Seminar zu Ursachen und Vermeidung von Fehlern in der Softwareentwicklung

Analyse vergangener Defektbehebungen

Freie Universität Berlin, Wintersemester 2005/2006

Philipp Schmidt <phils@inf.fu-berlin.de>

30. März 2006

Zusammenfassung

Um Defekte in Software zu erkennen und zu beheben wurden in der Vergangenheit vor allem statische Verfahren bemüht, die nahe an der Syntax und Semantik der Programmiersprache ungewöhnliche oder fehlerträchtige Konstrukte erkennen und diese dem Programmierer präsentieren. Die Akzeptanz dieser Verfahren leidet vor allem auf Grund der hohen Rate an „false positives", da diese Verfahren nur sehr eingeschränkt den tatsächlichen Programmfluß betrachten können, außerdem lassen sie sich nur mit extrem hohem Aufwand an die Bedürfnisse des Projekts anpassen.

Die im Folgenden vorgestellten Verfahren verwenden die Versionsgeschichte eines Softwareprojekts für verschiedene Zwecke:

- Für die Erkennung von Defektmustern anhand Statistischer Analyse der Änderungen,
- die Bewertung und Interpretation der Ergebnisse von statischen Defekterkennungsverfahren (z.B. Compiler-Warnings oder Werkzeugen wie Lint)
- und die Erkennung von Defekten anhand von Usage Pattern aus dem Code.

Inhaltsverzeichnis

1 Einführung

Einfache statische Code-Analyseverfahren haben sich über die letzten Jahre einen festen Platz in der Softwareentwicklung gesichert. Ob es nun die vielen aussagekräftigen Warnungen des GCC, die Hinweise in Eclipse oder das betage LINT sind, alle erkennen syntaktisch korrekte aber untypische Codeteile anhand statischer Regeln und sollen so helfen Fehler im Code zu finden.

Leider haben diese Verfahren Nachteile, sie lassen sich kaum für konkrete Projekte anpassen, so dass z.B. beim Übersetzen von Betriebssystemtreibern zwangsläufig diese Werkzeuge sehr viele false positives[1] produzieren, da diese sehr viele ungewöhnliche Typenumwandlungen enthalten müssen. Andererseits erkennen sie aber auch keine Projekt- oder Frameworkspezifischen Fehler oder strukturelle Fehler, wie nicht freigegebene Sperren, double-frees oder double-allocs.

Eine Lösung für dieses Problem versprechen Verfahren, die sich nicht nur auf den Ist-Zustand des Codes stützen, sondern unter anderem auf statistische Daten aus dem Projekt, auf Laufzeitdaten oder auf die Änderungen in der Versionsgeschichte des Projekts.

2 Überblick

Der Schwerpunkt wird in diesem Text auf Verfahren liegen, die die Versionsgeschichte des Projektes betrachten und diese statistisch oder qualitativ auswerten, um Informationen über für ein konkretes Projekt typische Defekte zu erhalten.

An hand von drei Arbeiten [1][2][3] sollen einige Verfahrensweisen illustriert werden. Angefangen mit der statistischen Auswertung von Fehlerberichten und deren Interpretation[1], wobei ausschließlich Prozessmaße zur Auswertung herangezogen werden, über die Bewertung von Ergebinissen statischer Analyse [2] mit Hilfe von Daten aus einem Versionskontrollsystem hin zu einem recht umfangreichen System zur Erkennung von Usage Pattern und deren Verletzungen [3] beschäftigen.

Zu allererst jedoch sollen die Grundideen, auf denen alle drei Artike basieren, hier vorgestellt werden.

[1] In diesem Fall: Meldungen das Fehler vorhanden sind, wobei der Code vollkommen korrekt ist

2.1 Charakter von Defektbehebungen

Für größere Projekte gilt immer die Devise „Ein Fehler kommt selten allein", die meisten Defekte entstammen ähnlichen Quellen und Gründen. Tritt ein Defekt mehrfach auf, so ist es sehr wahrscheinlich das ein ähnlicher Defekt an weiteren Stellen vorhanden ist. An dieser Stelle setzen alle verfahren zur Defekterkennung aus der Projektgeschichte an [1].

