tatsache, dass die folgende Modifikation in einer Revision integriert wurde, deutet dies auf einem tatsächlich stattgefundenen Merge hin. Mit dem zu Revisionsnummer 5005 zugehörigem Kommentar wird die Vermutung bestätigt[6].

Revision 5014 Diese Revision beinhaltet im Vergleich zu den Vorgängerrevisionen viele Änderungen. Da die umliegenden Revisionsgruppen keine ähnlichen Zahlen aufweisen, stellt sie keinen Kandidaten für ein Mergeszenario dar. Ein derartiges Bild wird häufig von Textersetzungs-Operationen im Quelltext, wie die Anpassung von Versionsnummern und Lizenztexten gezeigt. In der Commitnachricht der Revision findet man den Hinweis auf »Aufräumen« im Quelltext.

Revision 5019 Analog zu obiger Argumentation kann Revision für ein Mergeszenario gewählt werden. Hier zeigt der Commit-Kommentar ebenfalls einen tatsächlichen Merge an. Da der Abstand, sprich die Anzahl an Revisionen und geänderten Dateien, zum ersten Szenario ausreichend ist wird diese Revision gewählt.

[6] Über den Kommentar wäre diese Revision bereits durch die manuelle Suche entdeckt worden.

Revision 5044 Man erkennt, dass in Revision 5027 eine große Menge an Dateien geändert wurde. Der Umfang der Revisiongruppe 5031-5045 legt nahe, dass diese Änderungen in einer der Revision integriert werden. Zu dem zeigt diese Konstellation an, dass vor oder nach dem Merge am Branch drjava-concjunit weiter entwickelt wurde. Aus der zugehörigen Logdatei geht hervor, dass von Revision 5031 bis 5044 am Quellcode im Branch gearbeitet wurde und dass schließlich in der Folgerevision die Änderungen aus dem trunk integriert wurden.

Revision 5058 Diese Revision stellt einen Kandidaten dar, da sie die letzte Revision des Branches ist. In der zugehörigen Revisionsgruppe werden finale Änderungen getätigt. Aus den Logeinträgen geht hervor, dass in Revision 5059 der Branch abschließend in den trunk gemergt wird.

4.4 Durchführung

Ist ein Mergeszenario und damit die drei beteiligten Revisionen identifiziert, so kann der Mergelauf durchgeführt werden. Hierzu wird das betrachtete Projekt im Versionsstand der jeweiligen Versionsnummer aus der Versionsverwaltung ausgecheckt. In einem Vorverarbeitungsschritt werden aus den Quelltextdateien die Kommentare entfernt. Änderungen an Kommentaren stellen sich für den textuellen Merge dar wie Quelltextänderungen und würden so das Resultat verfälschen. Anschließend werden die drei Revisionen einmal textuellen Verfahren mit GNU Merge und einmal im semistrukturierten Verfahren verschmolzen und die resultierenden Dateien abgespeichert.

Die Laufzeit der Verschmelzung eines Mergetripels hängt von seinem Umfang ab. So dauert die Verarbeitung eines SQuirrelSQL-Szenarios bis zu 45 Minuten, während ein Jabref-Lauf 10 Minuten in Anspruch nimmt. Als Durchschnittswert lässt für die betrachteten Projekte lässt sich eine Zeitspanne von 15 Minuten angeben.

Insgesamt wurden 24 Projekte betrachtet. Aus ihnen konnte 180 Mergeszenarien konstruiert werden, die in der Summe ca. 50 Millionen Zeilen an Quellcode umfassen. Eine detaillierte Auflistung der einzelnen Projekte und zugehörigen Mergetripel befindet sich im Anhang A.1 ab Seite 83ff.

Kapitel 5

Auswertung & Diskussion

Bei der Durchführung des in Kapitel 4 beschriebenen Experiments werden für jedes betrachtete Projekt und jedes identifizierte Mergeszenario zwei zusammengeführte Versionen erzeugt. Eine Version entsteht durch die Anwendung des semistrukturierten Mergeverfahrens, die andere entsteht durch das unstrukturierte Verfahren.

5.1 Kennzahlen

Zum Vergleich der beiden Mergearten dienen vier Kennzahlen:

1. Die Anzahl der Konflikte
2. Die Anzahl der an Konflikten beteiligten Zeilen
3. Die Anzahl an Dateien, die Konflikte beinhalten
4. Die Anzahl an semantischen Konflikten

Die Bestimmung der Kennzahlen erfolgt auf Dateiebene. Konflikte zeigen sich in den gemergten Quelltextdateien beider Varianten nach dem in Abschnitt 2.1 auf Seite 21 gezeigten Muster. Darüber lässt sowohl die Konfliktmenge, als auch deren Umfang in Zeilen berechnet werden. Für die Ergebnisse des semistrukturierten Merge wird zusätzlich die Nummer der semantischen Konflikte extrahiert. Anschließend werden pro Revision die Werte der einzelnen Dateien aufsummiert und die Nummer der Konflikt tragenden Dateien ermittelt.

5.2 Validität

Der Quellcode der an Mergeszenarien beteiligten Revisionen wird verändert an die beiden Mergeverfahren übergeben. Um die Ergebnisse nur von der Programmstruktur bzw. dem Programmcode abhängig zu machen, wurden sämtliche Kommentare in einem Vorverarbeitungsschritt entfernt. Das semistrukturierte Verfahren könnte Quelltextkommentare erkennen und ignorieren, aber der unstrukturierte Merge kann nicht zwischen Code und Kommentar unterscheiden. Somit würden konkurrierende Kommentaränderungen textuelle Konflikte erzeugen. Zusätzlich unterliegen Kommentare nicht den strengen syntaktischen Regeln von Quelltext, wodurch mit häufigen Wandel zu rechnen ist. Durch die Elimination der Anmerkungstexte sind die Ergebnisse beider Mergeverfahren vergleichbar, da als Eingabe *reiner* Quelltext verwendet wird.

Als Unsicherheitsfaktor verbleibt die Auswahl der Mergeszenarien. Da, wie in Kapitel 4 erörtert, tatsächlich durchgeführte Merges nicht eindeutig vermerkt werden, mussten die Szenarien bestimmt werden. Da bei der Bestimmung Wert auf reelle Konstellationen, wie aus den Entwicklerkommentaren erkennbare Merges und vom Entwicklungsstand vernünftige Begebenheiten, gelegt wurde, wird eine verdeckte Beeinflussung des Ergebnisses verhindert.

Auf Grund des großen Umfangs der Testumgebung, drei Programmiersprachen, 24 Softwareprojekte, 180 Szenarien und 50.000.000 Zeilen an Quellcode, kann eine Beeinflussung des Ergebnisses durch individuelle Entwicklereigenschaften, wie Programmierstil und Programmiererfahrung ausgeschlossen werden. Weiter kann durch die breite Auswahl vermutet werden, andere Projekte ähnliche Ergebnisse liefern werden.

5.3 Ergebnisse

Das Ergebnis wird durch die in Abschnitt 5.1 definierten Kennzahlen bestimmt. In Tabelle 5.3 auf Seite 59 sind für die einzelnen Softwareprojekte die Mittelwerte pro Revision gelistet. Sie werden nach Maßzahltyp und Mergeart aufgeschlüsselt. Dabei verzeichnet die Spalte *UM* die Resultate für den unstrukturierten Merge und die Spalte *SM* die für den semistrukturierten Merge.

Zur Auswertung und zum Vergleich der Resultate bietet die Tabelle eine gute Übersicht. Eine Auflistung aller Projekte pro Mergeverfahren und Mergeszenario befindet sich im Anhang in den Tabellen A.1 und A.2 ab Seite 83ff. Die vollständigen Ergebnisse können im Internet auf der Seite `http://www.infosun.fim.uni-passau.de/spl/SSMerge/`

abgerufen werden. Dort werden die Kennzahlen bis auf Dateiebene aufgeschlüsselt und die Resultate beider Varianten in Quelltextform zur Verfügung gestellt.

	Konflikte		Zeilen		Dateien		sem. Konfl.	
Projekt	UM	SM	UM	SM	UM	SM	UM	SM
AutoWikiBrowser	20	17	418	362	6	6	–	0
BitPim	29	25	5240	471	6	3	–	4
CruiseControl.NET	26	16	1133	279	14	8	–	5
DrJava	20	16	293	210	7	7	–	0
emesene	7	4	400	17	7	4	–	17
Eraser	24	24	2146	637	12	11	–	0
eXe	5	3	112	35	4	2	–	2
FireIRC IRC Client	5	3	36	23	3	2	–	0
FreeCol	237	213	4606	3265	63	58	–	1
GenealogyJ	19	9	489	146	9	6	–	0
iFolder	44	10	228	156	6	4	–	34
iText	219	706	113757	35109	216	179	–	262
JabRef	75	49	1701	1782	28	24	–	2
jEdit	7	6	105	80	3	3	–	0
JFreeChart	560	361	18142	5500	193	109	–	21
Jmol	94	112	20094	6869	40	33	–	4
matplotlib	57	15	500	466	13	9	–	16
NASA WorldWind	23	36	5946	2328	12	9	–	10
PMD	55	371	18469	6596	48	36	–	17
Process Hacker	1	1	4	4	1	1	–	1
RSS Bandit	65	51	4086	1478	12	8	–	32
SpamBayes	25	22	1166	343	10	7	–	2
SQuirrelSQL	26	24	2620	579	16	13	–	1
Wicd	10	9	1267	218	4	3	–	2

Tabelle 5.1: Mittelwerte der Kennzahlen pro Szenario

Die Kennzahlen fallen in der überwiegenden Zahl der Projekte bei semistrukturierter Konfliktbehandlung niedriger aus als bei der textuellen Variante. Bei 19 der 24 Softwareprojekte zeigen sich in allen drei Kategorien geringere oder gleiche Werte. Die Kennzahlen der Programme, bei denen höhere Werte gemessen wurden, sind in der Tabelle mit gelber Hintergrundfarbe hervorgehoben.

Betrachtet man die Ergebnisse auf der Ebene der einzelnen Mergeszenarien, zeigt sich in 60% der Fälle, dass semistrukturiertes Zusammenführen weniger Konflikte verursacht. Im direkten Vergleich ergibt sich in diesem Ausschnitt eine durchschnittliche Reduktion um $34 \pm 21\%$. Bei 26 Szenarios ist die Konfliktzahl identisch. In 46 Konstellationen wurde eine Erhöhung um $109 \pm 170\%$ gemessen.

Der Vergleich der Anzahl der in Konflikte involvierten Quelltextzeilen zeichnet ein ähnliches Bild. Hier senkt sich in 82% der Szenarien die Kennzahl um $61 \pm 22\%$, während sie sich in 12% verdoppelt.

Für die dritte Kennzahl, die Anzahl der konfliktbehafteten Dateien, zeigt sich, dass sie in 96% der Mergetripel abnimmt oder gleich bleibt. In 8 Szenarien lässt sich eine Zunahme von $16 \pm 8\%$ feststellen.

