mehr zum thema:
www.hpi.uni-potsdam.de/doellner/research.html

KOMPLEXE SOFTWARE VERSTEHEN: SOFTWARE-MINING HILFT BEIM DURCHBLICK IM ENTWICKLUNGSPROZESS

der autor

Michael Ihringer
(E-Mail: michael@ihringer.de)
ist freier Journalist und Autor in Darmstadt. Als Entwicklungsleiter und Geschäftsführer verschiedener Softwarehäuser konnte er umfangreiche eigene Erfahrungen in der Softwareentwicklung sammeln.

Vom Programmierer bis zum Projektleiter: Wer an der Entwicklung komplexer Softwaresysteme beteiligt ist, verbringt ein Gutteil seiner Zeit damit, diese zu analysieren und verstehen zu versuchen. Während die Kollegen im Business ihre Entscheidungen längst auf Basis von Data-Mining und aussagekräftigen Dashboards fällen, sind die „Techies" bis heute ausschließlich auf Fleiß, Intuition und das Einholen persönlicher Auskünfte angewiesen. Die Disziplin des Software-Minings tritt an, um das zu ändern. Dieser Artikel stellt dar, wie sich die Methoden des Data-Minings auf komplexe Softwareprojekte anwenden lassen.

In der Anfangsphase sind Softwareprojekte überschaubar: Meist entwickelt man noch alleine daran, hat die Spezifikationen vielleicht sogar selbst geschrieben und das Wenige, was es zu dokumentieren gäbe, hat man auf alle Fälle im Kopf. Später wird die Sache dann schwieriger. Das Team und der Quellcode wachsen, es treten die ersten undokumentierten Seiteneffekte auf und irgendwann muss auch der letzte Entwickler zugeben, die Software nicht mehr aus dem Effeff zu beherrschen. Die alles entscheidende Frage in dieser Situation lautet: Wie kann man so schnell wie möglich den Überblick über sein Softwareprojekt zurückerlangen, um Release-Verzögerungen, hohe Wartungskosten oder gar ein Scheitern des Projektes zu vermeiden?

Statische Codeanalyse

Einen ersten Ansatzpunkt, um den Überblick wieder herzustellen, bietet die Analyse des vorhandenen Codes. Bei der statischen Codeanalyse – oder kurz der statischen Analyse – wird der Quelltext einer Reihe formaler Prüfungen unterzogen, bei denen bestimmte Arten von Fehlern sowie Inkonsistenzen und Abweichungen von definierten Programmierrichtlinien und erwünschtem Codierstil entdeckt werden. Unter Einsatz heuristischer Verfahren ist in gewissem Rahmen auch eine Bewertung der Codequalität möglich, oft mit der Maßgabe, Schwachstellen zu identifizieren, die potenziell für Probleme in späteren Projektphasen sorgen könnten.

Die angebotenen Tools reichen vom einfachen Style-Checker bis zu komplexen Analysewerkzeugen, die beim Code-Review auch die Konformität mit der Architekturspezifikation überprüfen oder diese sogar per Reverse-Engineering des Codes abzuleiten vermögen. Dahinter steht die Idee, dass Struktur und Verhalten des Systems, wenn diese schon nicht explizit dokumentiert sind, zumindest implizit in der Implementierung stecken und durch geeignete Analyseverfahren darstellbar sind.

Das typische Resultat der statischen Analyse besteht in den üblichen Quellcode-Metriken – von einfach ausgezählten *Lines of Code* über abgeleitete Werte wie die *Cyclomatic Complexity* bis zu einem anhand der Herstellerkriterien bewerteten *Maintainability Index*. Diese Kennzahlen gelten sehr granular jeweils für einzelne Module oder Funktionen und liegen gesammelt meist in Form umfangreicher Excel-Tabellen vor. Diese Datenmengen werfen dann natürlich die Frage der Exploration auf, ganz zu schweigen von der Bewertung der einzelnen Metriken. Denn da sich für die einzelne Kennzahl kein sicherer Schwellwert definieren lässt und erst projektspezifische Kombinationen von Metriken verlässliche Qualitätsindikatoren bilden, bleibt es am Ende dem Entwickler überlassen, tausende von Fehlalarmen von Hand durchzugehen, um die wenigen wirklichen Problemstellen zu identifizieren.

Dynamische Laufzeit-Analyse

Im Gegensatz zur statischen Analyse muss die Software für die dynamische Analyse – zumindest in den zu untersuchenden Teilen – bereits ablauffähig sein, da hier die Erkenntnisse aus der Beobachtung des Ablaufverhaltens zur Laufzeit gezogen werden.

