paluno
The Ruhr Institute for Software Technology
Institut für Informatik und Wirtschaftsinformatik
Universität Duisburg-Essen

Bachelor-Projekt

Implementierung eines Ruby Prüfmoduls

für das E-Assessment-System JACK

Patrick Czarnetzki
3042742

Herne, 12.11.2019

Betreuung: Dr. Michael Striewe

Studiengang: Angewandte Informatik – Systems Engineering

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe alle Stellen, die ich aus den Quellen wörtlich oder inhaltlich entnommen habe, als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Herne, am 12.11.2019

Zusammenfassung

Die Universität Duisburg-Essen nutzt zur Unterstützung von Vorlesungen und Übungen seit einigen Jahren ein selbst entwickeltes E-Assessment-System namens JACK. Dieses System ist dazu in der Lage, Aufgaben aus verschiedenen Themengebieten zu überprüfen und Studenten ein automatisches Feedback zu ihren Lösungen zu übermitteln. JACK wurde im Laufe der Zeit stetig weiterentwickelt und unterstützt mittlerweile eine Vielzahl von Programmiersprachen.

Das Ziel dieses Bachelor-Projekts lag in der Entwicklung einer solchen Erweiterung für die Programmiersprache Ruby. Es wird beschrieben, wie die Architektur von JACK aufgebaut ist und wie sich dieses System mit einer zusätzlichen Prüfkomponente für die Programmiersprache Ruby erweitern lässt. Weitere Themen, die in diesem Projekt behandelt wurden, sind Grundlagen über die Programmiersprache Ruby, verschiedene Testverfahren und Werkzeuge, die sich für diese Testverfahren eignen. Der Schwerpunkt des Projekts liegt auf die Implementierung und Nutzung der Ruby Prüfkomponente und dazugehörige Tests, um die Prüfkomponente auf seine korrekte Funktionalität zu überprüfen.

Abstract

The University Duisburg-Essen has been using a self-developed E-Assessment-System called JACK to support lectures and exercises for several years. This system is able to review tasks from different topics and provide students with automatic feedback on their solutions. JACK has been continuously developed over time and meanwhile it supports a variety of programming languages.

The goal of this bachelor's project was to develop such an extension for the programming language Ruby. It describes how the architecture of JACK is structured and how to extend this system with an additional check component for the Ruby programming language. Other topics covered in this project are basics of the Ruby programming language, various testing procedures and tools that are appropriate for these testing procedures. The project focuses on the implementation and use of the Ruby test component and associated tests to verify the correct functionality of the Ruby test component.

Abbildungsverzeichnis

Inhaltsverzeichnis

1 Das E-Assessment-System JACK

JACK ist ein E-Assessment-System, das im Jahr 2006/2007 von Paluno veröffentlicht und seit diesem Zeitpunkt vom Lehrstuhl „Spezifikation von Softwaresystemen" der Universität Duisburg-Essen betrieben wird, um Studenten und Lehrkörper bei Vorlesungen, Übungen und Prüfungen zu unterstützen. JACK wurde so entwickelt, dass eine Vielzahl verschiedener Aufgabentypen (Multiplechoice, Lückentexte, mathematische Aufgaben, Programmieraufgaben, u. v. m.) generiert, automatisch korrigiert und mit Feedback versehen werden können. Somit unterstützt JACK Studenten bei ihrem Lernprozess und entlastet Lehrkörper bei der Durchführung von Korrekturen. (Vgl. [St16], S. 36 f.)

1.1 Architektur

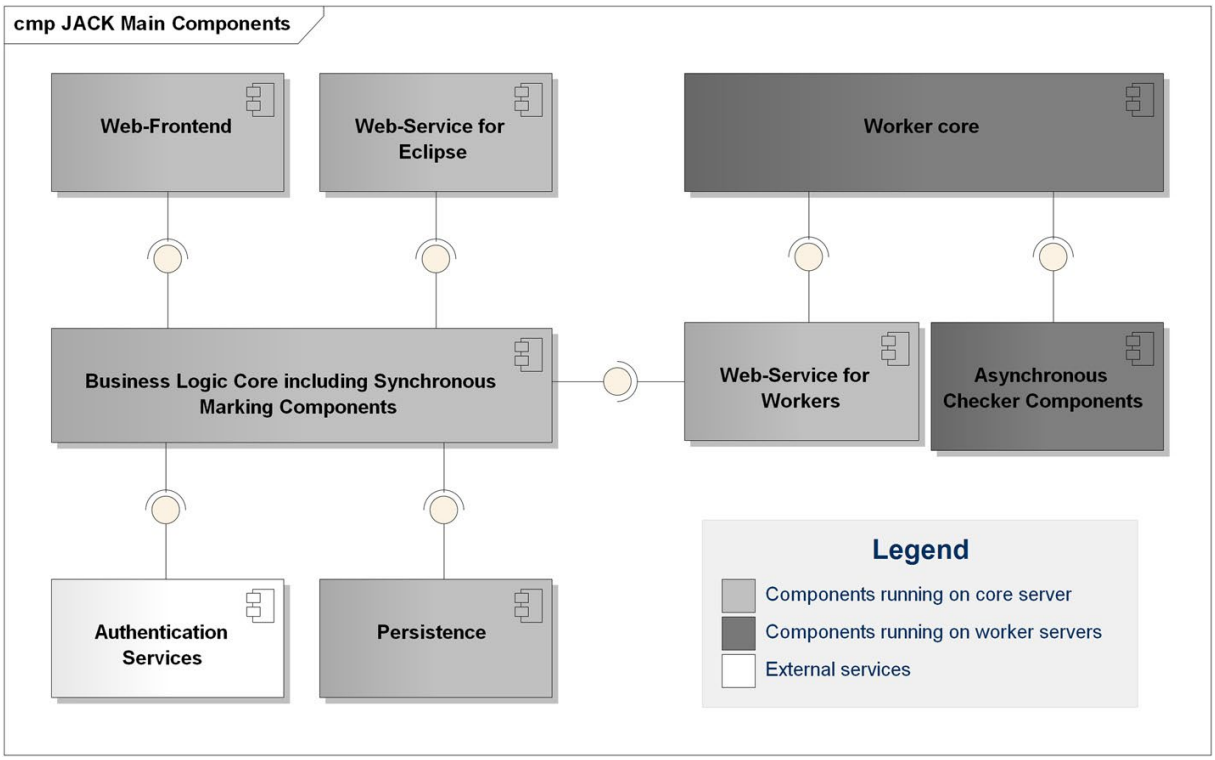

Abbildung 1: Komponentendiagramm der JACK Architektur
Quelle: Michael Striewe: An Architecture for Modular Grading and Feedback Generation for Complex Exercises. Science of Computer Programming 129. Special issue on eLearning Software Architectures, 2016.

Die JACK Architektur teilt sich in zwei wesentliche Bestandteile, dem Core Server und Worker Server. Der Core Server ist auf der linken Seite der Abbildung 1 zu sehen und stellt ein reaktives System dar. Die Komponenten dieses Systems ermöglichen den Zugriff auf den Core Server über Browser-Clients (Web-Frontend) oder über die integrierte Entwicklungsumgebung Eclipse (Web-Service for Eclipse). Weitere Aufgaben liegen in der lokalen Datenspeicherung innerhalb der Datenbank (Persistance), der Authentifizierung externer Systeme mit eigenem Datenspeicher (Authenti-

cation Service) und in der Gestaltung einer Zugriffsmöglichkeit durch Worker Server (Web-Services for Workers). Der Core Server reagiert also im Wesentlichen auf Anfragen, die sich an das Web-Frontend richten, verarbeitet diese Anfragen und speichert bzw. liefert Daten.

Die rechte Seite der Abbildung 1 repräsentiert den Worker Server, der im Gegensatz zum Core Server ein aktives System darstellt. Er besteht aus der Komponente „Worker Core", die sich mit der Kommunikation zum Core Server befasst und zusätzliche Komponenten zur Benotung der eingereichten Lösungen enthält. Die eigentliche Korrektur der eingereichten Lösung wird durch die Komponente „Asnychronous Checker Components" durchgeführt. (Vgl. [St16], S. 39)

1.2 Erweiterungsmöglichkeiten

Die Korrektur einer eingereichten Lösung wird durch die Checker-Komponente durchgeführt. Diese Komponente nimmt die eingereichten Lösungen entgegen, führt die eigentliche Korrektur durch und leitet die Ergebnisse anschließend wieder zurück an die „Worker Core" Komponente. Checker-Komponenten implementieren ein Interface namens „IChecker", durch das gewährleistet wird, dass der Informationsaustausch zwischen Checker-Komponente, „Worker Core" und „Core Server" fehlerfrei abgewickelt werden kann. (Vgl. [St16], S. 40)

JACK lässt sich somit aufgrund seines Architekturstils leicht erweitern, was vor allem der Tatsache geschuldet ist, dass die Checker als eigene Komponente betrachtet werden. Um JACK mit weiteren Prüfkomponenten zu erweitern, müssen dementsprechend nur weitere Checker-Komponenten entwickelt und in das System eingebunden werden.

1.3 Verwandte Systeme

Mir ist lediglich ein System bekannt, das mit JACK verwandt und dazu in der Lage ist, Ruby Quelltexte zu analysieren. Dieses System trägt den Titel „Wizard". Es wurde vom Institut für Informatik der Ryerson Universität aus Toronto, Kanada entwickelt. Außerdem wurde es an derselben Universität in verschiedenen Kursen über mehrere Jahre betrieben und unterstützt die Programmiersprachen Java, C, Perl, Python und Ruby. (Vgl. [Ha12], S. 95 – 98)

2 Die Programmiersprache Ruby

Ruby ist eine dynamische, objektorientierte Programmiersprache, die Mitte der 1990er Jahre von Yukihiro Matsumoto (aka Matz) entwickelt wurde. Inspiriert wurde Ruby durch Lisp, Smalltalk und Perl, greift jedoch auf eine Grammatik zurück, in der sich Java-Programmierer leicht zurechtfinden können. Matsumoto entwickelte Ruby, weil er laut eigener Aussage mit keiner anderen Programmiersprache völlig zufrieden war. Die Intention war eine Sprache zu erschaffen, die das Programmieren erleichtert und somit auch zugleich beschleunigt. (Vgl. [FlMa08], S. 2)

2.1 Popularität

Es gibt verschiedene Ansätze, mit denen versucht wird, die Popularität von Programmiersprachen messbar zu machen.

