Was Software-Archive erzählen

Stephan Diehl
Universität Trier
diehl@acm.org

Andreas Zeller
Universität des Saarlandes
zeller@acm.org

Thomas Zimmermann
Universität des Saarlandes
tz@acm.org

Abstract: Softwaretechnik basiert wie jede andere Wissenschaft auf historischen Erfahrungen: Was hat in der Vergangenheit funktioniert und was nicht? Aus der *Entstehungsgeschichte eines Programms,* wie sie in Software-Archiven aufgezeichnet wurde, kann man solche Erfahrungen bilden und nutzbar machen – etwa um verwandte Programmstellen vorzuschlagen (weil ähnliche Änderungen bereits früher auftraten) oder um Fehlerrisiken vorherzusagen (weil ähnliche Änderungen oder Komponenten sich in der Vergangenheit als fehlerträchtig herausgestellt haben). Erste Systeme, die die Software-Historie ausnutzen, bestechen durch hohe Vorhersagekraft und geringe Anforderungen bei einem wohlorganisierten Entwicklungsprozess.

1 Einleitung

Die *Analyse von Programmen* ist ein zentraler Bestandteil der Softwaretechnik: Überall dort, wo Entscheidungen getroffen werden müssen, die sich auf existierende Produkte beziehen, liefern Verfahren der Programmanalyse die Daten, die diese Entscheidungen unterstützen. Traditionelle Verfahren der Programmanalyse nutzen den (statischen) Programmcode oder (dynamische) Laufzeitinformationen, um Programmeigenschaften zusammenzufassen oder vorherzusagen.

Neben Code und Laufzeitinformation fallen in der Software-Entwicklung aber noch weitere Daten an, die sich mit automatisierten Verfahren weiter verarbeiten lassen und die die traditionelle Programmanalyse ergänzen und bereichern. Diese Daten stecken in *Software-Archiven,* die die Entstehungsgeschichte eines Systems dokumentieren – in Form von *Versionsarchiven,* die die Änderungen aufzeichnen, und *Fehlerdatenbanken,* die aufgetretene Fehler dokumentieren. Die systematische Analyse und Kopplung solcher Archive kann Fragen beantworten, wie sie in der Software-Entwicklung regelmäßig anfallen:

- Ich habe mein System folgendermaßen strukturiert. Ist dies so sinnvoll?
- Ich möchte die Funktion f() ändern. Muss ich noch mehr ändern?
- Welche meiner Komponenten sollte ich am sorgfältigsten testen?
- Ich habe die Wahl zwischen zwei Änderungen. Welche birgt das geringere Risiko?

Solche Fragen sind im Allgemeinen schwer zu beantworten. Die Analyse der Software-Historie eines Projekts kann jedoch erstaunlich zutreffende Vorhersagen treffen – und das

vollautomatisch, ohne dass mit großem Aufwand kontrollierte Experimente oder projektbegleitende Messungen durchgeführt werden müssten. Tatsächlich sind die Voraussetzungen zum Einsatz der Verfahren – ein Versionsarchiv und ggf. eine Fehlerdatenbank – Bestandteile jedweder systematischen Software-Entwicklung.

Die Analyse von Software-Archiven steht noch ganz am Anfang der Entwicklung. In diesem Artikel fassen wir den Stand der Forschung zusammen, und zeigen erste Anwendungen und Erfahrungen auf. Anschließend beschreiben wir kurz zentrale Techniken, die für die Analyse von Software-Archiven eingesetzt werden. Schließlich erörtern wir das zukünftige Potential, aber auch die Grenzen der Technik.

2 Evolutionsmuster

Ein Laie, der durch ein Software-Archiv stöbert, sieht zunächst nichts als eine schier endlose Liste von Änderungen und Fehlermeldungen. Zwar lässt sich die Suche leicht auf bestimmte Mengen eingrenzen – etwa auf die Komponente, für die der Programmierer verantwortlich ist – ein Bild des Gesamt-Systems erschließt sich jedoch erst, wenn die einzelnen Einträge zusammengefasst und abstrahiert dargestellt werden.

Einfache Visualisierungstechniken stellen zunächst das Archiv selbst grafisch dar. WinCVS [Str] etwa zeigt den Versionsgraphen des Archivs als zweidimensionalen Baum. Gall et. al. stellen verschiedene Visualisierungstechniken vor, die sich alle daran orientieren, 2D-Diagramme für die Struktur des Systems zu verwenden, die Zeit in die dritte Dimension zu kodieren und Farbmetriken zur Darstellung von Eigenschaften zu benutzen [GJR99].

Lanza vereint in seinem Werkzeug GEOCRAWLER verschiedene Visualisierungen, die zum Teil auch den zeitlichen Verlauf mit in Betracht ziehen. Besonders hervorzuheben sind hierbei die Evolutionsmatrizen [Lan01]. Diese stellen Klassen als Rechtecke dar, deren Form und Farbe Eigenschaften kodieren. Für die verschiedenen Zeitpunkte werden verschiedene Zeilen verwendet, so dass gleiche Klassen immer untereinander stehen.

