Analyse von Versionshistorien

Mark Wiesemann

Inhaltsverzeichnis

1 Einleitung **2**

2 eROSE – Reengineering of Software Evolution **2**

2.1 Einführung 3

2.2 Transaktionen 3

2.3 Regeln 5

2.4 Auswertung des Ansatzes 7

2.4.1 Navigation durch den Code 8

2.4.2 Verhinderung von Fehlern 8

2.4.3 Abgeschlossenheit 8

2.4.4 Granularität 8

2.5 Ähnlicher Ansatz von A. T. T. Ying 9

3 Weitere Ansätze **9**

3.1 Hipikat 10

3.2 CVSSearch 12

3.3 Version Editor 13

3.4 VssConneXion 16

4 Bewertung und Ausblick **17**

5 Zusammenfassung 18

1 Einleitung

Die Versionshistorie wird bei der Softwareentwicklung verwendet, um die verschiedenen Versionen des Quellcodes zu verwalten. Damit ist es z.B. möglich, zu prüfen, welcher Entwickler was und zu welchem Zeitpunkt geändert hat, alte Versionen wiederherzustellen, mehrere „Zweige“ eines Projekts (z.B. eine stabile und eine Beta-Version) zu verwalten und den Zugriff der Entwickler auf den Quellcode zu kontrollieren.

Die Informationen aus der Versionshistorie lassen sich u.a. nutzen, um Änderungen vorzuschlagen, um das Debugging zu erleichtern oder um problematische und fehleranfällige Module zu identifizieren. Auch zur Abschätzung von Release-Intervallen, von Codewachstum, von Wartungskosten oder der Fehlerzahl ist die Versionshistorie nützlich.

Versions-Kontrollsysteme (Version Control Systems, kurz: VCS) verwalten die Versionshistorie und erlauben es, jederzeit wieder auf ältere Versionen von Dateien zuzugreifen. Mit jeder neuen Version („commit“), die in einem VCS abgelegt wird, werden vom Entwickler auch Informationen über die vorgenommen Änderungen gespeichert. Beispiele für Versions-Kontrollsysteme sind CVS (Concurrent Versions System), das neuere und als CVS-Nachfolger gedachte Subversion oder das vor einem Jahr zur Verwaltung des Linux-Kernels entwickelte git.

Versions-Kontrollsysteme arbeiten meist entweder datei- oder änderungsbasiert. Während im ersten Fall jede Datei einzeln betrachtet wird, wird im zweiten Fall betrachtet, welche Dateien von einer Änderung des Quellcodes betroffen sind. Zum Beispiel arbeitet CVS dateibasiert, Subversion aber änderungsbasiert.

In dieser Arbeit werden verschiedene Ansätze vorgestellt, um mit Hilfe der Versionshistorie Vorschläge für weitere Änderungen zu generieren oder um einen Überblick über die letzten Änderungen einer Methode im Quellcode zu bekommen.

2 eROSE – Reengineering of Software Evolution

eROSE (Abkürzung für „Reengineering of Software Evolution“) ist ein am Lehrstuhl für Softwaretechnik der Universität des Saarlandes entwickeltes Plugin für die Software-Entwicklungsumgebung Eclipse. Mit Hilfe von Datamining auf der Versionshistorie sollen den Entwicklern verwandte Änderungen gezeigt werden. Es arbeitet ähnlich dem Vorschlagssystem von Amazon nach dem Prinzip „Programmers who changed these functions also changed . . .“.

2.1 Einführung

Abbildung 1 zeigt das Vorschlagsprinzip von eROSE am Beispiel des Source-Codes von Eclipse: In Zeile 173 wurde eine neue Konstante für eine zusätzliche Tabellenspalte eingefügt. Das eROSE-Plugin zeigt nun im unteren Bereich des Fensters verwandte Änderungen früherer Transaktionen an (Transaktionen bestehen aus mehreren Änderungen an mehreren Stellen im Code; vgl. Abschnitt 2.2). Während der erste Hinweis, dass die `COLUMN_COUNT`-Konstante (in Zeile 171) bei ähnlichen Transaktionen geändert wurde, dem Entwickler nicht weiterhilft, ist bereits der zweite Hinweis wichtig, denn auch das in Zeile 182 beginnende Array `columnHeaders[]` muss um ein weiteres Element erweitert werden, damit die neu definierte Konstante sinnvoll eingesetzt werden kann.

Abbildung 1: Das Eclipse-Plugin eROSE zeigt Stellen im Code (hier: Eclipse-Source-Code), an denen bei früheren ähnlichen Transaktionen weitere Änderungen vorgenommen wurden

2.2 Transaktionen

Abbildung 2: Datenfluss in eROSE

Abbildung 2 zeigt den Datenfluss in eROSE. Der eROSE-Server liest zunächst das Archiv eines CVS-Servers ein und gruppiert alle Änderungen zu Transaktionen. Mit Hilfe von Datamining bildet der Server aus den Transaktionen Regeln nach dem Prinzip: „Wenn die `COLUMN`-Konstanten geändert werden, dann wird üblicherweise auch das Array `columnHeaders[]` geändert." (vgl. Abschnitt 2.1) Wenn ein Entwickler einen Teil des Codes (Konstante, Variable, Methode etc.) ändert, sucht der eROSE-Client in den Regeln („Rule Set") nach einer anwendbaren Regel und macht passende Vorschläge (z.B. die Änderung von `columnHeaders[]`).

Formal ist eine ÄNDERUNG eine Abbildung $\delta : \mathcal{P} \to \mathcal{P}$, die ein PRODUKT $p \in \mathcal{P}$ in ein GEÄNDERTES PRODUKT $p' = \delta(p) \in \mathcal{P}$ überführt. $\mathcal{P}$ ist die Menge aller Produkte und $\mathcal{C} = \mathcal{P} \to \mathcal{P}$ ist die Menge aller Änderungen. Ein Produkt ist z.B. eine Datei, eine Klasse oder eine Methode.