Hierfür bietet es sich an etwaige Fehlerdatenbanken abzugrasen, um Informationen über Defektbehebungen zu erhalten, und hieraus Schlüsse zu ziehen. Freilich ist dies nur sehr eingeschränkte möglich, denn die meisten einfachen und damit maschinell auffindbaren Defekte schaffen es nicht in die Fehlerdatenbanken, weil sie schon während der Entwicklung entdeckt, behoben, aber nicht dokumentiert werden, da dies Softwareentwicklern oft zu mühselig erscheint.

2.2 Charakter von Änderungen

Sollten detaillierten Änderunge wie z.B. CVS-Revisions zur Verfügung stehen, bietet sich die Möglichkeit diese auch detailliert auszuwerten und aus den schon erfolgten Defektbehebungen auch qualitative Informationen zu ziehen [2] [3].

Die meisten Defekte sind keine großen strukturellen Mißverständnisse, sondern meist Kleinigkeiten - ein illegaler Cast in Java, eine vergessene Sperre, ein Listener der sich zu deregistrieren vergisst. Da Programmierer beim Weiterentwickeln von Software nur sehr selten wenige Zeile in ein Versionskontrollsystem einchecken, kann man davon ausgehen, dass alle sehr kurzen commits solche kleinen Defekte beheben. Man kann also von einer sehr starken Korrelation zwischen kurzen Commits und Defekten ausgehen und dies qualitativ wie quantitativ verwenden.

Andererseits werden zusammenhängende Strukturen wie Sperren/Entsperren, sofern sie korrekt angewandt wurden, meist in einem Commit auftauchen, da es wenig Sinn macht diese einzeln zu committen und es dann streng genommen auch Defektbehebungen sind.

Um kurze Änderungen auszumachen sind natürlich die Diffs zwischen Major-Releases eines Projekts vollkommen wertlos, da sie in der Fülle der Änderungen untergehen. Interessant sind vor allem Änderungen innerhalb eines Releases bis zu dessen Fertigstellung, da man darin die gesuchten Änderungen leicht ausfindig machen kann.

3 Statistische Auswertung von Fehlerbehebungen

Das erste hier vorgestellte Konzept stammt aus dem Artikel [1] und folgt einem eher traditionellen Weg, um besonders defektträchtige Konstrukte zu finden. Es wird lediglich die Defektdatenbank quantitativ auf Korrelationen zwischen Defekten und Vorkommen bestimmter Konstrukte untersucht.

Zunächst werden Größenmaße für das Design (z.B. Anzahl der Seiten) und den Code (z.B. Lines of Code) und qualitative Maße wie eingehende/ausgehende Verwendung im Design und Vorkommen ausgewählter Sprachkonstrukte im Code erhoben, dies geschieht für jede Komponente einzeln.

Danach werden die Korrelationen zwischen der Anzahl der Defekte und dem Auftreten der Konstrukte (z.B. durch faktorielle Analyse) untersucht.

Anschließend werden hieraus einige ausgewählt:

- Das Konstrukt dessen Häufigkeit am stärksten mit der Anzahl der Defekte korreliert.
- Das Konstrukt dessen Häufigkeit am stärksten mit der Anzahl der Defekte korreliert, bereinigt um die Größe der Komponente.
- Die Konstrukte deren Häufigkeit eine stärkere Korrelation mit dem Auftreten von Defekten als deren Häufigkeit mit der Größe.
- Die Konstrukte deren Häufigkeit eine stärkere Korrelation mit Auftreten von Defekten haben als das Auftreten von Defekten mit der Größe.

Die Interpretation bleibt dem geneigten Anwender überlassen.

Diese Methode hat den Vorteil, dass sie auch ohne Zugriff auf die Versionsgeschichte und den Quellcode funktioniert, sie ist dadurch jedoch sehr stark auf die Erhebung der verwendeten Maße angewiesen.

3.1 Bewertung

Der Artikel erweckte in mir den Eindruck, dass die Methode recht hilflos versucht aus Prozessmaßen irgendeinen Gewinn zu ziehen und scheint

auch in der Beispielsanwendung auf ein bereits abgeschlossenes Projekt, keine Interessanten Ergebnisse erbracht zu haben. Allerdings ist der Artikel schon über fünf Jahre alt und die Möglichkeiten des direkten Zugriffs auf den Quellcode und Versionsverwaltungsdaten in einem Closed-Source Projekt sind recht problematisch.