Auch SeeSoft [ESS92] stellt zeitliche Verläufe dar. Hier wird für eine Produktversion der jede Zeile des Quelltext verkleinert als Linie dargestellt. Die Farbe der Linie gibt Metrikwerte an – etwa das Datum der letzten Änderung. Ein Schieberegler erlaubt es, eine Zeitreise durch die verschiedenen Versionen eines Archivs zu machen. Das GEVOL-System [CKN+03] stellt verschiedene Typen von Graphen für Java-Programme dar (z.B. Vererbungs- und Aggregationsgraph) und kann mittels linearer Interpolation Animationen für die Übergänge zwischen verschiedenen Versionen des Systems anzeigen.

Die meisten dieser Ansätze dienen dem *Visual Data Mining:* Ziel ist, in den Visualisierungen "interessante" Zusammenhänge zu entdecken und diesen dann gezielt nachzuspüren. Als ein Beispiel für solche Zusammenhänge betrachten wir *historische Kopplung:* Wurden zwei oder mehr Software-Komponenten gemeinsam geändert, nennen wir sie *historisch gekoppelt* – eine Beziehung, die womöglich eine wie auch immer geartete Abhängigkeit ausdrückt [ZDZ03]. Beispielsweise muss nach der Änderung einer Schnittstelle der sie benutzende Code ebenfalls geändert werden – eine klassische Benutzt-Abhängigkeit, die einerseits durch klassische Programmanalyse aufgedeckt werden kann, aber auch unabhängig davon durch Analyse der historischen Änderungen.

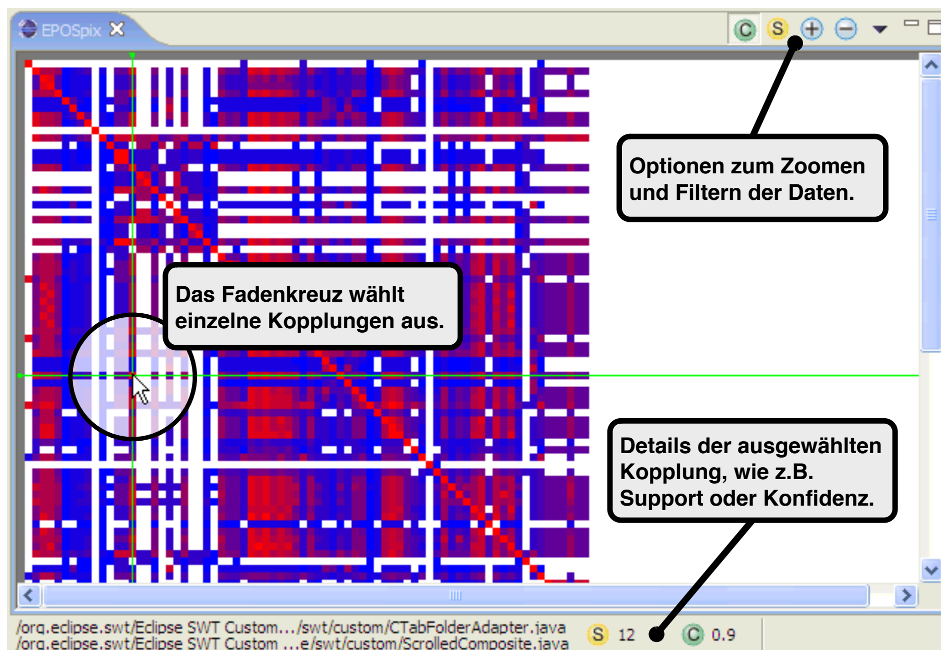

Abbildung 1: Eclipse Pixelmap Plug-In. Die Pixelmap zeigt, ob Dateien über gemeinsame Änderungen stark (rot), schwach (blau), oder gar nicht (weiß) gekoppelt sind.

Die historische Kopplung lässt sich durch *Pixelmaps* visualisieren, wobei auf die Achsen die Komponenten (Dateien) aufgetragen werden; der Schnittpunkt der Koordinaten wird entsprechend der Kopplung eingefärbt. Abbildung 1 etwa zeigt die Pixelmap des Pakets widgets von ECLIPSE SWT [BDW05]: Rote Pixel stehen für starke, blaue für schwache Kopplung, und ein weißer Pixel steht für keine Kopplung – d.h. die betreffenden Dateien wurden niemals gemeinsam geändert. (Die Diagonale ist komplett rot gezeichnet, da jede Datei mit sich selbst gekoppelt ist.)

Da die Dateien entlang der horizontalen und vertikalen Achse gemäß der Verzeichnisstruktur hierarchisch sortiert sind, entstehen entlang der Diagonalen häufig *Blöcke* roter Punkte. Dies ist dann ein Indiz dafür, dass Dateien im selben Verzeichnis häufig zusammen geändert werden – was für die gewählte Verzeichnisstruktur spricht. Besonders interessieren wir uns aber für die *Ausreißer.* Dies sind solche Punkte, die starke Kopplung zwischen *Dateien in verschiedenen Verzeichnissen anzeigen.* Solche Ausreißer können ein Hinweis auf eine schlechte Systemarchitektur sein. Kurz gesagt, wenn in der Pixelmap viele Ausreißer existieren und kaum Blöcke entlang der Diagonalen, könnte dies Anlass sein, den Programmcode des Softwaresystems umzustrukturieren.