Diesen Vergleich veranschaulicht Tabelle 5.3. Die Zeilen geben die Werte für die jeweilige Kennzahl wieder. Die Spalten teilen das Resultat in drei Kategorien auf: semistrukturierter Merge erzeugt geringere Werte, beide Verfahren liefern identische Zahlen und textueller Merge zeigt niedrigere Zahlen. Dabei listet jede Kategorie in der Spalte *absolut* die Szenarienzahl und in der Spalte *Änderung* die durchschnittliche Änderung von semistrukturiertem Ergebnis zu unstruktiertem an.

	SM besser		beide gleich		UM besser	
Kennzahl	absolut	Änderung	absolut	Änderung	absolut	Änderung
Konflikte	108	$-34 \pm 21\%$	26	$0 \pm 0\%$	46	$+109 \pm 170\%$
Konfliktzeilen	148	$-61 \pm 22\%$	10	$0 \pm 0\%$	22	$+96 \pm 86\%$
Konfliktdateien	139	$-28 \pm 12\%$	33	$0 \pm 0\%$	8	$+16 \pm 8\%$

Tabelle 5.2: Vergleich auf Szenarioebene und Änderung mit Standardabweichung

Als Beispiel für die überwiegende Zahl an Projekten, die von semistrukturierter Konfliktbehandlung profitieren, werden im Folgenden die Ergebnisse der einzelnen Szenarien für die Javaprogrammierumgebung DrJava dargestellt. Abbildung 5.1 zeigt die aufsummierte Konfliktzahl, Abbildung 5.2 die Summe der Konfliktzeilen[1]. In den meisten Revisionen kann die jeweilige Kennzahl gesenkt werden. Die entsprechenden Diagramme der übrigen Projekte stellen sich ähnlich dar und spiegeln den Trend aus Tabelle 5.3 wider.

Bei wenigen Projekten erzeugt semistrukturiertes Zusammenführen eine wesentlich höhere Zahl an Konflikten, aber gleichzeitig auch erheblich niedrigere Werte für an Konflikten beteiligten Zeilen. Die beiden Diagramme 5.3 und 5.4 des Programms iText visualisieren diese Situation.

Die Gründe für das Verhalten, bzw. der direkte Vergleich beider Verfahren, werden in Kapitel 5.4 präsentiert. Dort wird ebenfalls diskutiert, wie semistrukturiertes Zusammenführen trotz Kontextkenntnissen vermeintlich schlechtere Resultate produzieren kann.

Für das semistrukturierte Verfahren muss noch die Zahl der semantischen Konflikte mit der Zahl der Reihenfolgekonflikte verglichen werden. Dabei ergibt sich, dass die

[1] Die Diagramme aller Projekt können unter `http://www.infosun.fim.uni-passau.de/spl/SSMerge/` abgerufen werden.

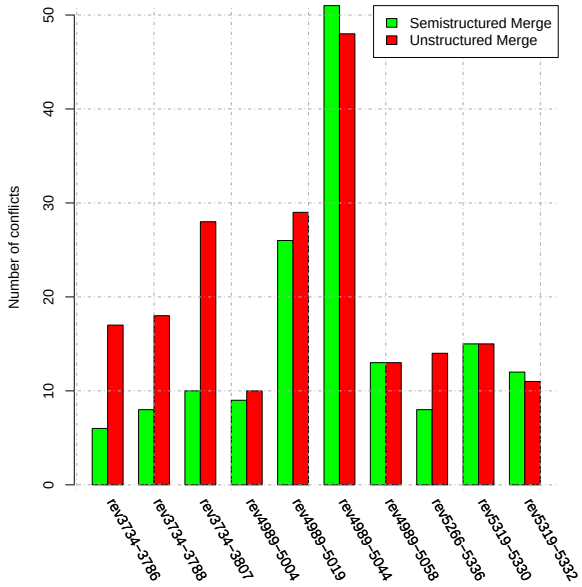


Abbildung 5.1: Konflikte DrJava

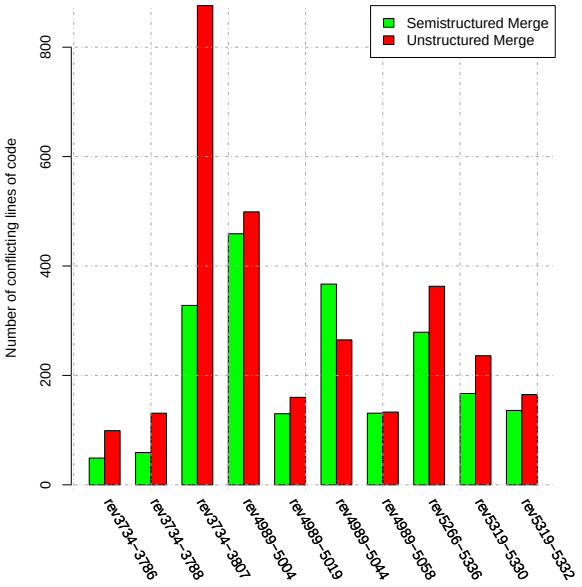


Abbildung 5.2: Konfliktzeilen DrJava

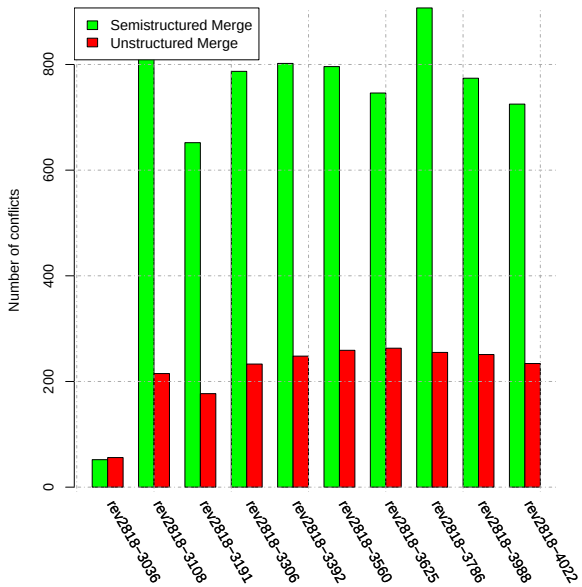


Abbildung 5.3: Konflikte iText

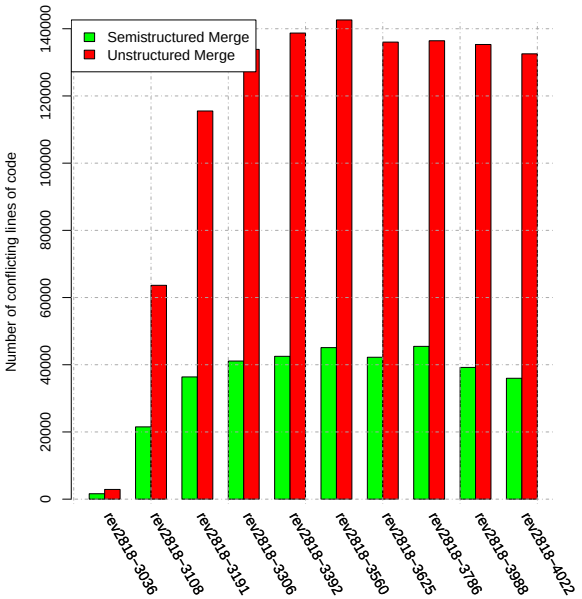


Abbildung 5.4: Konfliktzeilen iText

Zahl der semantischen Konflikte vergleichsweise gering ausfällt. Dies bedeutet, dass sich diese Programmelemente in den verschiedenen Szenarios stabil verhielten. Eine Ausnahme dieser Regel ist iText. Dort tritt diese Konfliktart in sehr hoher Zahl auf. In Abschnitt 5.4.3 werden die Ursachen hierfür erörtert.

5.4 Diskussion

Dieser Abschnitt vergleicht anhand aus den Mergeszenarien identifizierten typischen Konfliktsituationen das Verhalten des semistrukturierten Verfahrens mit dem des unstrukturierten Verfahrens. Dabei wird eine Erklärung für die im vorhergehenden Abschnitt gestellten Fragen gefunden.

Als Ausgangspunkt, bzw. Basisversion, wird eine einfache Implementierung eines Stack in der Programmiersprache Java verwendet. Dargestellt ist die entsprechende Klasse bzw. Datei in Listing 5.1 .Je nach nach zustellender Konfliktsituation werden, zwei Varianten abgeleitet. Diese Varianten repräsentieren den linken bzw. rechten Zweig der Basis, wie er in den Szenarien auftritt. Aus diesen drei Varianten mittels beider Mergeverfahren jeweils eine kombinierte Version erzeugt und die Ergebnisse gegenübergestellt.

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  public class Stack {
6
7    private LinkedList content = new LinkedList();
8
9    public void push(T item) {
10     content.add(item);
11   }
12
13   public T pop() {
14     return content.removeFirst();
15   }
16 }
```

Listing 5.1: Javaimplementierung eines Stack

5.4.1 Reihenfolge

Da semistrukturierte Konfliktbehandlung syntaktische Einheiten als Ganzes betrachtet, verursacht eine Änderung der Reihenfolge der einzelnen Elemente keine Konflikte. Sie werden durch insbesondere durch den Algorithmus selbst geordnet. Das unstrukturierte Verfahren arbeitet zeilenbasiert und kann nicht zwischen Textänderungen und Reihenfolgeänderungen unterscheiden. Daher wird es in unterstehender Konstellation Konflikte erzeugen.

Um die Situation zu nach zu stellen wird die Anordnung der Programmelemente in den abgeleiteten Revisionen wie folgt modifiziert:

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  public class Stack {
6
7    public void push(T item) {
8      content.add(item);
9    }
10
11   public T pop() {
12     return content.removeFirst();
13   }
14
15   private LinkedList content = new
         LinkedList();
16 }
```

Listing 5.2: Reihenfolge-Links

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  public class Stack {
6
7    private LinkedList content = new
         LinkedList();
8
9    public T pop() {
10     return content.removeFirst();
11   }
12
13   public void push(T item) {
14     content.add(item);
15   }
16 }
```

Listing 5.3: Reihenfolge-Rechts

Die Ergebnisse der Mergedurchläufe zeigen die Listings 5.4 und 5.5

```java
package datastructure;

import java.util.LinkedList;

public class Stack {

  private LinkedList content = new
      LinkedList();

  public T pop() {
    return content.removeFirst();
  }

  public void push(T item) {
    content.add(item);
  }
}
```

Listing 5.4: Reihenfolge-FSTMerge

