Qualitätssicherungsdokument

Hanselmann, Hecht, Klein, Schnell, Stapelbroek, Wohnig

18. März 2017
v0.4

Inhaltsverzeichnis

1 Einleitung **3**

2 Codereviews **4**
2.1 Planung 4
2.2 Ergebnis 4

3 Unit-Tests **5**
3.1 Planung 5
3.2 Übersicht über gefundene Fehler 5
3.3 Testüberdeckung 5

4 Performance und Verbrauch: **7**

5 Fehlerbehebungen **11**

6 Verbesserungen in der Phase **12**

7 Anhang **13**
7.1 Testprotokolle 13

1 Einleitung

2 Codereviews

2.1 Planung

Wir haben die Qualitätssicherungsphase mit Codereviews angefangen. Hierfür wurde unsere Gruppe in Gruppen zu je zwei Leuten unterteilt, wobei darauf Wert gelegt wird, dass diese, die sich gegenseitig ihren Code erklären müssen, möglichst wenig über den Code des anderen wissen. Wir haben hiermit angefangen, um möglichst schnell die gröbsten Fehler im Code zu finden, sodass wir uns im weiteren Verlauf des Qualitätssicherung auf versteckter liegende konzentrieren konnten. Ein weiterer wichtiger Aspekt dieser Codereviwes war, den Code zu refactorn, um die Lesbarkeit, Wartbarkeit und spätere Testbarkeit zu erhöhen.

2.2 Ergebnis

Das Ergebnis der Codereviews ist nicht ganz eindeutig. Während sie manchen Personen geholfen haben Fehler zu finden, die beim späteren Testen wahrscheinlich nicht entdeckt worden wären, und den Code an sich etwas robuster zu machen, haben andere eigentlich nur ein paar Style Fehler gefunden, und angegeben, dass ihnen die Codereviews eigentlich nicht geholfen hätten. Dies kann aber auch daran gelegen haben, dass die Leute zu schnell über den Code gegangen sind, und die andere Person nicht tiefgründig genug gehende Fragen gestellt hat.

3 Unit-Tests

3.1 Planung

Neben den Codereviews haben wir anfangs parallel (zum Beispiel weil ein Gruppenmitglied einer Zweiergruppe keine Zeit hat und sein Partner etwas zu tun braucht) und später auch verstärkt darauf hinarbeiten Testfälle für den Code zu schreiben. Zum einen werden wir alle Testfälle, welche im Pflichtenheft genannt wurden, implementieren. Sollte der Testfall GUI Bezug haben oder an sich nicht mit JUnit realisieren lassen wird er dann von Hand ausgefürt. Dabei ist es jedoch wichtig alle Schritte genau zu dokumentieren, damit der Test, im Falle einer Änderung, auch später noch reproduzierbar ist.

3.2 Übersicht über gefundene Fehler

Dank der Unit-Tests und dem Testen von Hand konnten in dieser Phase viele Fehler gefunden werden, sodass wir hier eine Übersicht über einige geben werde:

- Es gab einen Fehler in der Codegenerierung, sodass zum Beispiel Voting Arrays, die gleich sein sollten, unterschiedlich waren.
- In manchen Fälle ließ sich die Analyse nicht starten.
- Ein paar Nullpointer Exceptions.
- Die Ausgabe von CBMC konnte nicht immer richtig geparsed werden.
- Ein Fehler in der Codegenerierung, wenn man „EXISTSONE“ verwendet.
- Fehler bei der Präferenz Wahl, bei der Wähler Kandidaten die selbe Position geben konnten.

3.3 Testüberdeckung

Zur Bewertung unserer Tests setzten wir als Metrik auf die „Instruktionsüberdeckung“, da sich diese am leichtesten messen lässt, und für so ein komplexes Programm gut anzeigt, welche Bereiche noch weiterer Tests bedürfen. Weiterhin wird aber auch darauf geachtet, dass in den Methoden der einzelnen Klassen eine möglichst hohe Pfadüberdeckung gegeben ist. Da die Metrik-Werkzeuge, welche wir verwenden zwar nicht überdeckte Pfade anzeigen, daraus aber keine Ausdrucksvolle Metrik bauen können, fließt sie nicht in die

Metrik an sich mit ein, auch wenn darauf geachtet wurde.
Momentan erreiche wir eine Testabdeckung von ca 77 % (Stand 15.3.17 am Abend). Für das fertige Dokument kommt hier ein Graph hin, in dem man die Testabdeckung im Laufe der Zei (mindestens zu jedem milestone) erkennen kann.