Das vorherrschende Verfahren zur dynamischen Analyse ist das klassische Debugging. Dabei beobachten Entwickler zunächst die Auswirkungen eines Bugs, versuchen diesen dann zu reproduzieren und schließlich die Ursache im Code zu finden. Integrierte Entwicklungsumgebungen wie „Visual Studio" unterstützen dieses Vorgehen durch die Möglichkeit, *Breakpoints* zu setzen und den nachfolgenden Code schrittweise auszuführen. In aller Regel muss der Code dabei sehr häufig ausgeführt werden, da sich der Entwickler nur vorwärts durch den Programmfluss bewegen kann. Zudem ist bei jedem Funktionsaufruf eine schwierige Entscheidung zu treffen: Mit *StepOver* darüber hinweggehen und hoffen, dass der Fehler nicht in der Funktion liegt, oder die Funktion mit *StepInto* schrittweise analysieren, nur um am Ende festzustellen, dass sie nichts mit dem Fehler zu tun hat. Der Entwickler muss selbst den Überblick behalten, welche Codeteile er bereits untersucht und wie weit er eine mögliche Fehlerquelle damit bereits eingekreist hat. Nicht selten behilft er sich dabei mit zusätzlichen printf- oder cout-Anweisungen, um den Programm-

„Die Zeit ist reif für Software-Mining"

Interview mit Jürgen Döllner vom Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam.

Prof. Dr. Jürgen Döllner leitet am Hasso-Plattner-Institut das Fachgebiet Computergrafische Systeme, das sich mit der Analyse, Planung und Konstruktion computergrafischer und allgemein komplexer IT-Systeme befasst.

Wie unterscheidet sich Software-Mining von anderen Analyseverfahren?

Software-Mining kann Informationen aus der Implementierung, der Evolution und dem Laufzeitverhalten eines Softwaresystems miteinander verknüpfen. Schon die systematische Auswertung von Evolutionsdaten zählt bislang eher zu den Ausnahmen, aber dass alle drei Quellen miteinander verbunden werden, ist komplett neu. Das implementierte System kann dabei auf verschiedene Weisen dargestellt werden und ist nicht auf bestimmte Metriken ausgerichtet. Die automatische Analyse der drei Quellen erzeugt eine Momentaufnahme, die stets aktuell und objektiv ist. In Entwicklerprojekten mussten sich alle Beteiligten bisher auf subjektive Informationen oder dokumentierte Modelle verlassen, die meistens nicht mit der implementierten Realität in einem komplexen und langjährigen System übereinstimmen.

Welche Einsatzbereiche sehen Sie für Software-Mining?

Software-Mining ist ein Verfahren, das High-Level-Informationen verlässlich generiert und frühzeitig problematische Entwicklungstendenzen und Situationen deutlich macht, die aus Managementsicht hinterfragt werden müssen. Projektverantwortliche im Software-Engineering können sich dadurch ein Frühwarnsystem aufbauen, das sie bei Entscheidungen unterstützt und das hilft, Projektrisiken zu reduzieren und Problemsituationen in Entwicklungsprozessen zu vermeiden. Ein weiteres Einsatzgebiet betrifft eine der klassischen Aufgaben eines Softwareentwicklers: das Debugging. Software-Mining ermöglicht eine genaue und effiziente Fehlersuche, weil das gesamte Systemverhalten aufgezeichnet und nachträglich vollständig untersucht werden kann. Der Entwickler muss den Fehler im System also nicht wieder und wieder reproduzieren und nebenher debuggen, sondern kann einfach den einmal erstellten Fehlermitschnitt auswerten.

Unter welchen Bedingungen zeigen sich die Stärken von Software-Mining besonders deutlich?

Wenn beispielsweise fremde Software übernommen und weitergeführt werden muss, machen Software-Mining-Werkzeuge die wesentliche Strukturen und Verhältnisse des Systems um ein Vielfaches schneller erkennbar als mühsames, unsystematisches Durchforsten. Das gilt natürlich auch, wenn die Anwendung einfach zu groß geworden ist, um sie als einzelner, selbst als langjährig erfahrener Entwickler, noch verstehen zu können. Das ist spätestens ab 100.000 Codezeilen und einer Lebenszeit der Software von mindestens drei Jahren der Fall. Entwicklerteams brauchen dann vor allem mentale, nachvollziehbare Modelle der Software. Neben der Größe des Softwaresystems spielt auch die des Teams, das es bearbeitet, eine Rolle. Schon ab drei Personen gibt es in der Regel Kommunikationsprobleme, bei zehn Personen fehlt der Durchblick völlig. Integration, Spaltung, hohe Fluktuation und rasche Vergrößerung von Teams können die Probleme noch verschärfen.

Ist Software-Mining auch ohne Visualisierung denkbar?