Mehrere Änderungen werden als Komposition zusammengefasst ($\circ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$). Eine TRANSAKTION besteht aus mehreren Änderungen an mehreren Stellen im Code. Zum Beispiel wird die Transaktion $\Delta_{1,2}$ zwischen zwei Versionen $p_1, p_2 \in \mathcal{P}$ bestehend aus n individuellen Änderungen $\delta_1, \ldots, \delta_n$ geschrieben als $\Delta_{1,2} = \delta_1 \circ \delta_2 \circ \cdots \circ \delta_n$ mit $\Delta_{1,2}(p_1) = (\delta_1 \circ \delta_2 \circ \cdots \circ \delta_n)(p_1) = \delta_1(\delta_2(\cdots \delta_n(p_1))) = p_2$.

ENTITÄTEN werden benutzt, um alle syntaktischen Komponenten des Codes zusammenzufassen. Eine Entität ist ein Tripel (f, c, i). Dabei ist f der Name der betroffenen Datei, c ist die syntaktische Komponentenart (Konstante, Variable, Methode, Klasse,

Datei etc.) und i ist der Bezeichner der Komponente.

Die Abbildung $entities$ liefert alle Entitäten, die von einer Änderung oder einer Transaktion betroffen sind. Am Beispiel von Abschnitt 2.1 ergibt sich:

$$entities(\Delta) = entities(\delta_1) \cup \cdots \cup entities(\delta_n) = \left\{ \begin{array}{lll} (\text{EventsView.java}, & class, & \texttt{EventsView}), \\ (\text{EventsView.java}, & constant, & \texttt{COLUMN\_COUNT}), \\ (\text{EventsView.java}, & field, & \texttt{columnHeaders[]}), \\ (\text{EventsView.java}, & method, & \texttt{getForeground()}), \\ \ldots & & \end{array} \right\}$$

Der eROSE-Server versucht, einige Probleme von CVS zu umgehen. Während andere Versions-Kontrollsysteme die Versionierung änderungsbasiert vornehmen, geht CVS dateibasiert vor. Um eine änderungsbasierte Versionierung zu bekommen, gruppiert eROSE einzelne Dateiänderungen zu Transaktionen. Dabei werden zwei aufeinanderfolgende Änderungen δ_i und δ_{i+1} von dem gleichen Entwickler und mit der gleichen Anmerkung zu einer Transaktion Δ zusammengefasst, wenn der Abstand zwischen den Änderungen maximal 200 Sekunden beträgt. Zusammenfassungen („merge“) von Zweigen („branch“) werden genauso als Transaktionen erkannt. Dies ist aber nicht erwünscht, da die Änderungen in solchen großen Transaktionen meist keinen logischen Zusammenhang haben. eROSE ignoriert daher alle Änderungen, die mehr als 30 Entitäten umfassen.

Abbildung 3: eROSE fasst Code zu Entitäten zusammen (aus [ZWDZ04a])

CVS kann nur Dateien und Zeilennummern für Änderungen verwalten, nicht aber Details über die Syntax. Daher parst eROSE die Dateien und ordnet Entitäten Zeilennummern (von / bis) zu. Wie in Abbildung 3 zu sehen ist, kann eROSE jede Änderung (vom CVS-Server nur durch Dateinamen und Zeilennummern identifiziert) den Entitäten zuordnen.

2.3 Regeln

eROSE benutzt Datamining, um aus Transaktionen Regeln zu bilden. Ein Beispiel für eine Regel ist:

$$\{(\text{EventsView.java}, constant, \texttt{COLUMN\_TIME})\} \Rightarrow \left\{ \begin{array}{lll} (\text{EventsView.java}, & constant, & \texttt{COLUMN\_COUNT}), \\ (\text{EventsView.java}, & field, & \texttt{columnHeaders[]}) \end{array} \right\} \quad (1)$$

Der Pfeil $\Rightarrow$ ist keine logische Implikation, sondern besagt nur, dass möglicherweise auch die Konstante `COLUMN_COUNT` und das Array `columnHeaders[]` geändert werden sollten, wenn die Konstante `COLUMN_TIME` bzw. Code davor oder dahinter geändert wurde.

Formal ist eine REGEL ein Paar (x_1, x_2) von zwei disjunkten Entitäten-Mengen x_1 und x_2. In der Regel $x_1 \Rightarrow x_2$ wird x_1 als BEDINGUNGSTEIL („antecedent“) und x_2 als AKTIONSTEIL („consequent“) bezeichnet.

Die von eROSE berechneten Regeln sind nicht hundertprozentig genau, sondern nur stochastische Interpretationen der Transaktionen, aus denen die Regeln gebildet werden, und basieren auf der Menge und Qualität der vorliegenden Daten.

Die HÄUFIGKEIT einer Menge x in einer Menge T von Transaktionen ist formal definiert als $\mathrm{frq}(T, x) = |\{t \mid t \in T, x \subseteq t\}|$.

Die Qualität einer Regel wird durch zwei Werte bestimmt:

- UNTERSTÜTZUNG: Gibt an, aus wie vielen Transaktionen die Regel abgeleitet wurde. Formal ist die Unterstützung einer Regel $x_1 \Rightarrow x_2$ durch eine Menge von Transaktionen T definiert als $\mathrm{supp}(T, x_1 \Rightarrow x_2) = \mathrm{frq}(T, x_1 \cup x_2)$.

- KONFIDENZ: Gibt die Stärke des Aktionsteils der Regel an und wird aus dem Verhältnis der Anzahl der Transaktionen, die den Bedingungsteil betreffen, zu der Anzahl der Transaktionen, die den Aktionsteil betreffen, berechnet. Formal ist die KONFIDENZ einer Regel $x_1 \Rightarrow x_2$ definiert als $\mathrm{conf}(T, x_1 \Rightarrow x_2) = \frac{\mathrm{frq}(T, x_1 \cup x_2)}{\mathrm{frq}(T, x_1)}$.