Im Vergleich zur klassischen Programmanalyse liefert historische Kopplung weniger präzise Ergebnisse – nicht zuletzt, weil die Ursprungsdaten unvollständig oder verrauscht sein können. Auf der anderen Seite lässt sich historische Kopplung gleichermaßen für alle Arten von Artefakten bestimmen – und so Kopplungen zwischen Code, Bilddateien, oder Textdateien bestimmen, die der klassischen Programmanalyse verborgen blieben.

3 Verwandte Programmstellen

Die Visualisierung von Evolutionsdaten ist kein Selbstzweck: Aus jeder Messung sollten *Taten* folgen – im Fall der Pixelmap etwa eine Restrukturierung, die die Kopplung mindern soll. Historische Kopplung lässt sich aber noch anders einsetzen – nämlich für das Vorschlagen verwandter Programmstellen.

Wer schon einmal bei Amazon Bücher gekauft hat, hat vielleicht den Abschnitt "Kunden, die dieses Buch gekauft haben, haben auch diese Bücher gekauft..." bemerkt. Hier findet man andere Bücher, die üblicherweise zusammen mit dem angezeigten Buch bestellt wurden. Amazon bildet solche Empfehlungen automatisch aus vergangenen Bestellungen.

Wir entwickelten ein ähnliches Feature für Softwareentwicklung: "Programmierer, die diese Programmstellen geändert haben, haben auch diese Programmstellen geändert..." Unser ROSE-Werkzeug extrahiert diese Muster automatisch aus Versionsarchiven. Als Empfehlungen helfen sie dem Entwickler auf vielfältige Weise:

Verbessern der Navigation. Eine der Hauptanwendungen von ROSE ist es, den Benutzer durch Quelltext zu führen: Der Benutzer ändert eine Programmstelle und ROSE empfiehlt automatisch verwandte, auch zu ändernde, Programmstellen.

Abbildung 2 zeigt unser ROSE-Werkzeug als ein Plug-In für die ECLIPSE Programmierumgebung. Der Programmierer hat das Projekt um eine neue Präferenz erweitert, und hat bereits ein Element dem fKeys[]-Feld, das die Präferenzen verwaltet, hinzugefügt. ROSE schlägt nun zusätzliche Programmstellen vor; die Empfehlungen sind aus der Versionsgeschichte abgeleitet.

Zuerst kommen Programmstellen mit der höchsten Wahrscheinlichkeit, dass die Programmstelle tatsächlich zu ändern ist. In Abbildung 2 steht an Position 1 mit einer Wahrscheinlichkeit von 100% die Methode initDefaults(), die Standardwerte für Präferenzen setzt. An Position 3 steht eine HTML Datei mit einer Konfidenz von 72,7% – nach dem Hinzufügen einer neuen Präferenz sollte der Entwickler also auch die Dokumentation aktualisieren. Das letzte Beispiel zeigt eine Stärke von ROSE: Abhängigkeiten zu Dokumentation werden von klassischer Programmanalyse nicht erkannt. Solche „überraschendenden" Abhängigkeiten sind für den Programmierer besonders wertvoll, wie Ying et al. in einer zu ROSE ähnlichen Arbeit gezeigt haben [YMNC04].

Vermeiden von Fehlern. Auch wenn alle Änderungen vollzogen sind, hilft ROSE weiter, Fehler zu vermeiden. Wenn ein Benutzer sich entscheidet, seine Änderungen in das Versionsarchiv zu schreiben, überprüft ROSE vorher, ob es noch Programmstellen mit hoher Konfidenz gibt, die noch nicht geändert wurden. Ist das der Fall, zeigt ROSE eine Warnung an und verweist auf die Programmstellen. Dadurch vermeidet ROSE Fehler, die durch unvollständige Änderungen entstehen.

Jeder Empfehlung in Abbildung 2 liegt eine *Assoziationsregel* zu Grunde. Solche Regeln werden häufig im Data Mining verwendet, um Muster darzustellen. Als ein zusätzliches

