Erweiterung eines E-Assessment-Systems um eine Prüfkomponente für die Programmiersprache Python

Enno Lohmann[1]

Abstract: An der Universität Duisburg-Essen wird das E-Assessment-System JACK verwendet. Dieses wird dazu genutzt Vorlesungen zu unterstützen, indem es verschiedene Arten von Aufgaben überprüft und den Benutzern automatisch ein Feedback zu seinen Lösungen gibt. Das System JACK unterstützt beim Prüfen von Programmieraufgaben verschiedene Programmiersprachen und kann durch seinen modularen Aufbau leicht erweitert werden. In diesem Beitrag wird beschrieben, wie das System um eine Prüfkomponente für die Programmiersprache Python erweitert wurde. Es wird hierbei auf den Aufbau von JACK eingegangen. Außerdem werden die unterschiedlichen Testverfahren, die für Python-Quelltexte genutzt werden, näher betrachtet und die unterstützenden Werkzeuge erläutert.

Keywords: E-Assessment-System, automatische Prüfung von Programmieraufgaben, JACK

1 Einleitung

An der Universität Duisburg-Essen betreibt der Lehrstuhl „Spezifikation von Softwaresystemen“ das Übungs- und Prüfsystem JACK. JACK wird dazu benutzt, um verschiedene Vorlesungen zu unterstützen und Übungen und Prüfungen durchzuführen. Durch seine Architektur ist es möglich, JACK um weitere Komponenten zu erweitern.

In diesem Beitrag wird näher auf die Entwicklung von Prüfkomponenten für die Programmiersprache Python eingegangen. Diese sollen dann prototypisch in JACK implementiert werden, sodass es möglich ist, Aufgaben im System zur Verfügung zu stellen, diese zu prüfen und ein Feedback zu geben. Die Prüfung der einzelnen Aufgaben soll in zwei Schritten realisiert werden. Als erstes wird eine statische Prüfung des Quellcodes durchgeführt, deren Ziel es ist, Kompilierfehler zu finden. Der zweite Test ist ein dynamischer Test mittels Modultests, welches die inhaltliche Korrektheit der Abgaben prüft.

Für die verschiedenen Tests werden Werkzeuge eingesetzt, mit deren Hilfe sich die abgegebenen Quelltextdateien prüfen lassen. Das Feedback, das von diesen Programmen gegeben wird, wird daraufhin zusammengefasst und es wird eine Bewertung gebildet.

[1] Universität Duisburg-Essen, Holdenweg 97, 45143, Essen, Enno.Lohmann@stud.uni-due.de

2 E-Assessment-System JACK

JACK ist ein System zum Prüfen von verschiedenen Arten von Aufgaben. Es wird vom Lehrstuhl „Spezifikation von Softwaresystemen“ an der Universität Duisburg-Essen betrieben. Das Ziel von JACK ist es Vorlesungen zu unterstützen, indem Aufgaben automatisch geprüft und bewertet werden können.

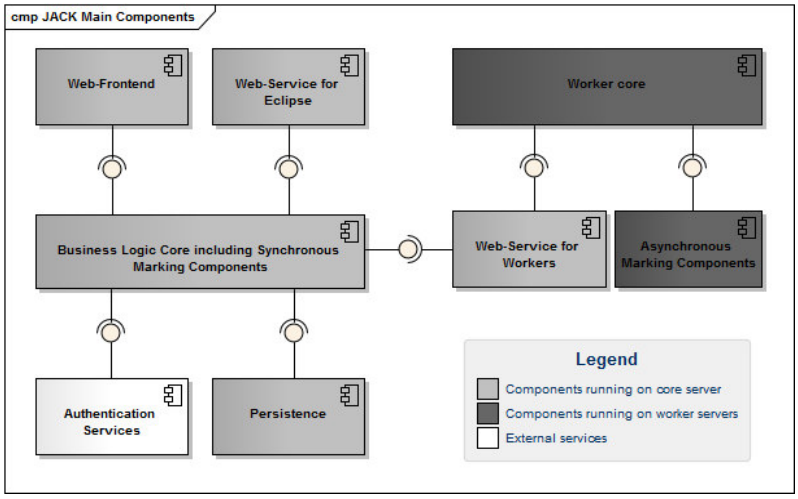

Abbildung 1 Komponentenmodell JACK – [SZG15]