Im Beispiel aus Abschnitt 2.1 bedeutet das: Die Konstante `COLOR_TIME` wurde in zwei Transaktionen geändert, bei einer dieser Transaktionen wurden auch die Konstante `COLUMN_COUNT` und das Array `columnHeaders[]` geändert. Damit ergibt sich für die Unterstützung der Wert 1. Der Wert für die Konfidenz berechnet sich als $1/2 = 0,5$.

Angenommen, ein Entwickler habe eine Reihe von Änderungen $\delta_1 \circ \delta_2 \circ \cdots \circ \delta_k$ vorgenommen. Die Menge der geänderten Entitäten (bezeichnet als SITUATION) sei $\Sigma = entitites(\delta_1 \circ \delta_2 \circ \cdots \circ \delta_k)$.

Im Beispiel wurde Code nach der Konstanten `COLOR_TIME` eingefügt. Die Situation ist daher:

$$\Sigma = \{(\text{EventsView.java}, constant, \texttt{COLUMN\_TIME})\} \tag{2}$$

Für die Berechnung der Vorschläge für weitere Änderungen wendet eROSE matchende Regeln an. Eine Regel matcht eine Menge von geänderten Entitäten, wenn diese Menge gleich dem Aktionsteil ist. Die Menge der Vorschläge für eine Situation Σ und eine Menge von Regeln R ist definiert als die Vereinigung aller matchenden Regeln:

$$\mathrm{apply}(\Sigma, R) = \bigcup_{(\Sigma \Rightarrow x_2) \in R} x_2$$

Im Beispiel der Situation Σ aus (2) und der Regel r aus (1) schlägt eROSE den Bedingungsteil von r vor:

$$\text{apply}(\Sigma, \{r\}) = \left\{ \begin{array}{lll} \text{(EventsView.java,} & constant, & \texttt{COLUMN\_COUNT}), \\ \text{(EventsView.java,} & field, & \texttt{columnHeaders[]}) \end{array} \right\}$$

Zur Berechnung der Regeln verwendet eROSE den Apriori-Algorithmus. Dieser Algorithmus beginnt mit minimalen Werten für Unterstützung und Konfidenz und berechnet dann die Menge aller Regeln in zwei Phasen. Eine Menge von Entitäten sei als HÄUFIG bezeichnet, wenn sie mehr als die minimale Unterstützung hat.

Phase 1 Der Algorithmus bildet für jede Transaktion eine Menge von Entitäten, die in der Transaktion vorhanden sind. Diese Menge vergrößert sich bei jedem Durchlauf, da eine solche Menge von Entitäten nur dann häufig sein kann, wenn ihre Teilmengen häufig sind. Am Ende ergibt sich die Menge F aller häufigen Mengen von Entitäten.

Phase 2 Der Algorithmus berechnet die Regeln aus den Mengen in F. Für jede Entitäten-Menge $E \in F$ werden die Regeln $E - X \Rightarrow X$ berechnet ($X \subseteq E$). Es werden also Regeln gebildet, die nicht die vollständige Menge der Entitäten im Bedingungsteil haben. Stattdessen kommt die Teilmenge X, die aus E herausgenommen wurde, in den Aktionsteil, um damit später die Vorschläge berechnen zu können. Diese Regeln haben alle die gleiche Unterstützung, aber unterschiedliche Konfidenz. Nur Regeln mit einer größeren als der minimalen Konfidenz werden zurückgegeben.

Normalerweise werden alle Regeln im Voraus berechnet. Da dies aber mehrere Tage dauern kann, setzen die eROSE-Entwickler zwei Optimierungen ein:

- ABHÄNGIGE AKTIONSTEILE: Es werden nur Regeln gebildet, deren Aktionsteile ähnlich zum Aktionsteil der aktuellen Situation Σ sind.
- EINZELNE BEDINGUNGSTEILE: Es werden nur Regeln gebildet, deren Bedingungsteil nur aus einer Entität bestehen. Das ist ausreichend, da eROSE sowieso die Vereinigung aller Bedingungsteile berechnet.

2.4 Auswertung des Ansatzes

Zimmer, Weißgerber, Diehl und Zeller haben für die folgenden vier Aspekte anhand von acht großen Open-Source-Projekten untersucht, welche Qualität die Änderungsvorschläge von eROSE haben (vgl. [ZWDZ04b]):

- NAVIGATION DURCH DEN CODE: Es sei eine einzelne geänderte Entität gegeben. Kann eROSE den Entwicklern auf andere Stellen im Code verweisen, die typischerweise auch geändert werden sollten?

- VERHINDERUNG VON FEHLERN: Kann eROSE Fehler verhindern? Weist eROSE z.B. auf eine nicht geänderte Entität hin, nachdem mehrere andere Entitäten geändert wurden?

- ABGESCHLOSSENHEIT: Angenommen, dass alle zu ändernden Entitäten geändert wurden. Schlägt eROSE fälschlicherweise weitere Änderungen vor?

- GRANULARITÄT: Wie gut sind die Ergebnisse von eROSE, wenn es nicht auf Methoden-Ebene (und kleineren Entitäten) arbeitet, sondern auf Dateien?

2.4.1 Navigation durch den Code

Nach einer geänderten Entität kann eROSE 15 Prozent aller weiteren zu ändernden Entitäten der gleichen Transaktion vorschlagen. In 64 Prozent aller Transaktionen verweisen die ersten drei Vorschläge auf die richtigen Stellen im Code.

Während beim Eclipse-Source-Code genau der Durchschnittswert von 15 Prozent erreicht wird, ist der Wert bei GCC mit 28 Prozent deutlich höher. KOffice kommt dagegen nur auf acht Prozent. Der Grund ist, dass in KOffice viele neue Funktionen eingefügt werden, während bei GCC der Code stabil ist und hauptsächlich nur Fehler behoben werden.