Der Einsatz von Visualisierungsverfahren ist naheliegend. Ein Punkt ist dabei ihre Methoden-, Sprach- und Vorgehensneutralität. Die Extraktionsverfahren für die Daten der drei Kategorien sind zwar sprachabhängig, der Visualisierungsapparat führt die Ergebnisse aber zusammen und funktioniert universell. Die Programmiersprache und der Entwicklungsprozess spielen keine Rolle. Ein weiteres Argument ist die Exploration der entstandenen Daten. Durch die Fusion der Information aus dem Quellcode, dem dynamischen Laufzeitverhalten und der Evolution des Softwaresystems sind leistungsstärkere Analysen möglich als beim klassischen Profiling. Durch die bildliche Darstellung abstrakter Informationen, wie z. B. Module, Klassen und Methoden des Softwaresystems, als konkrete Bausteine können diese leichter überblickt und wiedergefunden werden. Außergewöhnliche Muster oder nicht dokumentierte Phänomene werden so für die Projektbeteiligten ad hoc erkennbar und klassische Zufallsfunde kontrolliert sichtbar.

ablauf zu verfolgen und den Zustand von Variablen im Blick zu behalten.

Debugging dient allerdings nicht nur im ursprünglichen Sinne dem Auffinden von Fehlerstellen, sondern wird häufig auch zur Implementierung neuer Features oder bei Änderungsanforderungen eingesetzt, um das Design des Softwaresystems zu verstehen und die richtigen Stellen zu finden, an denen sich die neue Funktionalität integrieren oder die vorhandene anpassen lässt. Auch hier verbringen Entwickler wieder viel Zeit damit, Quellcode zu analysieren, sich mit dem Debugger schrittweise durch die Ausführung zu tasten und sich in Gedanken die Design- und Codierprinzipien zusammenzustellen, nach denen die Änderung sauber und an der richtigen Stelle implementiert werden kann.

Eine weitere wichtige Technik zur dynamischen Analyse bildet das Profiling, bei dem das Laufzeitverhalten einer Applikation aufgezeichnet und ausgewertet wird. So zeigen sich etwa Performance-

Hotspots, also besonderes „teure” Stellen im Code, mit deren Ausführung die Software unverhältnismäßig viel Zeit verbringt. Dadurch lassen sich Flaschenhälse frühzeitig identifizieren und beheben. Allerdings steckt auch hier der Teufel im Detail: Wenn die dynamische Analyse etwa ergibt, dass ein tausendfach ausgeführter Datenbankzugriff im Mittel teuer ist, kann es ja durchaus sein, dass der 17. und der 547. Zugriff besonders langsam sind, alle anderen aber normal ablaufen. Um das herauszufinden und die Ursachen dafür zu finden, sind also wieder die Methoden des Debuggings gefragt.

Evolutionsanalyse in Repositorien

Kein größeres Softwareprojekt kommt ohne Werkzeuge für die Verwaltung des Quellcodes und auftretender Probleme aus. Die für ersteres meist eingesetzten Konfigurationsmanagement-Werkzeuge erlauben es dem einzelnen Entwickler, Code auszuchecken, den er bearbeitet. Außerdem führen sie Buch darüber, wann er den Code wieder eingecheckt und welche Veränderungen er vorgenommen hat.

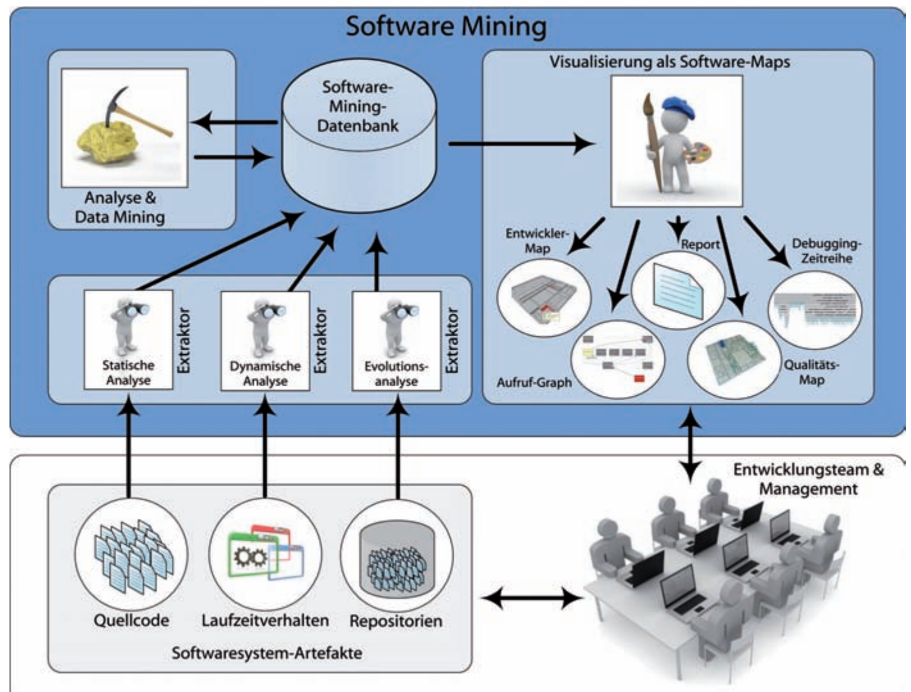

Abb. 1: Überblick Software-Mining (Quelle: HPI).