Zur Qualitätsverbesserung kann diese Methode nur sehr langfristig beitragen, da sie praktisch keine konkreten Ergebnisse liefert, die für die Mitarbeiter am Projekt nicht auch ohne Anwendung der Methode offensichtlich währen.

4 Bewertung statischer Analyseergebnisse

Viele, vor allem kleine[2] Defekte, lassen sich mit statischer Analyse recht gut erkennen, jedoch kommt man gerade bei komplexen Tests schnell an einen Punkt an dem das Werkzeug mehr false positives liefert, als echte Defekte. Ein möglicher Ausweg wird in [2] beschrieben und anhand Returnvalue-Checkers illustriert.

Die Grundidee ist dabei recht einfach: ein Algorithmus führt eine statische Analyse des Code durch, anschließend wird dessen Ergebnis anhand von Änderungen im Versionskontrollsystem und anderen Vorkommen ähnlicher Codeteile verifiziert.

4.1 Beispiel: Returnvalue-Checker

Ein Returnvalue-Checker prüft ob ein Rückgabewert einer Funktion (z.B. auf Nullpointer) geprüft oder ungeprüft verwendet wird. Eine solche Prüfung von z.B. Benutzereingaben ist zwingend notwendig, wohingegen sie bei internen Hilfsroutinen oft nur Rechenzeitverschwendung darstellen würde.

Um diese Differenzierung automatisiert zu ermöglichen finden nun zwei Tests statt:

- Findet eine Überprüfung der Rückgabewerte der Funktion in anderen Teilen des Projekts statt?
- Gibt es Commits im Versionskontrollsystem das solche Überprüfungen hinzufügt?

[2] wenige Codezeilen betreffende Fehler, unabhänig von ihren Auswirkungen

Ausgehend von der Annahme, die Entwickler arbeiten in großen Teilen des Projekts korrekt, andernfalls sollte man sich ernsthafte Sorgen um die Komplexität des Projekts, Entwicklungsprozess und Qualifikation der Entwickler machen, können nun alle Funktionsaufrufe ignoriert werden, die zum überwiegenden Teil nicht geprüft werden.

Als nächstes wird geprüft, zu welchen Funktionsaufrufen in einem Checkin eine Prüfung des Rückgabewertes hinzugefügt wurden. Diese sind besonders wahrscheinliche Kandidaten für Funktionen, die einer Überprüfung bedürfen.

Mit diesem Verfahren ist es den Autoren gelungen die Rate der False Positives von 99% bei ausschließlich statischer Analyse auf immerhin 78% zu senken (betrachtet man nur die Ergebnisse aus der Prüfung gegen die Checkins ins Versionskontrollsystem sind es „nur" 74% False Positives).

4.2 Bewertung

Die Idee, die Ergebnisse aus statischen Analyseverfahren mit Hilfe von Daten aus dem Versionskontrollsystem des Projekts zu Verifizieren ist vielversprechend: das Verfahren lässt sich hiermit ohne großen Aufwand an ein Projekt anpassen und seine false-postivie-Rate wird damit deutlich gesenkt. Wie gut sich diese Idee auf andere statische Analyseverfahren übertragen lässt und wie praxistauglich ein resultierendes Werkzeug dann ist, kann nur die Praxis zeigen.

5 Usage Pattern zur Defekterkennung

Wie schon in 2.1 beschrieben, treten gleichartige Defekte selten allein auf, und Codeteile die oft gemeinsam in einem Checkin auftauchen gehören meist zusammen. Diese Beobachtungen bilden die Basis für Artikel [3] und das darin vorgestellt System DynaMine zur Erkennung von Usage Pattern und deren Verletzung.

Als Usage Pattern wird hier bezeichnet, wenn Codeteile, z.B. Funktionsaufrufe, immer in bestimmter Weise gemeinsam auftreten. Ein Paar $< M, S >$ wird Usage Pattern genannt, wenn M eine Menge von Konstrukten ist, und S eine Beziehung wie $m_1 \Rightarrow m_2$, wobei $m_1, m_2 \in M$.

