Debugging mit der testgetriebenen Fehlernavigation

Forschungsartikel von Michael Perscheid – Lehrstuhl für Software-Architekturen

Die Beseitigung von Softwarefehlern dauert meist sehr lange, da die Suche nach der eigentlichen Fehlerursache sehr aufwändig und mühselig sein kann.

Angefangen mit einem Testfall, welcher den sichtbaren Fehler reproduziert, folgen Entwickler den Fehlerursachen entlang der Infektionskette bis hin zum ursprünglichen Defekt. Dieses idealisierte Vorgehen benötigt jedoch ein grundlegendes Verständnis über das Systemverhalten, da Fehler und Defekt oft sehr weit auseinander liegen und Entwickler somit häufig Programmteile verstehen müssen, welche ihnen nur wenig bekannt sind. Dabei sind sie aber auch weitestgehend auf sich allein gestellt, da gebräuchliche Entwicklungswerkzeuge nur wenig Unterstützung beim Verständnis und der Analyse von Fehlerursachen bieten. So erlauben weder symbolische Debugger noch Test-Runner die Identifikation und die systematische Nachverfolgung solcher Infektionsketten. Zwar ermöglichen Standarddebugger den Zugriff auf den letzten Ausführungsschritt, allerdings liegen die noch unbekannten Fehlerursachen in der nicht mehr verfügbaren Vergangenheit eines Programmes. Dementsprechend müssen Entwickler hauptsächlich auf ihr Können und Gespür vertrauen, um in den komplexen Programmabläufen Fehlerursachen zu finden. Da ihnen jedoch häufig eine gute Intuition und ein vertieftes Programmverständnis fehlen, enden ihre Debugging-Versuche meist in willkürlichem Herumprobieren. Dies nimmt nicht nur sehr viel Zeit in Anspruch, sondern macht die Beseitigung von Fehlern auch zu einer sehr mühseligen Angelegenheit.

> **"When something does go wrong, the people who write programs still lack good ways of figuring out exactly what went wrong. Debugging is still, as it was 30 years ago, largely a matter of trial and error."**
>
> — *Henry Lieberman*
>
> *The Debugging Scandal and What to Do About It, 1997.*

Michael Perscheid ist Doktorand am Lehrstuhl für Software-Architekturen.

Die von uns entwickelte *testgetriebene Fehlernavigation* löst einige dieser Probleme, in dem sie einen zusammenhängenden Wegweiser zur Beseitigung von Fehlern bereitstellt. Basierend auf fehlschlagenden Testfällen decken wir automatisch Anomalien auf, ermöglichen Entwicklern Zugriff auf die gesamte Ausführungsgeschichte und markieren die Infektionskette zum leichteren Auffinden von Fehlerursachen.

Anomalien helfen dabei, die Suche nach Fehlerursachen einzuschränken. Durch den Vergleich von fehlschlagenden und funktionierenden Testfällen entstehen Auffälligkeiten, welche eine hohe Wahrscheinlichkeit besitzen, direkt mit dem Fehler in Verbindung zu stehen. Zum Beispiel sind Methoden sehr verdächtig, wenn sie von allen fehlschlagenden aber nur von wenigen bis gar keinen funktionierenden Tests aufgerufen werden. Ebenso sind Zustände in fehlschlagenden Tests gute Indizien, wenn sie eine Diskrepanz zu bisher verwendeten Objekten aufweisen. Vorstellbar wäre eine Methode, die grundsätzlich mit positiven Zahlen als Argumenten aufgerufen wird, aber bei einem fehlschlagenden Test erstmals eine Null übergeben bekommt. Solche