Spätestens, wenn die Software beim Anwender im Einsatz ist, kommen *Issue-Tracking-* oder *Ticketing-*Systeme hinzu, die zusätzlich dokumentieren, auf welchen Bug oder welches neue Feature sich eine Änderung am Code bezieht.

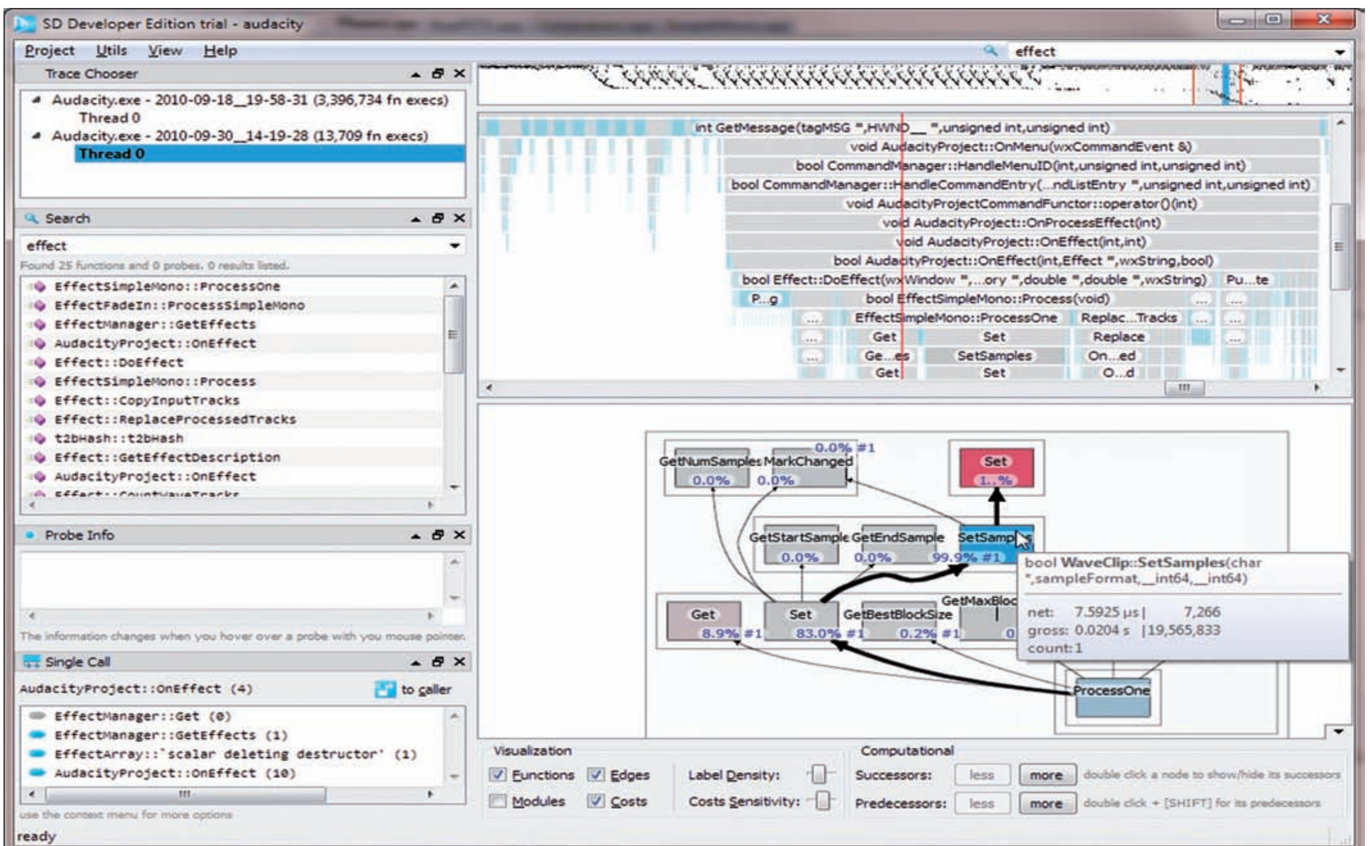

Abb. 2: Der „Call Graph View” deckt Aufrufbeziehungen und Performance-Engpässe auf (Quelle: Software Diagnostics).