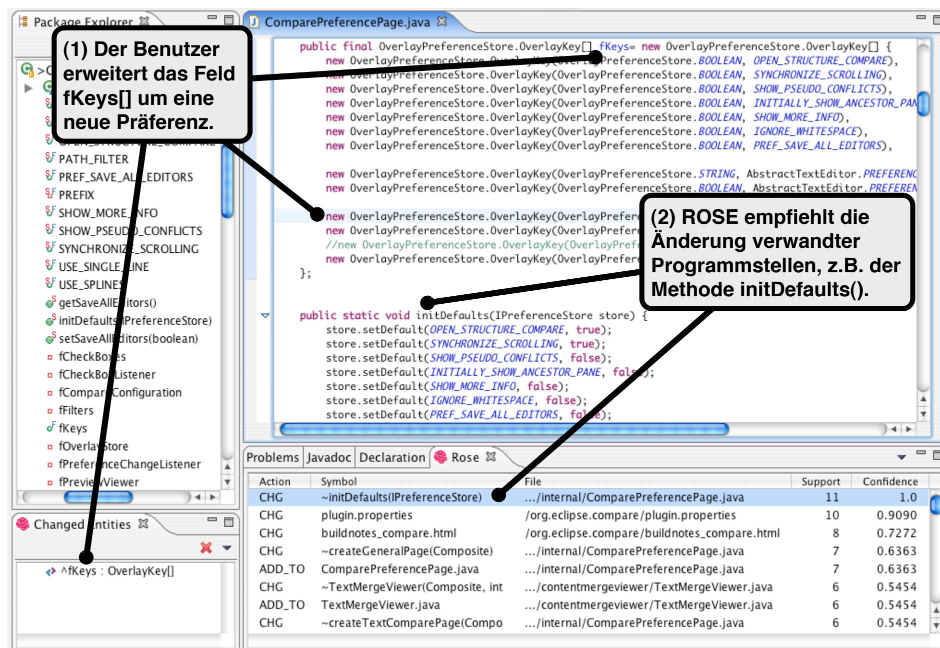

Abbildung 2: Nachdem der Entwickler den Quelltext (oben) geändert hat, empfiehlt ROSE weitere Programmstellen (unten), die von ähnlichen Transaktionen in der Vergangenheit geändert wurden.

Beispiel betrachten wir folgende Regel:

$$\mathit{change}(\text{fKeys[]}) \wedge \mathit{change}(\text{initDefaults()}) \Rightarrow \mathit{change}(\text{plugin.properties})$$
$$[\text{ support count=10; confidence=0.9090 }]$$

Diese Regel wird dann angewendet, wenn der Entwickler sowohl die Programmstellen fKeys[] als auch initDefaults() geändert hat. In diesem Fall, empfiehlt ROSE die Datei plugin.properties zu ändern, da diese in der Vergangenheit mit einer Häufigkeit von 90,9% zusammen mit fKeys[] und initDefaults() geändert wurde. Diese Konfidenz von 90,9% entspricht zehn von elf Änderungen – das heisst, alle drei Programmstellen wurden 10 mal zusammen geändert, fKeys[] und initDefaults() zusammen 11 mal.

Der klassische Ansatz zum Erzeugen von Assoziationsregeln mit dem Apriori Algorithmus [AS94] ist *alle* starken Regeln, also solche mit hoher Konfidenz und Support, *im Voraus* zu berechnen. ROSE dagegen berechnet Regeln erst *bei Bedarf,* was schnell und effektiv ist. Die daraus resultierenden Empfehlungen konnten für acht Open-Source-Projekte im Durchschnitt 33% aller noch zu ändernden Programmstellen (44% aller Dateien) korrekt vorhersagen. In 70% aller Fälle war eine der ersten drei Empfehlungen korrekt [ZWDZ05].

Neben der Versionsgeschichte lassen sich noch weitere Quellen nutzen, um den Programmierer beim Verständnis des Systems zu unterstützen. Čubranić et al. etwa kombinieren in ihrem HIPIKAT-Werkzeug verschiedene Projektdaten (wie Quelltexte, Dokumentationen, Fehlerberichte, Newsgroups und Versionsarchive) um den Entwickler eine gezielte Suche

nach Informationen zu ermöglichen [ČMSB05]. Der Schwerpunkt liegt hier auf der *Suche:* Welche Artefakte sind mit dem gegebenen Artefakt auf irgend eine Weise verwandt? ROSE hingegen betrachtet ausschließlich *Änderungen* und kann so präzisere (und kürzere) Vorschlagslisten präsentieren.

4 Vorhersage von Fehlerrisiken

Alle historien-basierten Werkzeuge lernen aus der Geschichte gute Beispiele genauso wie schlechte. Auch wenn wir optimistisch annehmen können, dass Programmierer mehr korrekte als fehlerträchtige Dinge tun, wäre es doch wünschenswert, könnten wir Änderungen *bewerten,* ob sie zu Fehlern oder zum Erfolg geführt haben. Dies wird möglich, indem man *Fehlerdatenbanken* als weitere Quelle einbezieht. Fehlerdatenbanken bieten im Vergleich zu Versionsarchiven zusätzliche Information: Sie enthalten eine Liste aller beobachteten Fehler, inklusive Beschreibung und Datum, wann ein Fehler zum ersten Mal beobachtet und später korrigiert wurde.

Manche Fehlerdatenbanken sind direkt mit Versionsarchiven gekoppelt und verweisen direkt auf die korrigierende Änderung. Viele Fehlerdatenbanken werden jedoch getrennt von Versionsarchiven verwaltet, so dass die korrigerenden Änderungen aus Datum, Beschreibung, und Autor des Eintrags erschlossen werden müssen. Hier hat es in den letzten Jahren erhebliche Fortschritte gegeben, so dass heute viele Korrekturen erfolgreich erkannt werden können [FPG03, SZZ05b].

Hat man Versionsarchive und Fehlerdatenbanken gekoppelt, verweist die korrigierende Änderung wiederum auf den Ort des Fehlers – nämlich die Stelle, die in eben dieser Änderung korrigiert wurde. Auf diese Weise kann man die faktische *Fehlerdichte* einzelner Komponenten bestimmen – und dies sogar nach Schwere oder Auswirkungen der Fehler aufschlüsseln, wie sie in der Fehlerdatenbank verzeichnet sind.