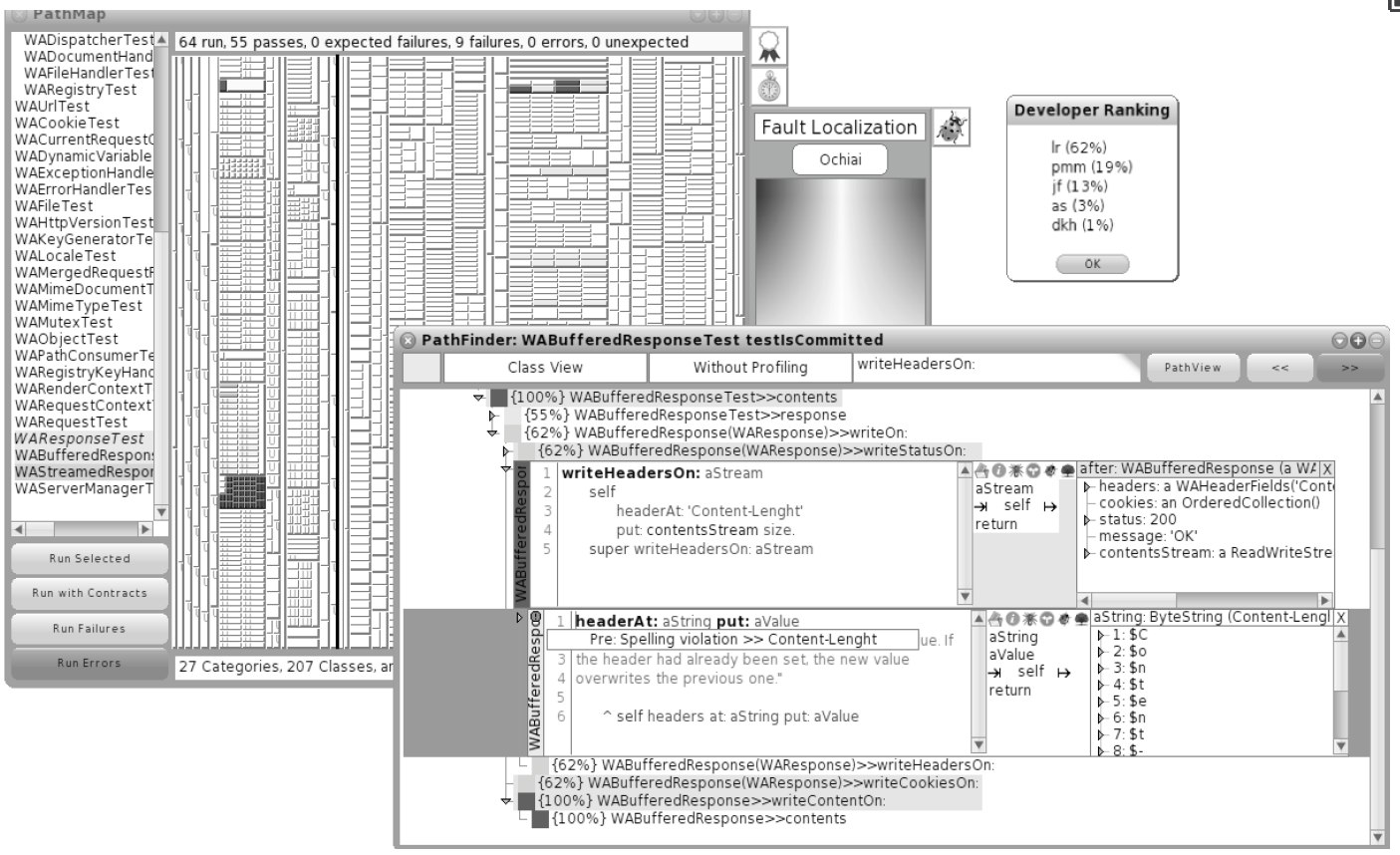

Abbildung 1: PathMap und PathFinder realisieren unsere testgetriebene Fehlernavigation in Form eines erweiterten Test-Runners und eines performanten Back-in-time Debuggers.

Unterschiede führen zwar meist noch nicht zum sichtbaren Fehler, geben aber wertvolle Hinweise für geeignete Fehlerhypothesen. Aus diesem Grund sollten Entwickler beim Debugging mit solchen Anomalien beginnen. Allerdings sind diese allein noch nicht ausreichend, um die ursprüngliche Fehlerursache direkt aufzudecken. Dazu bedarf es meist noch der aufwändigen Zurückverfolgung der Infektionskette.

Um in der Ausführung eines fehlschlagenden Testfalles zurückzugehen und somit Zugriff auf die Infektionskette zu erhalten, bedarf es einer effizienten Analyse der kompletten Ausführung. Bestehende Back-in-time-Debugger speichern jedoch alles, was während eines Programmlaufes geschieht. Dabei erzeugen sie nicht nur Unmengen an Laufzeitdaten, sondern benötigen auch unverhältnismäßig viel Zeit. Um diesen Nachteil zu beheben und die alltägliche Nutzung von Back-in-time-Debuggern zu ermöglichen, haben wir die *inkrementelle dynamische Analyse* entwickelt. Diese Technik teilt die hohen Berechnungskosten über mehrere Testdurchläufe abhängig von den Bedürfnissen des Nutzers auf. Wir nutzen dabei die Reproduzierbarkeit und den Determinismus von Testfällen aus – jede Ausführung eines Testfalls nimmt immer den gleichen Weg durch das System. Auf diese Weise können wir zuerst eine sehr oberflächliche Analyse durchführen und später Details bei Bedarf nachladen. Entwickler erhalten somit eine schnelle Navigation durch das Verhalten, welche inkrementell um weitere Ausführungsdetails ergänzt wird. Dazu müssen wir einfach nur den entsprechenden Test erneut ausführen und die angeforderten Daten aufsammeln. Dies ermöglicht uns eine sehr leichtgewichtige dynamische Analyse der Infektionskette, da nur das Notwendigste gesammelt wird und die hohen Kosten auf mehrere Durchläufe aufgeteilt werden. Entwickler bekommen von diesem Mechanismus nur mit, dass die Ausführungsgeschichte eines Testfalls sofort verfügbar ist.