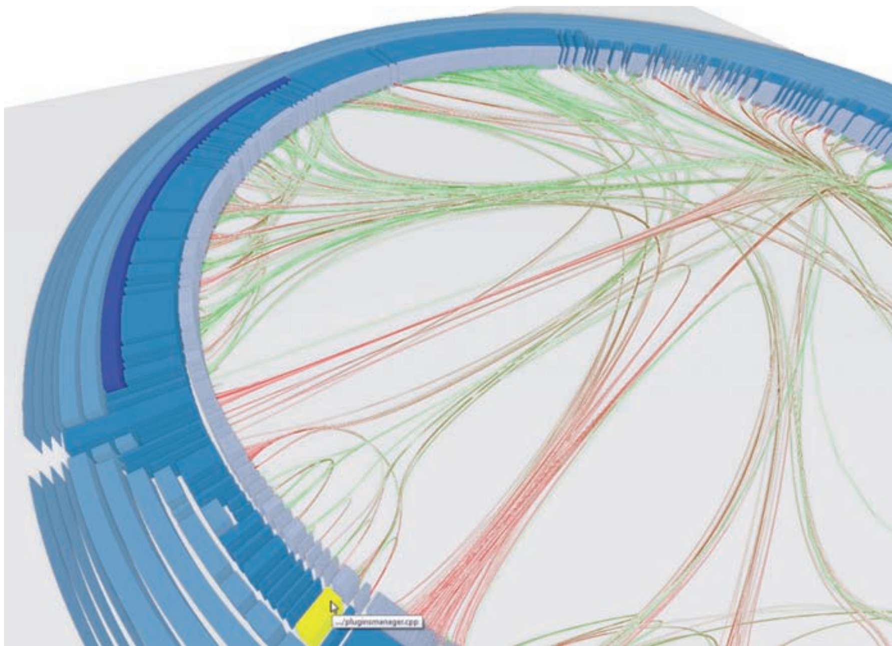

Abb. 3: Der „Bundle View" zeigt die Abhängigkeitsbeziehungen eines Softwaresystems (Quelle: Software Diagnostics).

Die in diesen Repositorien über die Zeit hinterlegten Informationen repräsentieren die Entwicklungsdynamik an einem Softwareprojekt, aus denen sich insbesondere für den Projektleiter wichtige Fragestellungen beantworten lassen: An welchen Modulen arbeiten die Entwickler zurzeit tatsächlich? Welche Module müssen besonders häufig angefasst werden? Erfordert eine Änderung an Modul *A* regelmäßig auch eine Änderung an Modul *B*? Wie viele Stellen wären betroffen, wenn wir Feature *X* implementieren würden?

Der übliche Weg, um an dieses Wissen zu gelangen, ist eine manuelle Auswertung der einzelnen Ereignisse. Die Werkzeuge selbst bieten maximal eine automatische Benachrichtigung per E-Mail an, sobald Code ein- oder ausgecheckt wird, oder stellen diese Informationen als tägliche Zusammenfassung zur Verfügung. Statt eines Überblicks erhält der Verantwortliche so nur eine Fülle von Details, mit denen er in der Praxis nichts anfangen kann.

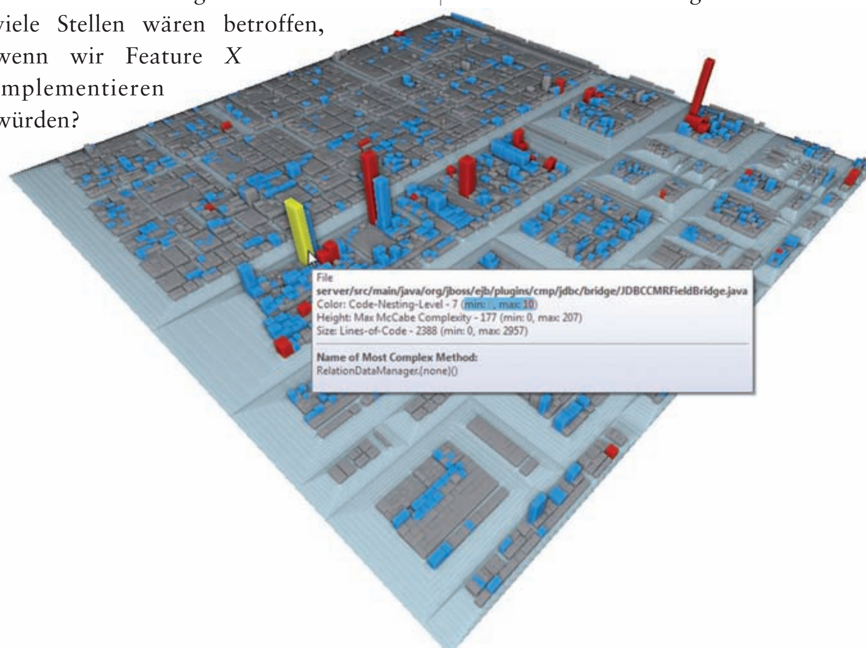

Abb. 4: Der „Treemap View" bildet die Software-Lagekarte, hier des „JBoss Application Servers" von Red Hat (Quelle: Software Diagnostics).