Eine klassische Anwendung der so bestimmten Fehlerdichte ist, fehlerträchtige Komponenten vorherzusagen – basierend auf der Annahme, dass eine Komponente, die in der Vergangenheit Fehler gezeigt hat, dies auch in Zukunft tun wird. Ostrand et al. konnten so etwa erfolgreich vorhersagen, welche Komponenten eines großen Telekommunikations-Systems die höchste Fehlerdichte zeigen würden [OWB05]. Offensichtlicher Nutzen: Der Test- und Validierungsaufwand lässt sich gezielter einsetzen – nämlich auf die Komponenten, für die eine hohe Fehlerdichte vorhergesagt wurde.

Neben dem Risiko von Fehlern kann man auch das *Risiko von Änderungen* untersuchen: anders ausgedrückt, die Wahrscheinlichkeit, dass eine Änderung später korrigiert oder rückgängig gemacht werden muss, weil sie einen Fehler verursacht hat. Traditionell lässt sich dieses Risiko aus Metriken der Änderung vorhersagen – etwa: Je größer die Änderung, desto größer das Risiko [MW00]. Wir bestimmen das Risiko aus Fehlerdatenbanken und erreichen so nicht nur eine höhere Präzision, sondern auch eine höhere diagnostische Qualität [SZZ05a]. Unser HATARI-Werkzeug für ECLIPSE visualisiert ein hohes Risiko in Form eines roten Balkens neben dem eigentlichem Code. Durch Wählen des Balkens erfährt der Programmierer in der *Risk History,* dass an dieser Stelle 8 von 11 Änderun-

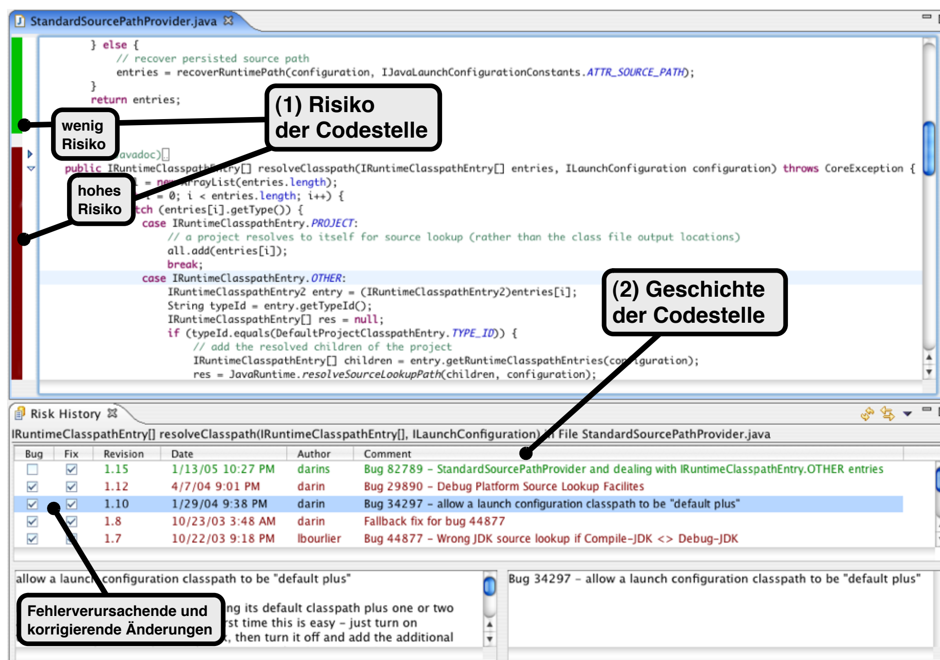

Abbildung 3: Das HATARI-Plug-in versieht Codestellen mit Balken, die das Risiko einer Änderung darstellen – ein roter Balken steht für hohes Risiko. Nach Anwählen eines Balken zeigt HATARI die Geschichte, aufgeteilt nach fehlerverursachenden und korrigierenden Änderungen.

gen später korrigiert werden mussten – was bedeutet, dass die geplante Änderung wohl bedacht sein will.

Welche Eigenschaften sind es, die Code oder Änderungen fehlerträchtig machen? In Zusammenarbeit mit Microsoft Research haben wir für fünf Projekte (u.a. Internet Explorer und Internet Information Server) untersucht, ob gängige *Komplexitätsmetriken* (wie etwa zyklomatische Komplexität, Anzahl der Codezeilen oder Vererbungstiefe) mit Fehlerdichte korrelieren [NBZ06]. Annahme war, dass bestimmte Code-Eigenschaften die Entstehung von Fehlern fördern. Ergebnis: Für jedes Projekt fanden wir signifikant korrelierende Metriken – jedoch für jedes Projekt andere. Keine der Metriken war universell geeignet, Fehlerdichte vorherzusagen.

Andererseits stellte sich heraus, dass *innerhalb* eines Projektes Metriken durchaus geeignet waren, Fehlerdichten vorherzusagen – sofern man eine Linearkombination derjenigen Metriken wählt, die sich über die Historie hinweg als gute Prädiktoren erwiesen haben. Das heißt: Hat man einmal anhand der Geschichte herausgefunden, welche Metriken für das Projekt geeignet sind, kann man mit guter Vorhersagekraft rechnen.