2.4.2 Verhinderung von Fehlern

Wenn in einer Transaktion eine Änderung vergessen wurde, kann eROSE im Schnitt nur vier Prozent dieser Fälle erkennen. Durchschnittlich ist jeder zweite Vorschlag korrekt.

Für GCC liegt die Korrektheit bei 81 Prozent und fehlende Änderungen werden bei jeder fünften Transaktion erkannt. Bei KOffice ist nur jeder vierte Vorschlag (24 Prozent) korrekt und fehlende Änderungen werden nur in 0,3 Prozent aller Fälle entdeckt.

2.4.3 Abgeschlossenheit

Durchschnittlich in nur zwei Prozent aller Fälle schlägt eROSE noch weitere Änderungen vor, obwohl bereits alle notwendigen Änderungen vorgenommen wurden.

Die Werte der acht Open-Source-Projekte liegen in diesem Aspekt mit einem bis zwei Prozent dicht beieinander. Nur bei GCC schlägt eROSE immerhin in etwa jedem zwanzigsten Fall noch eine weitere Änderung vor (4,7 Prozent).

2.4.4 Granularität

Wenn statt feiner Granularität (Methoden, Variablen, Konstanten etc.) eine gröbere Granularität verwendet wird, schlägt eROSE 26 statt 15 Prozent der Dateien vor, die in einer Transaktion geändert wurden. Die Korrektheit der ersten drei Vorschläge steigt von 64 auf 70 Prozent. Bei KOffice steigt der Wert sogar von acht auf 24 Prozent.

Trotz der höheren Prozentwerte sind die Vorschläge von eROSE mit gröberer Granularität aber gleichzeitig weniger nützlich, da nur noch Dateinamen angegeben werden, nicht aber die zu ändernde Methode.

2.5 Ähnlicher Ansatz von A. T. T. Ying

Abbildung 4: Drei Stufen im Ansatz von A. T. T. Ying (aus [Yin03])

Parallel und unabhängig zur Entwicklung von eROSE hat Annie Tsui Tsui Ying im Rahmen Ihrer Masterarbeit an der University of British Columbia einen ähnlichen Ansatz entwickelt. Wie in Abbildung 4 zu sehen ist, verwendet auch dieser Ansatz Datamining, um aus der Versionshistorie Muster und Regeln zu berechnen und daraus wiederum dem Entwickler Vorschläge für weitere Änderungen anbieten zu können.

Im Unterschied zu eROSE arbeitet dieser Ansatz aber nur auf der Ebene von Dateien und nicht z.B. auf der Ebene von Methoden. Ein weiterer Unterschied ist, dass die eROSE-Entwickler beim Datamining Beziehungsregeln verwenden, während Yings Ansatz analysiert, wie häufig sich Teile des Codes in einer Transaktion wiederholen. Dadurch ist hier nur der Wert für die Unterstützung relevant, nicht aber der Wert für die Konfidenz.

Ying hat im Gegensatz zu Zimmer, Weißgerber, Diehl und Zeller anhand der Open-Source-Projekte Eclipse und Mozilla auch die Qualität der korrekten Änderungsvorschläge untersucht. Für einen C-Programmierer ist es z.B. nicht sehr hilfreich, wenn er darauf hingewiesen wird, dass nach der Änderung von `example.h` auch die Datei `example.c` geändert werden sollte. Ying hat die Vorschläge daher in die Kategorien „überraschend“, „neutral“ und „offensichtlich“ eingeordnet. Die meisten Änderungsvorschläge für Eclipse und Mozilla fielen in die Kategorien „neutral“ oder „offensichtlich“. Die Kategorie „überraschend“ trat nur selten auf, da die meisten Quellcode-Dateien dieser Projekte direkt oder indirekt zusammenhängen.

3 Weitere Ansätze

Neben eROSE gibt es weitere Ansätze, um mit Hilfe der Versionshistorie die Entwicklung von Software zu erleichtern, indem Vorschläge für weitere Änderungen gegeben,

Abbildung 5: Beziehungen zwischen Artefakten, die Hipikat analysiert

verwandte Bugreports und Mailinglisten-Einträge angezeigt oder die letzten Änderungen einer Methode dargestellt werden.

3.1 Hipikat

Hipikat ist ein gemeinsames Projekt der University of British Columbia, des IBM Ottawa Software Lab und des National Research Council of Canada. Der Name stammt aus der Westafrikanischen Sprache Wolof und bedeutet übersetzt „weitgeöffnete Augen“.

Im Gegensatz zu eROSE berücksichtigt es nicht nur Versionhistorien, sondern auch Daten aus Bugtracking-Systemen wie Bugzilla, Online-Dokumentationen und Mailinglisten oder Newsgroups, um dem Entwickler bei der Programmierung von Software zu helfen. Ein weiterer Unterschied zu eROSE ist, dass Hipikat Vorschläge für weitere Änderungen nur auf Dateiebene bietet, während eROSE standardmäßig die Vorschläge auf kleinerer Ebene (z.B. Methoden) bietet.

Abbildung 5 zeigt die vier verschiedenen Artefakt-Typen, die Hipikat analysiert:

- Bug-Reports, Feature-Requests und Änderungswünsche aus Bugzilla-Einträgen
- Versionen von Dateien in Versions-Kontrollsystemen (bisher nur CVS)
- Nachrichten aus Mailinglisten und Newsgroup
- Dokumente, z.B. Beschreibungen über das Design einer Software auf den Webseiten eines Projekts

Diese Artefakte werden von den Mitgliedern des Projekts oder anderen Personen erstellt. Bugzilla-Einträge bilden einen zentralen Punkt in der Abbildung, da z.B. in einer Version einer Datei ein Änderungswunsch implementiert sein kann, da es weitere Dokumente zu diesem Wunsch geben kann und da mehrere Personen z.B. in einer Newsgroup über die gewünschte Änderung diskutieren können.