Zusammenhänge herstellen

Jede der etablierten Methoden zur Softwareanalyse bietet wertvolle Hinweise für die am Entwicklungsprozess Beteiligten, unterliegt aber auch spezifischen Einschränkungen. „Klassische Werkzeuge und Techniken im Umgang mit komplexen Softwaresystemen liefern nur punktuelle Erkenntnisse, so als wollte man ein dunkles Höhlensystem mit der Taschenlampe erforschen", erläutert Prof. Dr. Jürgen Döllner vom Lehrstuhl Analyse, Planung und Konstruktion komplexer Systeme des Hasso-Plattner-Instituts (HPI) für Softwaresystemtechnik an der Universität Potsdam (**siehe auch Interview in diesem Artikel**). Die von Döllner mitbegründete Disziplin des Software-Minings beschäftigt sich damit, die vielschichtigen Software-Engineering-Daten eines Projekts zusammenzufügen und mit den Methoden des Data-Minings zu analysieren. **Abbildung 1** gibt eine schematische Übersicht über das Verfahren.

Als Datenquelle dienen dem Software-Mining der Quellcode des Projekts, eine dynamische Laufzeitanalyse und die Evolutionsanalyse der vorhandenen Repositorien. Die gesammelten Fakten aus allen drei Bereichen werden zu einer aussagekräftigen Datenbasis zusammengeführt. Software-Mining ersetzt also die genannten Methoden nicht, sondern ergänzt sie um eine übergeordnete Instanz, die in der Lage ist, entscheidende Querverbindungen herzustellen.

„Schon aus der Verknüpfung zwischen Kennzahlen und Änderungsstellen ergeben sich wertvolle Handlungsempfehlungen", erläutert Marc Hildebrandt vom HPI-Spin-off Software Diagnostics. Um auch komplexere Querverbindungen aufzudecken, kann man Algorithmen zur Mustererkennung, Regelerkennung und zum Generieren von Datenabstraktionen entwickeln. Mit solchen Algorithmen lassen sich dann auch Muster erkennen, wie z. B.: „Wenn Änderungen am Netzwerk-Modul stattfinden, wird meistens auch ein Modul aus der GUI-Schicht geändert". Und es lassen sich dann die entsprechenden Konsequenzen ziehen, wie z. B.: „Hier passt vermutlich die Art der Modularisierung nicht mehr zur Programmierweise, unter Umständen kann eine Anpassung helfen, den *Change Impact* in Zukunft stärker zu begrenzen".

Software-Mining deckt außerdem Risiken und Entwicklungsbremsen auf: Wenn das viele Codezeilen umfassende Netzwerk-

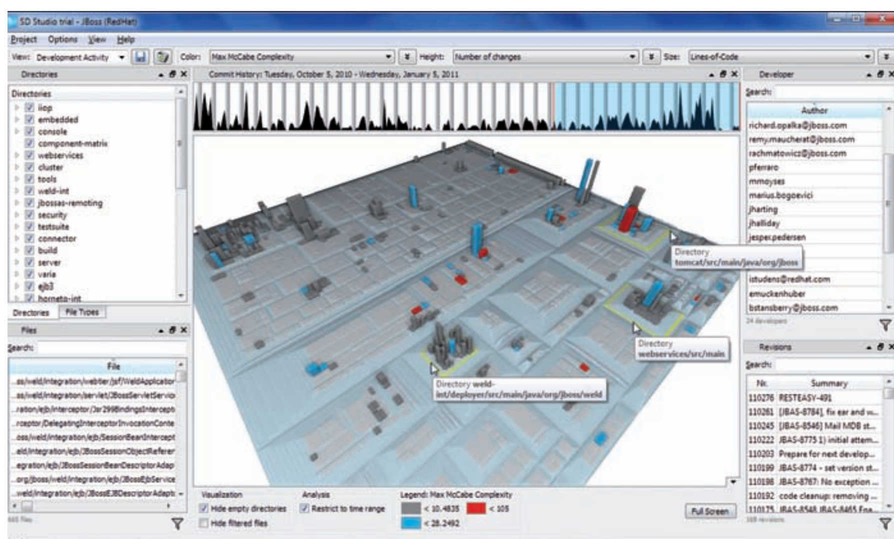

Abb. 5: Alle Entwicklungsaktivitäten zwischen 05.10.2010 und 05.01.2011 (Quelle: Software Diagnostics).