Neben Komplexitätsmetriken lassen sich weitere Code-Eigenschaften auf die Korrelation mit Fehlerdichte untersuchen. Wer etwa wissen möchte, wie sich die Verwendung von Zusicherungen, Vererbung, Parallelität oder anderen Sprachfeatures auf die Fehlerdichte auswirkt, kann eine entsprechende Untersuchung in wenigen Stunden implementieren und

durchführen. Auch wenn nicht sicher ist, dass hierbei *universelle* Aussagen enstehen – projekt-spezifische Korrelationen lassen sich in jedem Fall bestimmen, und auch statistisch hinsichtlich der Vorhersagekraft bewerten.

Neben dem Code gibt es natürlich noch weitere Eigenschaften, die die Fehlerdichte beeinflussen – nicht zuletzt den *Entwickler* des jeweiligen Codes. Hier sollte man jedoch voreilige Schlüsse vermeiden: In vielen Teams sind es die erfahrensten Programmierer, denen die risikoreichsten Änderungen zugewiesen werden – und die die meisten Fehler machen. Ähnliche Kompensations- und Rückkopplungseffekte treten auch bei Werkzeugen wie HATARI auf: Je mehr sich Entwickler eines Risikos bewusst sind, um so weniger Fehler werden sie machen – was letztendlich der Vorhersage zuwiderläuft.

5 Datenaufbereitung

Alle in den letzten Abschnitten beschriebenen Techniken nutzen zwei Datenquellen: Versionsarchive wie CVS und Fehlerdatenbanken wie BUGZILLA. Üblicherweise werden hierzu die Daten zunächst in einer gemeinsamen Datenbank gesammelt und geeignet aufbereitet. In diesem Abschnitt beschreiben wir die gängigsten Aufbereitungsverfahren, die im Detail in [FPG03], [ZW04] und [SZZ05b] beschrieben sind. Wer die Verfahren selbst nutzen möchte, dem sei die KENYON-Infrastruktur [BEJWKG05] empfohlen.

5.1 Erstellen von Transaktionen

Um verwandte Änderungen zu identifizieren (wie in Abschnitt 3 beschrieben) benötigt man Mengen von Programmstellen, zum Beispiel Dateien, die gleichzeitig geändert wurden. Diese Mengen bezeichnen wir als *Transaktionen*; in der Analogie zu Amazon entsprechen sie einer einzelnen Bestellung.

Für einzelne Versionsverwaltungssysteme, wie zum Beispiel SUBVERSION, können wir Transaktionen direkt abfragen. Andere Systeme wie CVS brechen eine Transaktion auf, sobald der Entwickler seine Änderungen übermittelt. Als Folge müssen wir bei der Analyse von CVS-Archiven die Transaktionen wiederherstellen. Eine Transaktion besteht in diesem Fall aus allen Änderungen eines Entwicklers mit derselben Änderungsbeschreibung und -zeit. Für die Änderungszeit verwenden wir zusätzlich Zeitfenster, da CVS manchmal mehreren Änderungen verschiedene Zeiten zuordnet.

5.2 Abbilden von Änderungen auf Programmstellen

Die meisten Versionsverwaltungssysteme zeichnen Änderungen für Dateien auf. Um in diesem Fall herauszufinden, welche Klassen oder Methoden geändert wurden, benötigt man bei der Vorverarbeitung einen zusätzlichen Schritt, der Änderungen auf feingranu-

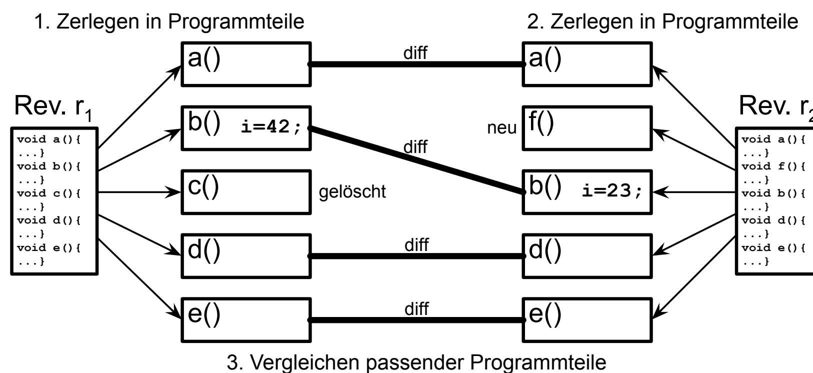

Abbildung 4: Bestimmen von Änderungen auf Methodenebene

lare Programmstellen abbildet. Abbildung 4 skizziert die Idee: Zuerst zerlegt man eine geänderte Datei in einzelne Programmteile; anschließend werden diese Programmteile miteinander verglichen. In dem Beispiel wurde b() geändert, c() gelöscht und f() hinzugefügt. Alle anderen Methoden blieben unverändert.