5.1 Erkennung und Verifikation von Usage Pattern

Die Erkennung von Usage Pattern erfolgt mit Methoden des DataMining, die hier nicht näher erläutert werden sollen.

In DynaMine wird der Apriori-Algorithmus, der auch in den meisten Webshops zur Ermittlung von ähnlichen Produkten dient[3], verwendet um in CVS-Commits Pattern zu finden. Je öfter so ein Pattern auftaucht und nicht nur ein Teil davon, um so stärker ist es.

Als geeignet zur Erkennung von Defekten scheinen vor allem kurze Pattern, weil diese leichter zu finden und zu verifizieren sind und die Datenmenge beim DataMining so beherrschbar bleibt. DynaMine beschränkt sich deshalb auf Funktionspaare wie z.B. hasNext und next oder lock und unlock .

Wie wird dies nun zu Defekterkennung genutzt? Die Verletzung eines starken Pattern, also eines, dessen Wahrscheinlichkeit vollständig zu sein hoch ist, deutet auf einen Defekt hin, tritt das Pattern jedoch oft in Teilen auf, ist ein Defekt unwahrscheinlicher.

Um False Positives bei der Verifikation von Pattern gering zu halten, etwa durch Pattern die an einigen Stellen über mehrere Aufrufebenen verteilt auftreten, wird der Code des Projekts um eine Protokollierung der Patternteilaufrufe erweitert, so können viele False Positives und nicht zutreffende Pattern zu Laufzeit der Applikation erkannt und eliminiert werden.

5.2 Bewertung

Das Verfahren ist sehr komplex, zu komplex um es hier in allen Details vorzustellen. Ein textuelles DataMinig der Applikation wirkt jedoch richtungsweisend, da es alle Informationen direkt aus der Versionsgeschichte des Projektes bezieht und man so recht genaue Annahmen über Defektwahrscheinlichkeiten erhält. Einschränkend ist jedoch anzumerken, dass ein Projekt eine gewisse Größe und Alter erreicht haben muss, damit genug Daten für das DataMining zur Verfügung stehen. Bei neuen oder kleinen Projekten sind keine sinnvollen Ergebnisse zu erwarten.

Das Projekt DynaMine wirkt sehr durchdacht und weit vorangeschritten und ist sogar als Eclipse-Plugin implemntiert worden.

[3] . . . und dafür sorgt, dass einem zur Kettensäge ein Klappspaten empfohlen wird

6 Fazit

Die drei Vorgestellten Konzepte gehen über das, was statische Verfahren leisten können, teilweise deutlich hinaus. Wenn man die Artikel liest und die Veröffentlichungsdaten dabei beachtet sieht man die deutlichen Fortschritte, die auf dem Gebiet der Analyse vergangener Defektbehebungen in den letzten fünf Jahren gemacht wurden. Auf dem Weg von [1], wo mit einfachen statistischen Verfahren neue statische Analysemethoden evaluiert werden, hin zu einem fast produktionsreifen Konzept wie DynaMine [3], haben die Methoden viel an Benutzbarkeit und Leistungsfähigkeit gewonnen, so dass in großen Projekten bald damit zu rechnen ist, dass diese Verfahren eingesetzt werden können.

Abschließend muss allerdings erwähnt werden, dass alle hier vorgestellten Verfahren eine gewisse Projektgröße voraussetzen, um ausreichend Daten für die Analyse zu sammeln. Bei ALP-Hausaufgaben oder kleinen Projekten werden diese Verfahren uns wohl nie helfen können.

Literatur

[1] Claes Wohlin, Martin Höst, Magnus C. Ohlsson. **Understanding the Sources of Software Defects: A Filtering Approach.** 8th International Workshop on Program Comprehension (IWPC'00), June 10 - 11, Limerick, Ireland 2000, p. 9

[2] Chadd C. Williams, Jeffrey K. Hollingsworth. **Bug Driven Bug Finders.** International Workshop on Mining Software Repositories (MSR), May 2004

[3] V. Benjamin Livshits and Thomas Zimmermann. **DynaMine: Finding Common Error Patterns by Mining Software Revision Histories.** In Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2005), September 2005