Modul häufig im Kontext von Bug-Fixes geändert wird und zudem nur in geringem Maße durch Tests abgedeckt ist, sollte der Projektleiter besonders wachsam sein und Tester und Entwickler sollten sich auf diese kritischen Codestellen fokussieren.

Software-Lagekarten in der Praxis

Ähnlich wie im businessorientierten Data-Mining, bei dem die Ergebnisse von Auswertungen dem Management als leicht verständliche Kurven, Tortendiagramme oder Ampelsymbole präsentiert werden, steht auch am Ende des Software-Minings eine visuelle Darstellung in Form virtueller Software-Lagekarten. Im Gegensatz zu einzelnen Kennzahlen oder Analyseergebnissen bietet eine Visualisierung dem Betrachter Unterstützung bei explorativen Fragestellungen, bei denen sich das gesuchte Ergebnis nicht von vornherein exakt spezifizieren lässt. Das ist die Standardsituation beim Debugging (siehe den *Call Graph View* in Abbildung 2) oder der Aufdeckung unerwünschter Abhängigkeiten zwischen Softwaremodulen (siehe den *Bundle View* in Abbildung 3).

Eine dritte Darstellungsart, *Treemap View* genannt, zeigt ihre besondere Stärke in der Analyse von Codemetriken. Durch freies Kombinieren mit weiteren Software-Mining-Ergebnissen und Auftragen auf die verschiedenen Dimensionen und Parameter entsteht eine Software-Lagekarte, die sich intuitiv interpretieren lässt. Um dies zu demonstrieren, soll der 850.000 Quellcode-Zeilen umfassende „JBoss Application Server" von Red Hat als Projektbeispiel dienen. Abbildung 4 zeigt die Ähnlichkeit mit der Häuserlandschaft einer Großstadt gut. Jedes Gebäude entspricht einer Quellcode-Datei, die Häuserblocks werden anhand der Modulhierarchie gebildet und mittels des *Treemap*-Algorithmus angeordnet.

Für dieses Beispiel wurde die Anzahl der Quellcodezeilen als Grundfläche, die McCabe-Komplexität des Codes als Höhe und die Kontrollfluss-Schachtelungstiefe als Farbe aufgetragen – mit der Anzahl verschachtelter if- und for-Anweisungen wechselt die Farbe von grau über blau zu rot. Bereits auf den ersten Blick springen einige Ausreißer mit hoher Komplexität und Schachtelungstiefe im server- und tomcat-Modul ins Auge, die teilweise auch beachtlichen Codeumfang aufweisen. Wenn das Team auf solche möglichen Schwachpunkte aufmerksam geworden ist, kann es die Gründe für die hohe Komplexität diskutieren und gemeinsam überlegen, wo sich beispielsweise durch eine Refaktorisierung die Qualität nachhaltig verbessern lässt.

Nimmt man zu den Codemetriken die Ergebnisse aus der Evolutionsanalyse hinzu, wird beispielsweise erkennbar, welchen Code die Entwickler tatsächlich bearbeitet haben. Im oberen Bereich von Abbildung 5 ist der Zeitstrahl mit den Änderungsaktivitäten zu erkennen, auf dem ein Drei-Monats-Zeitraum vom 05.10.2010 bis zum 05. 01.2011 ausgewählt wurde. In der Höhe ist die Anzahl an Änderungen aufgetragen, die Farbe gibt die McCabe-Komplexität des Codes an und die Grundfläche die Anzahl an Codezeilen. Hier ist sofort erkennbar, wie oft Entwickler große, monolithische Code-Blöcke verändern und/oder an komplexem Code arbeiten müssen. Mit Blick auf das Detail in Abbildung 6 kann ein Teamleiter etwa fragen, ob die häufigen Modifikationen an der Datei JBossCacheManager.java im tomcat-Modul übermäßigen Aufwand verursacht haben, weil es sich um die umfangreichste Codedatei der gesamten JBoss-Implementierung handelt. Falls ja, könnte ein sofortiges Aufräumen des Codes die zukünftige Entwicklungstätigkeit beschleunigen.

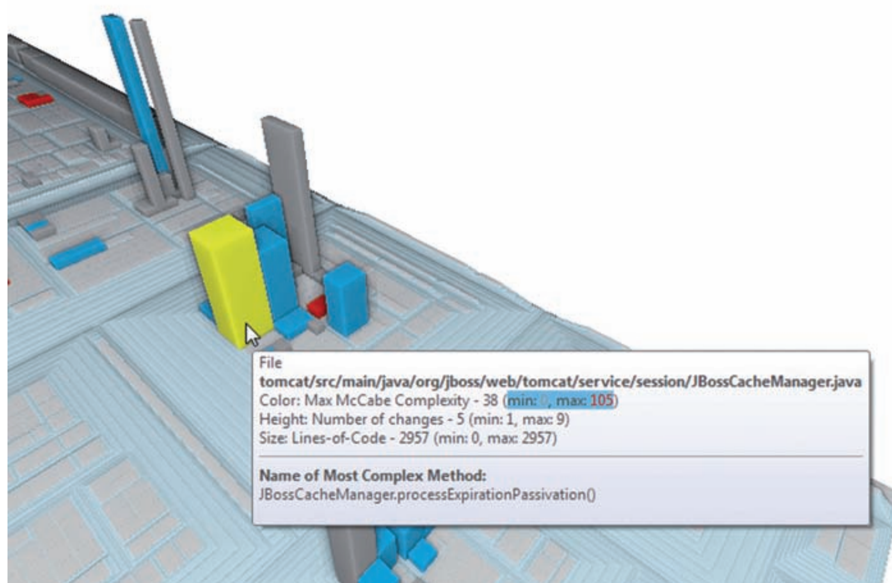

Abb. 6: Entwicklungsaktivitäten an der größten Datei des JBoss-Systems (Quelle: Software Diagnostics).

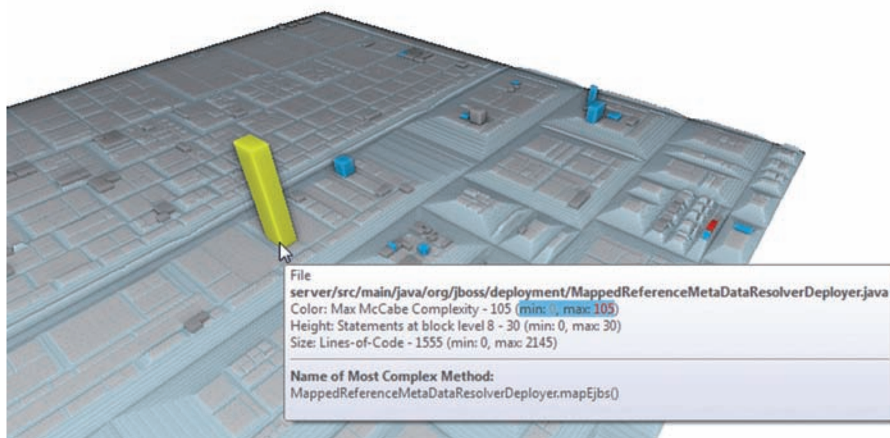

Abb. 7: *Dateien, die im Zusammenhang mit Bug-Fixes verändert wurden (Quelle: Software Diagnostics).*

Mit solchen Einblicken ausgerüstet, lassen sich zukünftige Qualitätsprobleme zumindest teilweise vorhersehen und ausräumen, bevor sie teure Konsequenzen haben. Steht dem Team – wie heute fast überall üblich – zusätzlich zum Konfigurationsmanagement auch ein *Issues/Bug-Tracking*-System zur Verfügung, eröffnet dies auch den umgekehrten Weg: Dann lassen sich durchgeführte Änderungen einzelnen Problemen zuordnen. So zeigt **Abbildung 7** alle im Zusammenhang mit Bug-Fixes durchgeführten Änderungen zwischen dem 31.08.2010 und dem 05.01.2011. Hier fällt die Datei MappedReferenceMetaDataResolverDeployer.java auf: Sie ist umfangreich, hat die größte McCabe-Komplexität überhaupt und ein signifikanter Teil des Codes befindet sich auf Schachtelungstiefe 8 oder höher. Hier wäre zu überprüfen, ob die angezeigten Qualitätsprobleme zur Notwendigkeit der Bug-Fixes geführt haben, und wie sich dies für die Zukunft vermeiden lässt.

Fazit

Die verschiedenen Darstellungen dienen dem besseren Verständnis der betreffenden Softwaresysteme bei den Entwicklern, der Aufdeckung unerwünschter Abhängigkeiten zwischen Softwaremodulen und der präzisen systembezogenen Kommunikation zwischen Management und Entwicklern. So sorgt Software-Mining dafür, dass die Beteiligten einer Softwareentwicklung über alle laufenden Aktivitäten genau informiert sind und den entscheidenden Kontext überblicken. Eine solche Umgebung bildet ein leistungsfähiges Frühwarnsystem, um Problemsituationen während des Projektverlaufs zu identifizieren. Software-Mining eignet sich daher besonders für große Softwareteams und solche, die mit komplexen Softwaresystemen zu tun haben. Ohne dass methodische Änderungen am Entwicklungsprozess nötig werden, können sie von Vorteilen wie einem beschleunigten Entwicklungsprozess, steigender Softwarequalität, Transparenz bei der Wartung und erheblich sinkenden IT-Projektrisiken profitieren. ■