5.3 Bewerten von Änderungen

Um herauszufinden, welche Änderungen an Fehlern beteiligt sind, muss man sie zunächst klassifizieren. Mockus und Votta haben drei Klassen vorgeschlagen [MV00]: *Adaptive* Änderungen erweitern ein Programm um neue Features; *Korrigierende* Änderungen bessern Fehler aus; unter *pflegenden* Änderungen versteht man zum Beispiel Refactoring. Sliwerski et al. haben außerdem *unvollständige* Änderungen eingeführt – nämlich Änderungen, die später korrigiert werden müssen [SZZ05b].

Für die Analyse von Versionsarchiven im Hinblick auf das Auftreten von Fehlern sind besonders *Korrekturen* und *unvollständige Änderungen* von Bedeutung:

Korrekturen. Eine Möglichkeit, Korrekturen zu erkennen ist mit Schlüsselwörtern, wie “Bug” oder “Fix”. Dieser Ansatz wurde ursprünglich von Mockus und Votta vorgeschlagen [MV00] und bisher in vielen Forschungsarbeiten eingesetzt. Neuere Arbeiten suchen zusätzlich nach Verweisen auf Fehlerdatenbanken, wie zum Beispiel “#42223”. Durch diese Kopplung erhält man zusätzliche Informationen – wie zum Beispiel die Schwere des ursprünglichen Fehlers.

Unvollständige Änderungen. Wir bestimmen unvollständige Änderungen ausgehend von Korrekturen. Dazu verfolgen wir die korrigierten Zeilen zurück bis zu den Revisionen, in denen sie zuletzt geändert wurden. Diese Revisionen enthalten dann unvollständige Änderungen (engl. *fix-inducing changes*), falls die Revision vor dem Berichtsdatum des Fehlers erzeugt wurde.

Aus Korrekturen lässt sich direkt die Fehlerdichte ableiten; aus unvollständigen Änderungen folgt das Risiko einer Änderung, wie es in HATARI dargestellt wird.

5.4 Qualität der Daten

Eine weitere Herausforderung bei der Analyse von Versionsarchiven und Fehlerdatenbanken ist der Umgang mit *Rauschen*. Bei CVS zum Beispiel kann Rauschen auf vielfältige Weise entstehen: große Transaktionen, die zum Beispiel in jeder Datei das Copyright ändern, oder das Verschmelzen von Entwicklungszweigen sind schwer erkennbar und führen zu Ungenauigkeiten. Im Bereich der *Datenbereinigung* für Versionsarchive wurde allerdings bisher wenig geforscht, abgesehen von Fischer et al. [FPG03] und Zimmermann und Weißgerber [ZW04], die das Thema leicht gestreift haben. Hier bleibt also noch viel zu tun.

Während es in der Industrie nicht an Fehlerdatenbanken mangelt, sieht es bei *Open-Source-Projekten* traurig aus. Kurz gesagt: Die meisten Open-Source-Autoren haben offensichtlich kein Interesse am systematischen Verfolgen von Fehlern. Die einzigen größeren gut gepflegten Fehlerdatenbanken fanden wir in Open-Source-Projekten mit industriellen Ursprüngen wie Eclipse (IBM), OpenOffice (Sun), sowie Mozilla (AOL).

Glücklicherweise sind diese Projekte und deren Fehlerdatenbanken umfangreich genug, um typische Fallstudien zu erstellen und auch Ansätze über Projekte (oder Subprojekte) hinweg zu vergleichen. Das stürmische Wachstum des 2003 eingeführten *Workshops on Mining Software Repositories* ist Beleg für die Produktivität und Kreativität im Gebiet.

6 Zusammenfassung und Ausblick

Was ist es, was Software fehlerträchtig macht? Software-Archive erzählen uns, wie es zum Produkt kam – und wie es zum Fehler kam. Jetzt liegt es an uns, aus der Geschichte zu lernen, damit wir diese Fehler nicht wiederholen.

In der Vergangenheit mussten Forscher und interessierte Entwickler ihr Wissen anekdotisch zusammentragen, sich auf (nicht immer übertragbare) Fallstudien Dritter stützen oder teure (und nicht immer skalierende) Experimente durchführen, um „Geschichte zu bilden“ und somit Entscheidungen in der Softwaretechnik zu begründen. Durch die allgemeine Verbreitung von Versionsarchiven und Fehlerdatenbanken können wir jetzt aus der projektspezifischen Geschichte lernen – und nach Zusammenhängen zwischen den aufgezeichneten Änderungen, den geänderten Codestellen, und den daraus entstehenden Fehlern und Erfolgen suchen.

Die allgemeine Verfügbarkeit öffentlicher Versionsarchive und Fehlerdatenbanken eröffnet neue Perspektiven für Forscher in der Softwaretechnik, die so Ansätze bewerten und vergleichen können. Langfristig erwarten wir, dass Projektgeschichten aus Software-Archiven einen wesentlichen Beitrag leisten werden, neue und bekannte Ansätze der Softwaretechnik empirisch zu validieren.