Die Architektur von JACK, welche man in Abbildung 1 sieht, ist modular aufgebaut. Dies hat den Vorteil, dass es einfach ist neue Komponenten zum System hinzuzufügen und somit den Funktionsumfang zu erweitern. Die Architektur von JACK besteht im Wesentlichen aus zwei Hauptkomponenten. Die erste Hauptkomponente ist das Core-System. Dieses besteht aus dem Benutzerinterface, dem Datenspeicher und den synchronen Checks. Die zweite Hauptkomponente ist das Backend-System, welches für die asynchronen Checks zuständig ist. Um das System zu erweitern muss nur eine neue Checker-Komponente hinzugefügt werden, die mit dem Backend zusammenarbeitet. [SZG15] Für die Prüfungen mit JACK existieren verschiedene Checker, sodass sich unterschiedliche Testverfahren durchführen lassen. Zu den Testverfahren für Quelltexte gehören z.B. Run Time Traces oder eine statische Codeanalyse mithilfe eines GReQL-Checkers.

2.1 Verwandte Systeme

Es gibt einige Systeme, die Ähnlichkeiten mit JACK haben und dabei auch die Programmiersprache Python unterstützen. Ein solches System ist ViLLE [K07]. Dieses soll den Benutzern, unabhängig von der Programmiersprache, grundlegende Funktionen verständlicher machen. Dies geschieht über eine Visualisierung des Quelltextes.

Ein weiteres System ist die ECAutoAssessmentBox [APR06]. Dieses System ermöglicht automatische Rückmeldungen zu Programmieraufgaben zu geben. Die Rückmeldungen basieren auf Modultests und es wird das erhaltene und erwartete Ergebnis angezeigt.

Das System Jype [HM10] kombiniert die anderen beiden Systeme. Mit diesem System lässt sich auch eine Rückmeldung zu Python-Programmen bilden. Diese basiert, wie bei der ECAutoAssessmentBox, auf Modultests. Zusätzlich wird eine Visualisierung der geforderten Werte erzeugt, die man dann mit seinen Ergebnissen vergleichen kann.

Diese Systeme bieten jedoch einen kleineren Funktionsumfang als JACK. Es fehlt z.B. die Möglichkeit einer statischen Prüfung des Quelltextes. Deswegen wird in diesem Beitrag die Erweiterung für JACK vorgestellt, welche die benötigten Funktionen bietet.

3 Dynamische Tests

3.1 Testverfahren

Es gibt verschiedene Testverfahren, die genutzt werden können, um Quelltexte dynamisch zu prüfen. Zwei dieser Verfahren sind für das Prüfen von Python Quelltexten im Umfeld von JACK relevant. Zum einen gibt es die Möglichkeit Modultests [W00] durchzuführen. Mit den Modultests wird die Funktionsfähigkeit von Methoden oder ganzen Klassen getestet. Es gibt verschiedenen Arten von Modultests.

Die erste Art von Modultests sind Blackbox-Tests. Diese prüfen das Programm ohne auf den eigentlichen Ablauf zu achten. Ein Programm wird mit bestimmten Parametern ausgeführt und die Ausgaben mit dem erwarteten Wert verglichen. Diese Art von Tests ist simpel, aber man erhält keine genauen Kenntnisse darüber, wo ein Fehler aufgetreten ist. Die zweite Art von Modultests sind Whitebox-Tests. Diese sind komplizierter durchzuführen, bieten aber auch mehr Informationen als die Blackbox-Tests. Bei den Whitebox-Tests wird der gesamte Ablauf des Programms beobachtet und man kann z.B. einzelne fehlerhafte Anweisungen finden.