```java
package datastructure;

import java.util.LinkedList;

public class Stack {

<<<<<<< Links
=======
  private LinkedList content = new
      LinkedList();

  public T pop() {
    return content.removeFirst();
  }
>>>>>>> Rechts
  public void push(T item) {
    content.add(item);
  }
<<<<<<< Links
  public T pop() {
    return content.removeFirst();
  }
  private LinkedList content = new
      LinkedList();
=======
>>>>>>> Rechts
}
```

Listing 5.5: Reihenfolge-GNU Merge

Das textuelle Verfahren erzeugt wie erwartet drei Konflikte. Diese Konstellation zeigt ebenfalls sich in den Ergebnissen der Mergeszenarien. Betroffenes Programmelement sind hier häufig `import`–Anweisungen zu Dateibeginn.

5.4.2 Neue Programmelemente

Die von der Basisrevision eines Branches abgeleiteten Varianten werden fortlaufend verändert und weiterentwickelt. Im Zuge der Erweiterung des Funktionsumfangs eines Programms, werden den einzelnen Programmeinheiten ständig neue Elemente hinzugefügt. In Sinne dieses Abschnitts sind Programmelemente, diejenigen, die frei im Quelltext angeordnet werden können. Hierunter fallen beispielsweise Methoden und Felder, nicht aber Anweisungen innerhalb von Methoden.

Wird in eine Variante ein solches Element eingefügt, so kann es beim semistrukturierten Zusammenführen immer konfliktfrei übernommen werden. Beim textuellen Verfahren können die modifizierten Abschnitte zwar identifiziert werden, wurde allerdings der zweite Mergepartner an der korrespondierenden Stelle verändert, so entsteht ein Konflikt.

Ein Beispiel verdeutlicht dies. In die linke Variante des Basis-Stacks, Listing 5.6, wird eine neue Methode `isEmpty()` eingefügt. Um einen textuellen Konflikt an in der rechten Variante, Listing 5.7, zu erzeugen, wurden die Leerzeilen zwischen den Methoden entfernt. Die Leerzeilen im Basis-Stack bleiben erhalten.

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  public class Stack {
6
7    private LinkedList content = new
         LinkedList();
8
9    public void push(Object item) {
10     content.add(item);
11   }
12   public boolean isEmpty()
13   {
14     return content.isEmpyt();
15   }
16   public Object pop() {
17     return content.removeFirst();
18   }
19 }
```

Listing 5.6: Methode eingefügt-Links

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  public class Stack {
6
7    private LinkedList content = new
         LinkedList();
8
9    public void push(Object item) {
10     content.add(item);
11   }
12   public Object pop() {
13     return content.removeFirst();
14   }
15 }
```

Listing 5.7: Methode eingefügt-Rechts

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  public  class  Stack {
6
7    private LinkedList content = new
         LinkedList();
8
9    public void push(Object item) {
10     content.add(item);
11   }
12
13   public Object pop() {
14     return content.removeFirst();
15   }
16
17   public boolean isEmpty()
18   {
19     return content.isEmpyt();
20   }
21 }
```

Listing 5.8: Methode eingefügt-FSTMerge

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  public class Stack {
6
7    private LinkedList content = new
         LinkedList();
8
9    public void push(Object item) {
10     content.add(item);
11   }
12 <<<<<<< Links
13   public boolean isEmpty()
14   {
15     return content.isEmpyt();
16   }
17 =======
18 >>>>>>> Rechts
19   public Object pop() {
20     return content.removeFirst();
21   }
22 }
```

Listing 5.9: Methode eingefügt-GNU Merge

In Listing 5.8 wird die Methode direkt eingefügt, während Listing 5.9 einen Konflikt anzeigt. Analoge Beobachtungen können beim Entfernen von Programmelementen gemacht werden.

5.4.3 Modifikationen

Im Zuge der Weiterentwicklung von Software werden auch an bestehenden Programmelementen Modifikationen vorgenommen. Anhand der annotierten Grammatik wird das betreffende Element identifiziert und, wie in Abschnitt 2.4 beschrieben mit dem entsprechenden hinterlegten Konfliktbehandlungsmethode zusammengeführt. Kann der Konflikt nicht gelöst werden, erfolgt ein Eintrag in der zusammengeführten Version.

Semantische Konflikte

Alle Programmelemente, die bei semistrukturierten Merge nicht mit dem zeilenbasierten Verfahren verschmolzen werden, erzeugen im Konfliktfall semantische Konflikte. Als Beispiel sei hier die überlappende Änderung eines Java-Klassenattributes präsentiert.

```
~~FSTMerge~~ private List content = new
    ArrayList();
##FSTMerge## private List content = new
    LinkedList();
##FSTMerge## private LinkedList content =
    new LinkedList();
```

Listing 5.10: Semantischer Konflikt-FSTMerge

```
<<<<<<< Links
  private List content = new ArrayList();
=======
  private LinkedList content = new
      LinkedList();
>>>>>>> Rechts
```

Listing 5.11: Semantischer Konflikt-GNU Merge

Unlösbare Konflikte, wie z.B. unvereinbare `implements`-Listen, erzeugen ebenfalls semantische Konflikte. Da bei der Inspektion der Resultate kein derartiger Fall auffiel, wird hier kein Beispiel geliefert.

In Anbetracht der hohen Zahl, 54, der Programmelemente, die diese Art der Konflikte verursachen kann, fallen die Werte verglichen mit den Ordnungskonflikten niedrig aus. Eine Ausnahme bildet iText. Hier liegen in den Szenarien, dargestellt in Abbildung 5.5, bis zu 486 Konflikte vor. Eine Untersuchung des iText–Quelltexts zeigt die Ursache für dieses Phänomen: Im Zuge des Versionssprunges von Java 1.4 auf Java 1.5, wurden generische Typen in den Sprachstandard aufgenommen. Bei der Umstellung auf diese Version wurden die verwendeten Collections parametrisiert. Ein Großteil der Konflikte

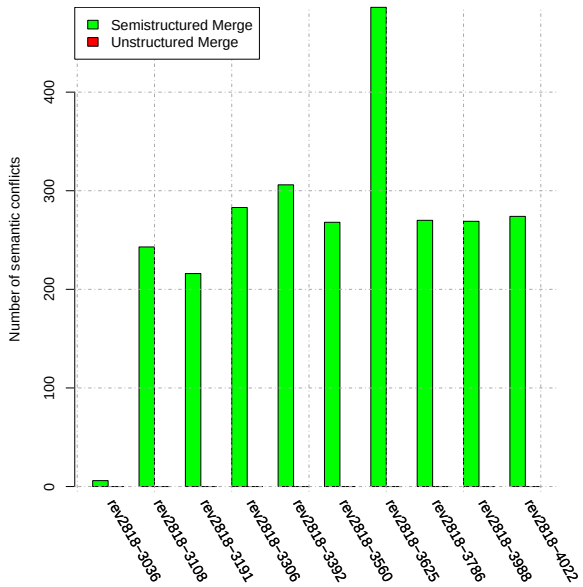


Abbildung 5.5: Entwicklung der semantischen Konflikte in iText

wird dadurch verursacht. Weiter können in verschiedenen Klassen mehrere statische Felder gefunden werden, deren Initialisierung geändert wird.

Da diese semantischen Konflikte textuelle Unterschiede zeigen, müssten beim unstrukturierten Zusammenführen ebenfalls Konflikte an diesen Stellen entstehen. In den bei iText betrachteten Revisionen ist dies nicht der Fall, da zusätzlich das in Abschnitt 5.4.4 erörterte Problem der Umbenennung zum Tragen kommt.

Methodenrumpf

In der annotierten Grammatik werden Methoden als terminale Knoten repräsentiert. Damit wird der Methodenrumpf als textueller Inhalt repräsentiert. Zusätzlich ist dort vermerkt, dass bei Verschmelzung das zeilenbasierte Verfahren angewandt werden soll. Somit gibt es auf dieser Ebene keine Unterschiede zwischen den beiden Mergeverfahren. Bei der Zusammenführung von Methoden mit identischen Signaturen in den drei beteiligten Revisionen werden die gleichen Resultate produziert.

Listing 5.12 zeigt das identische Ergebnis beider Verfahren. In der Ausgangssituation wurden an der Methode `push()` in linker und rechter Variante jeweils geringfügige Änderungen vorgenommen. Da Anweisungen in Methodenrümpfen Reihenfolge abhängig sind, stellen konkurrierende Änderungen Ordnungskonflikte dar. Diese werden in den Verfahren auf die gleiche Weise behandelt.

```
1 public void push(T item) {
2 <<<<<<< Links
3     content.addFirst(item);
4 =======
5     if (content != null)
6     content.addFirst(item);
7 >>>>>>> Rechts
8 }
```

Listing 5.12: Konflikt im Methodenrumpf

5.4.4 Umbenennung

Bei den bisher betrachteten Operationen kann semistrukturierte Konfliktbehandlung durch Kontextkenntnis mehr Konflikte lösen, als die unstrukturierte Variante. Dies erklärt die besseren Resultate der betrachteten Projekte. Die Ursache, das manche Projekte, die in Tabelle 5.3 mit gelber Farbe hervorgehoben sind, wesentlich mehr Konflikte bei weniger in Konflikt stehenden Zeilen verursachen, liegt in Umbenennung.

Die Zuordnung und der Vergleich der zu verschmelzenden Element geschieht bei semistrukturierter Konfliktbehandlung über den Namen des Elements im FST-Baum. Erfolgt eine Umbenennung eines Elements kann dieses nicht mehr seinem Partner zugeordnet werden. Beim Vergleich der FST-Bäume erscheint das Element unter seinem aktuellen Namen als neu eingefügtes Element, während es unter altem Namen als gelöscht erscheint. Die folgenden Beispiele illustrieren verschiedene Konstellationen.

Methodenumbenennung

Ausgangspunkt für dieses Beispiel ist die Basisimplementation eines Stack aus Listing 5.1 auf Seite 62. In der Variante des linken Programmzweigs, dargestellt in Listing 5.13, wird die Methode `push()` in `insert()` umbenannt. Gleichzeitig wird die ursprüngliche Methode in der rechten Variante, Listing 5.14, um eine Einfügebedingung erweitert.

Die vorgenommenen Änderungen umfassen textuell zwei Zeilen, 8 und 9, in der linken Variante und drei Zeilen, 8-10, in der rechten Variante. Somit stellt das unstrukturierte Verfahren diese Zeilen als Konflikt im Resultat (Listing 5.16) gegenüber.

Das semistrukturierte Verfahren erkennt die Methode `insert()` als neu eingefügt, da weder in der Basisrevision, noch in der gegenüberliegenden Revision ein Element mit diesem Namen vorhanden ist. Somit erfolgt eine Übernahme in das Resultat. Gleichzeitig verursacht die Methode `push()` einen vermeintlichen Auslöschungskonflikt. Da die

```java
package datastructure;

import java.util.LinkedList;

public class Stack {

  private LinkedList content = new
      LinkedList();

  public void insert(Object item) {
    content.add(item);
  }


  public Object pop() {
    return content.removeFirst();
  }
}
```

Listing 5.13: Umbenennung-Links

