Testbericht: Programmanalyse zum Durchklicken

Nils Jessen Anika Nietzer Patrick Petrovic
Sebastian Rauch Michael Schieber

2017-09-03

Inhaltsverzeichnis

I Einleitung

Dieses Dokument beschreibt das Vorgehen beim Testen, verwendete Tools sowie behobene Bugs. Zunächst werden kurz die beim Testen verwendeten Tools vorgestellt. Danach wird die Methodik erläutert, nach der Tests durchgeführt wurden. Dabei wird auch kurz auf Schwierigkeiten eingegangen, die sich beim Testen ergeben haben. Anschließend werden einige größere Bugs, die behoben wurden, aufgezählt und kurz erklärt. Kleine Bugs werden hier vernachlässigt. Schließlich wird ein Überblick über die aktuelle Testüberdeckung gegeben.

II Tools

Um effektiv zu Testen, wurden einige Tools eingesetzt. Dies sind zwar weitestgehend Standart-Tools für das Testen von Java-Code, dennoch werden diese hier kurz vorgestellt.

1. JUnit

JUnit ist ein Test-Framework für Java. Damit können Tests realisiert werden, die automatisch ausführbar sind. Die erwarteten Resultate können mittels Assertions festgelegt werden, die fehlschlagen, wenn die tatsächlichen Resultate von den Erwarteten abweichen. JUnit kann alle Tests einer bestimmten Klasse, eines Packages, oder alle vorhandenen Tests im Projekt ausführen und erstellt dann eine Übersicht über die erfolgreichen und fehlgeschlagenen Tests. Für fehlgeschlagene Tests wird präzise angezeigt, welche Assertion fehlgeschlagen ist und für auftretende Exceptions wird ein vollständiger Stacktrace ausgegeben. Dies ermöglicht komfortables Testen und relativ einfaches Debuggen. Außerdem lässt sich durch die automatischen Tests zuverlässig verhindern, dass einmal behobene Bugs wieder unbemerkt auftreten.

2. Mockito

@Patrick

3. EclEmma

EclEmma ist ein Eclipse-Plugin zum Messen der Code-Coverage. EclEmma basiert auf dem Code-Coverage-Tool JaCoCo, dem Nachfolger von EMMA. Mittels EclEmma lässt sich die Code-Coverage präzise und komfortabel messen: Die Coverage-Statistik wird nach Klassen und Packages aufgeschlüsselt dargestellt. Sehr hilfreich ist auch, dass getesteter Code je nach Coverage-Status eingefärbt wird: vollständig getestete Zeilen werden grün, teilweise getestete Zeilen gelb und nicht getestete Zeilen rot eingefärbt. Dies ermöglicht einen präzisen Überblick darüber, was genau getestet wurde.

4. Benutzertest

Um die Benutzerfreundlichkeit des Programmes zu testen, wurde es einigen Testern vorgestellt, welche mit dem Themengebiet der Informatik vertraut sind. Die Rückmeldung dieser Tester wurde genutzt um die Oberfläche des Programmes benutzerfreundlicher und übersichtlicher zu gestalten.

III Testmethodik

Zunächst wurden einzelne Klassen weitestgehend unabhängig voneinander in Unit-Tests getestet. Für Klassen, die viele Abhängigkeiten zu anderen komplexen Klassen hatten, wurde Mockito benutzt, um Mock-Objekte zu erzeugen, die statt der benötigten komplexen Objekte verwendet werden konnten. Die einzelnen Module wurden jeweils von den Personen getestet, die auch für deren Implementierung verantwortlich waren:

Modul	Person(en)
`dfa.framework`	Nils, Sebastian
`dfa.analyses`	Nils, Sebastian
`controller`	Anika
`codeprocessor`	Anika
`gui`	Michael
`gui.visualgraph`	Patrick

Test-Szenarien aus Pflichtenheft

manuelle Tests

5. Benutzertest

Beim Benutzertest wurden mehrere Tester, welche mit dem Themengebiet der Informatik vertraut sind dazu aufgefordert, dass Programm unter Anleitung eines Programmierers zu Benutzen und Rückmeldung über Benutzerfreundlichkeit und eventuell aufgetretene Bugs zu geben. Aufgrund dieser Rückmeldungen wurden Teile der Benutzeroberfläche überarbeitet, um das Programm für neue Benutzer intuitiver zu gestalten. Dazu zählt zum Beispiel Änderungen an der Größe und Farbe der verschiedenen Arbeitsbereiche um die Benutzerführung beim Start des Programmes zu verbessern. Außerdem wurden kleinere Fehler in der Programmlogik gefunden, welche keine wirklichen „Bugs“ waren, aber die Benutzbarkeit des Programmes eingeschränkt hätten. Dazu zählt zum Beispiel, dass der Benutzer beim Auswählen einer Datei mit Java Code, nur Dateien mit einer .java-Endung angezeigt bekam. Nicht hingegen konnte er Ordner, oder andere Dateitypen in welchen möglicherweise Code gespeichert werden könnte, sehen. Diese Tests bezogen sich hauptsächlich auf Teile der Grafischen Benutzeroberfläche, welche mit den standardmäßigen Test-Frameworks nur schwer sinnvoll getestet werden konnte.

IV Bugs

6. dfa.framework

- Vorberechnung einer DFAExecution geriet in eine Endlosschleife, falls der CFG leere Blöcke enthielt. Grund dafür war, dass ein leerer Block besonders behandelt wurde und in diesem Fall der Block für die nächste Iteration nicht aktualisiert wurde.
- Vorberechnung einer DFAExecution betrachtete nicht alle Blöcke, wenn sich Ausgangszustände von Beginn an nicht änderten. Grund dafür war, dass Blöcke nur auf die Worklist gesetzt wurden, wenn sich der Ausgangszustand eines Vorgängers geändert hatte.
- Künstlicher Endblock im SimpleBlockGraph hatte keine Vorgänger. Grund dafür war, dass nach Erstellen des künstlichen Endblocks die vorherigen Endblöcke nicht als Vorgänger des neuen künstlichen Endblocks gesetzt wurden.

7. dfa.analyses

8. gui

- Die Hintergrundfarbe von `JComponents` wurde auf MacOS nicht richtig angezeigt, da der Hintergrund dieser Komponenten auf MacOS standardmäßig durchsichtig ist. Da alle `JComponents` ihre Standardwerte für unser Programm über eine Art Decorator-Muster erhielten, konnte dies mit einer Zeile Code behoben werden.
- Um auf MacOS den Vollbildmodus zu aktivieren musste in Java Code hinzugefügt werden, sodass diese Funktion freigeschalten wird.

V Statistiken