Abbildung 6: Server-Architektur von Hipikat

Die Struktur des Hipikat-Servers ist in Abbildung 6 dargestellt. Die Hauptkomponente ist die Datenbank, die die Artefakte verwaltet. Der Server hat drei Funktionen, die in Modulen gekapselt sind.

1. AKTUALISIERUNG („Update“): Dieses Modul ist in vier Untermodule aufgeteilt. Ein Modul durchsucht die Bugzilla-Seiten, ein Modul durchsucht die Versi-

onshistorie eines CVS-Servers, ein weiteres Modul durchsucht über das NNTP-Protokoll Newsgroups und das letzte Modul durchsucht z.B. die Eclipse-Website. Neue oder geänderte Artefakte werden in die Datenbank eingefügt und Listener im Identifizierungsmodul werden über die Aktualisierungen benachrichtigt.

2. IDENTIFIZIERUNG („Identification“): Mehrere Untermodule analysieren die Artefakte in der Datenbank. Der so genannte „log-matcher“ durchsucht z.B. die Kommentare in der Versionshistorie nach IDs von Bugzilla-Einträgen und der „newsgroup-thread-matcher“ analysiert die „References“-Einträge von Newsgroup-Postings und (re)konstruiert daraus zusammenhängende Threads.

3. AUSWAHL („Selection“): Dieses Modul analysiert und beantwortet Anfragen des Entwicklers mit Hilfe der Artefakt-Datenbank. Die Ergebnisse werden im XML-Format zurückgegeben.

Für das Eclipse-Projekt konnte Hipikat im September 2002 auf 21.668 Bugzilla-Einträge (sowie weitere 72.536 Kommentare), 125.429 CVS-Versionseinträge, 36.864 Newsgroup-Postings und auf 1.459 Webseiten zurückgreifen.

Abbildung 7: Suchmaske und Ergebnisansicht von Hipikat (aus [CM03b])

Abbildung 7 zeigt die Suchmaske und die Ergebnisansicht innerhalb von Eclipse. Als Beispiel wurde hier über die Suchmaske nach dem Bugzilla-Eintrag mit der ID 116 gesucht. Unterhalb der Ergebnisansicht zeigt Hipikat Verweise auf ähnliche Artefakte an (vgl. Abbildung 8).

Neben der Suchmöglichkeit über die Suchmaske bietet Hipikat auch die Möglichkeit, über ein Kontextmenü die Funktion „Query Hipikat“ aufzurufen. Wenn z.B. in der „Console“-Ansicht eine Exception angezeigt wird, kann man über diese Funktion Hipikat nach verwandten Artefakten wie Bugzilla-Einträgen suchen lassen.

Die Ergebnisse von Hipikat sind nur Berechnungen und Analysen, die keine hundertprozentige Genauigkeit bieten können. In [CM03a] haben Davor Cubranic und Gail C. Murphy für das Eclipse-Projekt die Genauigkeit untersucht. Am 24. August 2002 gab es 9.418 Bugs, die als geschlossen markiert waren. Die Identifizierungs-Untermodule „log-matcher“ und „activity-matcher“ konnten für 60 Prozent (5.688) der Bugs Verknüpfungen zur Versionshistorie in der Artefakt-Datenbank herstellen. Allerdings wurden auch für weitere 2.810 Bugs, die noch nicht geschlossen waren, solche Verknüpfungen hergestellt.

3.2 CVSSearch

CVSSearch ist ein von Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang und Amir Michail an der University of New South Wales ent-

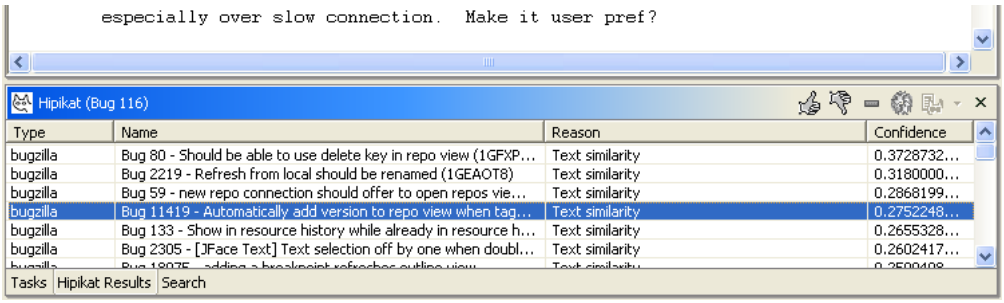

Abbildung 8: Hipikat zeigt ähnliche Bugs unterhalb der Ergebnisansicht in Eclipse an

wickeltes webbasiertes Programm, um nach Teilen des Quellcodes anhand von CVS-Kommentaren zu suchen. Der Vorteil gegenüber Kommentaren im Code ist, dass gerade bei größeren Open-Source-Projekten kaum Kommentare im Code vorhanden oder nur von schlechter Qualität sind, während die CVS-Kommentare hauptsächlich dazu dienen, die anderen Entwickler über die aktuellen Änderungen zu informieren, so dass die Qualität der Kommentare hier meist deutlich besser ist.

Abbildung 9: Suchmaske von CVSSearch

Abbildung 9[1] zeigt die Suchmaske von CVSSearch. Die Ansicht der gefundenen Dateien und eine Detailansicht zu einer Datei sind in Abbildung 10 zu sehen.

Abbildung 10: Ergebnisübersicht und -details von CVSSearch (aus [CCW+01])

CVSSearch benutzt eine Abbildung zwischen den CVS-Kommentaren und den Code-Zeilen, auf die sich die Kommentare beziehen. Diese Abbildung wird nur für die Zeilen der aktuellsten Version einer Datei erstellt, während aber alle Versionen dieser Datei betrachtet werden. Wenn z.B. die Zeile 3 in der aktuellsten Version zum ersten Mal in Version 1.2 eingefügt wurde und danach in den Version 1.4 und 1.5 geändert wurde, dann muss Zeile 3 mit den Kommentaren aus den Versionen 1.2, 1.4 und 1.5 verknüpft werden. CVSSearch benutzt die „Managing Gigabytes"-Datenbank und speichert für jede Zeile die verknüpften Kommentare.