Zum Aufdecken und dem einfachen Nachverfolgen der Infektionskette kombinieren wir nun die Anomalien mit der Ausführungsgeschichte eines fehlschlagenden Testfalles und erhalten somit unsere testgetriebene Fehlernavigation. Eingebettet in eine systematische Breitensuche bieten wir vier spezifische Navigationstechniken, welche Entwickler immer näher zum Defekt führen. Die *Strukturnavigation* grenzt ver-

dächtige Systemteile und somit den initialen Suchraum ein. Die *Teamnavigation* empfiehlt erfahrene Entwickler zur Behebung des Fehlers, auch wenn die Ursache noch im Verborgenen liegt. Die *Verhaltensnavigation* erlaubt es Entwicklern, die hervorgehobene Infektionskette eines fehlschlagenden Testfalls zurückzuverfolgen. Die *Zustandsnavigation* identifiziert fehlerhafte Zustände und legt auf diese Weise weitere Teile der Infektionskette automatisch offen. Folglich können wir für das Debugging essentielle Fragen in kurzer Zeit beantworten: Wo soll mit der Fehlersuche begonnen werden; wer versteht Fehlerursachen am Besten; was passierte, bevor der Fehler auftrat; und welche Programmteile sind eigentlich betroffen?

Alle vier Navigationstechniken wurden innerhalb unseres *Path tools framework* für die Squeak/Smalltalk Entwicklungsumgebung implementiert. Abbildung 1 zeigt die wesentlichen Werkzeuge *PathMap* und *PathFinder*. Dabei ist *PathMap* ein erweiterter Test-Runner, welcher nicht nur Testfälle ausführt, sondern auch zahlreiche Analysen über deren Verhalten anfertigt. Die Ergebnisse werden dann in Form einer TreeMap, welche das gesamte System repräsentiert, dem Entwickler angezeigt. In diesem Beispiel entsteht eine Heatmap, welche Anomalien in verdächtigen Methoden markiert. Auf diese Weise finden Entwickler heraus, welche Systemteile von dem Fehler betroffen sind (in der Abbildung sind oben rechts mehrere Methoden sehr auffällig gekennzeichnet). Weiterhin setzt *PathMap* die Teamnavigation um und empfiehlt geeignete Entwickler für diesen Fehler. In diesem Beispiel besitzt der Entwickler „lr“ die größte Kompetenz in den verdächtigen Systemteilen und sollte als erstes um Hilfe gebeten werden. Das zweite Werkzeug namens *PathFinder* realisiert unseren performanten Back-in-time-Debugger für Testfälle. Neben dem sofortigen Feedback über das Programmverhalten heben wir verdächtige Methoden und Zustände hervor, um Entwicklern die Rückverfolgung der Infektionskette zu erleichtern. In Abbildung 1 gibt es nicht nur auffällig eingefärbte Methoden; auch unerwartete Zustände werden dargestellt. So zum Beispiel wird hier auf einen Schreibfehler im String-Argument der Methode *#headerAt:Put:* hingewiesen. Diese Anomalie ist nur einen Ausführungsschritt vom ursprünglichen Defekt entfernt, in dem aus Versehen in „Content-Lenght“ ein Tippfehler passierte.

Die testgetriebene Fehlernavigation und unser *Path tools framework* wurden in den letzten Jahren kontinuierlich weiterentwickelt und haben sich als sehr hilfreich bei der Beseitigung von Fehlern erwiesen. In den letzten zwei Jahren konnten Studenten unserer Lehrveranstaltung diese Debugging-Werkzeuge detailliert testen. Dabei bekamen wir nicht nur zahlreiches Feedback für neue Funktionalitäten: Es wurde auch deutlich, dass die Suche nach Fehlern deutlich erleichtert wurde. In einer kontrollierten Nutzerstudie konnte dieser „Verdacht“ bestätigt werden und es zeigte sich, dass die notwendige Debugging-Zeit im Vergleich zu den Standardwerkzeugen halbiert werden konnte.

— *Michael Perscheid*

Wir danken Michael Perscheid für diesen Beitrag und sein Engagement in den Vorlesungen seines Lehrstuhls und wünschen viel Erfolg bei der Dissertation über das vorliegende Thema!

Für mehr Informationen

Debugging 3.0: Die Fehlersuche der Zukunft
http://www.youtube.com/watch?v=k6KUguy84H8

Michael Perscheid, Michael Haupt, Robert Hirschfeld and Hidehiko Masuhara. **Test-driven Fault Navigation for Debugging Reproducible Failures.** In Journal of the Japan Society for Software Science and Technology (JSSST) on Computer Software, vol. 29, no. 3, pages 188-211, 2012, Japan.
https://www.hpi.uni-potsdam.de/hirschfeld/publications/media/PerscheidHauptHirschfeldMasuhara_2012_TestDrivenFaultNavigationForDebuggingReproducibleFailures_JSSST.pdf

Michael Perscheid, Bastian Steinert, Robert Hirschfeld, Felix Geller, and Michael Haupt.
Immediacy through Interactivity: Online Analysis of Runtime Behavior. In Proceedings of the 17th Working Conference on Reverse Engineerings (WCRE) 2010, pages 77-86, 2010, IEEE.
https://www.hpi.uni-potsdam.de/hirschfeld/publications/media/PerscheidSteinertHirschfeldGellerHaupt_2010_ImxmediacyThroughInteractivityOnlineAnalysisOfRunTimeBehavior_IEEE.pdf