```java
package datastructure;

import java.util.LinkedList;

public class Stack {

  private LinkedList content = new
      LinkedList();

  public void push(Object item) {
   if(item != null)
    content.add(item);
  }

  public Object pop() {
    return content.removeFirst();
  }
}
```

Listing 5.14: Umbenennung-Rechts

Methode in der Basis vorhanden ist, in der rechten Revision bearbeitet und auf linker Seite nicht mehr aufgeführt ist, konkurrieren Löschung und Modifikation. Dies äußert sich wie in Listing 5.15 dargestellt. Die entfernte Methode zeigt sich als leerer linker Konfliktteil, dem die modifizierte Methode gegenübersteht. Da das semistrukturierte Verfahren in diesem Fall die gesamte Methode, Signatur und Methodenrumpf, zum Konflikt zählt, ist die Zahl der Konfliktzeilen höher.

Klassenumbenennung

Wird der in einer abgeleiteten Revision der Name einer Klasse geändert, stellt dies für semistrukturierte Konfliktbehandlung eine ähnliche Situation dar. Bei textueller Konfliktbehandlung verursacht diese Konstellation unter Umständen keine Konflikte.

In der linken Variante, Listing 5.17, wird die Klasse in `RenamedStack` umbenannt, während auf der rechten Seite, Listing 5.18, eine Erweiterung an der `push()`–Methode vorgenommen wurde. Zu bemerken ist, dass alle drei Versionen den gleichen Dateinamen, `Stack.java`, tragen. Es wird zwar empfohlen, dass der Dateiname einer Klasse ihrem Namen entspricht, aber wenn die Sichtbarkeit unter der `public`–Ebene liegt[2], gibt der Compiler bei Benennungsdifferenz keine Fehler- oder Warnmeldung aus.

Im unstrukturierten Verfahren können die drei Revisionen konfliktfrei zusammengeführt werden. Dies gelingt, da die Dateinamen übereinstimmen und damit die Varianten einander zugeordnet werden können. Zum anderen sind die Änderungen

[2] Die Sichtbarkeit sei bspw. `package-private`.

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  public  class  Stack {
6
7    private LinkedList content = new
         LinkedList();
8
9    <<<<<<< Links
10 =======
11 public void push(Object item) {
12     if( item != null )
13         content.add(item);
14   }
15 >>>>>>> Rechts
16
17   public Object pop() {
18     return content.removeFirst();
19   }
20
21   public void insert(Object item) {
22     content.add(item);
23   }
24 }
```

Listing 5.15: Umbenennung-FSTMerge

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  public class Stack {
6
7    private LinkedList content = new
         LinkedList();
8
9  <<<<<<< Links
10   public void insert(Object item) {
11     content.add(item);
12 =======
13   public void push(Object item) {
14     if( item != null )
15         content.add(item);
16 >>>>>>> Rechts
17   }
18
19   public Object pop() {
20     return content.removeFirst();
21   }
22 }
```

Listing 5.16: Umbenennung-GNU Merge

nur lokal und überschneiden sich nicht dateiübergreifend. Somit werden die modifizierten Stellen gemäß den Mergeregeln in das Ergebnis (Listing 5.20) übernommen.

Bei semistrukturierter Verschmelzung werden die Klassen `Stack` und `RenamedStack` getrennt behandelt, da sie auf Grund ihrer unterschiedlichen Namen nicht vereint werden können. Damit zeigt sich `RenamedStack` als neues Element, das vollständig in das Ergebnis (Listing 5.19) einfließt. Die Klasse `Stack` wird anhand der Basisrevision und der rechten Revision zusammengeführt. Als linke Revision dient eine leere Datei. Damit werden Methoden, die in Basis und rechter Revision bearbeitet wurden, als Löschkonflikt übernommen. Für Methoden, die in Basis und rechter Revision identisch sind, stellt sich die vermeintliche Löschung[3] als einzige Modifikation. Bei der Anwendung der Mergeregeln werden die Methoden folglich nicht in das Resultat eingebunden. Am Beispiel betrifft dies die Methode `pop()` der Klasse `Stack`.

[3] Das Element fehlt im linken FST-Baum.

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  class RenamedStack {
6
7    private LinkedList content = new
         LinkedList();
8
9    public void push(Object item) {
10     content.add(item);
11   }
12
13
14   public Object pop() {
15     return content.removeFirst();
16   }
17 }
```

Listing 5.17: Klassenumbenennung-Links

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  public class Stack {
6
7    private LinkedList content = new
         LinkedList();
8
9    public void push(Object item) {
10     if( item != null )
11     content.add(item);
12   }
13
14   public Object pop() {
15     return content.removeFirst();
16   }
17 }
```

Listing 5.18: Klassenumbenennung-Rechts

Verzeichnis–Umbenennung

Den schlechtest möglichen Fall stellt die Umbenennung eines Verzeichnisses, `package` in Java-Terminologie, dar. An einem Punkt der Versionsgeschichte wird ein Verzeichnis umbenannt oder verschoben. Damit können die beiden Mergeverfahren bei der Verschmelzung nur noch auf zwei Varianten zurückgreifen. Die dabei einstehenden Resultate sind sehr konfliktträchtig. Das semistrukturierte Verfahren erzeugt viele Konflikte und eine geringere Zahl an Konfliktzeilen, während das unstrukturierte Verfahren wenig Konflikte, aber dafür eine hohe Zahl an Konfliktzeilen verursacht. Die Auswirkungen des Umbenennungseffekts äußern sich entsprechend den Resultaten der Projekte iText, Jmol, PMD und NASA WorldWind. Sie überdecken durch den großen Ausschlag positive Effekte, die in anderen Dateien erzielt wurden.

Bei Verzeichnis–Umbenennung muss zwischen zwei Fällen unterschieden werden, da sich die Resultate dort unterschiedlich verhalten. Im ersten Fall wird das Verzeichnis einer Zweigrevision umbenannt, so dass diese beim Verschmelzen nicht auffindbar ist. Im zweiten Fall erfolgt die Verschiebung bei beiden Zweigrevisionen. Bei einem Merge kann also die Basisrevision nicht gefunden werden.

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  class  Stack {
6
7    <<<<<<< Links
8  =======
9  public void push(Object item) {
10     if( item != null )
11     content.add(item);
12   }
13 >>>>>>> Rechts
14 }
15
16 class  RenamedStack {
17
18   private LinkedList content = new
         LinkedList();
19
20   public void push(Object item) {
21     content.add(item);
22   }
23
24   public Object pop() {
25     return content.removeFirst();
26   }
27 }
```

Listing 5.19: Klassenumbenennung-FSTMerge

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  class RenamedStack {
6
7    private LinkedList content = new
         LinkedList();
8
9    public void push(Object item) {
10     if( item != null )
11     content.add(item);
12   }
13
14   public Object pop() {
15     return content.removeFirst();
16   }
17 }
```

Listing 5.20: Klassenumbenennung-GNU Merge

Zweigrevision fehlt Um die Auswirkung einer fehlenden Zweigrevision zu simulieren, werden die Basisversion des Stacks, Listing 5.1 auf Seite 62, und die davon abgeleitete Version aus Listing 5.7 auf Seite 65 miteinander verschmolzen.

Semistrukturierte Konfliktbehandlung verhält sich hier, wie im Fall einer Klassenumbenennung. Bei unstrukturierter Behandlung verursacht die Verschmelzung mit einer leeren Zweigrevision einen Konflikt, sich um die linke Zweigrevision spannt.

Basisrevision fehlt Ein anderes Verhalten zeigt sich, wenn ein Merge ohne Basisrevision durchgeführt wird. Als Demonstration werden die Listings 5.23 und 5.24 zusammengeführt.

Da keine Basisrevision zur Verfügung steht, werden beim semistrukturierten Verfahren Elemente, die in nur einer Revision oder in beiden unverändert vorkommen, in das Ergebnis, Listing 5.25, übernommen. Für Elemente, die sich in beiden Revisionen unterscheiden, wird ein entsprechender Konflikt vermerkt.

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5    class  Stack {
6
7    <<<<<<< Links
8  public void push(T item) {
9      content.addFirst(item);
10   }
11 =======
12 >>>>>>> Rechts
13 }
```

Listing 5.21: Zweigrevision fehlt-FSTMerge

```
1  <<<<<<< Links
2  package datastructure;
3
4  import java.util.LinkedList;
5
6  public class Stack {
7
8    private LinkedList content = new
         LinkedList();
9
10   public void push(T item) {
11     content.addFirst(item);
12   }
13
14   public T pop() {
15     return content.removeFirst();
16   }
17 }=======
18 >>>>>>> Rechts
```

Listing 5.22: Zweigrevision fehlt-GNU Merge

Im Gegensatz dazu erkennt die textuelle Mergevariante beim Vergleich der beiden Blätter mit der leeren Basis einen unvereinbaren textuellen Unterschied. Daher besteht die Ergebnisdatei, Listing 5.26, aus einem Konflikt, der die linke und rechte Variante vollständig beinhaltet.

Diese unterschiedliche Behandlung von Umbenennungen führt dazu, dass die Kennzahlen beider Mergeverfahren in diesen Fällen gegensätzlich ausfallen.

```java
package datastructure;

import java.util.LinkedList;

public class Stack {

  private LinkedList content = new
      LinkedList();

  public void push(T item) {
    content.addFirst(item);
  }

  public T pop() {
    return content.removeFirst();
  }
}
```

Listing 5.23: Basisrevision fehlt-Links

```java
package datastructure;

import java.util.LinkedList;

public class Stack {

  private LinkedList content = new
      LinkedList();

  public void push(T item) {
      content.add(item);
  }

  public T pop() {
    return content.removeFirst();
  }
}
```

Listing 5.24: Basisrevision fehlt-Links

```java
package datastructure;

import java.util.LinkedList;

public  class  Stack {

  private LinkedList content = new
      LinkedList();

  <<<<<<< Links
public void push(T item) {
    content.addFirst(item);
  }
=======
public void push(T item) {
      content.add(item);
  }
>>>>>>> Rechts

  public T pop() {
    return content.removeFirst();
  }
}
```

Listing 5.25: Basisrevision fehlt-FSTMerge

```java
<<<<<<< Links
package datastructure;

import java.util.LinkedList;

public class Stack {

  private LinkedList content = new
      LinkedList();

  public void push(T item) {
    content.addFirst(item);
  }

  public T pop() {
    return content.removeFirst();
  }
}=======
package datastructure;

import java.util.LinkedList;

public class Stack {

  private LinkedList content = new
      LinkedList();

  public void push(T item) {
      content.add(item);
  }

  public T pop() {
    return content.removeFirst();
  }
}>>>>>>> Rechts
```

Listing 5.26: Basisrevision fehlt-GNU Merge

JabRef

Durch Umbenennung lassen sich die Werte der Projekte erklären, deren Konfliktzahl bei semistrukturierten Merge deutlich erhöht ist. Nun zeigt sich bei JabRef, dass die Konfliktzahl semistrukturiert niedriger, die Konfliktzeilenzahl allerdings höher ausfällt. Als Ursache kann hier Umbenennung von Methodensignaturen angegeben werden. In wenigen Dateien kommt es vor, dass sich von Basisrevision zu den Zweigrevisionen die Signaturen ändern. Da die betroffenen Methoden in den Zweigrevisionen nicht übereinstimmen, zeigt sich im Resultat an betroffener Stelle der Effekt der fehlenden Basisrevision. Bei unstrukturierter Konfliktbehandlung zeigen sich die Auswirkungen weniger deutlich, da die textuelle Änderung nur wenige Zeilen umfasst oder sogar gelöst werden kann.