Wenn über die Suchmaske nach einem oder mehreren Begriffen gesucht wird, kombiniert CVSSearch die Ergebnisse aus seiner Datenbank mit den Ergebnissen von *grep*. Auf diese Weise wird aus zwei Perspektiven gesucht: zum einen der Blick in den Code, zum anderen der Blick auf das, was die Entwickler über den Code schreiben.

Für eine Auswertung wurden durch Studenten insgesamt 703 Suchanfragen gestellt. Davon wurden 40 Prozent durch die Suche in den CVS-Kommentaren am besten beantwortet, 32 Prozent am besten durch die Suche mit *grep* und 28 Prozent wurden mit beiden Varianten gleich gut beantwortet.

3.3 Version Editor

Version Editor ist ein von David L. Atkins an den Bell Laboratories entwickelter Editor, der auf die Editoren *vi* und *Emacs* aufsetzt. Er arbeitet mit Versionskontrollsystemen zusammen, die das Source Code Control System (SCCS) oder das Revision Control System (RCS) unterstützen – SCCS und RCS wurden inzwischen von CVS abgelöst.

Die Idee des Version Editor ist es, nicht nur den aktuellen Code zu zeigen, sondern auch diejenigen Zeilen anzuzeigen und zu kennzeichnen, die seit dem letzten Checkout geändert oder gelöscht wurden. In der Standardkonfiguration des Version Editor zeigt er Zeilen, die geändert wurden in fetter Schrift an. Gelöschte Zeilen werden unterstrichen dargestellt. Zusätzlich werden für die Zeile, in der sich der Cursor befindet, Informationen zur letzten Änderung angezeigt (Zeilennummern der Änderung, Kommentar zum Commit).

Als Beispiel sei ein großes Programm angenommen, das u.a. Dateien ausliest und diese auch mit Daten füllt. Das Programm sei lange Zeit ohne Probleme gelaufen, aber es wurde nun entdeckt, dass es plötzlich zu viele gleichzeitig geöffnete Dateien gibt. Abbildung 11 zeigt eine der Methoden, die Dateien öffnen und schließen. Während

[1] http://web.archive.org/web/20030806094001/horn.cse.unsw.edu.au/~cvssearch/Query.cgi

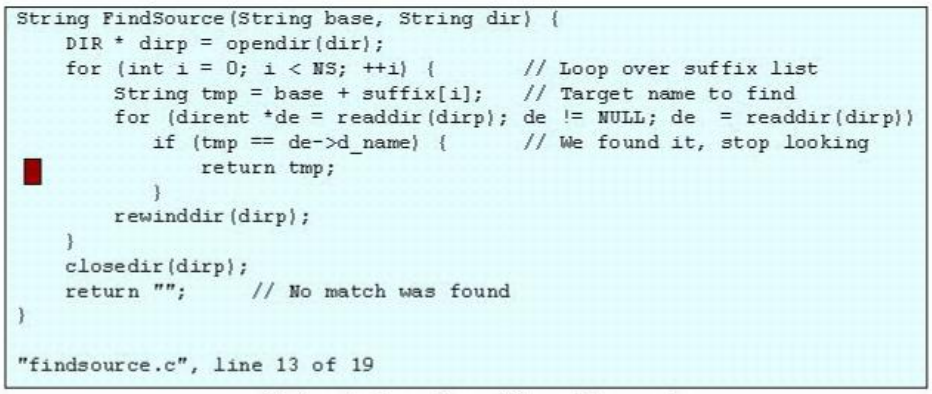

Abbildung 11: Codebeispiel (aus [Atk04])

die Methode `FindSource` auf den ersten Blick richtig erscheinen mag, ist die Anzeige der Funktion mit Version Editor (vgl. Abbildung 12) hilfreich, um den Fehler zu erkennen. Der Editor zeigt hier zusätzlich die geänderten und gelöschten Zeilen an, so dass der Entwickler auf einen Blick sehen kann, was an dieser Methode geändert wurde. Der Entwickler kann nun besser erkennen, dass die Methode aus der `for`-Schleife heraus verlassen wird, dass aber weiterhin die Datei erst nach der `for`-Schleife geschlossen wird.

3.4 VssConneXion

VssConneXion ist ein von EPocalipse Software entwickeltes Plugin für Borland Delphi und bietet eine Anbindung von Delphi an Visual SourceSafe, ein von Microsoft entwickeltes Versions-Kontrollsystem. Neben Standardfunktionen wie Checkout und Commit bietet es auch eine Option namens „Source Analysis“.

Das Konzept ist das gleiche wie beim Version Editor, allerdings werden die Informationen zu gelöschten und geänderten Zeilen nicht direkt im Editor angezeigt, sondern in einem separaten Fenster. Dieses Fenster bietet vier verschiedene Ansichten:

- Änderungen der einzelnen Zeilen mit Angabe des Änderungsdatums
- Änderungen des Entwicklers, der sich an der Visual SourceSafe-Datenbank angemeldet hat
- Entwickler, die die einzelnen Zeilen geänderten Zeilen geändert haben (vgl. Abbildung 13)
- Anzahl der Änderungen der einzelnen Zeilen (vgl. Abbildung 14)