2.1.1 TIOBE-Index

Der TIOBE-Index wird von der Firma Tiobe Software BV herausgegeben. Er verfolgt den Ansatz, eine Suchanfrage der Form [Programmiersprache] + programming in den populärsten Suchmaschinen auszuführen und anschließend die Anzahl der Treffer miteinander zu vergleichen. (Vgl. [Tă15], S. 3) Im TIOBE-Index ist Ruby innerhalb eines Jahres (Oktober 2018 – Oktober 2019) vom 18. Platz auf den 13. Platz gestiegen.

Oct 2019	Oct 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.884%	-0.92%
2	2		C	16.180%	+0.80%
3	4	^	Python	9.089%	+1.93%
4	3	v	C++	6.229%	-1.36%
5	6	^	C#	3.860%	+0.37%
6	5	v	Visual Basic .NET	3.745%	-2.14%
7	8	^	JavaScript	2.076%	-0.20%
8	9	^	SQL	1.935%	-0.10%
9	7	v	PHP	1.909%	-0.89%
10	15	^^	Objective-C	1.501%	+0.30%
11	28	^^	Groovy	1.394%	+0.96%
12	10	v	Swift	1.362%	-0.14%
13	18	^^	Ruby	1.318%	+0.21%

Abbildung 2: TIOBE-Index Oktober 2019
Quelle: Tiobe Software BV
https://www.tiobe.com/tiobe-index/
Abgerufen am: 06.10.2019

2.1.2 PYPL-Index

Die Herausgeber von „Popularity of Programming Languages" (PYPL) verfolgen einen ähnlichen Ansatz, wie ihn auch die Herausgeber des TIOBE-Index verfolgen. Genau wie beim TIOBE-Index, wird eine Suchanfrage in den populärsten Suchmaschinen ausgeführt und anschließend die Anzahl der Treffer miteinander verglichen. Der Unterschied von PYPL gegenüber TIOBE liegt darin, dass PYPL eine andere Form von Suchbegriff für seine Suchanfragen nutzt: [Programmiersprache] + tutorial. (Vgl. [En15], S. 2) Im PYPL-Index ist Ruby innerhalb eines Jahres (Oktober 2018 – Oktober 2019) vom 11. Platz auf den 13. Platz gefallen.

Worldwide, Oct 2019 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	29.49 %	+4.5 %
2		Java	19.57 %	-2.4 %
3		Javascript	8.4 %	+0.1 %
4		C#	7.35 %	-0.4 %
5		PHP	6.34 %	-1.2 %
6		C/C++	5.87 %	-0.4 %
7		R	3.82 %	-0.2 %
8		Objective-C	2.6 %	-0.7 %
9		Swift	2.57 %	-0.1 %
10		Matlab	1.87 %	-0.2 %
11	↑	TypeScript	1.87 %	+0.3 %
12	↑↑↑↑	Kotlin	1.61 %	+0.6 %
13	↓↓	Ruby	1.47 %	-0.1 %

Abbildung 3: PYPL-Index Oktober 2019
Quelle: PYPL
https://pypl.github.io/PYPL.html
Abgerufen am: 06.10.2019

2.1.3 RedMonk Programming Language Rankings

Im Gegensatz zu den beiden bereits erwähnten Verfahren nutzt RedMonk eine andere Herangehensweise. RedMonk setzt Hashtags und Anzahl an GitHub-Projekten, die auf einer bestimmten Sprache basieren, miteinander in Bezug und wertet die Ergebnisse anschließend auf einer Skala von 1 bis 100 aus. (Vgl. [Tă15], S. 4 f.) Ruby belegte im RedMonk Programming Language Ranking im Juni 2019 den 8. Platz.

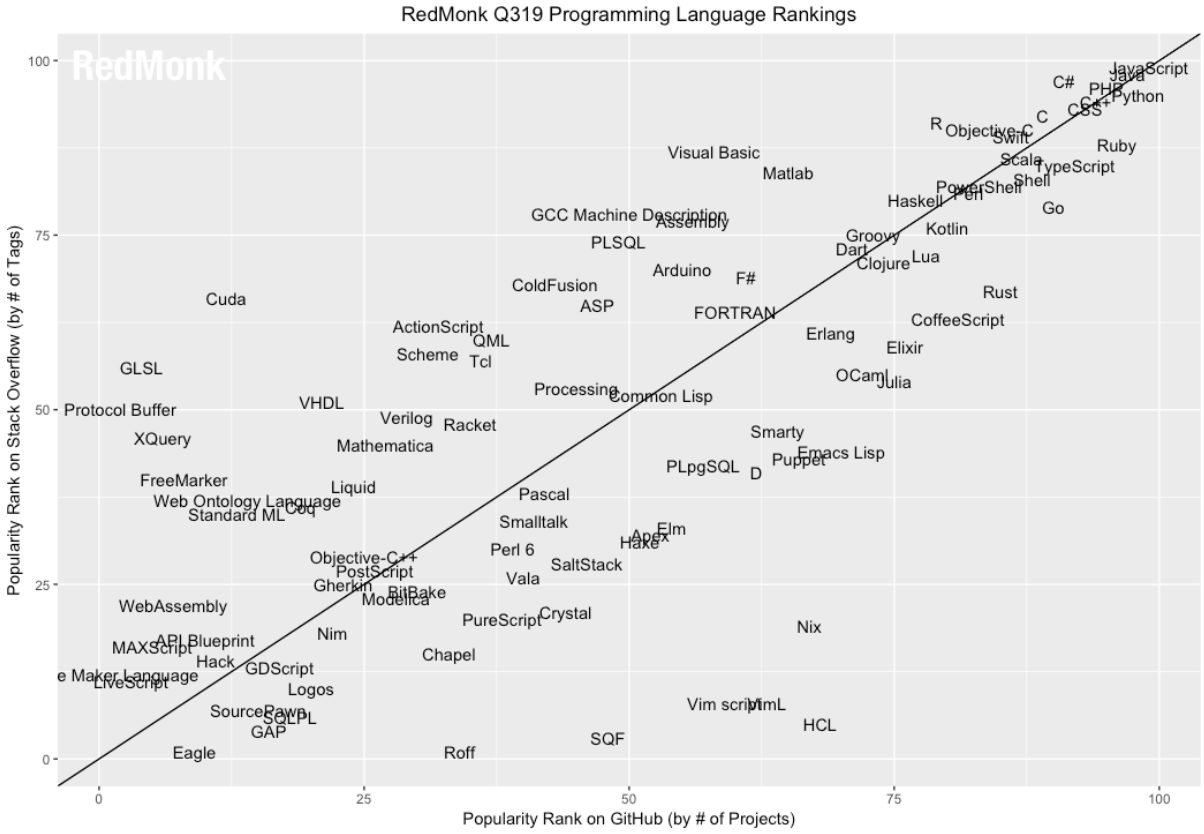

Abbildung 4: RedMonk Programming Language Ranking Juni 2019
Quelle: RedMonk
https://redmonk.com/sogrady/2019/07/18/language-rankings-6-19/
Abgerufen am: 06.10.2019

3 Testverfahren

3.1 Klassifizierungen

Testverfahren lassen sich anhand verschiedener Kriterien klassifizieren. Typisch sind Klassifizierungen nach Teststufen (Modultest, Integrationstest, Systemtest, Abnahmetest) und Prüfmethoden (Statisch, dynamisch). (Vgl. [Wi19], S. 75 ff.)

3.1.1 Statische Tests

Statische Testverfahren analysieren den Quelltext eines Programms, ohne diesen Quelltext und somit das eigentliche Programm ausführen zu müssen. Daraus lässt sich schließen, dass statische Tests sich vorwiegend auf die Syntax eines Programms beziehen und zur Verifikation genutzt werden. Geprüft wird vor allem, ob der Quelltext überhaupt ausgeführt werden kann und ob gewisse Standards eingehalten werden.
(Vgl. [Wi19], S. 84)

3.1.2 Dynamische Tests

Bei dynamischen Testverfahren wird der Quelltext des Programms ausgeführt und die eigentliche Eignung des Programms basierend auf dessen Einsatzzweck analysiert. Dynamische Testverfahren werden dementsprechend zur Validierung eines Programms genutzt. Diese Testverfahren lassen sich anhand der Prüftechnik (strukturorientiert, funktionsorientiert, diversifizierend) und des Informationsstandes (White-Box-Test, Black-Box-Test, Grey-Box-Test) weiter unterteilen. (Vgl. [Wi19], S. 84 f.)

Black-Box-Test

Bei einem Black-Box-Test sind dem Tester die inneren Strukturen des zu testenden Objekts nicht bekannt und die Testfälle werden aus den Spezifikationen abgeleitet. Das Programm wird bei diesen Tests ausgeführt und mit Eingaben gespeist, dessen Ausgaben mit den erwarteten Ausgaben verglichen werden. Die Eingaben und die dazugehörigen, erwarteten Ausgaben werden zuvor in mehreren Testfällen definiert. Das Ziel des Testers liegt darin, ein Systemverhalten zu finden, das den Spezifikationen widerspricht. (Vgl. [Wi19], S. 84)

White-Box-Test

Der Begriff White-Box-Test klassifiziert Testverfahren, bei denen die Tests auf der Grundlage basieren, dass die inneren Strukturen des zu testenden Objekts bekannt sind. Aufbauend auf dieser Grundlage, können Tests entwickelt werden, mit denen die Übereinstimmung der internen Operationen gegenüber den Spezifikationen geprüft werden kann.
(Vgl. [Wi19], S. 84 f.)

Grey-Box-Test

Beim Grey-Box-Test handelt es sich um einen White-Box-Test, der jedoch vor der Implementierung entwickelt wird. Zu diesem Zeitpunkt sind kaum Informationen über den eigentlichen Quellcode vorhanden, was diesen White-Box-Test mit einem Black-Box-Test kombiniert. (Vgl. [Wi19], S. 85)

3.2 Werkzeuge

3.2.1 Statische Testwerkzeuge für Ruby

Es existieren verschiedene Werkzeuge, mit denen statische Tests für die Programmiersprache Ruby durchgeführt werden können. Zu diesen Werkzeugen zählen RuboCop, Reek, Laser und Ruby-Lint. Es gibt natürlich noch viele weitere Werkzeuge, mit denen statische Tests für Ruby durchgeführt werden können, allerdings würde es den Rahmen sprengen, auf alle diese Werkzeuge explizit einzugehen.