Außerdem gibt es für Python noch die Möglichkeit Doctests [S14] zu schreiben. Bei den Doctests werden Testfälle direkt in der Dokumentation definiert. Doctests sind nicht so mächtig wie die Modultests, können aber genutzt werden um z.B. Beispiele für die Nutzung von Funktionen direkt in der Dokumentation anzugeben. Es wäre auch möglich die Doctests zusammen mit JACK zu nutzen, da aber die Modultests mächtiger sind ist dies nur bedingt sinnvoll.

3.2 Werkzeuge

Für das Durchführen von Modultests in Python gibt es zwei verbreitete Werkzeuge. Diese

Werkzeuge sind Python's Nose[2] und Pytest[3]. Der Funktionsumfang, der von beiden geboten wird, ist zu großen Teilen identisch und es hängt hauptsächlich von den Präferenzen der Tester ab, welches sie verwenden wollen. Beide Werkzeuge bieten nützliche Funktionen an, die das Durchführen der Testfälle erleichtern. Der erste Vorteil, den die Tools bieten, ist das automatische Finden der Testfälle. Man kann einen Ordner angeben und es werden automatisch alle Testfälle, die sich in dem Ordner und der darunter befindlichen Ordnerstruktur befinden, gefunden. Ein weiterer Vorteil ist, dass die Syntax zum Erstellen der Testfälle erleichtert wird, sodass man sich stärker auf das Definieren der eigentlichen Testfälle konzentrieren kann. Zusammenfassend kann man sagen, dass die Werkzeuge das Erstellen von Testfällen stark erleichtern und die Möglichkeiten bei der Testfallerstellung erhöhen.

4 Statischen Checks

4.1 Testverfahren

Neben den dynamischen Tests gibt es auch statische Tests des Quelltextes. Diese Tests werden vor der Kompilierung durchgeführt. Die Prüfung kann dabei direkt auf dem Quelltext durchgeführt werden oder auf einem auf dem Quelltext basierenden Graphen. Die statischen Checks prüfen hauptsächlich auf Syntaxfehler. Hierbei wird geprüft, ob der Quelltext lauffähig ist oder ob Syntaxfehler vorhanden sind. Außerdem wird in manchen Fällen noch geprüft, ob der Pep 0008[4] Standard, welcher bestimmte Konventionen zum Schreiben von Python Quelltexten festlegt, eingehalten wurde.

4.2 Werkzeuge

Für die statischen Checks gibt es drei Werkzeuge, die sich in Funktionsweise und Qualität unterscheiden. Die vorhandenen Tools sind PyChecker[5], Pyflakes[6], und Pylint[7].

PyChecker ist das älteste der drei Werkzeuge. Da die Weiterentwicklung schon im Jahr 2008 eingestellt wurde, ist es möglich, dass PyChecker nicht mehr mit neueren Python-Quelltexten kompatibel ist. PyChecker beschränkt sich bei der Prüfung auf die Syntax und ignoriert Verstöße gegen den Pep0008 Standard. Außerdem muss der Quelltext für die Prüfung importiert werden, wodurch es zu Seiteneffekten kommen kann.

Das zweite Werkzeug ist Pylint. Mit Pylint können ohne Probleme aktuelle Quelltexte in Python 2.5 oder 3.4 geprüft werden. Das Werkzeug wird heute noch weiterentwickelt und

[2] https://nose.readthedocs.org/en/latest/ (29.07.2015)
[3] http://pytest.org/latest/ (29.07.2015)
[4] https://www.python.org/dev/peps/pep-0008/ (03.09.2015)
[5] http://pychecker.sourceforge.net/ (29.07.2015)
[6] https://pypi.python.org/pypi/pyflakes (29.07.2015)
[7] http://www.pylint.org/ (29.07.2015)

gepflegt. Es werden viele Tests von Pylint durchgeführt. Es wird auf Syntaxfehler in den Quelltexten und auf verschiedene Verstöße gegen den Pep 0008 Standard geprüft. Pylint bietet über eine Konfigurationsdatei die Möglichkeit, die Tests einzuschränken und auf seine Bedürfnisse anzupassen. Seiteneffekte werden von Pylint vermieden.