5.4.5 Quelltextformatierung

Moderne Entwicklungsumgebungen bieten Assistenten zum Formatieren von Quelltexten an. Über diese Module können verschiedene Texteigenschaften festgelegt werden. Konfigurierbare Optionen sind beispielsweise Zeilenumbrüche und Einrückung in verschiedenen Situationen. Eine Entwicklergruppe eines Projekt teilt sich eine Formatierungskonfiguration. Vor jedem Check-in in die Versionsverwaltung wird die Konfiguration angewandt und der Quelltext bzw. die Änderung in die gemeinsame Form gebracht.

Wird dieser Konsens gebrochen, d.h. die Formatierungskonfiguration wird geändert oder nicht eingehalten, so entstehen textuelle Differenzen zwischen den Revisionen. Diese betreffen nicht mehr ausschließlich die lokalen Codeänderungen, sondern alle Programmzeilen, die durch die Formatierungseinstellung berührt werden.

Dieses Problem zeigte sich bei iText und NASA WorldWind. In diesen beiden Projekten wurde im Entwicklungsverlauf die Art der Einrückung am Zeilenbeginn umgestellt. Im Fall iText von Leerzeichen auf Tabulatoren. Alleine durch diese Anpassung entsteht textuelle Differenz zwischen den Revisionen, so dass das zeilenbasierte Mergeverfahren eine hohe Anzahl an Konflikten anzeigt. Da sich die Formatierungsänderung auf die Einrückung beschränkt, wurden bei iText in einem Vorverarbeitungsschritt die Tabulatoren wieder durch Leerzeichen ersetzt.

Vermutlich wegen des gravierenden Einflusses auf den textuellen Merge, behalten die übrigen untersuchten Softwareprojekte die Formatierungsregeln bei.

Semistrukturierte Konfliktbehandlung zeigt sich gegenüber Formatierungsänderungen stabiler, da Identifikatoren der Programmelemente nicht betroffen sind. Konflikte tun sich nur da auf, wo das textuelle Verfahren zum Zusammenführen verwendet wird. Im Vergleich werden dennoch mehr Konflikte erzeugt, da alle Methodenrümpfe markiert werden, während GNU Merge größere Textbereiche zusammenfasst und somit weniger Konflikte und mehr Konfliktzeilen verursacht.

5.4.6 Strukturelle Grenzen

Bei der Auswertung der Ergebnisse der Mergeszenarien zeigte sich ein weiterer Vorteil des semistrukturierten Verfahrens. Da dieser Ansatz die Verschmelzung auf Ebene der Programmelemente durchführt, umfassen entstehende Konflikte ebenfalls nur ein Programmelement. Wird, wie bei Methodenrümpfen, ein textueller Algorithmus verwendet, so kann dieser Konflikte nur innerhalb des jeweiligen Rumpfes markieren. Gleichzeitig verringert sich die Fehleranfälligkeit des textuellen Merges, da der Eingabeumfang kleiner ist. Zusätzlich erhöht sich durch die Eingrenzung der Lesbarkeit der Konflikte und macht es einfacher deren Ursachen zu erkennen.

Im Gegensatz dazu kann das reine textuelle Verfahren Konflikte struktur-übergreifend markieren. Es kann häufig beobachtet werden, dass ein Konflikt innerhalb einer Methode beginnt und anschließend innerhalb einer anderen endet. Zwischen diesen getrennten Methoden können wiederum Programmelemente liegen. Somit werden auf Kosten der Lesbarkeit Konfliktsituationen zusammengefasst. Listing 5.27 zeigt ein Beispiel eines methoden-übergreifenden Konflikts.

```
1  package datastructure;
2
3  import java.util.LinkedList;
4
5  public class Stack {
6
7    private LinkedList content = new LinkedList();
8
9    public void push(T item) {
10 <<<<<<< Links
11     content.addFirst(item);
12   }
13
14   public void clear()
15   {
16     content = new LinkedList();
17 =======
18     if( item != null )
19         content.add(item);
20   }
21
22   public boolean isEmpty()
23   {
24     return content.isEmpty();
```

```
25 >>>>>>> Rechts
26   }
27
28   public T pop() {
29     return content.removeFirst();
30   }
31 }
```

Listing 5.27: Textueller Konflikt über Strukturgrenzen

Man ist verleitet in diesem Beispiel Quelltext über Konfliktgrenzen zu lesen und dadurch falsch zu interpretieren. Beim semistrukturierten Verfahren besteht diese Gefahr nicht.

5.4.7 Zusammenfassung

In 74% der Mergeszenarien erzeugt semistrukturierte Konfliktbehandlung weniger oder gleich viele Konflikte wie das zeilenbasierte textuelle Mergeverfahren. Somit kann durch die Unabhängigkeit von der Reihenfolge der Programmelemente und das konfliktfreie Einfügen neuer Programmelement die Anzahl der Konflikte um $34 \pm 21\%$ gesenkt werden. Betrachtet man innerhalb dieser Gruppe Dateien, in denen das textuelle Verfahren weniger Konflikte hervorbringt, so liegen die Ursachen im Umgang mit Umbenennung und in Konstellationen, in denen Konflikte über Strukturgrenzen hinweg zusammengefasst werden.

Als Ursache für Szenarien, in denen die Ergebnisse bei unstrukturierter und semistrukturierter Konfliktbehandlung gegenläufig erscheinen, konnte ebenfalls Umbenennung identifiziert werden. Der Effekt zeigt sich, so bald die Mergetripel nicht vollständig sind. Den schlechtest möglichen Fall stellt dabei die Umbenennung oder das Verschieben von Verzeichnissen dar.

Eine potentielle Lösung, die die Auswirkungen von Umbenennung mindern könnte, wäre eine Kombination der Mergeverfahren. Umbenennung kann auf Grund der Resultate pro Datei erkannt werden: bei semistrukturierter Behandlung viele Konflikte und wenige Konfliktzeilen; bei unstrukturierter Behandlung ein Konflikt und Konfliktzeilen Umfang von linker und rechter Revision. Liegt diese Konstellation vor, könnte das unstrukturierte Ergebnis verwendet werden, da es keine Auslöschungseffekte zeigt.

Ein weiterer Ansatz wäre bereits bei der Verschmelzung auf Vollständigkeit des Mergetripels zu achten. Wird entdeckt, dass eine Datei im Tripel fehlt, könnte wie folgt agiert werden: Fehlt die Basisrevision im Tripel kann semistrukturierte Konfliktbehandlung verwendet werden. Identische gemeinsame und neue Programmelemente werden in

das Resultat übernommen, während nicht identische gemeinsame Programmelemente, wie Methoden mit identischen Signaturen und unterschiedlichen Rümpfen, in Konflikt gesetzt werden. Fehlt hingegen eine Zweigrevision wäre das textuelle Verfahren die bessere Wahl, da keine Programmelemente entfernt werden. Alternativ könnte die Basisrevision temporär als Zweigrevision behandelt und eine semistrukturierte Verschmelzung wie im ersten Fall durchgeführt werden.

Kapitel 6

Zusammenfassung

Unstrukturierte textuelle Konfliktbehandlung ist unabhängig vom verwendeten Versionsverwaltungssystem und der verwendeten Programmiersprache. Sie stellt den Standard für die Verschmelzung von Programmrevisionen im Drei-Wege-Merge dar. Da das textuelle Verfahren zeilenbasiert arbeitet und keine Kenntnis von der Struktur der Programmiersprache hat, ist es nicht möglich auf individuelle Spracheigenschaften einzugehen. Konflikte stellen sich hier immer als sich textuell überlappende Änderungen dar und können nicht automatisch gelöst werden.

Im Gegensatz dazu besitzt strukturierte Konfliktbehandlung Detailwissen über die Programmiersprache. In Kombination mit einer angepassten Versionsverwaltung ist es möglich, Verschmelzungen bis auf Anweisungsebene durchzuführen. Dieser Ansatz löst Konflikte, die die Struktur der Sprache betreffen, kann aber Verhaltenskonflikte einführen. Wegen der enormen Menge an benötigtem Kontextwissen zu Sprache und Versionshistorie sind diese Ansätze nicht allgemein anwendbar und auf eine bestimmte Sprache und Versionsverwaltung begrenzt.

Semistrukturierte Konfliktbehandlung verwendet Aspekte aus beiden Bereichen. So erhält das Verfahren über eine annotierte Grammatik genügend Hintergrundinformation zu Programmiersprachen, um Ordnungskonflikte zu erkennen und zu behandeln. Mit Hilfe dieses Wissens können die an einem Merge beteiligten Programme in Program Structure Tree überführt und durch den generischen Mergealgorithmus verschmolzen werden. Da zu jedem Programmelement ein eigener Konfliktbehandlungsmechanismus hinterlegt werden kann, lässt sich die Menge der behandelbaren Konflikte erweitern. Kann ein Ordnungskonflikt nicht gelöst werden, werden die in Konflikt stehenden Quelltextstellen mit den unstrukturierten Verfahren hervorgehoben.

Der Ansatz zeigt sich als generisch, da bei Fehlen einer passenden Grammatik auf das zeilenbasierte Verfahren zurückgegriffen werden kann. Zudem ist die Schnittstelle zur Integration neuer Sprachen gering gehalten. So reicht die Formulierung der Grammatik

und die Hinterlegung des passenden Mergeverfahrens pro Element, um das Verfahren anzupassen.

Diese Arbeit vergleicht anhand breit ausgewählter Open Source Programme und realistischer Mergeszenarien das Verhalten von unstrukturierter und semistrukturierter Konfliktbehandlung. Mangels zur Verfügung stehender Projekte konnten keine Daten zu strukturierter Konfliktbehandlung erhoben werden. Zum Vergleich wurde die Projekthistorie der Programme analysiert. Aus diesen Daten ließen Mergeszenarien identifizieren. Mergeszenarien bestehen dabei aus Revisionen, die an einem Merge beteiligt waren oder aus Revisionen für ein Merge realistisch wäre.