RuboCop

Abbildung 5: RuboCop Logo
Quelle: https://www.rubocop.org/en/stable/
Abgerufen am: 08.10.2019

Ein statischer Code-Analysierer und Code-Formatierer für Ruby, der von Bozhidar Batsov entwickelt wurde. Der Name ist eine Anspielung auf den Film „RoboCop" aus dem Jahr 1987. RuboCop prüft den Ruby Quelltext auf Verstöße gegen die Konventionen, die von der Ruby-Community festgelegt werden. Es existiert keine grafische Benutzeroberfläche, stattdessen wird RuboCop in der Konsole ausgeführt. Die Analyse lässt sich mit verschiedenen, zur Verfügung stehenden Kommandozeilenbefehlen anpassen.

Es ist beispielsweise möglich, die Analyse auf Warnungen und Fehlermeldungen (Error, Fatal Error) zu beschränken und somit Verstöße gegen die Konventionen zu ignorieren. Selbstverständlich kann die Analyse auch ausschließlich für Fehlermeldungen erfolgen. Auch die Ausgabe, die standardisiert auf der Konsole ausgegeben wird, lässt sich in verschiedenen Formaten darstellen und speichern. RuboCop ist außerdem in der Lage, einige gefundene Verstöße automatisch zu beheben.
(Vgl. [Ra16], S. 12 f.)

Reek

Abbildung 6: Reek Logo
Quelle: https://github.com/troessner/reek
Abgerufen am: 08.10.2019

Reek wurde von Kevin Rutherford entwickelt und fokussiert sich bei der Analyse auf Ruby Klassen, Ruby Module und Ruby Methoden. Das Programm sucht sogenannte „Code Smells", bei denen es sich um Indikatoren handelt, die sich auf Stellen im Code beziehen, die möglicherweise schwer zu interpretieren, entwickeln und zu pflegen sind. Reek schließt dabei Prüfungen für verschiedene „Smells" ein, wie beispielsweise Attribute, Klassenvariablen, doppelte Methodenaufrufe, zu große Klassen bezüglich der Nutzung zu vieler Konstanten und Methoden, ungenutzte Parameter bzw. Methoden und vieles mehr. Genau wie RuboCop wird Reek mittels Kommandozeilenbefehlen in einer Konsole bedient. (Vgl. [Sc15], S. 11 f.)

Ruby-Lint

Das von Yorick Peterse entwickelte Ruby-Lint ist ein weiteres Werkzeug zur statischen Analyse von Ruby Quelltext, das sich hauptsächlich auf logische Fehler anstelle der Semantik fokussiert, wie beispielsweise die Verwendung nicht vorhandener Variablen. Ruby-Lint analysiert eine Reihe von Dateien, die dem Programm als Argumente übergeben werden. Der Unterschied zu RuboCop liegt darin, dass sich dieses Programm auf technische Probleme fokussiert, anstelle von stilbezogenen Problemen. Genau wie RuboCop, führt dieses Programm verschiedene Prüfstile durch, ist allerdings nicht so komplex wie RuboCop oder Reek. Die Bedienung erfolgt wie bei allen anderen Programmen auch, über Kommandozeilenbefehle in einer Konsole. (Vgl. [Sc15], S. 12)

Laser

Laser ist ein Werkzeug, um die lexikalische Struktur und die semantische Bedeutung von Ruby Programmen zu analysieren. Es wurde 2011 von Michael Edgar entwickelt und ist in der Lage Fehler aufzudecken, die Ruby normalerweise erst zur Laufzeit auffallen. Das Programm kann beispielsweise überprüfen, ob ein bestimmter Codeblock auslöst und welche Zugriffsmodifikatoren für Methoden vergeben wurden. Genau wie RuboCop, wird Laser mittels Kommandozeilenbefehlen in einer Konsole bedient. (Vgl. [Ed11])

3.2.2 Dynamische Testwerkzeuge für Ruby

Genau wie bei den statischen Testwerkzeugen, existieren auch bei den dynamischen Testwerkzeugen verschiedene Werkzeuge, mit denen dynamische Tests für die Programmiersprache Ruby durchgeführt werden können. Zu diesen Werkzeugen zählen RSpec, Unittest und MiniTest.

RSpec

Abbildung 7: RSpec Logo
Quelle: https://rspec.info/
Abgerufen am: 08:10.2019

RSpec ist ein Test-Framework für die verhaltensorientierte Entwicklung (behavior-driven development (BDD)) von Ruby Software, das im Jahr 2005 von Steven Baker entwickelt und im Mai 2007 als RSpec 1.0 veröffentlicht wurde. Mittels RSpec können exzellente Tests entwickelt werden, mit denen das Verhalten eines Ruby Programms spezifiziert und kontrolliert werden kann. Um diese Tests zu spezifizieren, bietet RSpec eine Reihe von Werkzeugen, mit denen Testszenarios, Behauptungen (Assertions) und benutzerspezifische Fehlermeldungen generiert werden können. RSpec wird nun seit mehr als 10 Jahren von einer großen Community verwendet und wurde seit diesem Zeitpunkt stetig weiterentwickelt. (Vgl. [Ta16], S. V)

Test::Unit

Nathaniel Talbott entwickelte das Framework Test::Unit, das in Ruby ab Version 1.8 mitgeliefert und somit bei der Installation von Ruby automatisch vorinstalliert wird. Genau wie in RSpec, stehen eine Reihe von Werkzeugen zur Verfügung, um Testfälle und Behauptungen (Assertions) zu generieren. (Vgl. [TFH09], S. 199 f.)

MiniTest::Unit

Ryan Davis und Eric Hodel entwickelten das Framework MiniTest::Unit, das ab Ruby Version 1.9 das zuvor mitgelieferte Framework Test::Unit ablöste. Dieses Framework wird dementsprechend ab Ruby Version 1.9 automatisch mitgeliefert und bei der Installation von Ruby vorinstalliert. (Vgl. [TFH09], S. 199 f.)

4 Implementierung

Um die Überprüfung von Ruby Quelltexte mittels JACK durchführen zu können, wurden zwei verschiedene Checker-Komponenten entwickelt und in die JACK Architektur implementiert. Beide Komponenten führen dabei verschiedene Testverfahren durch. Die erste Komponente befasst sich mit einem statischen Testverfahren, während die zweite Komponente ein dynamisches Testverfahren durchführt. Beide Komponenten arbeiten dabei mit dem JACK Backend zusammen und nutzen zur Erfüllung ihrer Aufgaben ein vom Backend angebotenes Interface mit dem Namen IChecker. Dieses Interface wurde in beiden Checker Komponenten implementiert.

4.1 Statisches Ruby Prüfmodul

Die statische Ruby Checker Komponente prüft den Quelltext der eingereichten Lösungen auf Verstöße gegen die Konvention und Syntax, ohne den Quelltext dabei auszuführen. Verstöße gegen die Konvention sind Abweichungen von Programmierrichtlinien, die von der Ruby Community über mehrere Jahre definiert wurden. Verstöße gegen die Syntax hingegen sind Verstöße gegen die Regeln der Programmiersprache selbst, die von den Entwicklern von Ruby festgelegt wurden. Die Syntax gibt also vor, wie Ausdrücke in Ruby gebildet werden müssen, damit diese korrekt ausgeführt werden können. Der wesentliche Unterschied zwischen den beiden Verstößen liegt dementsprechend darin, dass bei Verstößen gegen die Konventionen ein Ruby Programm trotzdem ausgeführt werden kann, während bei Verstößen gegen die Syntax ein Programm nicht mehr korrekt ausgeführt werden kann.

Diese Komponente wurde so implementiert, dass sie auf ein externes Werkzeug zurückgreift, um den Quelltext zu analysieren. Es standen mehrere Werkzeuge zur Auswahl, von denen einige bereits im Kapitel 3.2.1 vorgestellt wurden. Die Entscheidung fiel letztendlich aus diversen Gründen auf RuboCop. Bei Ruby-Lint war es nicht möglich, stilbezogene Fehler in der Analyse mit einzubeziehen, wodurch die Komponente keine Verstöße gegen die Konventionen hätte berücksichtigen können. (Vgl. [Sc15], S. 12) Bei Reek waren die Fehlermeldungen für Programmieranfänger nicht informativ genug. Reek findet und benennt die Fehler zwar, schlägt aber keine Lösungen vor, wie sich diese Fehler korrekt beheben lassen. RuboCop hingegen findet und benennt die Fehler und schlägt zusätzlich noch Lösungen vor, die dazu beitragen, die gefundenen Fehler zu korrigieren. (Vgl. [Wa14])

4.1.1 Optionen des statischen Ruby Checkers

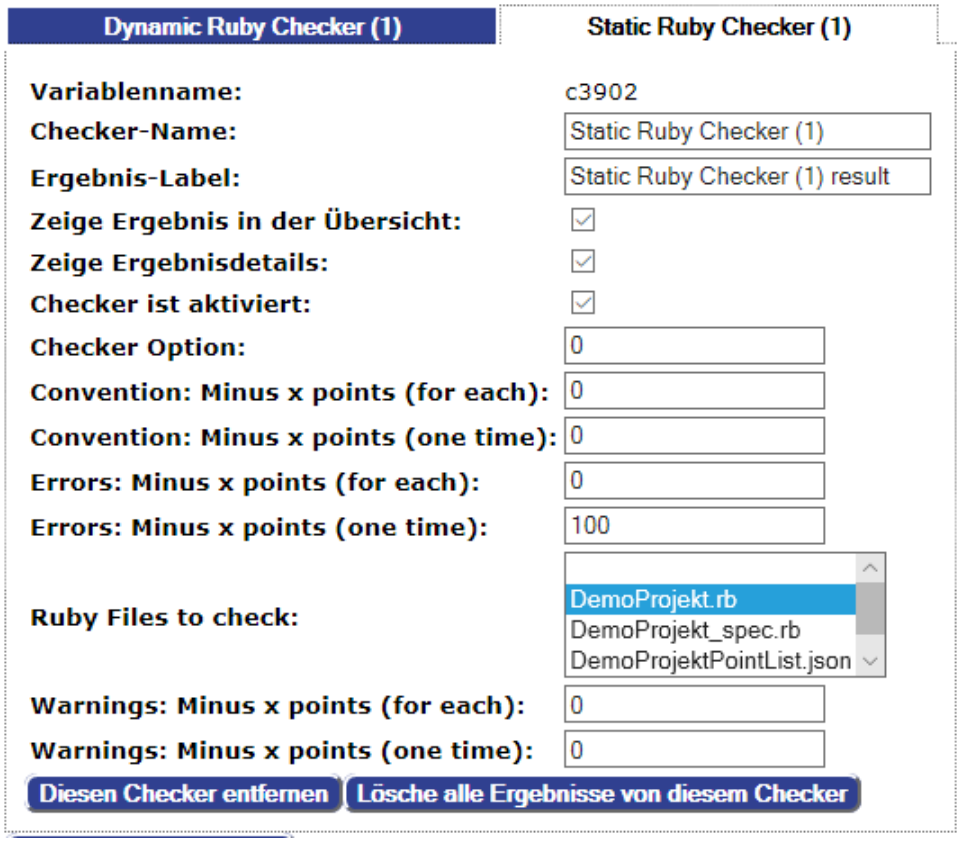

Abbildung 8: Statische Ruby Checker Komponente

Wählen Lehrkörper für ihre Tests die statische Ruby Checker Komponente aus, dann wird ihnen das obige Fenster angezeigt, in dem sie verschiedene Optionen auswählen können.

