Vollautomatischer explorativer Crawler-Test von grafischen Oberflächen

Vorschlag Master-Abschlussarbeit Matthäus Poloczek, Matrikel-Nr. 126826

e-Spirit AG Dortmund

Ein Hauptproblem der Softwareentwicklung ist der langwierige, häufig nichtdeterministische Prozess des Debuggens und der damit verbundenen hohen Kosten, insbesondere, wenn die Entwickler selbst beteiligt sind. Bekannte Ansätze zur Lösung dieses Problems sind die Durchführung von Tests durch dedizierte (weniger kostspielige) Tester, oder auch die Automatisierung von Tests.

Speziell für den Fall von Java GUI mittels Swing gibt es zahlreiche Möglichkeiten variierender Komplexität wie z.B. [QF-Test](http://www.qfs.de/en/qftest/index.html), [uispec4j](http://www.uispec4j.org/), [FEST](https://code.google.com/p/fest/) oder auch [Jemmy](https://jemmy.java.net/). Viele bieten selbst interaktive GUI zur Testerstellung, erweiterbare Module und die Möglichkeit eigener Skripte. Auch Integration von JUnit oder TestNG und Eclipse-Unterstützung sind verbreitet. Was diese Möglichkeiten allerdings gemeinsam haben, ist die notwendige VORGABE der durchzuführenden Eingaben und der erwarteten, zu überprüfenden Resultate. Dies ist insofern ein Problem, dass ein Entwickler, der eine bestimmte abwegige Eingabe nicht voraussieht, vermutlich auch keinen Test dafür vorsehen wird. Selbst ein Tester wird unter Umständen nicht ALLE möglichen Eingabekombinationen probieren, insbesondere auch ungültige und unsinnige Varianten.

Der Vorschlag ist nun ein neues Werkzeug, welches vollautomatisch funktionieren soll und das sich nicht primär mit dem gewünschten und erzeugten Verhalten der GUI beschäftigt, sondern vielmehr systematisch Vertreter aller technisch möglichen Eingaben (Sonderzeichen, extrem lange Eingaben etc.) durchführt. Es würde völlig selbsttätig Knöpfe und Eingabemasken einer Applikation erkennen und systematisch Kombinationen durchprobieren, und für eventuelle Folgebildschirme dasselbe tun. Wenn ein sinnhaltiger Login o.Ä. nötig ist, um an die „Innereien“ eines Programms zu kommen, muss das natürlich manuell vorgesehen oder durchgeführt werden.

Das Vorgehen eines solchen Werkzeugs müsste dann in einer Baum- oder Graphartigen Struktur mitprotokolliert werden, um z.B. auftretende Zyklen zu erkennen und auch, um einen erzeugten und erkannten Fehler reproduzieren zu können. Ebenso ist es unrealistisch, z.B. in einem Textfeld jeden existierenden String zu probieren, hier muss eine maximal problematische und dennoch endliche Untermenge gefunden werden. Ein weiterer Aspekt ist die Beobachtung des Zustands der Applikation – wenn ein „blinder Knopfdruck“ nicht in einer Zustandsänderung resultiert (auf interner Ebene der Swing-Applikation!), ist entweder der Knopf kaputt, in der Situation unangebracht drückbar, oder sogar überflüssig.

Es gibt auch verschiedene Ansätze, so einen Test durchzuführen: Sollte überhaupt die grafische Oberfläche erzeugt, und dann Mausklicks darauf simuliert werden? Oder sollte der Tester eher alle möglichen Kontrollflüsse quasi statisch bzw. offline ausführen? Resultierende Fehler und Meldungen sollten geeignet von den System- und Errorstreams gelesen und protokolliert werden, und man könnte auch eine Schnittstelle für auftretende Exceptions oder Runtime-Errors vorsehen oder sie aus der Applikation abfangen oder mitschneiden.

Die Planung und Implementierung sowie die Anwendung dieses Werkzeugs auf eine Java Swing Applikation der e-Spirit AG wäre der Umfang der Abschlussarbeit.

Existierende Arbeiten Ähnlichem Inhalts sind z.B. [GUITest: a Java library for fully automated GUI robustness testing](http://dl.acm.org/citation.cfm?id=2351739): Eine Java-Bibliothek (allerdings speziell für OSX entwickelt), die vollautomatisch mögliche Inputs in einer beliebigen (allerdings hier nur an einer getestet) Applikation der "Mac OSX Accessibility API" aufspürt und durchführt. Ein Unterschied zum Vorschlag wäre, dass hier zufällig Inputs gewählt werden (anstelle eines systematischen, abdeckenden Ansatzes) und das Tool auch nicht auf die Applikation beschränkt ist (es wird erwähnt, dass die Shutdown und Reset Tasten des Betriebssystems ausgenommen werden mussten). Meine Arbeit wäre für das vermutlich allgemeingültigere Java Swing. Es werden hierin auch weitere Arbeiten referenziert.

Ein anderes Beispiel wäre [A GUI Crawling-Based Technique for Android Mobile Application Testing](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5954416&tag=1). Hierbei wird in zwei seperaten Schritten zunächst eine graphische Applikation vollautomatisch auf mögliche Eingaben und anzeigbare Bildschirme durchsucht, also ein „Wegnetz“ ähnlich dem Vorschlag durch das Programm gelegt, um dann in einem zweiten Schritt dann mit üblichen (regressiven) Test-Applikationen und –Vorgehensweisen Abläufe zu planen und tatsächlich zu prüfen. Der Unterschied und die Einschränkung hier ist hauptsächlich die Android-GUI-API sowie die Testumgebung eines Android-Emulators am Rechner. Auch ist der zweite Vorgehensschritt nicht automatisch und erfordert in jedem Fall erhebliche Nutzerangaben, die Arbeit ist also eher ein Hilfsmittel als vollständige Lösung.

Weitere Überlegungen, die vermutlich nicht mehr im Umfang dieser Master-Arbeit sein würden, aber die z.B. als Erweiterungsmöglichkeiten genannt werden könnten, sind die Nutzung von Bilderkennung / Computer Vision, um auch über Java Swing hinaus GUI testen zu können. Hierbei muss natürlich die Beschriftung von Knöpfen wiedererkannt werden können, und viele GUI benutzen Glyphen anstelle aussagekräftiger Namen.

Vorschlag für Strukturierung der Arbeit:

[Abstract, Einleitung]

1. Hintergrund automatischer GUI-Tests, [QF-Test](http://www.qfs.de/en/qftest/index.html), [uispec4j](http://www.uispec4j.org/), [FEST](https://code.google.com/p/fest/) oder auch [Jemmy](https://jemmy.java.net/)
2. Vollautomatische Tests [GUITest](http://dl.acm.org/citation.cfm?id=2351739), [Crawling-Based Android Mobile Application Testing](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5954416&tag=1)
3. Konzept Vollautomatischer Crawler-Test von grafischen Oberflächen Java Swing
4. Tests bzgl. Fragestellung „GUI simulieren oder offline testen?“
5. Zustandserfassung/-Sicherung/-Verfolgung sowie Kontrollmechanismen
6. Tests an einer konkreten Applikation, Vergleich mit klassischer Methodik
7. Fazit und Erweiterungsmöglichkeiten