Das letzte Werkzeug für die statische Prüfung von Python-Quelltexten ist Pyflakes, welches sich insbesondere in der Geschwindigkeit von den anderen beiden Werkzeugen abhebt. Pyflakes wird bis heute weiterentwickelt und gewartet, sodass es keine Probleme bereitet aktuelle Python 2.5 und 3.4 Quelltexte zu prüfen. Pyflakes führt die Überprüfung, anders als die beiden anderen Werkzeuge, auf einem Syntaxbaum durch.

Die drei Werkzeuge haben alle ihre Vor- und Nachteile. Pychecker ist nicht mehr zu empfehlen, da es schon seit längerem nicht mehr gepflegt wird. Für eine schnelle Überprüfung eignet sich Pyflakes am besten und für eine genaue Prüfung des Quelltextes ist Pylint am besten geeignet.

5 Implementierung

Für die Prüfung der Python-Quelltexte wurden zwei Checker-Komponenten entwickelt und in JACK prototypische implementiert. Die erste Komponente führt eine statische Prüfung des Quelltextes mithilfe von Pylint durch. Die zweite Komponente führt die dynamische Prüfung des Quelltextes durch. Hierfür werden Testfälle von Dozenten erstellt, die mithilfe von Python's Nose durchgeführt werden. Die Checker-Komponenten arbeiten eng mit dem Backend von JACK zusammen. Hierzu wird ein vom Backend angebotenes Interface in den Checkern implementiert.

Der Ablauf der Prüfung ist in beiden Komponenten identisch. Beim Aufruf der Methode doCheck übergibt das Backend eine Map mit den zu prüfenden Dateien und den zusätzlich benötigten Informationen wie z.B. den Testtreibern oder der Konfiguration. Diese Dateien werden dann lokal abgelegt. Anschließend wird das passende Werkzeug auf die Dateien aufgerufen und der Test durchgeführt. Die Ausgabe des Werkzeugs wird dann von der Checker-Komponente eingelesen und in ein Format umgewandelt, welches an das Backend zurück gesendet werden kann. Die Rückgabe an das Backend besteht dann aus einer Liste der gefundenen Fehler und einer Bewertung die von 0 bis 100 Punkte reicht.

5.1 Beispielablauf der Implementierung

Um die Implementierung unter passenden Bedingungen testen zu können, wurde eine Programmieraufgabe aus der Vorlesung Programmierung an der Universität Duisburg-Essen, in Python umgeschrieben. Für die Durchführung des Tests werden einige Dateien benötigt. Dazu gehört die Quelltextvorlage, welche dem Studenten bereitgestellt wird, ein Testtreiber für die dynamischen Tests und die Konfigurationsdatei für Pylint. Ein Ausschnitt aus dem Testtreiber kann man in Abbildung 2 sehen. Hierbei ist wichtig, dass

die fünf Variablen im oberen Teil belegt und über die print-Funktion ausgegeben werden. Dies hat den Zweck zusätzliche Informationen bei den fehlgeschlagenen Testfällen auszugeben. In der letzten Zeile wird geprüft, ob das Ergebnis korrekt ist.

```
42 def test_nicht_reserviert_two():
43     var_erwartet = 2
44     var_erhalten = m.nichtreservierttwo(8,19)
45     var_functionname = "nichtReserviertTwo(8, 19)"
46     var_testname = "test_nicht_reserviert_two"
47     var_weight = "5"
48 
49     print strTestname + var_testname
50     print strAufuf + var_functionname
51     print strErwartet + var_erwartet
52     print strErhalten + var_erhalten
53     print strGewichtung + var_weight
54     assert var_erwartet == var_erhalten
```