Da aus der Versionsgeschichte nicht immer eindeutig hervorgeht, wann ein Merge durchgeführt wurde, ist man auf Kommentare der Entwickler angewiesen. Da auch Projekte gefunden wurden, in den Merges nur selten durchgeführt werden, werden zusätzliche hypothetische Mergeszenarien konstruiert. Diese simulieren die Reintegration eines Entwicklungszweiges zu realistischen Zeitpunkten in der Versionshistorie. Diese Zeitpunkte sind markiert durch das Ende eines Entwicklungszweiges, den Abschluss einer längerfristigen Entwicklung im Zweig oder das Ende einer abwechselnden Commitfolge in verschiedenen Zweigen.

Die Analyse anhand der Projekthistorie ist mühsam. Aus diesem Grund wurde ein Algorithmus entworfen, der die Projekthistorie zweier Entwicklungszweige gegenüberstellt und hinsichtlich der Commitreihenfolge gruppiert. Als weitere Information liefert er die Zahl der gemeinsam geänderten Datei. Da Konflikte beim Merge dieser Dateien entstehen, konnten so zusätzliche Szenarien bestimmt werden.

Insgesamt wurden für den Vergleich 24 Softwareprojekte mit insgesamt 180 Mergeszenarien herangezogen. Dabei umfassen die Mergeszenarien ca. 50.000.000 Zeilen an Quellcode aus drei verschiedenen Programmiersprachen. Diese Mergeszenarien wurden jeweils mit unstrukturierter und semistrukturierter Konfliktbehandlung durchgespielt. Zum Vergleich der Ergebnisse wurde pro Revision die Zahl der Konflikte, Konfliktzeilen und Zahl der Dateien mit Konflikten bestimmt.

Es lässt sich festhalten, dass semistrukturierte Konfliktbehandlung in den meisten Szenarien bessere Ergebnisse, niedrigere Kennzahlen, generiert. Bei 108 Szenarien reduziert sich die Zahl der Konflikte im Schnitt um $34 \pm 21\%$. Die Zahl der Konfliktzeilen reduziert sich bei 148 Szenarien um durchschnittlich $61 \pm 21\%$.

Für 46 Mergeszenarien konnte allerdings eine Erhöhung der Konflikte um $109 \pm 170\%$ und ein ähnlicher Anstieg bei den Konfliktzeilen gemessen werden. Die Ursache hierfür lässt sich auf das Umbenennungsproblem zurückführen. Durch Umbenennung oder Verschiebung von Methoden bis hin zu Ordnerstrukturen, kann das Mergetripel

nicht vollständig erzeugt werden. Da die Mergeverfahren diese Situation gegensätzlich behandeln, kommt es in diesen Fällen zu einer großen Abweichung der Kennzahlen.

Eine Lösung dieses Problems ist noch offen, aber die Effekte könnten bei einem kombinierten Ansatz gemildert werden. So wird jeder Merge mit beiden Verfahren durchgeführt und als Resultat wird das jenige mit der geringsten Konfliktzahl verwendet. Ebenso könnte die »Zweigrevision fehlt« Konstellation in eine »Basisrevision fehlt« Konstellation umgewandelt werden. Dies würde Mergeregel-bedingte Auslöschungen im Ergebnis verhindern.

Unstrukturierte wie semistrukturierte Konfliktbehandlung zeigt sich anfällig für Änderungen der Quelltextformatierung. Dieser Fall äußert sich ähnlich wie Umbenennung und betrifft Stellen an denen das zeilenbasierte Verfahren zum Einsatz kommt. Glücklicherweise treten derartige Änderungen selten, so waren nur Ausschnitte von zwei betrachteten Projekten betroffen.

Nicht unerwähnt darf ein durch die Aufgliederung erzielter Effekt bleiben. Konflikte in semistrukturierter Behandlung sind an strukturelle Grenzen gebunden. So werden Programmelement übergreifende Konflikte ausgeschlossen. Dies stellt eine Erleichterung bei der manuellen Konfliktbehandlung durch den Entwickler dar. Die Grenzen der jeweiligen Konflikte sind klar gezeichnet und somit lässt sich die Konfliktursache und die Lösung leichter überblicken.

Die positiven Ergebnisse semistrukturierter Konfliktbehandlung hängen von der Projektstruktur ab. Bei Projekten in denen die Verzeichnis und Klassenstruktur stabil, bzw. die Branchstruktur aktuell gehalten wird, sind bessere Ergebnisse zu erwarten. Ist sie hingegen lückenhaft, kommt das Umbenennungsproblem zum Tragen und erzeugt schlechtere Kennzahlen.

Anhang A

Tabellen

A.1 Ergebnisse semistrukturierter Konfliktbehandlung

Tabelle A.1: Ergebnisse bei semistrukturierter Konfliktbehandlung: Konflikte (K), Zeilen mit Konflikten (K-LOC), Dateien mit Konflikten (K-Dat) und semantische Konflikte (SK)

Projekt	Revisionen	K	K-LOC	K-Dat	SK
AutoWikiBrowser	rev6386-6479	7	160	3	0
AutoWikiBrowser	rev6386-6496	8	132	4	0
AutoWikiBrowser	rev6386-6553	11	303	6	0
AutoWikiBrowser	rev6386-6607	13	433	5	1
AutoWikiBrowser	rev6386-6663	40	969	6	1
AutoWikiBrowser	rev6386-6682	6	122	4	0
AutoWikiBrowser	rev6386-6724	15	226	9	0
AutoWikiBrowser	rev6386-6735	5	140	5	0
AutoWikiBrowser	rev6386-6799	45	775	11	0
BitPim	rev2895-2929	3	154	1	0
BitPim	rev2895-2954	10	396	5	1
BitPim	rev2895-2991	18	512	4	1
BitPim	rev2895-2993	4	201	1	0
BitPim	rev3177-3237	13	311	2	7
BitPim	rev3177-3275	63	767	5	10
BitPim	rev3177-3296	64	957	6	9
CruiseControl.NET	rev4657-4714	4	63	4	0
CruiseControl.NET	rev4657-4902	10	249	7	0

Fortsetzung auf nächster Seite

Tabelle A.1 – Fortsetzung von vorheriger Seite

Projekt	Revisionen	K	K-LOC	K-Dat	SK
CruiseControl.NET	rev4657-5050	54	1150	23	1
CruiseControl.NET	rev5315-5419	32	253	11	39
CruiseControl.NET	rev6234-6868	15	340	10	1
CruiseControl.NET	rev6234-6870	19	394	13	1
CruiseControl.NET	rev6984-7213	0	0	0	0
CruiseControl.NET	rev6984-7321	0	0	0	0
CruiseControl.NET	rev6984-7492	10	62	4	0
DrJava	rev3734-3786	6	49	5	0
DrJava	rev3734-3788	8	59	5	0
DrJava	rev3734-3807	10	328	6	1
DrJava	rev4989-5004	9	459	2	0
DrJava	rev4989-5019	26	130	11	1
DrJava	rev4989-5044	51	367	26	0
DrJava	rev4989-5058	13	131	4	0
DrJava	rev5266-5336	8	279	4	0
DrJava	rev5319-5330	15	167	2	1
DrJava	rev5319-5332	12	136	1	1
emesene	rev1286-1505	4	17	4	17
Eraser	rev1386-1611	2	57	1	0
Eraser	rev1518-1611	2	56	1	0
Eraser	rev1518-1680	21	766	14	0
Eraser	rev1518-1744	33	785	14	0
Eraser	rev1518-1768	40	982	17	0
Eraser	rev1518-1801	46	1177	19	0
eXe	rev2283-2337	1	7	1	0
eXe	rev2283-2366	0	0	0	1
eXe	rev2283-2409	7	82	3	1
eXe	rev2669-2722	11	157	7	2
eXe	rev2735-2828	3	18	3	6
eXe	rev2735-2877	7	52	6	6
eXe	rev3426-3457	0	0	0	0
eXe	rev3426-3500	1	3	1	0
eXe	rev3426-3513	0	0	0	0
FireIRC IRC Client	rev20-50	3	23	2	0
FreeCol	rev5884-5962	4	155	4	0
FreeCol	rev5884-6055	12	492	8	0
FreeCol	rev5884-6110	16	724	6	0
FreeCol	rev5884-6265	28	1257	12	0

Fortsetzung auf nächster Seite

Tabelle A.1 – Fortsetzung von vorheriger Seite

Projekt	Revisionen	K	K-LOC	K-Dat	SK
FreeCol	rev5884-6362	43	1494	19	0
FreeCol	rev5884-6440	245	4319	66	1
FreeCol	rev5884-6616	439	5637	111	2
FreeCol	rev5884-6672	445	5877	116	2
FreeCol	rev5884-6742	443	6147	117	2
FreeCol	rev5884-6843	455	6548	121	3
GenealogyJ	rev5531-5561	0	0	0	0
GenealogyJ	rev5531-5725	5	89	3	0
GenealogyJ	rev5537-5610	30	341	17	2
GenealogyJ	rev5537-5673	30	341	17	2
GenealogyJ	rev5676-6013	1	27	1	0
GenealogyJ	rev5676-6125	1	27	1	0
GenealogyJ	rev6127-6244	3	95	3	0
GenealogyJ	rev6127-6310	5	151	5	0
GenealogyJ	rev6127-6410	5	189	5	0
GenealogyJ	rev6127-6531	6	201	6	0
iFolder	rev6767-6829	0	0	0	134
iFolder	rev6893-6939	11	144	5	0
iFolder	rev7104-7145	4	55	2	0
iFolder	rev7104-7277	24	427	11	0
iText	rev2818-3036	52	1613	24	6
iText	rev2818-3108	815	21510	198	243
iText	rev2818-3191	652	36374	157	216
iText	rev2818-3306	787	41102	204	283
iText	rev2818-3392	802	42509	204	306
iText	rev2818-3560	796	45099	208	268
iText	rev2818-3625	746	42231	200	486
iText	rev2818-3786	907	45473	208	270
iText	rev2818-3988	774	39204	201	269
iText	rev2818-4022	725	35976	187	274
JabRef	rev2119-2213	5	167	3	0
JabRef	rev2119-2290	33	1377	18	0
JabRef	rev2119-2422	72	2745	43	3
JabRef	rev2119-2471	88	3254	40	3
JabRef	rev2119-2563	112	4401	52	4
JabRef	rev2119-2619	120	4676	54	4
JabRef	rev2620-2839	17	290	11	1
JabRef	rev2620-2863	24	488	11	1