Checker Option

Lehrkörper können hierüber bestimmen, inwiefern der Quelltext der eingereichten Lösungen untersucht wird. Wird eine 0 eingegeben, werden die Quelltexte auf Verstöße gegen Konventionen als auch Warnungen und Fehlermeldungen untersucht. Wird der Wert 1 eingegeben, werden die Quelltexte auf Warnungen und Fehlermeldungen untersucht, ohne Verstöße gegen Konventionen dabei zu berücksichtigen. Wird eine 2 eingegeben, werden die Quelltexte ausschließlich auf Fehlermeldungen untersucht und Verstöße gegen Konventionen als auch Warnungen werden ignoriert.

Ruby Files to check

Lehrkörper können in diesem Bereich bestimmen, welche der hochgeladenen Ruby Dateien für den Test berücksichtigt werden sollen.

Convention: Minus x points (for each)

Der Wert, der in dieser Zeile eingetragen wird, bestimmt, wie viele Punkte von der Gesamtpunktzahl für jeden einzelnen Verstoß gegen die Konventionen abgezogen wird. Werden beispielsweise 10 Verstöße gegen die Konventionen gefunden und es wurde der Wert 2 eingetragen, dann werden 10 * 2 = 20 Punkte von der Gesamtpunktzahl abgezogen, wobei die Gesamtpunktzahl 100 Punkte beträgt.

Convention: Minus x points (one time)

Der Wert, der in dieser Zeile eingetragen wird, bestimmt, wie viele Punkte von der Gesamtpunktzahl abgezogen werden, wenn ein Verstoß gegen die Konventionen gefunden wurde. Der Unterschied zur vorherigen Option besteht darin, dass dieser Wert nur einmalig abgezogen wird, ganz gleich, wie viele Verstöße gegen die Konventionen gefunden wurden.

Errors: Minus x points (for each)

Dieser Wert bestimmt, wie viele Punkte von der Gesamtpunktzahl abgezogen werden, wenn ein Fehler gefunden wurde. Werden beispielsweise 5 Fehler gefunden und es wurde der Wert 3 eingetragen, dann werden 5 * 3 = 15 Punkte von der Gesamtpunktzahl abgezogen.

Errors: Minus x points (one time)

Dieser Wert bestimmt, wie viele Punkte von der Gesamtpunktzahl abgezogen werden, sobald ein Fehler gefunden wurde. Der Unterschied zur vorherigen Option besteht wiederum darin, dass dieser Wert nur einmalig abgezogen wird, ganz gleich, wie viele Fehler gefunden wurden.

Warnings: Minus x points (for each)

Dieser Wert bestimmt, wie viele Punkte von der Gesamtpunktzahl für jede einzelne gefundene Warnung abgezogen werden. Werden beispielsweise 5 Warnungen gefunden und es wurde der Wert 2 eingetragen, dann werden 5 * 2 = 10 Punkte abgezogen.

Warnings: Minus x points (one time)

Dieser Wert bestimmt, wie viele Punkte von der Gesamtpunktzahl abgezogen werden, sobald eine Warnung gefunden wurde. Der Unterschied zur vorherigen Option besteht wiederum darin, dass dieser Wert nur einmalig abgezogen wird, ganz gleich, wie viele Warnungen gefunden wurden.

4.1.2 Vorgehensweise des statischen Ruby Checkers

Übertragung der getätigten Optionen und Dateien

Sobald eine Lösung von Studenten eingereicht wird, werden die getätigten Optionen übertragen und die Ruby Dateien, die für den statischen Test berücksichtigt werden sollen, lokal zwischengespeichert.

Ausführung von RuboCop

Die lokal gespeicherten und zu überprüfenden Ruby Dateien werden nun mittels RuboCop geprüft. Dazu muss RuboCop zuvor auf dem System, auf dem auch JACK betrieben wird, installiert werden. RuboCop wird nach der Übertragung der getätigten Optionen und Dateien mittels Kommandozeilenbefehl gestartet. Der Kommandozeilenbefehl ist unterschiedlich, je nach dem, welche Option zuvor für den Test ausgewählt wurde. Wurde bei der Checker Option eine 0 ausgewählt, dann lautet der Kommandozeilenbefehl rubocop –format json –out Dateiname.json. Wurde anstelle der 0 eine 1 ausgewählt, dann lautet der Befehl rubocop –only Lint –format json –out Dateiname.json. Wurde eine 2 als Checker Option ausgewählt, dann lautet der Kommandozeilenbefehl rubocop –only Syntax –format json –out Dateiname.json.

Speichern der Ergebnisse in JSON-Datei

RuboCop speichert die Ergebnisse anschließend lokal in einer JSON-Datei, dessen Dateiname automatisch und zufällig generiert wird. Der Dateiname der JSON-Dateien unterscheidet sich dementsprechend von Test zu Test. Die zufällige Generierung des Dateinamens wurde implementiert, um sicherzustellen, dass Studenten keine Möglichkeit haben, selbst eine JSON-Datei hochzuladen. Würde immer der gleiche Dateiname für die JSON-Datei verwendet werden, könnten die Ergebnisse unter Umständen manipuliert werden, indem Studenten selbst eine solche JSON-Datei erzeugen, die keinerlei Fehlermeldungen enthält und diese hochladen. Eingereichte Lösungen wären somit immer korrekt, selbst wenn die eingereichte Lösung eigentlich völlig falsch wären. Um das zu verhindern, werden die Dateinamen immer wieder zufällig erzeugt, mit einer Länge von 20 Zeichen aus den Zeichen a bis z und 0 bis 9.

Einlesen der Ergebnisse aus JSON-Datei

Die statische Checker Komponente wartet nun solange, bis RuboCop eine entsprechende JSON-Datei mit den Ergebnissen erstellt hat. Danach liest die Komponente die relevanten Informationen aus der erstellten JSON-Datei ein und generiert den Output für die Fehlermeldungen, vorausgesetzt, dass auch Fehler gefunden wurden.

Benotung durchführen

Der nächste Schritt besteht darin, die Benotung durchzuführen. Dazu wird zunächst die Anzahl der gefundenen Verstöße gegen die Konventionen und gefundenen Warnungen sowie Fehlermeldungen gezählt. Danach wird die Anzahl der jeweiligen Verstöße mit den durch die Lehrkörper angegebenen Fehlerpunkte multipliziert und die Ergebnisse von der Gesamtpunktzahl abgezogen. Würde der Prozess in einer mathematischen Formel übertragen werden, würde diese wie folgt lauten: 100 – (Anzahl der gefundenen Verstöße gegen die Konventionen * angegebene Fehlerpunkte für Verstöße gegen die Konventionen) – (Anzahl der gefundenen Warnungen * angegebene Fehlerpunkte für gefundene Warnungen) – (Anzahl der gefundenen Fehlermeldungen * Fehlerpunkte für gefundene Fehler). Übrig bleiben dementsprechend die von den Studenten erreichten Punkte.

Vorbereitung der zu versendenden Resultate

Alle bisher beschriebenen Vorgänge werden durch die Methode doCheck ausgeführt, die sich innerhalb der Checker Komponente wiederfindet und ausgeführt wird, sobald Studenten eine Lösung einreichen. Diese Methode gibt ein Objekt des Typs BackendResult zurück. Dieses Objekt besteht aus einer Liste von Strings mit den generierten Fehlermeldungen, zwei String Objekten, die beschreibende Informationen enthalten, die auf JACK ausgegeben werden und den erzielten Punkten. Dieses Objekt wird in diesem Schritt erstellt und am Ende der Methode doCheck zurückgegeben.

Löschung temporärer Dateien und getätigter Optionen

Bevor das Objekt BackendResult zurückgegeben wird, werden zunächst die heruntergeladenen und durch RuboCop erstellten Dateien, die lokal zwischengespeichert wurden, gelöscht. Auch die durch die Lehrkörper getätigten Optionen werden wieder auf den Ausgangszustand zurückgestellt und die Einträge in den jeweiligen Listen gelöscht.

Versenden der Resultate

Am Ende der Methode doCheck wird das zuvor erstellte BackendResult Objekt zurückgegeben und vom JACK Backend verarbeitet. Das Backend zeigt dann die gefundenen Fehlermeldungen sowie die beschreibenden Informationen und die erzielten Punkte auf der JACK Plattform an.

4.2 Dynamisches Ruby Prüfmodul

Im Gegensatz zur statischen Ruby Checker Komponente führt die dynamische Ruby Checker Komponente den Quelltext aus und vergleicht die Ergebnisse mit zuvor definierten Ruby Testdokumenten. Diese Testdokumente werden in einer bestimmten Form geschrieben, die anschließend von einem entsprechenden externen Werkzeug interpretiert und verarbeitet werden kann. Für diese Komponente wurde das Werkzeug RSpec verwendet, dementsprechend müssen die Testdokumente auch in einer Form geschrieben werden, die von dem Werkzeug RSpec interpretiert und verarbeitet werden können.