Abbildung 2 Testtreiber für dynamic Checker

In der Abgabe, welche für diesen Test verwendet wurde, sind einige Fehler eingebaut. Diese sind sowohl inhaltlich, als auch syntaktisch. Abbildung 3 zeigt einen Quelltextausschnitt, in welchem man einige Verstöße gegen den Pep0008 Standard sehen kann. Außerdem wurden in zwei Funktionen Fehler in der Logik eingebaut. In Abbildung 4 wird die Ausgabe der Webschnittstelle von JACK gezeigt. Außerdem wird hier die Bewertung, welche aus den Bewertungen der einzelnen Checker gebildet wurde, gezeigt.

```
12    def platzieren(self, Tischnummer, anzahl_personen):
13         """Personen an einem Tisch platzieren"""
14         for i in self.alle_tische:
15             if i.tischnr == Tischnummer:
16                 i.setAnzahlPersonen(anzahl_personen)
17                 break
18 
19    def zusatz(self, unbenutzt):
20         print "Test"
21         platzieren(4)
22         return 0
23         test = 5;
```

Abbildung 3 Lösung mit Verstößen gegen Pep 0008 Standard

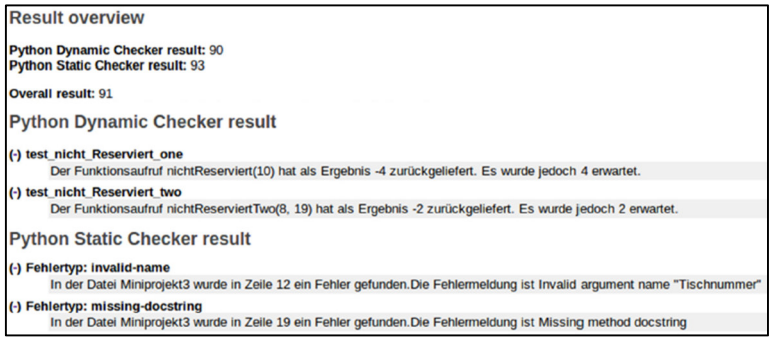

Abbildung 4 Rückgabe in JACK

6 Ausblick

Die hier implementierten Checker bieten die Grundlagen zum Prüfen von Python-Quelltexten und geben ein Feedback, welches dazu geeignet ist, den Benutzern aufzuzeigen, an welchen Stellen ihr Quelltext noch verbessert werden muss. Es gibt jedoch einige Einschränkungen bei der Betrachtung des Quelltextes. So könnte man neben der Syntax und Semantik auch noch andere Aspekte des Quelltextes überprüfen. Einige dieser Aspekte wurden in Kapitel 2.3 genannt. So könnte es sinnvoll sein, den Checker noch um weitere Prüfungen, z.B. der Performanz, zu erweitern. Man könnte auch die statische Prüfung erweitern. Hierzu könnte eine graphbasierte Prüfung [SG13C] implementiert werden. Hierdurch gäbe es mehr Möglichkeiten zur Prüfung des Quelltextes und die Aufgabenstellungen könnten flexibler gestellt werden.

Auch die Ausgabe kann noch erweitert werden. So könnten Modelle der abgegeben Lösung erstellt werden. Diese könnten die Anforderungen für den Benutzern besser kenntlich machen. Eine Erstellung der Objektstruktur ließe sich durch die verwendeten Werkzeuge umsetzen. Pylint bietet diese Funktion an. Über das zusätzliche Tool Pyreverse[8] lassen sich Klassen und Paketdiagramme erstellen. Abbildung 5 zeigt ein solches Paketdiagramm. Durch die Grafiken wäre es leichter den Benutzern die Unterschiede zwischen seiner Lösung und der Geforderten zu zeigen.

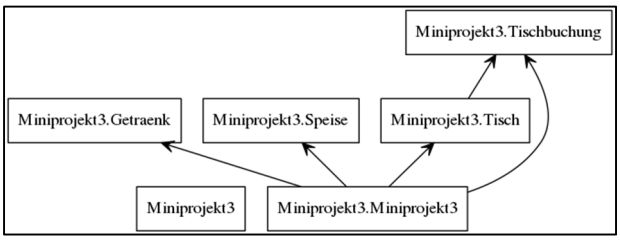

Abbildung 2 Durch Pyreverse erstelltes Komponentendiagramm

Es gibt noch einige Erweiterungen, die für die Prüfung von Python-Quelltexten gemacht werden können, aber man hat durch die hier implementierten Checker eine Grundlage geschaffen, die ausreichend ist, um die Korrektheit von Quelltexten zu prüfen.

7 Fazit

In diesem Beitrag wurde das E-Assessment-System JACK näher betrachtet. Es wurde näher auf die Architektur eingegangen und gezeigt, wie man das System erweitern kann. Anschließend wurden alle Grundlagen, die man für das Prüfen von Python-Quelltexten benötigt, erläutert. Hierzu gehörte zuerst eine genauere Betrachtung von Python und den Prinzipien, die in der Sprache angewendet werden. Außerdem wurden die verschiedenen

[8] https://www.logilab.org/blogentry/6883 (29.07.2015)

Formen von Tests gezeigt, die auf Quelltexte angewendet werden können und welche Werkzeuge hierfür genutzt werden können.

Als letztes wurde die Implementierung der Checker-Komponenten vorgestellt. Hierbei wurde darauf eingegangen, welche Werkzeuge von ihnen genutzt werden und wie der Ablauf der Prüfung ist. Anschließend wurde noch ein Ausblick darüber gegeben, wie die Checker-Komponenten noch erweitert werden könnten. Zusammenfassend kann man sagen, dass in dieser Arbeit eine Komponente für das E-Assessment-System JACK vorgestellt wurde, welche das Prüfen und Bewerten von Python-Quelltexten ermöglicht.

Literaturverzeichnis

[APR06] Amelung, A.; Piotrowski, M.; Rösner, D.: Experience in E-Assessment in Computer Science Education. In ITICSE '06 Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education, 2006, 88-92

[BW00] Barrys Styleguide: http://barry.warsaw.us/software/STYLEGUIDE.txt , 29.07.2015

[GSB08] Goedicke, M.; Striewe, M.; Balz, M.: Computer Aided Assessment and Programming Exercises with JACK. Technical report 28,ICB, University of Duisburg-Essen, 2008.

[HM10] Helminen, J.; Malmi, L.: Jype – A program visualization and programming exercise tool for Python. In SOFTVIS '10 Proceedings of the 5th international symposium on Software visualization, 2010, S. 153-162

[K07] Kaila, E. et.al. : Automatic Assessment of Program Visualization Exercises. In Koli Calling '07 Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88, 2007, S. 151-159

[R00] Guidos original Python style guide: https://www.python.org/doc/essays/styleguide/ 29.07.2015

[S14] Sale, D.: Testing Python, John Wiley & Sons-Verlag, 2014

[SG13A] Striewe, M.; Goedicke, M.: Trace Alignment for Automated Tutoring. In: Proce21edings of International Computer Assisted Assessment (CAA) Conference 2013, Southampton, 2013.

[SG13B] Striewe, M.; Goedicke, M.: Analyse von Programmieraufgaben durch Softwareproduktmetriken. In: SEUH, 2013, 59-68.

[SG13C] Striewe, M.; Goedicke, M.: A Review of Static Analysis Approaches for Programming Exercises. In: Computer Assisted Assessment. Research into E-Assessment : International Conference, 2014, 100-113

[SZG15] Striewe M.; Zurmaar, B.; Goedicke, M.: Evolution of the E-Assessment Framework JACK. In: Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015, Dresden, Germany, 17.-18. März 2015., 2015, 118-120.

[W00] Wirtz, G: Enzyklopädie der Wirtschaftsinformatik – Modultest http://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie, 29.07.2015