Fortsetzung auf nächster Seite

Tabelle A.1 – Fortsetzung von vorheriger Seite

Projekt	Revisionen	K	K-LOC	K-Dat	SK
JabRef	rev2620-2959	4	134	3	1
JabRef	rev2959-3203	12	285	6	0
jEdit	rev16588-16755	2	9	2	0
jEdit	rev16588-16883	2	9	2	0
jEdit	rev16588-17060	3	24	2	0
jEdit	rev16588-17316	3	20	2	0
jEdit	rev16588-17492	3	20	2	0
jEdit	rev16588-17551	3	20	2	0
jEdit	rev16883-16964	14	305	6	0
jEdit	rev4676-4998	14	233	2	0
jEdit	rev9022-9450	9	78	3	0
jEdit	rev9022-9462	11	77	3	0
JFreeChart	rev91-1012	442	6712	121	13
JFreeChart	rev91-1269	509	5898	166	21
JFreeChart	rev91-1538	567	8800	166	26
JFreeChart	rev91-1825	440	4995	136	45
JFreeChart	rev91-2020	555	7224	164	48
JFreeChart	rev91-208	12	209	6	0
JFreeChart	rev91-2272	864	17756	247	52
JFreeChart	rev91-389	34	868	12	1
JFreeChart	rev91-588	59	1240	29	1
JFreeChart	rev91-775	125	1301	40	1
Jmol	rev11338-11438	24	1895	7	1
Jmol	rev11338-11538	50	3556	12	2
Jmol	rev11338-11638	57	4751	16	3
Jmol	rev11338-11738	79	5772	24	3
Jmol	rev11338-11838	102	7003	33	4
Jmol	rev11338-11938	118	7880	35	4
Jmol	rev11338-12038	134	8162	41	6
Jmol	rev11338-12138	159	9110	45	6
Jmol	rev11338-12238	185	10005	57	6
Jmol	rev11338-12338	217	10558	63	7
matplotlib	rev7315-7404	0	0	0	6
matplotlib	rev7315-7444	1	6	1	9
matplotlib	rev7315-7531	6	260	4	12
matplotlib	rev7315-7618	6	264	4	11
matplotlib	rev7315-7734	7	290	5	12
matplotlib	rev7315-7844	6	264	4	19

Fortsetzung auf nächster Seite

Tabelle A.1 – Fortsetzung von vorheriger Seite

Projekt	Revisionen	K	K-LOC	K-Dat	SK
matplotlib	rev7315-7944	14	514	10	21
matplotlib	rev7315-8057	24	629	15	21
matplotlib	rev7315-8135	35	865	20	22
matplotlib	rev7315-8375	53	1573	25	22
NASA WorldWind	rev2714-2853	3	133	2	0
NASA WorldWind	rev2714-2956	19	1098	4	1
NASA WorldWind	rev2714-3196	63	4503	24	2
NASA WorldWind	rev2714-3419	65	4614	25	3
NASA WorldWind	rev3136-3275	2	106	2	0
NASA WorldWind	rev3136-3458	3	135	2	0
NASA WorldWind	rev3136-3555	28	2379	6	0
NASA WorldWind	rev3136-3650	105	5659	8	77
PMD	rev5929-6010	5	111	4	0
PMD	rev5929-6135	55	668	9	1
PMD	rev5929-6198	70	1148	15	1
PMD	rev5929-6296	75	1339	20	1
PMD	rev5929-6425	563	9593	41	14
PMD	rev5929-6595	579	10172	47	18
PMD	rev5929-6700	583	10537	50	19
PMD	rev5929-6835	593	10695	55	20
PMD	rev5929-7018	593	10831	61	47
PMD	rev5929-7073	595	10863	63	47
Process Hacker	rev2147-2158	1	4	1	1
RSS Bandit	rev295-327	32	647	10	4
RSS Bandit	rev295-365	52	1269	7	27
RSS Bandit	rev295-417	60	2621	10	47
RSS Bandit	rev295-501	59	1373	7	48
SpamBayes	rev2415-2547	8	94	1	0
SpamBayes	rev2415-2601	9	105	2	1
SpamBayes	rev2415-2698	19	190	6	2
SpamBayes	rev2415-2775	26	305	9	3
SpamBayes	rev2415-2874	36	670	12	3
SpamBayes	rev2415-2881	36	692	12	3
SQuirrelSQL	rev4007-4051	0	0	0	0
SQuirrelSQL	rev4007-4103	3	33	2	0
SQuirrelSQL	rev4007-4212	20	331	10	1
SQuirrelSQL	rev4007-4321	29	574	15	1
SQuirrelSQL	rev4007-4394	34	852	19	1

Fortsetzung auf nächster Seite

Tabelle A.1 – Fortsetzung von vorheriger Seite

Projekt	Revisionen	K	K-LOC	K-Dat	SK
SQuirrelSQL	rev4007-4516	37	899	22	1
SQuirrelSQL	rev4007-4908	55	1496	28	5
SQuirrelSQL	rev4007-5081	57	1510	29	5
SQuirrelSQL	rev5082-5155	3	37	2	0
SQuirrelSQL	rev5082-5351	4	61	3	0
Wicd	rev519-537	6	154	3	2
Wicd	rev519-546	7	168	3	2
Wicd	rev519-557	8	222	3	2
Wicd	rev519-568	11	272	4	3
Wicd	rev519-578	11	272	4	3

A.2 Ergebnisse unstrukturierter Konfliktbehandlung

Tabelle A.2: Ergebnisse bei unstrukturierter Konfliktbehandlung: Konflikte (K), Zeilen mit Konflikten (K-LOC) und Dateien mit Konflikten (K-Dat) und semantische Konflikte (SK)

Projekt	Revisionen	K	K-LOC	K-Dat	SK
AutoWikiBrowser	rev6386-6479	8	124	3	0
AutoWikiBrowser	rev6386-6496	12	152	4	0
AutoWikiBrowser	rev6386-6553	12	159	6	0
AutoWikiBrowser	rev6386-6607	18	510	6	0
AutoWikiBrowser	rev6386-6663	50	885	10	0
AutoWikiBrowser	rev6386-6682	6	137	3	0
AutoWikiBrowser	rev6386-6724	19	590	8	0
AutoWikiBrowser	rev6386-6735	6	167	5	0
AutoWikiBrowser	rev6386-6799	48	1039	12	0
BitPim	rev2895-2929	3	836	2	0
BitPim	rev2895-2954	11	2831	7	0
BitPim	rev2895-2991	16	3041	7	0
BitPim	rev2895-2993	6	1943	4	0
BitPim	rev3177-3237	4	7295	3	0
BitPim	rev3177-3275	75	9483	9	0
BitPim	rev3177-3296	86	11254	9	0
CruiseControl.NET	rev4657-4714	9	256	8	0
CruiseControl.NET	rev4657-4902	21	869	14	0
CruiseControl.NET	rev4657-5050	75	3676	38	0
CruiseControl.NET	rev5315-5419	32	2229	13	0
CruiseControl.NET	rev6234-6868	34	1038	18	0
CruiseControl.NET	rev6234-6870	45	1278	23	0
CruiseControl.NET	rev6984-7213	1	6	1	0
CruiseControl.NET	rev6984-7321	2	372	2	0
CruiseControl.NET	rev6984-7492	15	472	6	0
DrJava	rev3734-3786	17	99	7	0
DrJava	rev3734-3788	18	131	7	0
DrJava	rev3734-3807	28	876	8	0
DrJava	rev4989-5004	10	499	2	0
DrJava	rev4989-5019	29	160	12	0
DrJava	rev4989-5044	48	265	23	0
DrJava	rev4989-5058	13	133	5	0
DrJava	rev5266-5336	14	363	4	0

Fortsetzung auf nächster Seite

Tabelle A.2 – Fortsetzung von vorheriger Seite

Projekt	Revisionen	K	K-LOC	K-Dat	SK
DrJava	rev5319-5330	15	236	2	0
DrJava	rev5319-5332	11	165	1	0
emesene	rev1286-1505	7	400	7	0
Eraser	rev1386-1611	3	88	1	0
Eraser	rev1518-1611	3	87	1	0
Eraser	rev1518-1680	18	2454	13	0
Eraser	rev1518-1744	35	3237	16	0
Eraser	rev1518-1768	38	3306	18	0
Eraser	rev1518-1801	46	3704	20	0
eXe	rev2283-2337	1	2	1	0
eXe	rev2283-2366	2	175	2	0
eXe	rev2283-2409	7	259	5	0
eXe	rev2669-2722	15	311	9	0
eXe	rev2735-2828	6	82	4	0
eXe	rev2735-2877	10	135	7	0
eXe	rev3426-3457	1	5	1	0
eXe	rev3426-3500	3	20	2	0
eXe	rev3426-3513	2	17	2	0
FireIRC IRC Client	rev20-50	5	36	3	0
FreeCol	rev5884-5962	5	363	5	0
FreeCol	rev5884-6055	14	1094	9	0
FreeCol	rev5884-6110	19	1121	8	0
FreeCol	rev5884-6265	31	1802	12	0
FreeCol	rev5884-6362	43	2630	20	0
FreeCol	rev5884-6440	258	5784	72	0
FreeCol	rev5884-6616	484	7017	120	0
FreeCol	rev5884-6672	493	7598	124	0
FreeCol	rev5884-6742	502	8702	125	0
FreeCol	rev5884-6843	517	9949	135	0
GenealogyJ	rev5531-5561	1	3	1	0
GenealogyJ	rev5531-5725	15	89	3	0
GenealogyJ	rev5537-5610	74	2255	34	0
GenealogyJ	rev5537-5673	74	2255	34	0
GenealogyJ	rev5676-6013	2	3	2	0
GenealogyJ	rev5676-6125	2	3	2	0
GenealogyJ	rev6127-6244	2	30	2	0
GenealogyJ	rev6127-6310	5	56	4	0
GenealogyJ	rev6127-6410	5	94	4	0

Fortsetzung auf nächster Seite

Tabelle A.2 – Fortsetzung von vorheriger Seite

Projekt	Revisionen	K	K-LOC	K-Dat	SK
GenealogyJ	rev6127-6531	6	106	5	0
iFolder	rev6767-6829	134	268	1	0
iFolder	rev6893-6939	11	144	5	0
iFolder	rev7104-7145	4	55	2	0
iFolder	rev7104-7277	29	447	16	0
iText	rev2818-3036	56	2891	25	0
iText	rev2818-3108	215	63643	215	0
iText	rev2818-3191	177	115527	177	0
iText	rev2818-3306	233	133863	233	0
iText	rev2818-3392	248	138718	248	0
iText	rev2818-3560	259	142619	259	0
iText	rev2818-3625	263	136028	263	0
iText	rev2818-3786	255	136424	255	0
iText	rev2818-3988	251	135318	251	0
iText	rev2818-4022	234	132535	234	0
JabRef	rev2119-2213	16	356	11	0
JabRef	rev2119-2290	49	640	22	0
JabRef	rev2119-2422	97	1741	44	0
JabRef	rev2119-2471	139	2535	51	0
JabRef	rev2119-2563	174	4049	59	0
JabRef	rev2119-2619	183	4204	61	0
JabRef	rev2620-2839	27	1251	12	0
JabRef	rev2620-2863	35	1580	13	0
JabRef	rev2620-2959	18	430	6	0
JabRef	rev2959-3203	12	223	6	0
jEdit	rev16588-16755	2	9	2	0
jEdit	rev16588-16883	2	9	2	0
jEdit	rev16588-17060	3	24	2	0
jEdit	rev16588-17316	3	20	2	0
jEdit	rev16588-17492	3	20	2	0
jEdit	rev16588-17551	3	20	2	0
jEdit	rev16883-16964	21	192	6	0
jEdit	rev4676-4998	4	442	3	0
jEdit	rev9022-9450	12	145	3	0
jEdit	rev9022-9462	14	169	3	0
JFreeChart	rev91-1012	514	15027	169	0
JFreeChart	rev91-1269	752	18640	264	0
JFreeChart	rev91-1538	781	25503	290	0