Genau wie bei der statischen Ruby Checker Komponente standen mehrere Werkzeuge zur Auswahl, von denen einige bereits im Kapitel 3.2.2 vorgestellt wurden. Die Entscheidung fiel aus diversen Gründen auf RSpec. Das Framework Test::Unit wurde nicht in Betracht gezogen, weil es ab Ruby Version 1.9 bereits durch das schlankere Framework MiniTest::Unit abgelöst wurde. (Vgl. [TFH09], S. 199 f.) Eine Entscheidung zwischen RSpec und MiniTest::Unit zu finden war schwieriger, so haben beide Frameworks ihre Vor- und Nachteile. Ein Vorteil von MiniTest::Unit ist beispielsweise, dass es ab Ruby Version 1.9 mitgeliefert wird und somit keine externen Programme integriert werden müssen. Ein Vorteil von RSpec hingegen ist die erstklassige Formatierungs-API, die bereits von vielen Drittanbietern genutzt wird. Die Entscheidung fiel letztendlich auf RSpec, weil mir die Möglichkeiten bezüglich der Ausgabeformate mächtiger erschienen. Mittels RSpec war es beispielsweise möglich, die Resultate der Analysen unkompliziert in einem JSON-Dateiformat zu exportieren, das anschließend von der Ruby Prüfkomponente eingelesen werden konnte. (Vgl. [Ma12])

4.2.1 Optionen des dynamischen Ruby Checkers

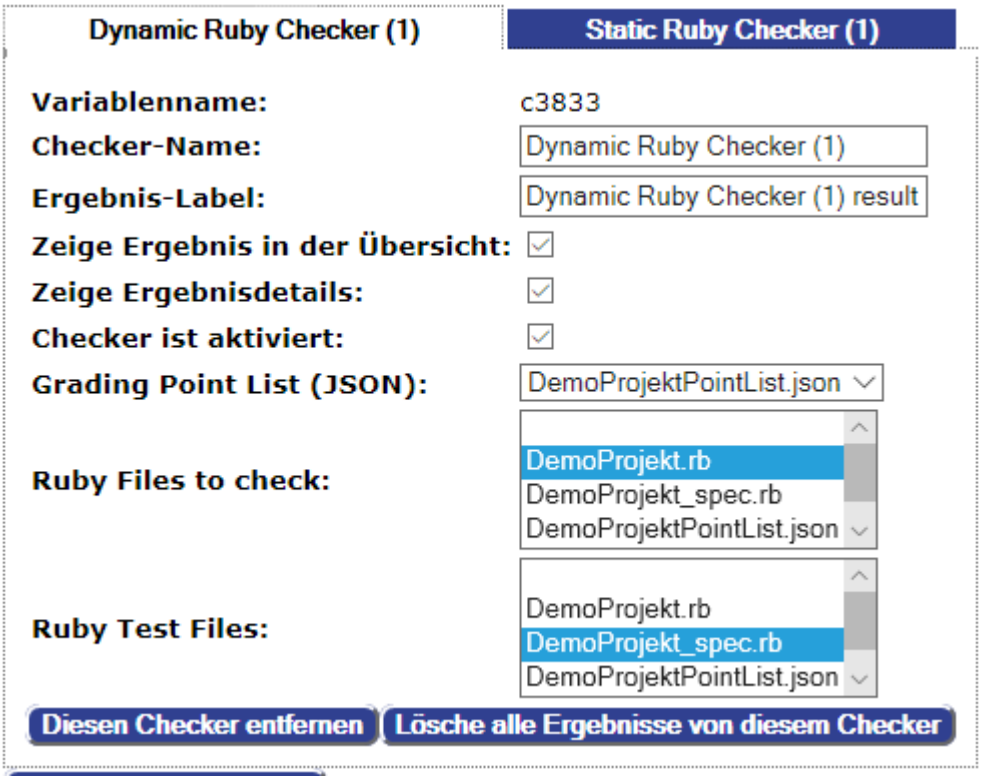

Abbildung 9: Dynamische Ruby Checker Komponente

Wählen Lehrkörper für ihre Tests die dynamische Ruby Checker Komponente aus, dann wird ihnen das obige Fenster angezeigt, in dem sie verschiedene Angaben tätigen müssen. Der Komponente muss mitgeteilt werden, welche der hochgeladenen Ressourcen überprüft und welche dieser Ressourcen zur Testdurchführung und Benotung genutzt werden sollen.

Grading Point List (JSON)

Lehrkörper müssen eine JSON-Datei erstellen, in der festgehalten wird, wie die Benotung der einzelnen Teilaufgaben erfolgen soll. Die Werte für lesson (name, author) sowie exercise (id, name) sind für den eigentlichen Check und dessen Bewertung irrelevant und dienen lediglich zur besseren Zuordnung von JSON-Dateien zu den dazugehörigen Aufgaben. Wirklich relevant für den eigentlichen Check sind lediglich die Werte „points", die sich innerhalb eines JSON-Objekts wiederfinden, wobei diese Objekte wiederum innerhalb eines JSON-Arrays „exercise" enthalten sind. Nachdem eine solche JSON-Datei zur Benotung erstellt und hochgeladen wurde, muss diese unter Grading Point List (JSON) ausgewählt werden, damit die dynamische Ruby Checker Komponente diese Datei zur Benotung heranziehen kann. In der folgenden Abbildung 10 ist der Aufbau einer solchen JSON-Datei zur Benotung zu erkennen.

hello_world_point_list.json

```json
{
  "lesson":
    {
      "name": "Ruby Einstiegskurs: Hello World",
      "author": "Patrick Czarnetzki",
      "exercise":
      [
        {
          "id": "0",
          "name": "Hello World",
          "points": "100"
        }
      ]
    }
}
```

Abbildung 10: JSON-Datei zur Benotung

Ruby Files to check

In diesem Bereich der dynamischen Ruby Checker Komponente müssen die Ruby Dateien angegeben werden, die von der dynamischen Ruby Checker Komponente geprüft werden sollen.

Ruby Test Files

Lehrkörper müssen für jede zu überprüfende Ruby Datei eigene Testdateien formulieren und diese als Ruby Datei speichern. Anschließend müssen diese Dateien im Bereich „Ruby Test Files" ausgewählt werden, damit diese Testdateien von der dynamischen Ruby Checker Komponente für den eigentlichen Test berücksichtigt werden können. Die Testdateien müssen in einem Format definiert werden, das vom Werkzeug RSpec interpretiert und verarbeitet werden kann. Damit RSpec die Testdateien den dazugehörigen und zu überprüfenden Ruby Dateien zuordnen kann, müssen die Testdateien mit dem gleichen Dateinamen beginnen, mit dem auch die zu überprüfenden Ruby Dateien beginnen. Außerdem müssen die Dateinamen auf „_spec.rb" enden, damit diese als Testdateien von RSpec erkannt werden können. Soll beispielsweise die Ruby Datei „Arrays.rb" geprüft werden, muss der Dateiname der dazugehörigen Testdatei „Arrays_spec.rb" lauten. Hierdurch wird gewährleistet, dass RSpec die Testdateien den dazugehörigen Ruby Dateien korrekt zuordnen kann.

Lehrkörper können die Fehlermeldungen, die den Studenten auf der JACK Plattform für bestimmte Aufgaben angezeigt werden, selbst bestimmen. Innerhalb von RSpec wird mit dem Tag „it" angegeben, was eine Methode eigentlich bewirken soll. Die Angaben, die nach dem Tag „it" getroffen werden, werden auf der JACK Plattform als Fehlermeldungen ausgegeben. In der Abbildung 11 wird beispielsweise bei der Methode minimum(array) die Fehlermeldung „Error: Operation minimum(array) should return minimul value in array" ausgegeben, sobald ein Fehler gefunden wurde. Zusätzlich wird noch automatisch ausgegeben, welches Ergebnis eigentlich erwartet wurde und welches Ergebnis letztendlich herausgekommen ist.

DemoProjekt_spec.rb

```
require_relative 'DemoProjekt'

# values to check
testArray1 = [6, 15, -23, 1, -9, 4, 123, 44, -96, -1]
testarray2 = [[111, -23, 654, 234, 4], [17, -34, -50, 11, 12], [678, -238, -22, -213, -1001],
              [-34, -222, -184, 273, -9]]

describe DemoProjekt do
  let!(:testObject) {DemoProjekt.new}
  it "Error: Operation minimum(array) should return minimum value in array" do
    expect(testobject.minimum(testArray1)).to eq(-96)
  end
  it "Error: Operation kleinsterIndex(array) should return index of array with lowest sum" do
    expect(testObject.kleinsterIndex(testArray2)).to eq(2)
  end
end
```

Abbildung 11: Ruby Testdatei im RSpec Format

4.2.2 Vorgehensweise des dynamischen Ruby Checkers

Übertragung der hochgeladenen Dateien

Genau wie bei der statischen Ruby Checker Komponente, wird auch bei der dynamischen Ruby Checker Komponente die Methode doCheck ausgeführt, sobald Studenten ihre Lösungen einreichen. Diese Methode erfüllt anschließend verschiedene Aufgaben. Die erste Aufgabe besteht darin, die hochgeladenen Dateien herunterzuladen und diese lokal zwischenzuspeichern. Bei diesen zwischengespeicherten Dateien handelt es sich um die eingereichten Lösungen der Studenten als auch um die von den Lehrkörpern hochgeladenen Ruby Testdateien und die dazugehörige Grading Point List.

Ausführung von RSpec

Nachdem die Dateien heruntergeladen und lokal zwischengespeichert wurden, wird RSpec ausgeführt. RSpec muss dementsprechend auf dem System, auf dem auch JACK läuft vorinstalliert sein. Der dynamische Ruby Checker startet RSpec mittels Kommandozeilenbefehl, der wie folgt definiert ist: rspec --format j --out workDir/Dateiname.json workDir. Der Begriff workDir ist hierbei ein Platzhalter und definiert das Verzeichnis, in dem RSpec die zu prüfenden Dateien sucht.

Speichern der Ergebnisse in JSON-Datei

RSpec speichert die Ergebnisse der Prüfung im workDir in einer JSON-Datei, dessen Dateiname aus 20 Zeichen besteht und der sich aus den Zeichen a – z als auch 0 – 9 zusammensetzt. Der Dateiname wird zufällig erzeugt, um zu verhindern, dass Studenten eine Lösung einreichen, dessen Quelltext eine eigene JSON-Datei erzeugt. Würde stattdessen ein fester Dateiname verwendet werden und Studenten würden diesen Dateinamen herausfinden, könnten Studenten einen Quelltext hochladen, der selbst eine solche JSON-Datei mit entsprechenden Dateinamen erzeugt. Diese JSON-Datei könnte dann so gestaltet sein, dass sie keinerlei Fehler enthält und somit würde jede eingereichte Lösung immer als korrekte Lösung erkannt werden, selbst wenn die eingereichte Lösung die Aufgabenstellungen nicht erfüllt.

Einlesen der Ergebnisse aus JSON-Datei

Genau wie die statische Checker Komponente, wartet auch die dynamische Checker Komponente solange, bis RSpec eine entsprechende JSON-Datei mit den Ergebnissen erstellt hat. Danach liest die Komponente die relevanten Informationen aus der erstellten JSON-Datei ein und generiert den Output für die Fehlermeldungen, vorausgesetzt, dass auch Fehler gefunden wurden.

Benotung durchführen

Die Herangehensweise der dynamischen Checker Komponente zur Benotung der eingereichten Lösungen ist eine andere, als das bei der statischen Checker Komponente der Fall ist. In der durch RSpec erzeugten JSON-Datei werden Methoden, die korrekte Ausgaben liefern, mittels „passed" bezeichnet und Methoden, die falsche Ausgaben generieren, mittels „failed" bezeichnet. Diese Bezeichnungen (passed / failed) entsprechen also den Ergebnissen der eingereichten Lösungen und werden durch die dynamische Checker Komponente eingelesen und in einer Liste gespeichert. Anschließend liest die Komponente, die von den Lehrkörpern erstellte Grading Point List im JSON-Format ein, entnimmt dieser Datei die angegebenen Punkte für die jeweiligen Aufgabenstellungen und speichert diese Punkte ebenfalls in einer Liste. Die Komponente geht nun die Liste mit den Ergebnissen durch und überprüft, ob das jeweilige Ergebnis „passed" lautet. Sollte das der Fall sein, werden die Punkte, die sich in der Punkteliste an der gleichen Stelle des Durchlaufs befinden einem Wert hinzuaddiert, der bei 0 beginnt. Im Gegensatz zur statischen Ruby Checker Komponente, die bei einer Gesamtpunktzahl von 100 startet und Fehlerpunkte für falsche Lösungen von dieser Gesamtpunktzahl subtrahiert, wird bei der dynamischen Ruby Checker Komponente bei einem Wert von 0 begonnen und die Punkte für korrekte Aufgaben werden diesem Wert hinzuaddiert, bis alle Aufgaben durchlaufen und somit die erreichte Gesamtpunktzahl erreicht wurde.

Vorbereitung der zu versendenden Resultate

Genau wie auch bei der statischen Ruby Checker Komponente, gibt die Methode doCheck, die alle bisher beschriebenen Vorgänge ausführt, ein Objekt des Typs BackendResult zurück. Dieses Objekt besteht wie bereits erwähnt aus einer Liste von Strings mit den generierten Fehlermeldungen, zwei String Objekten, die beschreibende Informationen enthalten, die auf JACK ausgegeben werden und den erzielten Punkten. Dieses Objekt wird nun erzeugt und am Ende der Methode doCheck zurückgegeben.

Löschung temporärer Dateien

Die von der dynamischen Ruby Checker Komponente zwischengespeicherten Dateien werden nun gelöscht und die Listen sowie die erreichte Punktzahl auf den Ausgangszustand zurückgesetzt.

Versenden der Resultate

Am Ende der Methode doCheck wird das zuvor erstellte BackendResult Objekt zurückgegeben und vom JACK Backend verarbeitet. Das Backend zeigt dann die gefundenen Fehlermeldungen sowie die beschreibenden Informationen und die erzielten Punkte auf der JACK Plattform an.

5 Tests für die entwickelten Prüfmodule

Um die Implementierung des Ruby Prüfmoduls unter realen Bedingungen testen zu können, wurde ein kleiner Ruby Einstiegskurs mit verschiedenen Aufgaben verfasst. Die Aufgaben behandeln unterschiedliche Themengebiete, wie beispielsweise einfache Ausgaben auf der Konsole, String Manipulationen, mathematische Funktionen als auch der Umgang mit Vergleichsoperatoren und Arrays. Jede Aufgabe beinhaltet eine Aufgabenbeschreibung in Form einer PDF-Datei und einer Quelltextvorlage, die von den Studenten heruntergeladen, bearbeitet und anschließend wieder hochgeladen werden muss. Weiterhin beinhaltet jede Aufgabe ein Testdokument, das in einem RSpec kompatiblen Format verfasst wurde und eine Grading Point List in Form einer JSON-Datei, um die Punktevergabe für die Teilaufgaben festzulegen. Die beiden zuletzt erwähnten Dateien sind für den dynamischen Ruby Checker erforderlich, um den dynamischen Test durchführen und die Benotung abschließen zu können.

5.1 Testvorbereitung

In dieser schriftlichen Ausarbeitung wird lediglich auf eine Aufgabe des Ruby Einstiegskurs eingegangen, weil dieser Teil der Ausarbeitung sonst den Rahmen sprengen würde.

5.1.1 Aufgabenbeschreibung

Bei der hier erwähnten Aufgabe müssen Studenten verschiedene mathematische Funktionen als Methoden implementieren. Bei den zu implementierenden Methoden handelt es sich um Addition, Subtraktion, Multiplikation, Division, Modulo, Potenzieren. Weitere Methoden, die implementiert werden müssen, sind Flächen- und Umfangberechnungen von Rechtecken, Dreiecken und Kreisen. Auch Volumen- und Oberflächenberechnungen von Würfeln, Quadern und Pyramiden gehören dazu. Zuletzt sind noch drei weitere mathematische Funktionen als Aufgaben gegeben, die von den Studenten implementiert werden müssen, dabei handelt es sich um die Fakultätsberechnung, Fibonacci-Formel und eine Methode, die ermitteln soll, ob eine als Parameter übergebene Zahl eine Primzahl ist. Die hier erwähnten Aufgaben finden sich in der Aufgabenbeschreibung mit dem Titel „Mathe.pdf“ wieder, die von den Studenten heruntergeladen werden muss.

5.1.2 Quelltextvorlage

Die zu implementierenden Methoden finden sich in einer Quelltextvorlage namens „calculator.rb" wieder. Die Methoden sind dort bereits definiert und die Studenten müssen diese Datei herunterladen und lediglich die Logik zu den dazugehörigen Methoden implementieren. Ein Ausschnitt dieser Datei ist in der Abbildung 12 zu sehen.

calculator.rb

```ruby
# Ruby Einstiegskurs fuer das E-Assessment-System JACK
# Aufgabe 3: Calculator
# Aufgabenstellung: Programmieren Sie eine Anwendung, die verschiedene
# mathematische Funktionen beinhaltet

class Calculator
  # Aufgabe 1) Addiere
  # Diese Methode soll a und b addieren und das Ergebnis zurueckgeben
  # Runden Sie auf 2 Stellen nach dem Komma
  def addiere(a,b)
    # Hier ergaenzen
  end

  # Aufgabe 2) Subtrahiere
  # Diese Methode soll b von a subtrahieren und das Ergebnis zurueckgeben
  # Runden Sie auf 2 Stellen nach dem Komma
  def subtrahiere(a,b)
    # Hier ergaenzen
  end
```

Abbildung 12: Quelltextvorlage

5.1.3 Testdokument im RSpec Format

Um den dynamischen Test mit der dynamischen Ruby Checker Komponente durchführen zu können, wurde ein Testdokument in einem RSpec kompatiblen Format verfasst. In Abbildung 13 ist ein Teil dieses Dokuments zu erkennen. Der Ausschnitt zeigt, wie die ersten beiden Methoden mit den Parametern 8 und 6 auf die Ergebnisse 14 und 2 geprüft werden.

calculator_spec.rb

```ruby
require_relative 'calculator'

describe "calculator.rb" do
  describe Calculator do
    let(:calculator) {Calculator.new}

    it "Die Methode addiere soll Parameter a und b miteinander addieren und das Ergebnis
        zurueckgeben" do
      expect(calculator.addiere(8,6)).to eq(14)
    end

    it "Die Methode subtrahiere soll Parameter b von a subtrahieren und das Ergebnis
        zurueckgeben" do
      expect(calculator.subtrahiere(8,6)).to eq(2)
    end
```

Abbildung 13: Testdokument im RSpec Format

5.1.4 Grading Point List

Um die Benotung des dynamischen Tests durchzuführen, wurde eine Punkteliste im JSON-Format verfasst. Ein Teil dieser Punkteliste ist in Abbildung 14 dargestellt. Für die ersten beiden Aufgaben werden jeweils 6 Punkte vergeben, sofern diese korrekt gelöst wurden.

calculator_point_list.json

```json
{
  "lesson":
    {
      "name": "Ruby Einstiegskurs: Mathe",
      "author": "Patrick Czarnetzki",
      "exercise":
      [
        {
          "id": "0",
          "name": "addiere",
          "points": "6"
        },
        {
          "id": "1",
          "name": "subtrahiere",
          "points": "6"
        },
```

Abbildung 14: Grading Point List

5.1.5 Einstellungen der dynamischen Checker Komponente

Damit die dynamische Ruby Checker Komponente erfährt, welche Dateien überhaupt geprüft werden sollen und mit welchen Testdokumenten und Punktelisten die Prüfung und Benotung erfolgen soll, müssen innerhalb von JACK bei der Aufgabenerstellung die jeweiligen Dateien in den Optionen der Komponente angegeben werden. In der Abbildung 15 ist zu erkennen, wie diese Einstellungen vorgenommen wurden.

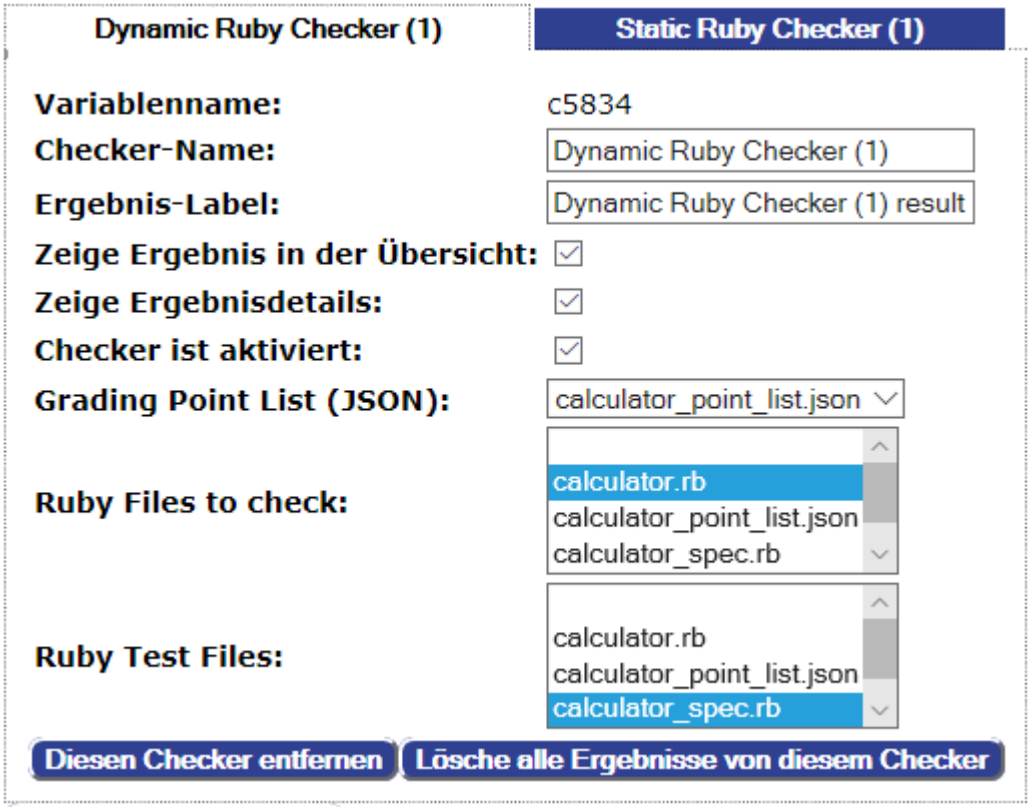

Abbildung 15: Dynamische Ruby Checker Komponente

5.1.6 Einstellungen der statischen Checker Komponente

In Abbildung 16 ist zu erkennen, welche Optionen für die statische Ruby Checker Komponente vorgenommen wurden. Diese Komponente wurde so eingestellt, dass Verstöße gegen die Konventionen als auch Warnungen und Fehler berücksichtigt werden. Punktabzüge werden allerdings nur bei einem gefundenen Fehler vorgenommen, wird ein solcher Fehler gefunden, werden 100 Punkte abgezogen und die Lösung wäre dementsprechend nicht korrekt, weil die eingereichte Lösung nicht kompilierbar wäre.

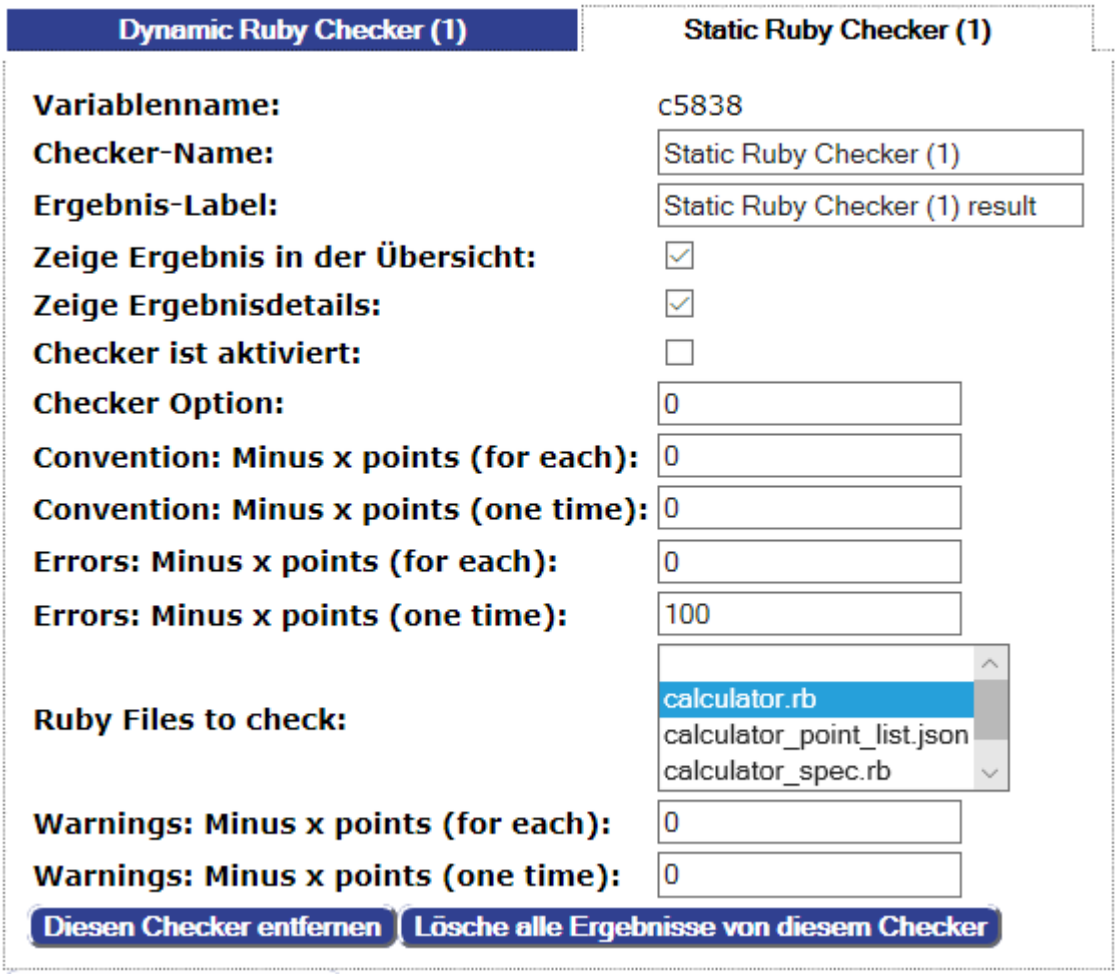

Abbildung 16: Statische Ruby Checker Komponente

5.2 Testergebnisse

5.2.1 Ergebnisse der dynamischen Ruby Checker Komponente

Um zu überprüfen, ob die dynamische Ruby Checker Komponente Fehler erkennt, Ausgaben erzeugt und die Benotung korrekt durchführt, wurde eine fehlerhafte Lösung erstellt und eingereicht. Die Fehler wurden bei den Logiken für die Methoden kreis_flaeche, kreis_umfang, subtrahiere, rechteck_flaeche und dreieck_umfang eingebaut. Alle diese Methoden berechnen nicht die korrekten Ergebnisse und sind dementsprechend fehlerhaft. In Abbildung 17 ist zu erkennen, dass 70 Punkte vergeben wurden. Dieses Ergebnis ergibt sich aus den 5 gefunden Fehlern, für die jeweils 6 Punkte und somit 30 Punkte nicht vergeben wurden. Außerdem ist in dieser Abbildung zu erkennen, welche Methoden falsch implementiert wurden, was die Methoden eigentlich hätten realisieren sollen, welche Ausgaben erwartet wurden und welche falschen Ausgaben letztendlich generiert wurden.

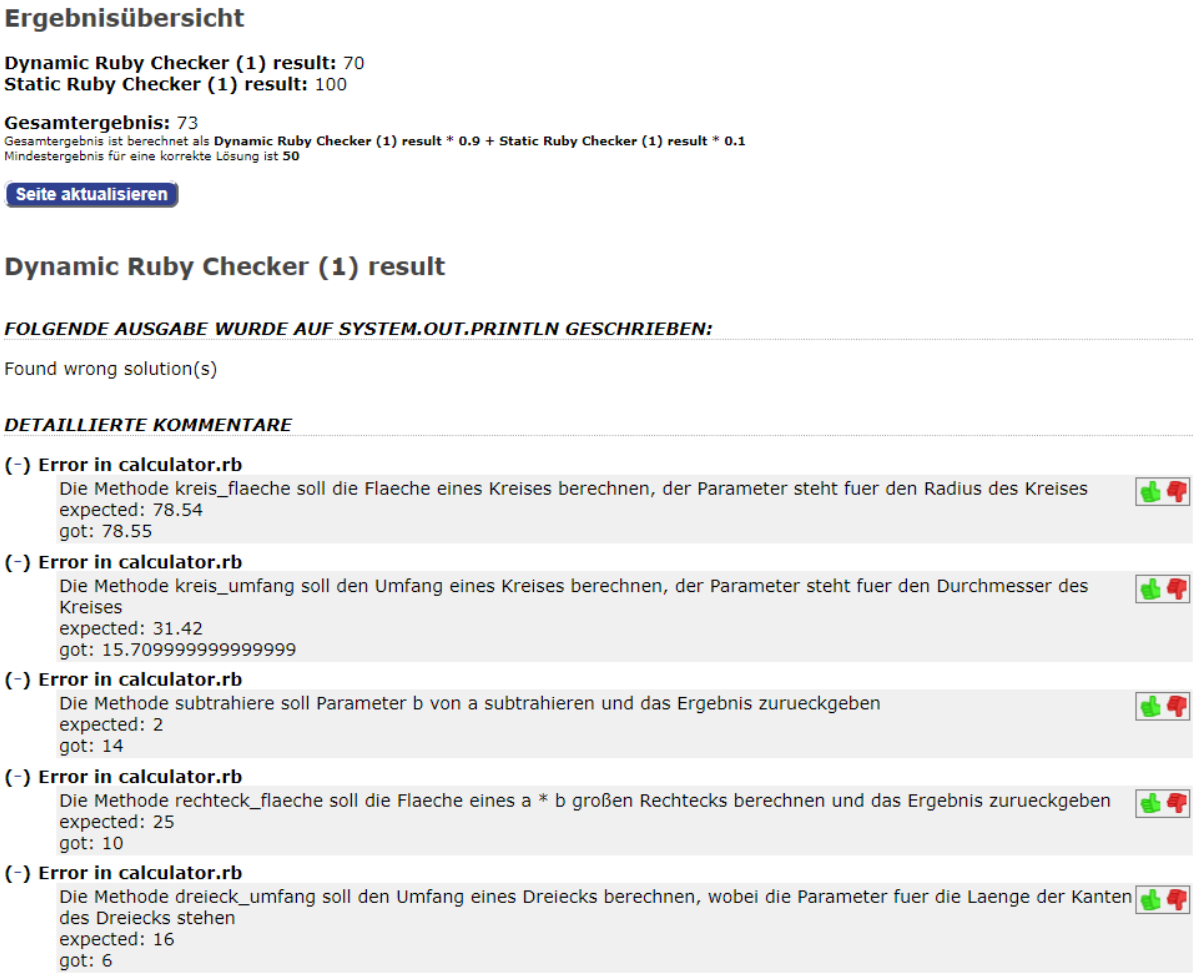

Abbildung 17: Ergebnisse der dynamischen Ruby Checker Komponente

5.2.2 Ergebnisse der statischen Ruby Checker Komponente

Die Ergebnisse der statischen Ruby Checker Komponente beziehen sich auf die gleiche fehlerhafte Lösung, die auch zuvor für den Test der dynamischen Ruby Checker Komponente verwendet wurde. In Abbildung 18 ist zu erkennen, dass Verstöße gegen die Konventionen erkannt wurden und dass Ausgaben erzeugt wurden, die beschreiben, um welche Verstöße es sich genau handelt. In Abbildung 17 ist zu erkennen, dass für diese Lösung trotzdem 100 Punkte vergeben wurden, was an den Optionen liegt, die für die statische Ruby Checker Komponente verwendet wurden. Die Komponente wurde so eingestellt, dass zwar Verstöße gegen die Konventionen als auch Warnungen und Fehlermeldungen erkannt und ausgegeben werden, allerdings werden nur Punkte abgezogen, sofern es sich um einen Fehler handelt.

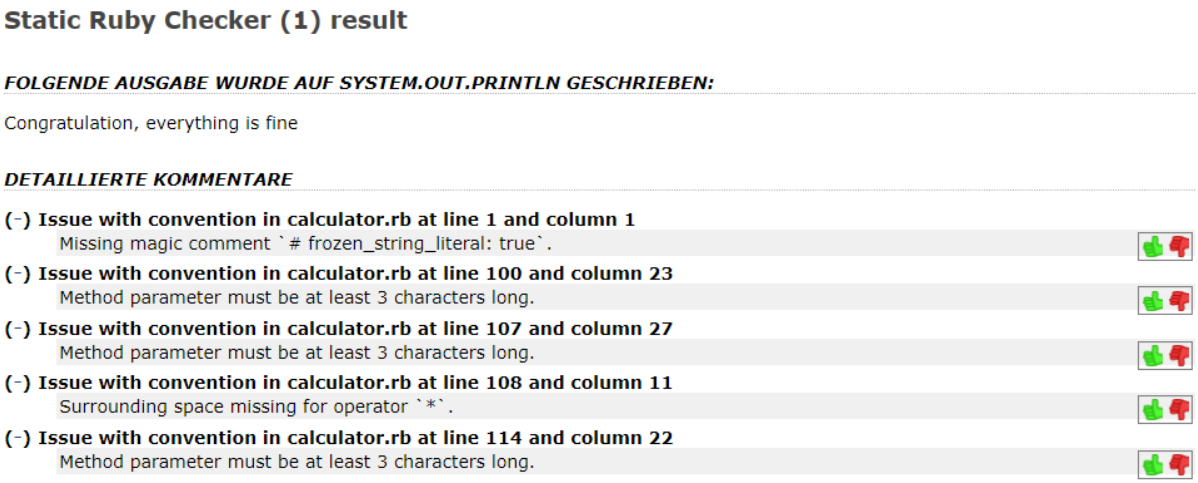

Abbildung 18: Ergebnisse der statischen Ruby Checker Komponente

5.2.3 Berechnung des Gesamtergebnisses

Das Gesamtergebnis ist in Abbildung 17 zu erkennen und ergibt sich aus der eingestellten Gewichtung für die jeweiligen Checker Komponenten. In Abbildung 19 ist zu erkennen, wie die Gewichtung eingestellt wurde. #{c5976} ist eine ID, die der dynamischen Checker Komponente zugewiesen wurde. Das Ergebnis dieser Komponente wird mit 0,9 multipliziert. #{c5980} ist eine ID, die der statischen Checker Komponente zugewiesen wurde. Das Ergebnis dieser Komponente wird mit 0,1 multipliziert. Beide Ergebnisse werden anschließend miteinander addiert, wodurch sich das Gesamtergebnis mit 73 Punkten ergibt. Die dynamische Checker Komponente hat dementsprechend eine Gewichtung von 90 % und die statische Checker Komponente eine Gewichtung von 10 %.

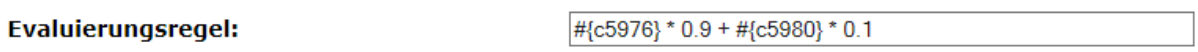

Abbildung 19: Berechnung des Gesamtergebnisses

6 Ausblick

Die in diesem Bachelor-Projekt entwickelte Ruby Prüfkomponente, die sich aus zwei separaten Checker-Komponenten zusammensetzt, legt das Fundament zur Überprüfung von Ruby Quelltexten und zur Generierung von Feedback, mit dem Studenten ihre Ruby Kenntnisse verbessern können. Es existieren allerdings noch einige Aspekte, die von dieser Prüfkomponente nicht berücksichtigt werden. Zu diesen Aspekten zählen beispielsweise die Performanz, Benutzerfreundlichkeit, Wartbarkeit, Wiederverwendbarkeit, Portierbarkeit und Interoperabilität des eingereichten Quelltexts. Es könnte sinnvoll sein, die Prüfkomponente hinsichtlich dieser Aspekte zu erweitern, sodass die Prüfkomponente auch diese Eigenschaften des Quelltexts analysiert und bewertet.

Eine graphbasierte Analyse des Quelltexts könnte eine weitere sinnvolle Erweiterung der Ruby Prüfkomponente darstellen. Eine graphbasierte Analyse würde es ermöglichen, die Aufgabenstellungen dynamischer zu gestalten. Dadurch könnten für die Aufgabenstellungen gewisse Lösungsansätze vorausgesetzt und überprüft werden. Ein Beispiel hierfür wäre es, für die Lösung einer Aufgabe den Einsatz von Rekursion vorauszusetzen. Durch eine graphbasierte Analyse könnte anschließend überprüft werden, ob diese Aufgabe auch tatsächlich mittels Rekursion gelöst wurde. Ein solcher Ansatz wurde bereits für die Programmiersprache Java im GReQLJavaChecker implementiert. (Vgl. [St16], S. 41).

Die Ausgabe der Ruby Prüfkomponente könnte ebenfalls erweitert werden. Anstatt lediglich Feedback in schriftlicher Form auszugeben, könnte die Prüfkomponente so erweitert werden, dass sie zusätzliche UML-Diagramme ausgibt. Diese UML-Diagramme sollten sich auf den Quelltext der eingereichten Lösungen beziehen und beispielsweise das Zusammenwirken verschiedener Klassen anhand eines Klassendiagramms visuell darstellen. Natürlich könnten auch weitere UML-Diagramme aus den eingereichten Lösungen generiert werden, wie beispielsweise Komponentendiagramme, Aktivitätsdiagramme oder Sequenzdiagramme.

Die Möglichkeiten, die in diesem Bachelor-Projekt entwickelte Ruby Prüfkomponente zu erweitern, sind also noch lange nicht erschöpft. Die Prüfkomponente ist allerdings ausreichend, um dynamische und statische Tests für Ruby Quelltexte durchzuführen, was auch den eigentlichen Schwerpunkt dieses Projekts darstellte.

7 Fazit

In diesem Bachelor-Projekt konnte das E-Assessment-System JACK mit einer zusätzlichen Prüfkomponente für die Programmiersprache Ruby erweitert werden. Zunächst wurden Architektur und Erweiterungsmöglichkeiten von JACK näher betrachtet. Dabei stellte sich heraus, dass JACK im Wesentlichen aus 2 Bestandteilen besteht, dem Core- und Worker Server. Außerdem stellte sich heraus, dass sich JACK durch die Implementierung einer weiteren Checker Komponente leicht erweitern ließ.

Anschließend wurden die Grundlagen zu der Programmiersprache Ruby und den verschiedenen Testverfahren ermittelt. Hierbei stellte sich heraus, dass sowohl statische als auch dynamische Testverfahren von der Prüfkomponente abgedeckt werden sollten. Es folgte eine Analyse der verschiedenen Werkzeuge für die jeweiligen Testverfahren, bevor sich mit der Implementierung der Prüfkomponente befasst wurde. Um sowohl dynamische als auch statische Testverfahren abzudecken, wurden zwei separate Prüfkomponenten entwickelt, die beide verschiedene externe Werkzeuge nutzen und eng mit dem JACK Backend zusammenarbeiten. Die Nutzung und Vorgehensweisen der entwickelten Prüfkomponenten wurden anschließend näher erläutert.

Zuletzt konnten die Prüfkomponenten erfolgreich getestet werden. Hierzu wurde ein kleiner Ruby Einstiegskurs mit Aufgaben zu verschiedenen Themengebieten genutzt, der zuvor verfasst wurde. Das Ergebnis einer dieser Testaufgaben wurde anschließend dokumentiert und näher erläutert. Letztendlich lässt sich festhalten, dass in diesem Projekt eine Prüfkomponente für das E-Assessment-System JACK entwickelt und dessen Entwicklung dokumentiert wurde. Die Prüfkomponente ist dazu in der Lage, Ruby Quelltexte anhand bestimmter Kriterien zu prüfen und zu benoten.

Literatur

[Ed11] Michael Edgar: Beschreibung von Laser auf Github. https://github.com/michaeledgar/laser/ Abgerufen am 04. November 2019.

[En15] Eduard Paul Enolu: Programming Languages Popularity and Implications to Testing Programmable Logic Controllers. PeerJ PrePrints Tech. Rep., 2015.

[FlMa08] David Flanagan, Yukihiro Matsumoto, übersetzt durch Sascha Kersken, Thomas Demmig: Die Programmiersprache Ruby. O'Reilly Verlag GmbH & Co. KG, 2008.

[Ha12] Zenon Harley, Eric R. Harley: A Wizard For E-Learning Computer Programming. International Conference on E-Learning and E-Technologies in Education (ICEEE), 2012.

[Ma12] Myron Marston: Minitest and Rspec. Answer from Myron Marston (Developer of RSpec) on Stackoverflow in September 2012. Quelle: https://stackoverflow.com/questions/12470601/minitest-and-rspec Abgerufen am: 11. November 2019.

[Ra16] Katharina Rahf: Überprüfung von Stilrichtlinien für deklarative Programme. Masterarbeit. Christian-Albrechts-Universität zu Kiel, Juli 2016.

[Sc15] Vincent Sceraert: A Tool for Offering Immediate Feedback on Violations of Structural Regularities in Ruby Source Code. Masterarbeit. Universite Cathologque De Louvain, 2015.

[St16] Michael Striewe: An Architecture for Modular Grading and Feedback Generation for Complex Exercises. Science of Computer Programming 129. Special issue on eLearning Software Architectures, 2016.

[Tă15] Alexandru Tăbuscă: Learning a Programming Language for Today. Journal of Information Systems & Operations Management, Vol. 9, No 1, 2015.

[Ta16] Mani Tadayon: RSpec Essentials – Develop testable, modular, and maintainable Ruby software for the real world using RSpec. Packt Publishing, 2016.

[TFH09] Dave Thomas, Chad Fowler, Andy Hunt: Programming Ruby 1.9 – The Pragmatic Programmers' Guide. The Pragmatic Programmers, LLC, 2009.

[Wa14] Eugene Wang: What's That (Code) Smell? How to Monitor Code Quality in Ruby and Rails. Eugenius Blog, 2014. http://eewang.github.io/blog/2014/01/18/how-to-smell-check-your-code-quality-in-ruby-and-rails/ Abgerufen am: 11. November 2019.

[Wi19] Frank Witte: Testmanagement und Softwaretest – Theoretische Grundlagen und praktische Umsetzung. 2. Erweiterte Auflage, Springer Fachmedien Wiesbaden GmbH, 2019.