4 Performance und Verbrauch:

Über die Phase haben wir unser Programm stetig in einem Profiler betrachtet, um schnell reagieren zu können, sollte eine Änderung in dieser Phase die Lauffähigkeit unseres Programmes stärker als Erwartet beeinflussen.
Die war jedoch nicht der Fall, sodass der Resoucenverbraucht vor und nach der Qualitätssicherungsphase relativ konstant geblieben ist.
Wie man in 4.1 und 4.2 erkennen kann, ist der Verlauf des Speicherverbrauches so gut wie identisch mit ca 30MB, bevor der „garbage collector“ es wieder auf ca 10 MB herunterbringt. Anscheinend haben viele unserer Objekte nur eine kurze Lebensdauer, woraus sich auch schließen ließe, dass unser Programm im „Leerlauf“ einen insignifikanten Speicherverbrauch hat, der Computersysteme von heute vor keine große Aufgabe stellen sollte.
Vergleicht man nun 4.3 mit 4.4 sieht man, dass sich die Unterschiede der Versionen, während eine Eigenschaft überprüft wird, schon stärker unterscheiden. Während der Arbeitsspeicherverbrauch zwar noch relativ ähnlich zwischen den beiden Versionen ist, sieht man, dass die Auslastung des Prozessors schon deutliche Unterschiede aufweist, welche jedoch vor allem darauf zurückzuführen sind, dass nicht die exakt gleichen Wahlverfahren verglichen wurden, da sich im Laufe der Qualitätssicherungsphase etwas am System zum Speichern der Wahlverfahren geändert hatte.
Der Grund, aus dem der Resourcenverbrauch bei der Überprüfung so viel höher liegt, ist, dass in dieser Phase zum einen der Code, welcher an CBMC gesendet werden muss, für jede Eigenschaft einzeln erzeugt wird, und auch mehrere Threads konstant die Ausgabe von CBMC auffangen müssen. Ist die Überprüfung jedoch abgeschlossen normalisiert sich der Resourcenverbrauch wieder relativ schnell.

Abbildung 4.1: Dies ist der Resoucenverbrauch des Programmes, während es auf eine Eingabe vom Nutzer wartet und momentan keine Verifikation durchführt

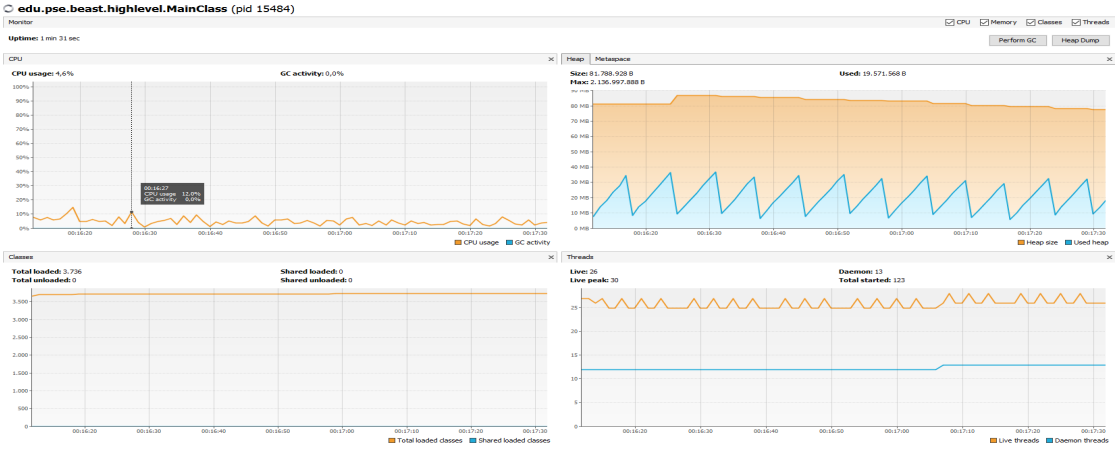

Abbildung 4.2: Der Resourcenverbrauch der momentanten Version des Programmes, während keine Überprüfung durchgeführt wird

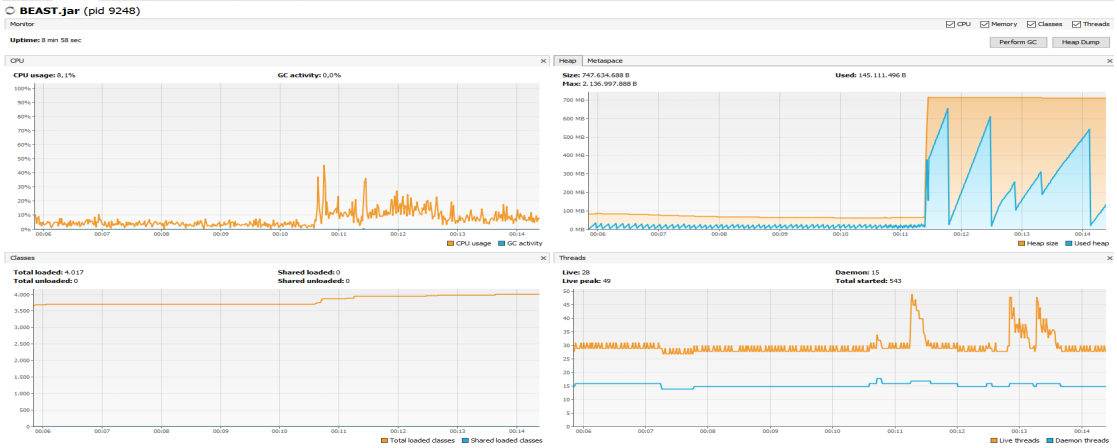

Abbildung 4.3: Der Resourcenverbrauch der originalen Version von BEAST, während Eigenschaften überprüft werden

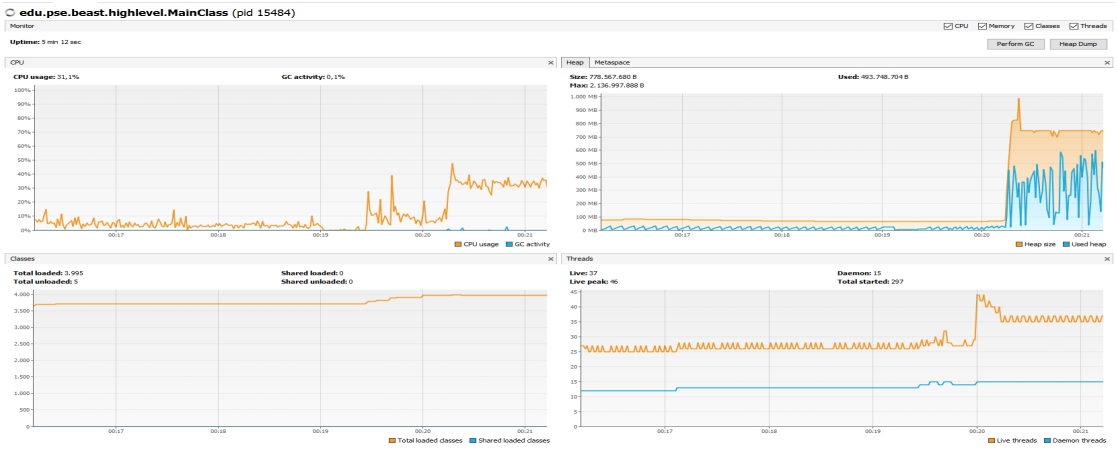

Abbildung 4.4: Der Resourcenverbrauch der momentanten Version des Programmes, während Eigenschaften überprüft werden

Betrachtet man die Verteilung der neusten BEAST Version während einer Analyse (siehe 4.5) fällt auf, dass die Methoden, welche die meiste Prozessorzeit in Anspruch nehmen, die sind, die dafür sorgen, dass das Programm so angenehmt für wie möglich läuft. Würde man zum Beispiel die konstante Überprüfung auf Fehler weniger häuftig ausführen, so müsste der Nutzer länger auf eine Rückmeldung warten, was er noch ändern müsste. Ähnlich verhält es sich zu den „ThreadedBufferedReader“ Instanzen, die auch noch einen großen Anteil an der Prozessorzeit haben. Dies liegt daran, dass sie die gesamte Kommunikation zu außerhalb laufenden Prozessen übernehmen, und deshalb die gesamte Zeit ohne Unterbrechung laufen müssen, solange der Prozess, den sie Überwachen, auch noch läuft.

Hot Spots - Method	Self Time [%]	Self Time	Total Time	Invocations
edu.pse.beast.codearea.ErrorHandling.ErrorFinderThread.**run** ()		176.096 ms (31,1%)	176.181 ms	4
edu.pse.beast.codearea.Autocompletion.FindWordsConcurrently.**run** ()		160.039 ms (28,3%)	160.041 ms	4
edu.pse.beast.highlevel.BEASTCommunicator$1.**run** ()		79.022 ms (14%)	79.171 ms	1
edu.pse.beast.propertychecker.CheckerFactory.**run** ()		76.151 ms (13,4%)	76.446 ms	2
edu.pse.beast.toolbox.ThreadedBufferedReader.**run** ()		34.087 ms (6%)	34.087 ms	15
javax.swing.RepaintManager$ProcessingRunnable.**run** ()		29.220 ms (5,2%)	29.232 ms	11.197
edu.pse.beast.propertychecker.Checker.**run** ()		9.008 ms (1,6%)	9.025 ms	2
edu.pse.beast.highlevel.BEASTCommunicator.**checkForErrors** (edu.pse.beast.highlevel.CentralObjectProvider)		1.832 ms (0,3%)	1.878 ms	1
edu.pse.beast.highlevel.BEASTCommunicator.**createTimeString** (double)		144 ms (0%)	144 ms	1.560
edu.pse.beast.propertychecker.CBMCResult.**mergeLinesToOne** (java.util.Iterator, String)		92,4 ms (0%)	92,4 ms	1.869
edu.pse.beast.highlevel.BEASTCommunicator.**startCheck** ()		85,2 ms (0%)	1.964 ms	1
edu.pse.beast.propertychecker.CBMCCodeGenerator.**generateAST** (String)		68,0 ms (0%)	79,6 ms	2
edu.pse.beast.propertychecker.CBMCResult.**readLongs** (String, java.util.List)		33,8 ms (0%)	63,1 ms	1
edu.pse.beast.propertychecker.CBMCResult.**readTwoDimVar** (String, java.util.List)		27,6 ms (0%)	55,8 ms	1
sun.awt.GlobalCursorManager$NativeUpdater.**run** ()		16,0 ms (0%)	16,0 ms	52
edu.pse.beast.highlevel.BEASTCommunicator.**stopReacting** (edu.pse.beast.highlevel.CentralObjectProvider)		12,9 ms (0%)	12,9 ms	1
edu.pse.beast.toolbox.FileSaver.**writeStringLinesToFile** (java.util.List, java.io.File)		12,0 ms (0%)	12,0 ms	2
edu.pse.beast.propertychecker.WindowsProcess.**createProcess** (java.io.File, int, int, int, String)		11,4 ms (0%)	16,2 ms	2
sun.awt.windows.WComponentPeer$2.**run** ()		11,2 ms (0%)	11,2 ms	28

Abbildung 4.5: Der Prozentuale Anteil einzelner Methoden an der gesamt benutzen Prozessorzeit

5 Fehlerbehebungen

Nummer problem — Ursache — Lösung

6 Verbesserungen in der Phase

Neben Fehlerbehebungen haben wir BEAST in dieser Phase auch in einigen Punkten verbessert:

- Im Eigenschafteneditor gibt es nun einen Knopf, welcher eine Erklärung über die BooleanExpressionLanguage gibt, mit der der Nutzer hier Befehler schreiben kann.
- Der Nutzer kann nun als Nach-Eigenschaft angeben, dass die Analyse erfolgreich war, wenn ein bestimmter Kandidat gewählt wurde.

7 Anhang

7.1 Testprotokolle

Tabelle 7.1: Testfall 8.2 (Testfall für Rückgängig machen und Wiederherstellen)

Sub-Testfall	**Abgedeckte Funktionalitäten**	**Beschreibung**	**Ergebnis**	**Lukas** (Windows 10) Version 1.4.13	**Jemand anderes** Ubuntu 14.0 Version 1.4.13)
/T100/	/FS1100/ /FS2150/ /F0010/ /F0050/	Man startet das Programm ganz normal. Nun gibt man in jedes Feld, das die „Rückgängig machen“ Funktionalität unterstützt, einen kleinen Text ein, und drückt dann, während der Fokus auf dem zu testendem Feld liegt „Strg + z“	Der zuletzt eingegebene Buchstabe oder Textblock (im Falle des Einfügens mit „Strg + c“) wird gelöscht	✓	X
/T110/	/FS1100/ /FS2150/ /F0010/ /F0050	Man startet das Programm ganz normal. Nun gibt man in jedes Feld, das die „Rückgängig machen“ Funktionalität unterstützt, einen kleinen Text ein, und drückt dann, während der Fokus auf dem zu testendem Feld liegt „Strg + z“. Nun drückt man „Strg + r”	Der vorher durch das rückgängig machen verschwundene Buchstabe oder Textblock erscheint wieder	✓	X