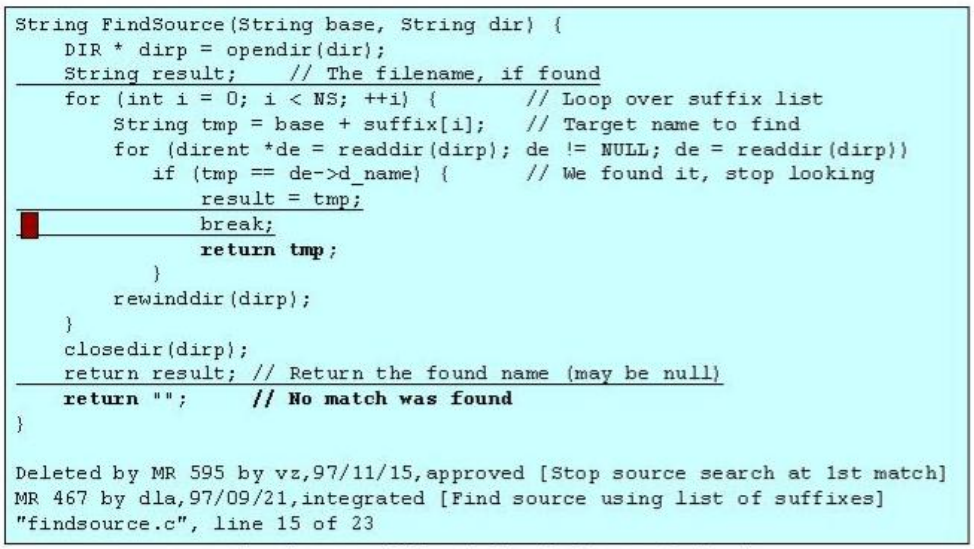

Abbildung 12: Version Editor zeigt geänderten Code (hinzugefügte Zeilen fett, gelöschte Zeilen unterstrichten; aus [Atk04])

Die Angaben zum Entwickler, zur Anzahl oder zum Datum werden durch farbliche Unterlegung dargestellt. Zusätzlich wird zur aktuell markierten Zeile die Versionshistorie mit der Versionsnummer, dem Namen des Entwicklers, dem Änderungsdatum und dem Kommentar angezeigt.

4 Bewertung und Ausblick

eROSE kann zwar durchschnittlich nur einen relativ geringen Teil (15 Prozent) der weiteren zu ändernden Entitäten in einer Transaktion vorschlagen, allerdings verdoppelt sich dieser Anteil, wenn der Code wie bei GCC stabil ist. Gleichzeitig sind die Vorschläge durchschnittlich in zwei Drittel aller Fälle korrekt. Somit kann man sich nicht allein auf die Vorschläge von eROSE verlassen, bekommt aber häufig eine wertvolle Hilfe.

Hipikat konnte in der Genauigkeitsanalyse 60 Prozent der Bugs mit der Versionshistorie verknüpfen. Somit kann man sich auch auf Hipikat nicht allein verlassen, bekommt aber durch die Verknüpfung von Code, Bug-Reports und Mailinglisten-Einträge eine gute Unterstützung.

Die Analyse zu CVSSearch zeigte, dass die Strategie, sowohl mit Hilfe der Versionshistorie als auch mit Hilfe von *grep* zu suchen, sinnvoll ist. Zwar konnten 28 Prozent der etwa 700 Suchanfragen mit beiden Suchvarianten gleich gut beantwortet werden, aber 40 bzw. 32 Prozent der Anfragen besser über die Kommentare in der Historie oder mit

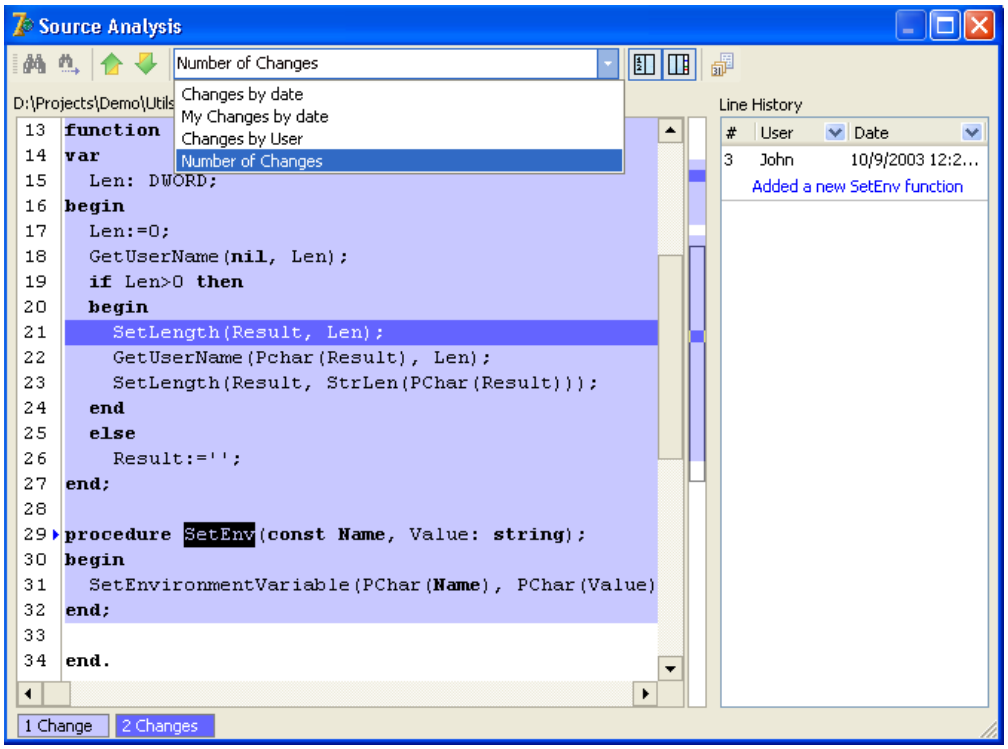

Abbildung 13: VssConneXion zeigt an, wie häufig die einzelnen Zeilen geändert wurden (aus [EPo06])

grep.

Hinter Version Editor und VssConneXion steckt das gleiche Konzept, durch die grafische Oberfläche ist das „Source Analysis“-Fenster aber deutlich übersichtlicher und damit auch sinnvoller zu verwenden als im Version Editor.

Das Eclipse-Plugin eROSE ist als Alpha-Version 0.0.5 zum Download verfügbar, die Version 0.1.0 ist für Juni 2006 geplant. Version 1.8.0 von Hipikat wurde im Juli 2003 fertiggestellt, zur Zeit ist aber – wegen eines geplanten Redesign des Clients – kein Download möglich. Von CVSSearch war die Version 2.0 Beta2 verfügbar[2], zur Zeit ist aber nur der direkte Zugriff auf den CVS-Server möglich. Vom Version Editor konnte keine Download-Version gefunden werden. VssConneXion ist in Version 4 als 30-Tage-Testversion und als Vollversion verfügbar.

eROSE und VssConneXion werden weiterentwickelt, bei Hipikat und CVSSearch liegen die letzten Änderungen mehr als ein Jahr zurück und Version Editor war nie zum freien Download verfügbar. Die drei erstgenannten Tools sind aus meiner Sicht nützlich, leider ist aber keine installierbare Version von Hipikat im Internet zu finden.

[2] http://web.archive.org/web/20031217162927/http://cvssearch.sourceforge.net

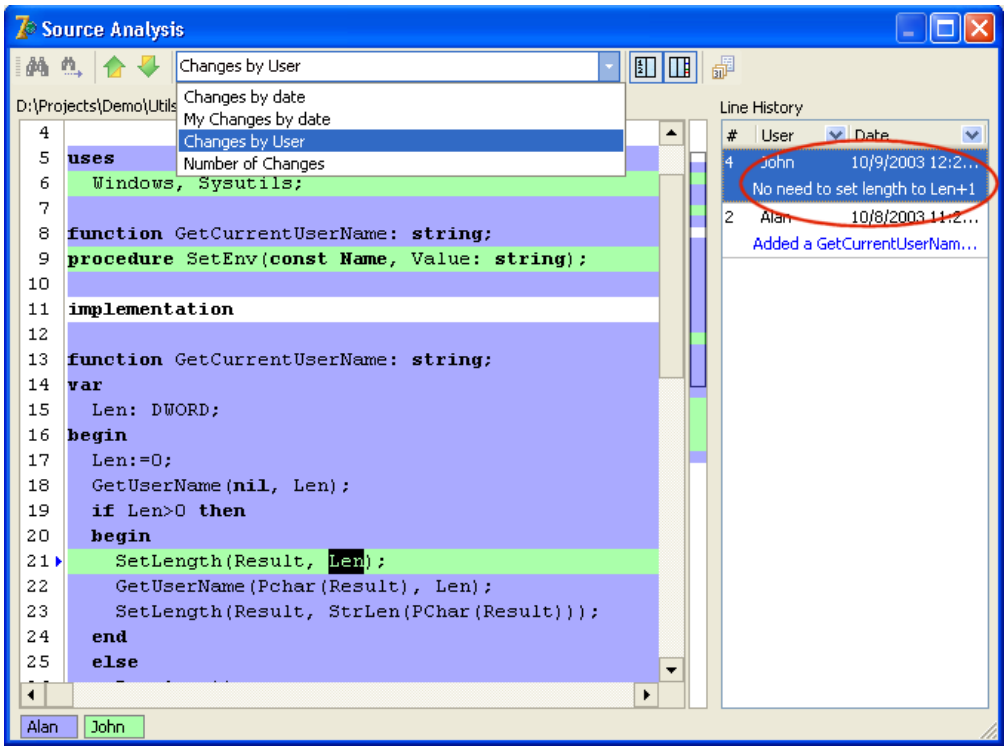

Abbildung 14: VssConneXion zeigt an, welcher Entwickler welche Zeile (zuletzt) geändert hat (aus [EPo06])

5 Zusammenfassung

In dieser Arbeit wurden verschiedene Ansätze vorgestellt, um dem Entwickler auf verschiedene Weisen bei der Programmierung zu unterstützen. eROSE und Hipikat sind Plugins für die weit verbreitete Software-Entwicklungsumgebung Eclipse. CVSSearch ist webbasiert und Version Editor setzt auf die bekannten Editoren *vi* und *Emacs* auf. VssConneXion ist ein Plugin für Borland Delphi.

Alle Ansätze verwenden die Versionshistorie, um entweder Vorschläge für weitere Änderungen (eROSE, Hipikat) anzuzeigen, Code-Fragmente zu suchen (CVSSearch), hinzugefügte bzw. gelöschte Code-Zeilen (Version Editor) oder die Anzahl bzw. Autoren der Änderungen (VssConneXion) anzuzeigen. Hipikat greift zusätzlich zur Versionshistorie auch auf Webseiten, Bugzilla-Datenbanken, Mailinglisten und Newsgroups zu. CVSSearch verwendet neben der Versionshistorie auch *grep*, um aus zwei Perspektiven heraus Code-Fragmente zu suchen.

Literatur

[Atk04] David L. Atkins. *The Story of the Version Editor*, 2004. `http://www.cs.uoregon.edu/~datkins/ve.html`.

[CCW+01] Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang, and Amir Michail. CVSSearch: Searching through Source Code using CVS Comments. In *IEEE International Conference Software Maintenance (ICSM)*, pages 364–374, 2001.

[CM03a] Davor Cubranic and Gail C. Murphy. Hipikat: Recommending Pertinent Software Development Artifacts. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 408–418, Mai 2003.

[CM03b] Davor Cubranic and Gail C. Murphy. *Sample Scenarios*, 2003. `http://www.cs.ubc.ca/labs/spl/projects/hipikat/scenarios.html`.

[EPo06] EPocalipse Software. *EPocalipse Software – VssConneXion*, 2006. `http://www.epocalipse.com/vcx.htm`.

[Yin03] Annie Tsui Tsui Ying. Predicting Source Code Changes by Mining Revision History. Master's thesis, University of British Columbia, Canada, Oktober 2003.

[ZWDZ04a] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. *Data Mining Version Histories*, 2004. `http://www.st.cs.uni-sb.de/papers/icse2004/gradkoll.pdf`.

[ZWDZ04b] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining Version Histories to Guide Software Changes. In *26th International Conference on Software Engineering (ICSE)*, 2004.