Solche Validierungen auf bekannten Projekten werden nicht nur aus wissenschaftlicher Sicht sinnvoll sein, sie werden auch die Verbreitung in der Praxis unterstützen. Nicht zuletzt kann jeder Anwender selbst die Vorhersagekraft eines Ansatzes für ein gegebenes Projekt bestimmen und so bewerten, wie sich der Ansatz übertragen oder sogar verfeinern lässt. All dies wird helfen, die Anerkennung von Softwaretechnik zu steigern – bei Wissenschaftlern wie bei Anwendern.

Weiterführende Verweise zu Projekten und Werkzeugen finden Sie auf unserer Projektseite

`http://www.st.cs.uni-sb.de/softevo/`

Danksagung. Die *Deutsche Forschungsgemeinschaft* förderte die in diesem Artikel beschriebenen Forschungsarbeiten im Projekt „Evolutionsmuster", Kennzeichen Ze 509/1-1. Thomas Zimmermann wird zusätzlich vom *DFG-Graduiertenkolleg "Leistungsgarantien für Rechnersysteme"* gefördert . *IBM* unterstützte mit zwei *Eclipse Innovation Awards* die Entwicklung von ROSE, HATARI, und des Pixelmap-Plug-Ins. *Microsoft* ermöglichte es Andreas Zeller, seine Versions- und Fehlerdatenbanken zu untersuchen und die Ergebnisse zu veröffentlichen. Allen Institutionen gilt unser herzlicher Dank für ihre Unterstützung.

Literatur

[AS94] R. Agrawal und R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th Very Large Data Bases Conference (VLDB)*, Seiten 487–499. Morgan Kaufmann, 1994.

[BDW05] Michael Burch, Stephan Diehl und Peter Weißgerber. Visual Data Mining in Software Archives. In *Proceedings of ACM Symposium on Software Visualization (SOFTVIS05), to appear*, St. Louis, USA, May 2005.

[BEJWKG05] Jennifer Bevan, Jr. E. James Whitehead, Sunghun Kim und Michael Godfrey. Facilitating software evolution research with kenyon. In *Proc. ESEC/FSE-13*, Seiten 177–186, New York, NY, USA, 2005. ACM Press.

[CKN+03] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts und Kevin Wampler. A System for Graph-Based Visualization of the Evolution of Software. In *Proceedings of the ACM Symposium on Software Visualization*, San Diego, USA, Juni 2003.

[ČMSB05] Davor Čubranić, Gail C. Murphy, Janice Singer und Kellogg S. Booth. Hipikat: A Project Memory for Software Development. *IEEE Transactions on Software Engineering*, 31(6):446–465, Juni 2005.

[ESS92] S.G. Eick, J.L. Steffen und E.E. Summer. Seesoft - A Tool For Visualizing Line Oriented Software Statistics. In *Proc. of IEEE Transactions on Software Engineering*, Seiten 957–968, , 1992. IEEE Press.

[FPG03] Michael Fischer, Martin Pinzger und Harald Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, September 2003. IEEE.

[GJR99] Harald Gall, Mehdi Jazayeri und Claudio Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. In *Proc. International Conference on Software Maintenance (ICSM 1999)*, Seiten 99–108, Oxford, England, UK, August 1999. IEEE.

[Lan01] Michele Lanza. The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques. In *Proceedings of International Workshop on the Principles of Software Evolution IWPSE 2001*, Vienna, 2001.

[MV00] Audris Mockus und Lawrence G. Votta. Identifying Reasons for Software Changes using Historic Databases. In *Proc. International Conference on Software Maintenance (ICSM 2000)*, Seiten 120–130, San Jose, California, USA, Oktober 2000. IEEE.

[MW00] Audris Mockus und David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[NBZ06] Nachiappan Nagappan, Thomas Ball und Andreas Zeller. Mining Metrics to Predict Component Failures. In *Proc. 28th International Conference on Software Engineering (ICSE)*, Shanghai, China, Mai 2006.

[OWB05] Thomas J. Ostrand, Elaine J. Weyuker und Robert M. Bell. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[Str] Strata Inc. WinCVS Homepage. `http://www.wincvs.org`.

[SZZ05a] Jacek Śliwerski, Thomas Zimmermann und Andreas Zeller. HATARI: Raising Risk Awareness. In *Proc. ESEC/FSE-13*, Seiten 107–110, New York, NY, USA, 2005. ACM Press.

[SZZ05b] Jacek Śliwerski, Thomas Zimmermann und Andreas Zeller. When Do Changes Induce Fixes? In *Proc. International Workshop on Mining Software Repositories (MSR)*, St. Louis, Missouri, U.S., Mai 2005.

[YMNC04] Annie T.T. Ying, Gail C. Murphy, Raymond Ng und Mark C. Chu-Carroll. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering*, 30(9):574–586, September 2004.

[ZDZ03] Thomas Zimmermann, Stephan Diehl und Andreas Zeller. How History Justifies System Architecture (or not). In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2003)*, Seiten 73–83, Helsinki, Finland, September 2003. IEEE Press.

[ZW04] Thomas Zimmermann und Peter Weißgerber. Preprocessing CVS Data for Fine-Grained Analysis. In *Proc. Intl. Workshop on Mining Software Repositories (MSR)*, Edinburgh, Scotland, Mai 2004.

[ZWDZ05] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl und Andreas Zeller. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, Juni 2005.