Fortsetzung auf nächster Seite

Tabelle A.2 – Fortsetzung von vorheriger Seite

Projekt	Revisionen	K	K-LOC	K-Dat	SK
JFreeChart	rev91-1825	802	24915	298	0
JFreeChart	rev91-2020	948	30416	326	0
JFreeChart	rev91-208	19	1943	14	0
JFreeChart	rev91-2272	1514	41780	414	0
JFreeChart	rev91-389	45	3562	27	0
JFreeChart	rev91-588	85	9097	52	0
JFreeChart	rev91-775	139	10536	73	0
Jmol	rev11338-11438	14	17375	9	0
Jmol	rev11338-11538	32	18551	15	0
Jmol	rev11338-11638	43	18714	22	0
Jmol	rev11338-11738	63	19076	30	0
Jmol	rev11338-11838	84	19705	41	0
Jmol	rev11338-11938	93	19788	42	0
Jmol	rev11338-12038	107	19959	48	0
Jmol	rev11338-12138	132	21661	53	0
Jmol	rev11338-12238	171	22633	65	0
Jmol	rev11338-12338	202	23483	73	0
matplotlib	rev7315-7404	3	15	3	0
matplotlib	rev7315-7444	4	21	3	0
matplotlib	rev7315-7531	56	271	6	0
matplotlib	rev7315-7618	56	290	6	0
matplotlib	rev7315-7734	58	305	8	0
matplotlib	rev7315-7844	59	334	9	0
matplotlib	rev7315-7944	65	397	14	0
matplotlib	rev7315-8057	82	803	24	0
matplotlib	rev7315-8135	86	843	26	0
matplotlib	rev7315-8375	99	1725	31	0
NASA WorldWind	rev2714-2853	3	44	2	0
NASA WorldWind	rev2714-2956	12	1135	4	0
NASA WorldWind	rev2714-3196	53	15052	33	0
NASA WorldWind	rev2714-3419	54	15565	34	0
NASA WorldWind	rev3136-3275	4	343	2	0
NASA WorldWind	rev3136-3458	4	332	2	0
NASA WorldWind	rev3136-3555	11	6193	6	0
NASA WorldWind	rev3136-3650	41	8902	9	0
PMD	rev5929-6010	4	338	4	0
PMD	rev5929-6135	11	11533	11	0
PMD	rev5929-6198	23	12619	18	0

Fortsetzung auf nächster Seite

Tabelle A.2 – Fortsetzung von vorheriger Seite

Projekt	Revisionen	K	K-LOC	K-Dat	SK
PMD	rev5929-6296	27	12784	22	0
PMD	rev5929-6425	60	23022	55	0
PMD	rev5929-6595	70	23263	62	0
PMD	rev5929-6700	76	24168	65	0
PMD	rev5929-6835	80	24698	71	0
PMD	rev5929-7018	95	26116	84	0
PMD	rev5929-7073	101	26147	86	0
Process Hacker	rev2147-2158	1	4	1	0
RSS Bandit	rev295-327	39	1555	12	0
RSS Bandit	rev295-365	48	4471	11	0
RSS Bandit	rev295-417	84	5119	12	0
RSS Bandit	rev295-501	89	5198	12	0
SpamBayes	rev2415-2547	6	99	2	0
SpamBayes	rev2415-2601	9	136	3	0
SpamBayes	rev2415-2698	19	305	8	0
SpamBayes	rev2415-2775	33	432	12	0
SpamBayes	rev2415-2874	41	2996	17	0
SpamBayes	rev2415-2881	41	3028	17	0
SQuirrelSQL	rev4007-4051	1	4	1	0
SQuirrelSQL	rev4007-4103	7	299	3	0
SQuirrelSQL	rev4007-4212	23	1598	12	0
SQuirrelSQL	rev4007-4321	33	4076	18	0
SQuirrelSQL	rev4007-4394	38	4432	22	0
SQuirrelSQL	rev4007-4516	44	4472	26	0
SQuirrelSQL	rev4007-4908	52	6325	34	0
SQuirrelSQL	rev4007-5081	54	4788	35	0
SQuirrelSQL	rev5082-5155	3	17	2	0
SQuirrelSQL	rev5082-5351	7	194	4	0
Wicd	rev519-537	8	474	4	0
Wicd	rev519-546	9	585	4	0
Wicd	rev519-557	9	582	4	0
Wicd	rev519-568	13	2340	5	0
Wicd	rev519-578	13	2354	5	0

Literaturverzeichnis

[AKL09] Sven Apel, Christian Kästner und Christian Lengauer: *FEATUREHOUSE: Language-independent, automated software composition*. In: *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, Seiten 221–231, Washington, DC, USA, Mai 2009. IEEE Computer Society.

[AL08] Sven Apel und Christian Lengauer: *Superimposition: A Language–Independent Approach to Software Composition*. In: C. Pautasso und É. Tanter (Herausgeber): *Proceedings of the ETAPS International Symposium on Software Composition*, Band 4954 der Reihe *SC 2008*, Seiten 20–35. Springer-Verlag Berlin Heidelberg, März 2008.

[All80] Eric Allman: *An Introduction to the Source Code Control System*. Computer Science Division, Berkeley, CA:Department of Electrical Engineering and Computer Science, University of California, 1980.

[ALL+10] Sven Apel, Jörg Liebig, Christian Lengauer, Christian Kästner und William R. Cook: *Semistructured Merge in Revision Control Systems*. In: *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, Seiten 13–19, Universität Duisburg-Essen, Januar 2010.

[ALMK08] Sven Apel, Christian Lengauer, Bernhard Möller und Christian Kästner: *An Algebra for Features and Feature Composition*. In: J. Meseguer und G. Roşu (Herausgeber): *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology (AMAST)*, Band 5140 der Reihe *Lecture Notes in Computer Science*, Seiten 36–50. Springer-Verlag Berlin Heidelberg, Juli 2008.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi und Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2. Auflage, 2006.

[Aza07] Kalid Azad: *Intro to Distributed Version Control*, Oktober 2007. http://betterexplained.com/articles/intro-to-distributed-version-control-illustrated/, Abgerufen: April 2011.

[BB95] Don Bolinger und Tan Bronson: *Applying RCS and SCCS*. O'Reilly & Associates Inc., 1. Auflage, August 1995.

[BIB90] Brian Berliner, Prisma Inc und Mark Dabling Blvd: *CVS II: Parallelizing Software Development*, 1990.

[Buf95] Jim Buffenbarger: *Syntactic Software Merging*. In: *Selected papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*, Seiten 153–172, London, UK, 1995. Springer-Verlag.

[Cha10] Scott Chacon: *Git Community Book*, 2010. http://book.git-scm.com, Abgerufen: April 2011.

[CSFP10] Ben Collins-Sussman, Brian W. Fitzpatrick und C. Michael Pilato: *Version Control with Subversion – For Subversion 1.6*, 2010. http://svnbook.red-bean.com, Abgerufen: April 2011.

[CW98] Reidar Conradi und Bernhard Westfechtel: *Version models for software configuration management*. ACM Comput. Surv., 30:232–282, Juni 1998, ISSN 0360-0300.

[EK06] André Eickler und Alfons Kemper: *Datenbanksysteme*. Oldenbourg Wissenschaftsverlag, 6. Auflage, 2006.

[Fis] Shlomi Fish: *Version Control System Comparison*. http://better-scm.berlios.de/comparison/comparison.html, Abgerufen: April 2011.

[Her99] Helmut Herold: *Linux-Unix-Kurzreferenz*. Pearson Education, 1999.

[HM76] J. W. Hunt und M. D. McIlroy: *An Algorithm for Differential File Comparison*. Technischer Bericht CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

[JL94] D. Jackson und D. A. Ladd: *Semantic Diff: a tool for summarizing the effects of modifications*. In: *Proceedings of the International Conference on Software Maintenance, 1994.*, Seiten 243–252, 1994.

[KKP07] Sanjeev Khanna, Keshav Kunal und Benjamin C. Pierce: *A formal investigation of Diff3*. In: *FSTTCS'07: Proceedings of the 27th international conference on Foundations of software technology and theoretical computer science*, Seiten 485–496, Berlin, Heidelberg, 2007. Springer-Verlag.

[Men02] Tom Mens: *A State-of-the-Art Survey on Software Merging*. IEEE Trans. Softw. Eng., 28:449–462, Mai 2002.

[NFH05] Iulian Neamtiu, Jeffrey S. Foster und Michael Hicks: *Understanding source code evolution using abstract syntax tree matching*. In: *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, Seiten 1–5, New York, NY, USA, Mai 2005. ACM.

[O'S09a] Bryan O'Sullivan: *Making sense of revision-control systems*. Commun. ACM, 52:56–62, September 2009.

[O'S09b] Bryan O'Sullivan: *Mercurial: The Definitive Guide*. O'Reilly Media, 1. Auflage, 2009.

[Pod09] Igor Podolskiy: *Verteilte Versionsverwaltung von hierarchisch-sequentiellen Datenstrukturen am Beispiel von Eisenbahnfahrplänen*. Diplomarbeit, Universität Stuttgart, November 2009.

[S+10] Richard Stallman *et al.*: *Comparing and Merging Files - Manual for GNU Diffutils*. Free Software Foundation Inc., April 2010. http://www.gnu.org/software/diffutils/manual/, Abgerufen: April 2011.

[Tic85] Walter F. Tichy: *RCS—a system for version control*. Softw. Pract. Exper., 15(7):637–654, 1985.

[Tic95] Walter F. Tichy: *Manpage for GNU Merge*. RCS Programmpaket, Juni 1995.

[TvS07] Andrew S. Tanenbaum und Maarten van Steen: *Verteilte Systeme*. Pearson Studium, 2. Auflage, 2007.

[Wes91] Bernhard Westfechtel: *Structure-oriented merging of revisions of software documents*. In: *Proceedings of the 3rd international workshop on Software configuration management*, SCM '91, Seiten 68–79, New York, NY, USA, 1991. ACM.

[Yan94] Wuu Yang: *How to merge program texts*. J. Syst. Softw., 27(2):129–135, 1994.

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis