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

Bachelor-Arbeit

Erweiterung eines Ruby Prüfmoduls

zur Messung qualitativer Softwareaspekte

Patrick Czarnetzki
3042742

Herne, 28.07.2020

Betreuung:	Dr. Michael Striewe
Erstgutachter:	Prof. Dr. Michael Goedicke
Zweitgutachter:	Prof. Dr. Volker Gruhn
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 28.07.2020

Zusammenfassung

Seit mehreren Jahren nutzt die Universität Duisburg-Essen ein E-Assessment-System mit dem Namen JACK. Mittels JACK können verschiedene Aufgabentypen generiert, automatisch korrigiert und mit Feedback versehen werden. JACK wurde über die Jahre stetig weiterentwickelt und unterstützt mittlerweile zahlreiche Programmiersprachen. Für mein Bachelor-Projekt wurde eine zusätzliche Prüfkomponente für das System entwickelt, mit der sich statische und dynamische Testverfahren für die Programmiersprache Ruby durchführen lassen. Die Komponente fokussierte sich allerdings darauf, ob der Quellcode kompilierbar und die Funktionalität korrekt implementiert wurde. Qualitative Softwareaspekte blieben bei dieser Analyse aber weitgehend unberücksichtigt. Das Ziel dieser Bachelor-Arbeit liegt in der Erweiterung dieser Ruby Prüfkomponente, dahingehend, dass sie auch qualitative Softwareaspekte bei ihrer Analyse, Bewertung und Erzeugung von Feedback berücksichtigt. Es werden zunächst die Grundlagen von Softwarequalität erläutert, bevor darauf eingegangen wird, wie sich dieser abstrakte Begriff durch verschiedene Maße fassen lässt. Anschließend wird die bisherige Ruby Prüfkomponente und ihre Vorgehensweise genauer erläutert, bevor sich mit der Erweiterung dergleichen befasst wird, was auch den Schwerpunkt dieser Arbeit darstellt. Zuletzt wird erklärt, wie die Erweiterung auf korrekte Funktionalität geprüft und inwiefern sich die Verfahren auf andere Projekte und Programmiersprachen übertragen lassen.

Abstract

For several years the University of Duisburg-Essen has been using an e-assessment-system called JACK. Various types of tasks can be generated, automatically corrected and provided with feedback by JACK. This system has been constantly developed over years and supports now numerous programming languages. An additional test component for the system has been developed for my bachelor project which can be used to perform static and dynamic test procedures for the Ruby programming language. However, the component focused on whether the source code was compilable and functionality was implemented correctly. So Qualitative software aspects were largely disregarded in this analysis. Goal of this bachelor thesis was to extend the actually Ruby test component to take into account qualitative software aspects in its analysis, evaluation and generation of feedback. The basics of software quality are explained first before discussing how softwarequality can be grasped by different metric values. The actually Ruby test component and its approach are explained then in more detail before extension of the same is discussed which is also the focus of this work. Finally it explains how the extension is checked for correct functionality and how the procedures can be transferred to other projects and programming languages.

Abbildungsverzeichnis

Inhaltsverzeichnis

1 Einleitung

Es ist allgemein bekannt, dass sich die Studierendenzahlen an deutschen Hochschulen stetig erhöhen, was dazu führt, dass eine individuelle Betreuung einzelner Studierender unmöglich wird. Um diesem Problem entgegenzuwirken, setzen Hochschulen immer häufiger auf E-Assessment-Systeme, mit denen Aufgaben generiert, automatisch korrigiert und mit Feedback versehen werden können. Diese Automatisierung reduziert den Arbeitsaufwand und unterstützt Studierende beim Lernprozess, indem ihnen zeitnahes Feedback bereitgestellt wird. (Vgl. [ScHo14], S. 11)

Auch die Universität Duisburg-Essen nutzt ein solches E-Assessment-System mit dem Namen JACK. Während meines Bachelor-Projekts wurde eine Ruby Prüfkomponente für dieses System entwickelt, mit der sowohl statische als auch dynamische Testverfahren für die Programmiersprache Ruby durchgeführt werden können. Wie die meisten programmiertechnischen Prüfkomponenten fokussiert sich auch die Ruby Prüfkomponente bei ihrer Analyse auf die Kompilierbarkeit und Funktionalität eines Quelltexts und lässt qualitative Softwareaspekte weitgehend unberücksichtigt. Das hat zur Folge, dass Studenten ihre eigene Codequalität nicht hinterfragen, obwohl dieser Aspekt bei realen Projekten von großer Bedeutung ist. Qualitative Softwareaspekte bleiben von den meisten Komponenten unberücksichtigt, weil es sich als schwierig erweist, Qualität in Einheiten zu fassen, um diese messbar und vergleichbar zu machen. (Vgl. [ArSeFi16], S. 1 f.)

Diese Bachelor-Arbeit setzt genau an diesen Punkt an und versucht Verfahren zu finden, mit denen dieser abstrakte Begriff fassbar und messbar gemacht werden kann. Insbesondere wird innerhalb dieser Arbeit versucht, diese Verfahren zu automatisieren und in die bestehende Ruby Prüfkomponente zu implementieren. Die Komponente wird somit dahingehend erweitert, dass sie neben Kompilierbarkeit und Funktionalität auch qualitative Softwareaspekte bei ihrer Analyse berücksichtigen kann. Zuletzt wird sich mit der Frage befasst, inwiefern sich die erarbeiteten Methoden und Erkenntnisse auf andere Programmiersprachen übertragen lassen.

2 Softwarequalität

Um Softwarequalität messbar zu machen, muss zunächst geklärt werden, was Softwarequalität überhaupt ist. Softwarequalität spielt eine wichtige Rolle für die Kundenzufriedenheit und den Erfolg von Softwareprodukten. Der Begriff ist allerdings weit gefächert und umfasst eine Vielzahl verschiedener Qualitätseigenschaften und Verfahren, mit denen sich diese Eigenschaften überprüfen lassen. Hierbei gilt es zu berücksichtigen, dass sich nicht alle diese Qualitätskriterien gleich gut für alle erdenklichen Softwareprodukte eignen. Qualitätskriterien, die für Softwareprodukt A eine große Relevanz besitzen, können für Softwareprodukt B völlig unbedeutend sein. Die gewünschte Qualität muss zunächst durch eine Qualitätszielbestimmung definiert werden. Erst durch die Qualitätszielbestimmung können geeignete Verfahren gewählt werden, um die festgelegten Qualitätskriterien zu erreichen oder diese einzuhalten. (Vgl. [Li09], S. 1 f.)

2.1 Qualität

Es gibt verschiedene Definitionen für den Begriff Qualität. Eine dieser Definitionen stammt aus der Norm DIN 8402 und lautet wie folgt: „Qualität ist die Gesamtheit von Eigenschaften und von Merkmalen eines Produkts oder einer Leistung, die sich auf die Erfüllung vordefinierter Anforderungen beziehen". In der Norm ISO 9126 wird Qualität als „Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen" beschrieben. (Vgl. [BrSc07], Folie 3)

2.2 Produktqualität und Prozessqualität

Softwarequalität lässt sich in Produktqualität und Prozessqualität unterteilen. Die Produktqualität bezieht sich auf das Maß an Qualität, das ein Produkt vorweist. Die Prozessqualität hingegen bezieht sich auf das Maß an Qualität, das ein Herstellungsprozess eines Produkts vorweist. Die Produktqualität hat einen Einfluss auf die Kundenzufriedenheit und die Prozessqualität wiederum einen Einfluss auf die Produktqualität. In der Regel führt eine gute Prozessqualität auch zu einer guten Produktqualität und somit zu Kundenzufriedenheit. (Vgl. [La10], S. 2)

2.3 Innere und äußere Qualität

Softwarequalität lässt sich auch zwischen innerer und äußerer Qualität differenzieren. Innere Qualität befasst sich mit Qualitätsmerkmalen, die mit der Entwicklung in Verbindung stehen, wie beispielsweise Wartbarkeit, Verständlichkeit und Wiederverwendbarkeit. Äußere Qualität befasst sich hingegen mit Qualitätsmerkmalen, die mit der Nutzung des Produkts und somit auch mit den Kunden zusammenhängen, wie beispielsweise Benutzerfreundlichkeit, Fehlertoleranz und Zuverlässigkeit. (Vgl. [La10], S. 2)

2.4 Qualitätsmodelle

Laut Liggesmeyer beschreiben Qualitätsmodelle „die kausalen Beziehungen zwischen nicht greifbaren Sichten auf Qualität und greifbaren Softwaremaßen". ([Li09], S. 16) Im Folgenden wird auf einige der populärsten Modelle für Softwarequalität eingegangen, in denen verschiedene Qualitätsmerkmale definiert und für diese Klassifizierungen vorgenommen wurden. Hierzu zählen die Qualitätsmodelle von McCall, Boehm und Dromey als auch das Qualitätsmodell FURPS und der internationale Standard ISO 9126. (Vgl. [AMA13], S. 1)

2.4.1 Softwarequalitätsmodell nach McCall

Im Jahr 1977 entwickelte Jim McCall für die US Air Force ein nach ihn benanntes Softwarequalitätsmodell. Es handelt sich um ein dreistufiges Modell mit 3 Hauptperspektiven, 11 Qualitätsfaktoren, 24 Qualitätskriterien und dazugehörigen Metriken, siehe Abbildung 1. Bei den Metriken handelt es sich vorwiegend um Einschätzungen, die mit Ja oder Nein zu beantworten sind. (Vgl. [La10], S. 5)

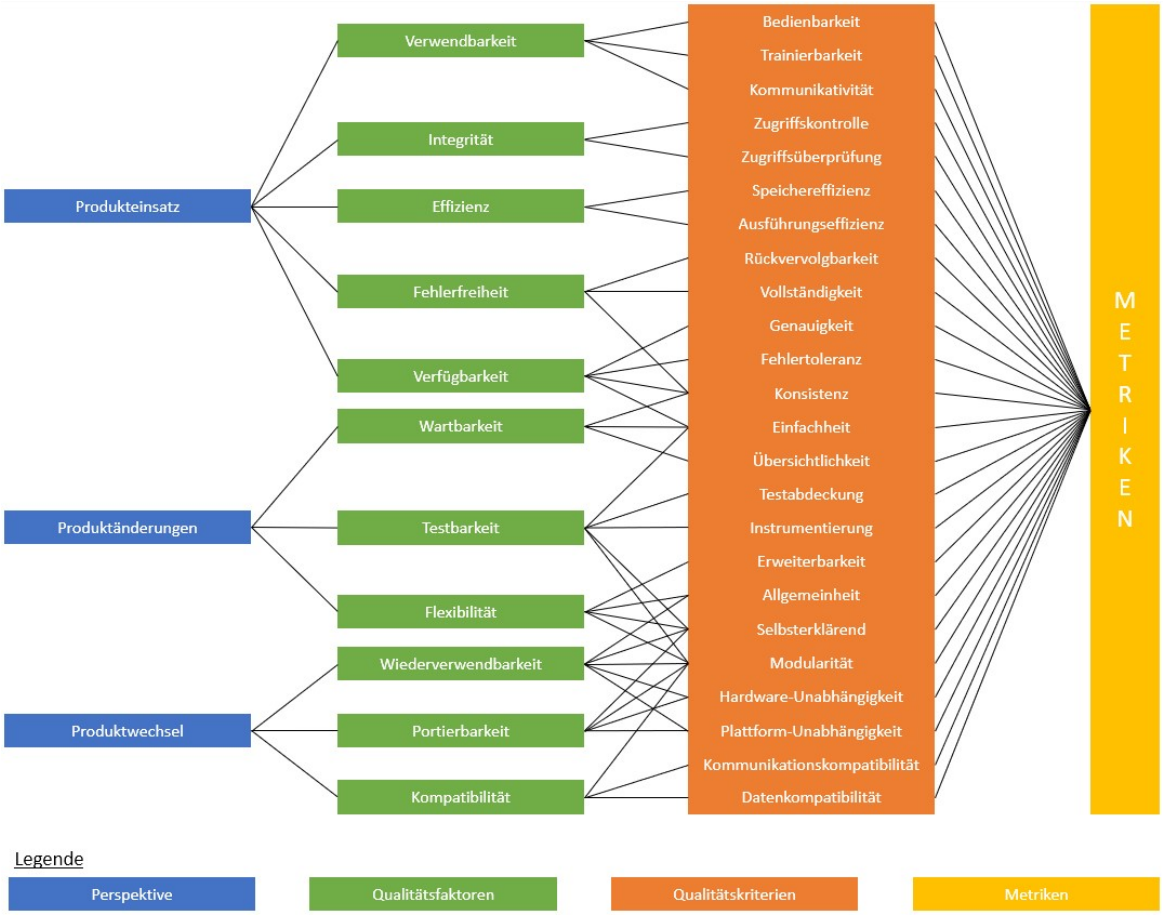

Abbildung 1 – Softwarequalitätsmodell nach Jim McCall.
Abbildung basiert auf Dietmar Winkler: Methoden der Qualitätssicherung – Vortragsreihe „Software Engineering for Everyday Business", Folie 7. Technische Universität Wien – Institut für Softwaretechnik und Interaktive Systeme.

2.4.2 Softwarequalitätsmodell nach Boehm

Barry W. Boehms Softwarequalitätsmodell ähnelt dem Qualitätsmodell von McCall. (Vgl. AMA13, S. 2) Genau wie bei dem Ansatz von McCall, handelt es sich um ein hierarchisches und zugleich dreistufiges Modell. Boehm geht von 3 Ebenen (Hoch, Mittel, Niedrig) aus, in denen sich verschiedene Qualitätsfaktoren als auch Qualitätskriterien befinden. (Vgl. [Al10], S. 3 f.)

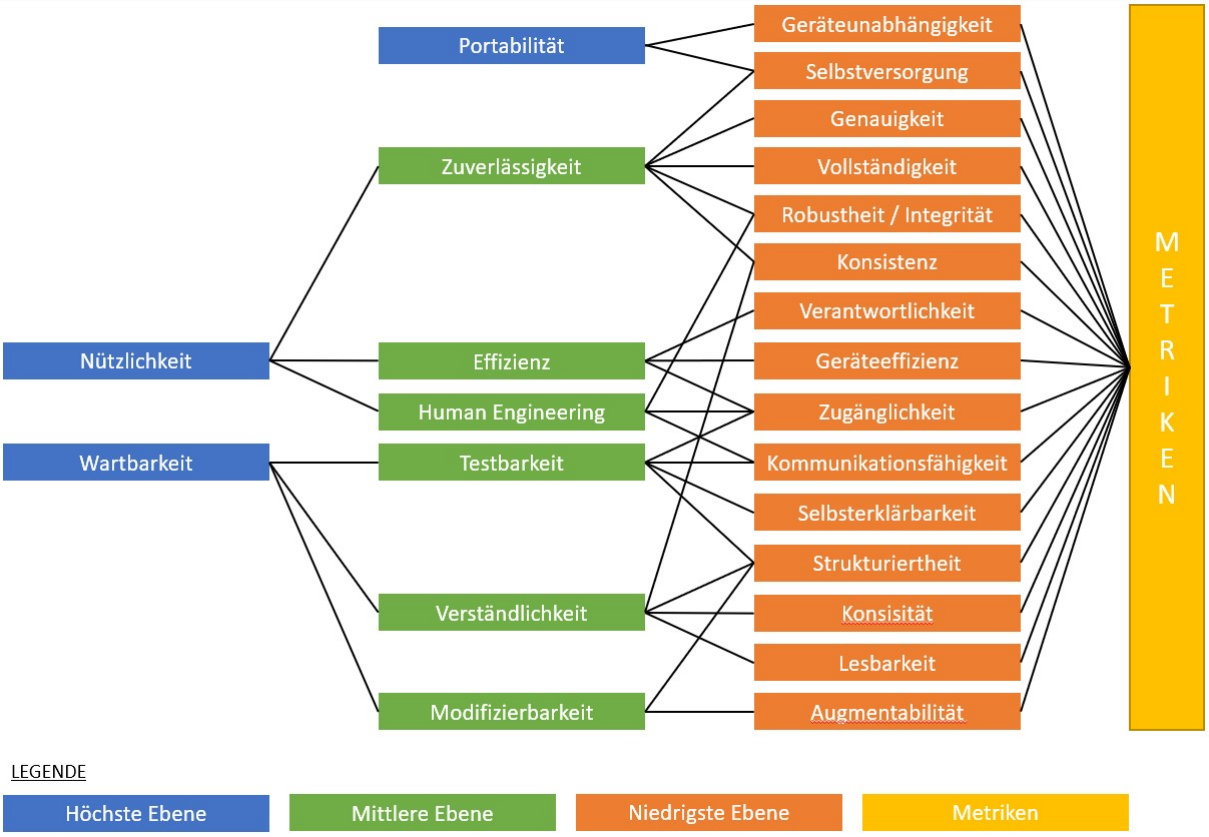

Abbildung 2 – Softwarequalitätsmodell nach Barry W. Boehm.
Abbildung basiert auf Marc-Alexis Côté, Witold Suryn, Elli Georgiadou: In search for a widely applicable and accepted software quality model for software quality engineering. Software Quality Journal 15:401-416, 2007.

2.4.3 Softwarequalitätsmodell FURPS

Im Jahr 1992 führte Robert Grady das Softwarequalitätsmodell FURPS ein. Der Name setzt sich aus den fünf Qualitätsmerkmalen Functionality (Funktionalität), Usability (Verwendbarkeit), Reliability (Zuverlässigkeit), Performance (Leistung) und Supportability (Unterstützbarkeit) zusammen. Grady unterteilt diese Qualitätsmerkmale in 2 Kategorien, funktionale und nicht funktionale Anforderungen. Funktionale Anforderungen sind durch Eingaben und erwartete Ausgaben definiert. Nichtfunktionale Anforderungen hingegen bestehen aus den verbleibenden 4 Qualitätsmerkmalen. (Vgl. [AMA13], S. 2)

Diesen 5 Kategorien von Qualitätsmerkmalen werden anschließend verschiedene Qualitätskriterien zugewiesen. Das Modell soll sicherstellen, dass Kunden ein Softwareprodukt erhalten, das zuverlässig und gut wartbar ist, sich vom Benutzer leicht bedienen lässt und über die gewünschte Funktionalität verfügt. (Vgl. [Gr06], S. 16)

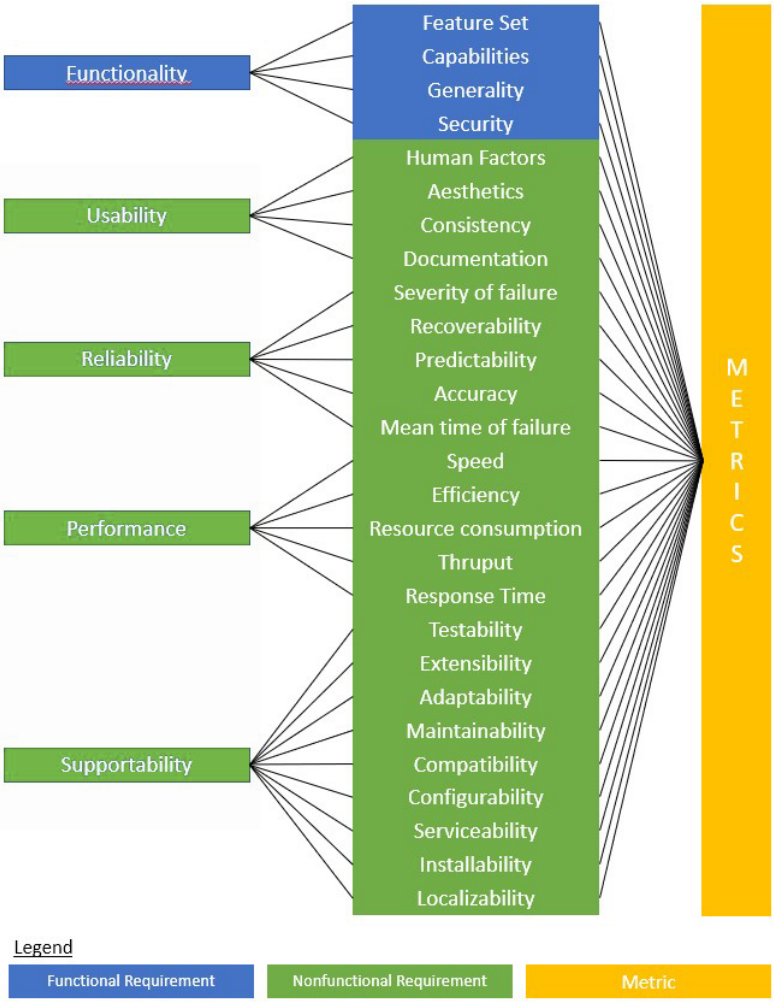

Abbildung 3 – Softwarequalitätsmodell FURPS
Abbildung basiert auf Prof. Dr. Hans-Gert Gräbe: Software-Qualitätsmanagement – Kernfach Angewandte Informatik. Institut für Informatik, Betriebliche Informationssysteme, Universität Leipzig, 2006.

2.4.4 Das ISO / IEC 9126 Model

1991 veröffentlichte die internationale Organisation für Normung einen Standard mit dem Titel „ISO / IEC 9126: Software Produktbewertung – Qualitätsmerkmale und Richtlinien für ihre Verwendung". Bei diesem Standard handelt es sich um ein Softwarequalitätsmodell, inklusive Richtlinien zur Messung der darin enthaltenen Merkmale. (Vgl. [Al10], S. 171)

Dieses Modell trennt die interne und externe Qualität von der Benutzungsqualität. Die interne und externe Qualität besteht bei diesem Modell aus sechs Qualitätsmerkmalen, während die Benutzungsqualität aus vier Qualitätsmerkmalen besteht. Die Mehrzahl dieser Qualitätsmerkmale wird in weitere Qualitätsteilmerkmale zerlegt. In einen separaten Bereich der ISO / IEC 9126 wurden für alle diese Qualitätsmerkmale und ihre Qualitätsteilmerkmale Maße definiert, mit denen sich diese Merkmale messen lassen. Das Modell wurde über die Zeit stetig weiterentwickelt und mittlerweile durch die Norm ISO / IEC 25000 ersetzt. (Vgl. [Li09], S. 16 f.)

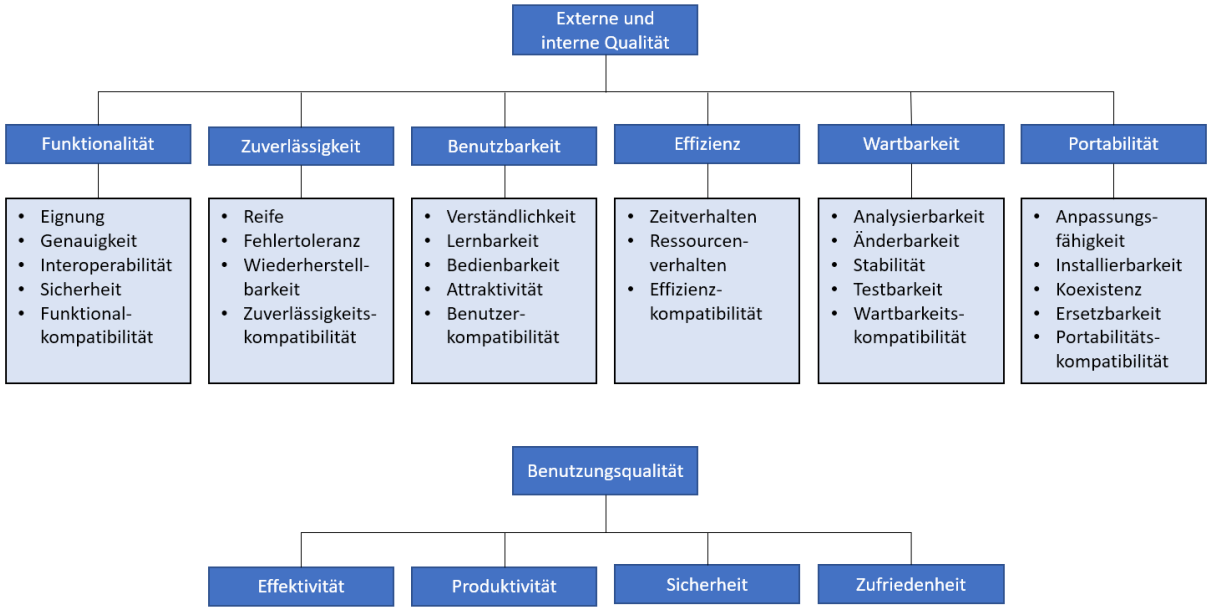

Abbildung 4 - ISO / IEC 9126 Model
Abbildung basiert auf Peter Liggesmeyer: Software-Qualität – Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag Heidelberg, 2. Auflage, 2009.

2.4.5 Softwarequalitätsmodell nach Dromey

Bei diesem Qualitätsmodell handelt es sich um ein produktbasiertes Modell, das 1995 von R. Geoff Dromey entwickelt wurde. Es besteht aus vier Produkteigenschaften, denen verschiedene Qualitätsmerkmale zugeordnet werden. Dromey versucht, mit diesem Modell zu berücksichtigen, dass sich die Qualitätsansprüche verschiedener Produkte voneinander unterscheiden. Aus diesem Grund wird mit diesem Modell versucht, den Modellierungsprozess so dynamisch wie möglich zu halten, um ihn auf möglichst viele verschiedene Systeme anwenden zu können. (Vgl. [Al10], S. 170)

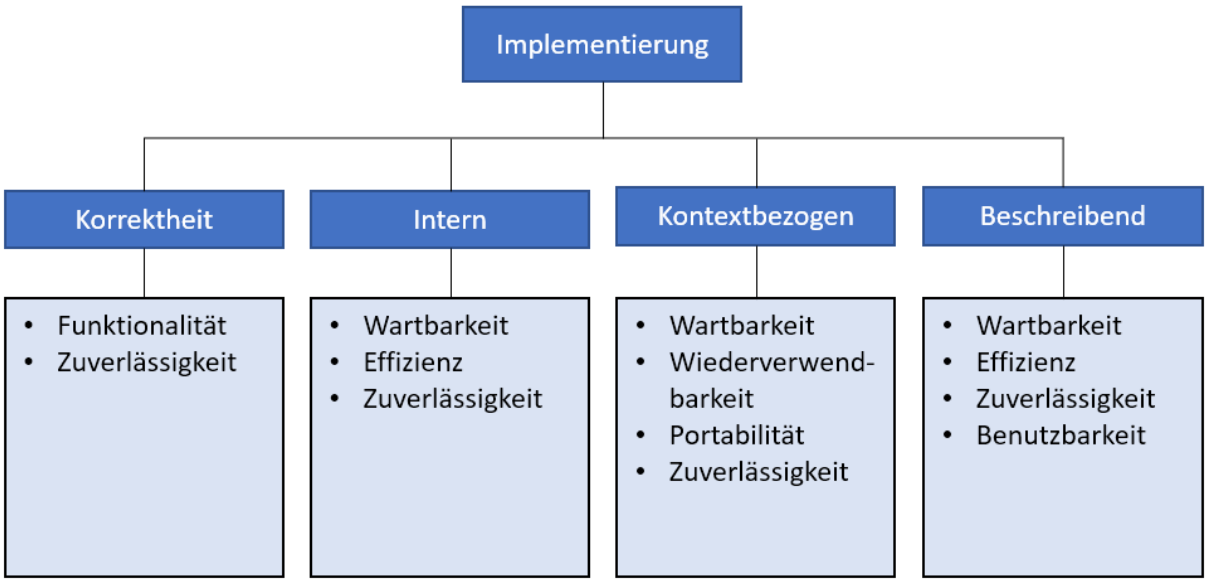

Abbildung 5 - Qualitätsmodell nach Dromey
Abbildung basiert auf Rafa E. Al-Qutaish: Quality Models in Software Engineering Literature: An Analytical and Comparative Study. Journal of American Science, 2010.

2.5 Qualitätsmerkmale

Qualitätsmerkmale sind ein Mittel, mit dem sich Qualität festlegen und beurteilen lässt. Laut der Norm DIN EN ISO 9000 / DIN EN ISO 9000 05 handelt es sich bei einem Qualitätsmerkmal um „ein inhärentes Merkmal eines Produkts, Prozesses oder Systems, das sich auf eine Anforderung bezieht". Eine Qualitätsanforderung wiederum ist eine Erwartung, die zuvor festgelegt wird und üblicherweise verpflichtend ist. Diese Merkmale lassen sich in weitere Teilmerkmale unterteilen. Es existieren verschiedene Vorschläge unterschiedlicher Autoren für Qualitätsmerkmale, wie beispielsweise von Balzert, Pagel und Six. Im Folgenden werde ich auf einige wichtige Qualitätsmerkmale eingehen. (Vgl. [Li09], S. 5)

2.5.1 Korrektheit

Software ist immer dann korrekt, wenn sie mit ihrer funktionalen Spezifikation übereinstimmt. Daraus folgt, dass die Korrektheit nur mit einer gegebenen Spezifikation überprüft werden kann. (Vgl. [Li09], S. 7 f.)

2.5.2 Zuverlässigkeit

Liggemeyer definiert Zuverlässigkeit als „die Wahrscheinlichkeit des ausfallsfreien Funktionierens einer betrachteten Einheit über eine bestimmte Zeitspanne bei einer bestimmten Betriebsweise". ([Li09], S. 9) Ein gewisser Grad an mangelnder Zuverlässigkeit kann bei Softwareprodukten allerdings toleriert werden, was sie von fast allen anderen industriellen Produkten unterscheidet. (Vgl. [Pa13], S. 8)

2.5.3 Robustheit

Ein System kann korrekt sein und trotzdem ausfallen, beispielsweise dann, wenn Eingaben verarbeitet werden müssen, die nicht in den Spezifikationen definiert wurden. Robustheit ist dementsprechend der Grad an Toleranz gegenüber nicht spezifizierten Eigenschaften. (Vgl. [Li09], S. 10)

2.5.4 Performanz

Bezieht sich auf die Leistung eines Softwareprodukts, hinsichtlich dessen Antwortzeitverhalten und Umgang mit den eigenen Ressourcen (CPU, Speicher, Bandbreite, etc.). (Vgl. [Pa13], S. 14)

2.5.5 Benutzungsfreundlichkeit

Dieses Merkmal bezieht sich auf den Schwierigkeitsgrad der Bedienung. Benutzerfreundliche Software sollte intuitiv erlernbar, fehlerrobust und sich leicht konfigurieren lassen. (Vgl. [Pa13], S. 16)

2.5.6 Wartbarkeit

Stellt den Schwierigkeitsgrad für Anpassungen, Änderungen und Weiterentwicklungen eines Softwareprodukts dar. (Vgl. [Pa13], S. 17)

2.5.7 Wiederverwendbarkeit

Gibt an, wie gut sich das gesamte Softwareprodukt oder ein Teil dieses Produkts für einen anderen Kontext einsetzen lässt. Wiederverwendbarkeit entsteht nicht zufällig, sondern muss zuvor geplant werden, damit das Produkt oder Teile des Produkts auch mit den zukünftigen Strukturen anderer Produkte kompatibel sind. (Vgl. [Pa13], S. 22)

2.5.8 Portierbarkeit

Bezieht sich auf den Schwierigkeitsgrad, das Softwareprodukt in andere Hard- und Softwareumgebungen zu integrieren, wie beispielsweise Windows und Macintosh Systeme, die beide auf verschiedene Architekturen als auch Betriebssysteme zurückgreifen. (Vgl. [La10], S. 6)

2.5.9 Interoperabilität

Stellt die Fähigkeit der Zusammenarbeit mit anderen, verschiedenen Systemen dar. (Vgl. [Pa13], S. 24)

2.5.10 Sicherheit

Bezieht sich auf die Fähigkeit, Informationen und Daten vor unautorisierten Personen und System zu schützen, dahingehend, dass diese weder gelesen noch beschrieben werden können. (Vgl. [Al10], S. 172)

2.5.11 Effizienz

Die Effizienz ist die Fähigkeit eines Softwareprodukts, angemessene Leistungen im Verhältnis zu einem minimalen Ressourcenverbrauch zu erbringen. Dieses Qualitätsmerkmal wird auch häufig der Performanz untergeordnet, wie beispielsweise beim Qualitätsmodell FURPS. (Vgl. [La10], S. 5)

2.5.12 Vollständigkeit

Laut Liggemeyer ist Software „funktional vollständig, wenn alle in der Spezifikation geforderten Funktionen realisiert sind. Das betrifft sowohl die Behandlung von Normalfällen als auch das Abfangen von Fehlersituationen“. (Vgl. [Li09], S. 8)

2.6 Messen von Qualität

Allgemein wird beim Messen von Qualität versucht, zuvor definierte Eigenschaften quantifizierbar zu machen, um den Fortschritt eines Projekts prüfen, steuern und messen zu können. Das gleiche Ziel wird auch beim Messen von Softwarequalität verfolgt. (Vgl. [Schn12], S. 52)

Mit den bereits erwähnten Qualitätsmodellen werden relevante Qualitätsmerkmale quantifizierbar gemacht, in dem diese Merkmale in kleinere und leichter zu messende Teilmerkmale zerlegt werden, die in einer Beziehung zu messbaren Eigenschaften stehen. (Vgl. [Gl05], S. 4)

2.7 Metriken und Maße

Eine Softwarequalitätsmetrik ist nach dem IEEE Standard 1061 „eine Funktion, deren Eingabe aus Softwaredaten besteht und deren Ausgabe ein einzelner numerischer Wert ist, der als Erfüllungsgrad einer Qualitätseigenschaft interpretiert werden kann". (Vgl. [IE98], S. 2)

Liggemeyer macht darauf aufmerksam, dass der Begriff der Metrik in der Fachliteratur und Praxis häufig falsch verwendet wird. Streng genommen handelt es sich laut Liggemeyer um eine Maßbestimmung, „da Objekten der realen Welt (z. B. einer Systemkomponente) Messwerte zugeordnet werden". (Vgl. [Li09], S. 234)

2.7.1 Maßtypen

Softwarequalität wird zwischen Produkt- und Prozessqualität differenziert. Diese Differenzierung wird auch bei Maßen vorgenommen. In diesem Fall wird zwischen Produkt- und Prozessmaßen differenziert, außerdem kann die Berücksichtigung von Projektmaßen sinnvoll sein. (Vgl. [Li09], S. 234)

Produktmaße

Bei Produktmaßen handelt es sich um Informationen, die sich auf die Eigenschaften eines Produkts beziehen. Sie werden verwendet, um kritische Produktteile zu identifizieren und um Produktvergleiche vornehmen zu können. (Vgl. [Li09], S. 234)

Prozessmaße

Bei Prozessmaßen handelt es sich um Informationen, die sich auf die Eigenschaften eines Entwicklungsprozesses beziehen. Sie werden beispielsweise verwendet, um Probleme im Prozess zu identifizieren und Gegenmaßnahmen einzuleiten. (Vgl. [Li09], S. 234)

Projektmaße

Bei Projektmaßen handelt es sich um Informationen, die sich auf die Eigenschaften eines Projekts beziehen. Sie werden beispielsweise verwendet, um den Projektfortschritt zu kontrollieren. (Vgl. [Li09], S. 234)

2.7.2 Gruppen von Maßen

Umfangsmaße

Umfangsmaße zählen zu den einfachsten und frühesten Maßen. Sie erfassen einfache, direkt verfügbare Informationen, wie beispielsweise die Größe einer Programmdatei. (Vgl. [Li09], S. 255) Vorteilhaft bei diesen Maßen ist, dass sie sich leicht berechnen und auf alle Variationen von Programmen anwenden lassen. (Vgl. [Ze02], S. 29)

Textuelle Maße

Ein Beispiel für textuelle Maße ist das Umfangsmaß Lines of Code. Hierbei handelt es sich um ein textuelles Maß, weil es direkt auf dem Programmtext aufsetzt. (Vgl. [Li09], S. 256)

Datenmaße

Laut Liggemeyer verwenden Datenmaße „die in einem Programm vorhandenen Daten als Basisgrößen zur Definition von Maßen. Dies kann […] auf Basis des Programmcodes oder mithilfe von Modellen […] geschehen". (Vgl. [Li09], S. 256)

Kontrollflussmaße

Orientieren sich an der Kontrollstruktur eines Programms und nutzen diese Struktur, die in der Regel durch einen Kontrollflussgraphen dargestellt wird, für die Maßdefinition. (Vgl. [Li09], S. 256)

Zuverlässigkeitsmaße

Zuverlässigkeitsmaße lassen sich nicht anhand der Elemente eines Programms erfassen, weil sie sich auf Eigenschaften beziehen, die sich aus der Beobachtung während der Nutzung eines Programms ergeben. Die reale Nutzung des Programms kann alternativ durch Tests simuliert werden. (Vgl. [Li09], S. 256)

2.7.3 Populäre Maße

Im Folgenden werden einige der wichtigsten und populärsten Maße in Bezug auf Softwarequalität erläutert.

Lines of Code (LOC)

Dieses Maß ist sowohl Umfangsmaß als auch textuelles Maß. Es handelt sich um eine Funktion, die Programmzeilen zählt und das Ergebnis als Wert zurückgibt. Das Maß lässt sich auf alle Programmiersprachen anwenden, allerdings lässt sich hierbei kritisieren, dass nicht genau definiert wurde, ob Leer- und Kommentarzeilen bei der Berechnung berücksichtigt werden sollen. Weiterhin lässt sich kritisieren, dass eine Anordnung mehrerer Anweisungen innerhalb einer Programmzeile zu einer Abnahme des Messwertes führen. Daraus folgt, dass Programme mit gleichem Umfang unterschiedliche Werte erzielen können. (Vgl. [Li09], S. 255 f.)

Non Commented Source Statements (NCSS)

Ein ähnliches Maß wie Lines of Code, das ebenfalls ein Umfangsmaß als auch ein textuelles Maß darstellt. Auch hierbei handelt es sich um eine Funktion, die Programmzeilen zählt. Der Unterschied zu Lines of Code besteht darin, dass Kommentar- und Leerzeilen bei der Berechnung ausgeschlossen werden. Auch hier lässt sich kritisieren, dass mehrere Anweisungen innerhalb einer Programmzeile zu einer Abnahme des Messwertes führen. (Vgl. [Ze02], S. 28)

Mean Time Between Failure (MTBF)

Die Mean Time Between Failure (MTBF) ist ein Maß für die Zuverlässigkeit eines Systems mit reparierbaren Einheiten. Es bezieht sich auf die Betriebsdauer zwischen zwei aufeinanderfolgenden Ausfällen einer solchen reparierbaren Einheit. Der Wert lässt sich am einfachsten ermitteln, indem die gesamte Betriebszeit durch die Anzahl der Ausfälle dividiert wird, allerdings wird bei dieser Art der Wertermittlung davon ausgegangen, dass die MTBF ein konstanter Wert ist, weil keine Reparaturzeiten anfallen. In der Realität ist das allerdings nicht der Fall, schließlich müssen immer wieder Funktionserweiterungen und Fehlerkorrekturen vorgenommen werden. (Vgl. [Li09], S. 263)

Mean Time To Failure (MTTF)

Dieses Maß ist das gleiche Maß wie Mean Time Between Failure (MTBF), allerdings bezieht es sich auf Systeme mit nicht reparierbaren Einheiten, was bedeutet, dass diese Einheiten nicht repariert, sondern ausgetauscht bzw. erneuert werden müssen. (Vgl. [Li09], S. 442)

McCabe Maße

Wurden 1976 von Thomas J. McCabe eingeführt. Bei diesem Ansatz wird versucht, die strukturelle Komplexität zu berücksichtigen und für diese einen Wert zu ermitteln. Dieser Wert wird auch als zyklomatische Zahl (Z) bezeichnet. Das Prozedere stammt aus der Graphentheorie und lässt sich auch auf Kontrollflussgraphen anwenden. (Vgl. [Li09], S. 256 f.)

Vorteile der McCabe Maße liegen darin, dass sie sich mit einem einfachen Parser ermitteln lassen, allerdings beschränkt sich dieser Ansatz ausschließlich auf den Kontrollfluss und lässt beispielsweise die Komplexität des Datenflusses unberücksichtigt. (Vgl. [Ze02], S. 46)

Die zyklomatische Zahl (Z) für einen Kontrollflussgraphen (G) lässt sich wie folgt ermitteln: (Vgl. [Ze02], S. 44)

$Z(G) = e - n + 2p$
$e: Anzahl\ der\ Kanten\ von\ G$
$n: Anzahl\ der\ Knoten\ von\ G$
$p: Anzahl\ der\ Komponenten$

Für das Beispiel aus Abbildung 6 ergibt sich somit folgende Rechnung.

$Z(G) = 7 - 6 + 2 * 1 = 1 + 2 = 3$

Für Programme mit nur einer Komponente lässt sich der Wert auch einfacher ermitteln. Dazu wird die Anzahl der Entscheidungen abgezählt und dessen Summe mit eins addiert. Entscheidungen können immer dann getroffen werden, wenn Programmieranweisungen verwendet werden, die zu Verzweigungen im Kontrollflussgraphen führen. In Abbildung 7, was auch dem Graphen aus Abbildung 6 entspricht, sind zwei solcher Anweisungen enthalten. Somit ergibt sich, 1 + 1 + 1 = 3. (Vgl. [Li09], S. 256 f.)

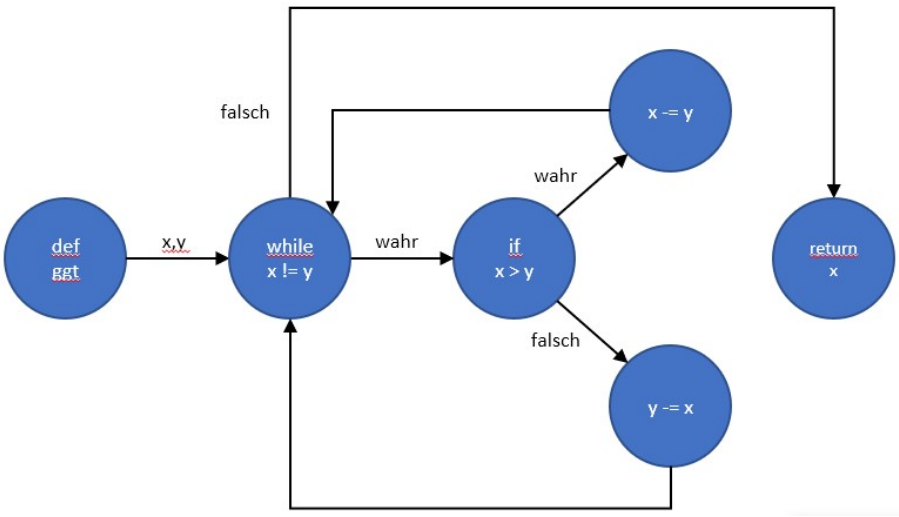

Abbildung 6 – McCabe Beispiel

Halstead Maße

Halstead Maße sind Umfangsmaße und textuelle Maße zugleich. Es sind verschiedene Maße, die sich auf unterschiedliche Softwareeigenschaften (Schwierigkeit, Umfang, Aufwand) beziehen. Alle diese Maße stellen Funktionen dar, bei denen der Quelltext als Eingabe verwendet und deren Ausgabewerte mithilfe der Anzahl unterschiedlicher Operatoren und Operanden, sowie der Gesamtanzahl verwendeter Operatoren und Operanden ermittelt werden. (Vgl. [Li09], S. 260) Operatoren sind Symbole oder Schlüsselwörter, die Aktionen kennzeichnen (+, while, etc.). Operanden sind alle übrigen Symbole und Schlüsselwörter, wie beispielsweise Variablen und Konstanten (Vgl. [Gl05], S. 8) Vorteile der Halstead Maße bestehen darin, dass sie sich für alle Programmiersprachen eignen, bereits durch einen einfachen Scanner ermittelt werden können und ein gutes Maß für Komplexität darstellen. Kritisieren lässt sich, dass ausschließlich die textuelle Komplexität berücksichtigt wird. (Vgl. [Ze02], S. 40)

Basisgrößen

η_1 → Anzahl unterschiedlicher Operatoren.
η_2 → Anzahl unterschiedlicher Operanden.
N_1 → Gesamtanzahl verwendeter Operatoren.
N_2 → Gesamtanzahl verwendeter Operanden.

Größe des Vokabulars und Länge der Implementierung

Mit den 4 Basisgrößen lassen sich zwei weitere Maße berechnen, die Größe des Vokabulars und die Länge der Implementierung. (Vgl. [Gl05], S. 8)

$\eta = \eta_1 + \eta_2$ → Größe des Vokabulars.
$N = N_1 + N_2$ → Länge der Implementierung.

Halstead Schwierigkeit (Difficulty)

Die Halstead Schwierigkeit, die im Englischen als Halstead Difficulty (D) bezeichnet wird, stellt die Schwierigkeit dar, um einen Algorithmus in einer Programmiersprache zu kodieren. (Vgl. [Li09], S. 261)

$$D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}$$

Halstead Umfang (Volume)

Der Halstead Umfang, der im Englischen als Halstead Volume (V) bezeichnet wird, stellt die Größe des Programms in Bits dar. (Vgl. [Li09], S. 260)

$$V = N * \log_2 \eta$$

Level

Das Level (L) ist der Kehrwert des Schwierigkeitsgrades und stellt das Programmniveau dar, wobei Programme mit niedrigem Niveau anfälliger für Fehler sind. (Vgl. [Wi18], S. 24)

$L = 1 / D$

Halstead Aufwand (Effort)

Der Halstead Aufwand, der im Englischen als Halstead Effort (E) bezeichnet wird, stellt den Implementierungsaufwand dar. (Vgl. [Wi18], S. 24)

$E = D * V$

Implementierzeit (Time to implement)

Laut Witte sind die Implementierzeit (T) und die Zeit, die benötigt wird, um ein Programm zu verstehen, proportional zum Implementierungsaufwand (E). Weiterhin macht Witte darauf aufmerksam, dass empirische Untersuchungen darauf schließen lassen, dass ein realistischer Wert für den Implementierungsaufwand in Sekunden entsteht, wenn dieser Aufwand (E) durch 18 dividiert wird. (Vgl. [Wi18], S. 24)

$T = E / 18$

Beispiel für die Berechnung der Halstead Maße

Bei diesem Beispiel (Abbildung 7) handelt es sich um eine Funktion, die in der Programmiersprache Ruby definiert wurde und den größten gemeinsamen Teiler der Zahlen x und y zurückgibt. Die Operatoren wurden blau gekennzeichnet, während die Operanden rot gekennzeichnet wurden.

```
def ggt(x,y)
        while x != y
                if x > y
                        x -= y
                else
                        y -= x
                end
        end
        return x
end
```

Abbildung 7 – Halstead Beispiel

$$\eta_1 = 11$$
$$\eta_2 = 3$$
$$N_1 = 15$$
$$N_2 = 12$$
$$\eta = \eta_1 + \eta_2 = 11 + 3 = 14$$
$$N = N_1 + N_2 = 15 + 12 = 27$$
$$D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2} = \frac{11}{2} * \frac{12}{3} = 5{,}5 * 4 = 22$$
$$L = \frac{1}{D} = \frac{1}{22}$$
$$V = N * \log_2 \eta = 27 * \log_2 14 = 102{,}8$$
$$E = D * V = 22 * 102{,}8 = 2261{,}6$$
$$T = \frac{E}{18} = \frac{2261{,}6}{18} = 125{,}64\ Sekunden$$

Live Variables (LV)

Ein Maß zur Messung der Komplexität, bei dem davon ausgegangen wird, dass die Schwierigkeit der Generierung einer Anweisung durch die Anzahl der zu berücksichtigenden Variablen bestimmt wird. Gezählt werden ausschließlich die lebendigen Variablen jeder Codezeile. Variablen gelten innerhalb eines Moduls als lebendig, wenn sie sich zwischen ihrer ersten und letzten Referenz befinden. Anschließend wird aus der ermittelten Anzahl lebendiger Variablen das arithmetische Mittel berechnet, was letztendlich den Wert dieses Maß darstellt. (Vgl. [Li09], S. 262)

$$Live\ Variables\ (LV) = \frac{Gesamtzahl\ lebendiger\ Variablen}{Anzahl\ ausführbarer\ Anweisungen}$$

Coupling Between Objects (CBO)

Bezieht sich auf die Kopplung eines Systems und stellt einen Indikator für den Verwaltungs- und Testaufwand dar. Berechnet wird die Summe der Anzahl verwendeter Klassen und der Anzahl von Klassen, die auf verwendete Klassen referenzieren. Ein höherer Wert führt zu höheren Testaufwand und zu größerer Fehlerwahrscheinlichkeit. (Vgl. [Wi18], S. 31)

$$CBO = Anzahl\ verwendeter\ Klassen + Anzahl\ von\ referenzierenden\ Klassen$$

Number Of Services (NOS)

Bezieht sich ebenfalls auf die Kopplung eines Systems, allerdings wird in diesem Fall die Anzahl der Dienste berechnet. Unter einem Dienst werden Methoden verstanden, die sich innerhalb einer Klasse befinden und von einer anderen Klasse genutzt werden. (Vgl. [Gl05], S. 10)

Number Of Accesses (NOA)

Ebenfalls ein Indikator für die Kopplung eines Systems. Gibt die Anzahl der Zugriffe an, die eine bestimmte Klasse auf eine andere Klasse vornimmt. (Vgl. [Gl05], S. 10)

Lack Of Cohesion in Methods (LCOM)

Stellt ein Maß für den Mangel an Kohäsion dar, wobei Kohäsion als Grad der Verknüpfungen zwischen einzelnen Teilen eines Programmelements verstanden wird. Laut Globke ist der Wert „definiert durch die Differenz zwischen der Anzahl der Paare von Methoden einer Klasse, die eine gemeinsame Instanzvariable verwenden, und der Anzahl der Paare, die das nicht tun". Globke macht allerdings darauf aufmerksam, dass diese Definition nicht völlig zufriedenstellend war und über die Zeit mehrere Verbesserungen vorgeschlagen wurden. (Vgl. [Gl05], S. 11)

Tight Class Cohesion (TCC)

Bezieht sich auf die Kohäsion eines Systems. Der Wert, der bei diesem Maß ermittelt wird, stellt die Anzahl von Methoden einer Klasse dar, die sich mindestens eine Instanzvariable teilen. (Vgl. [Gl05], S. 11)

Loose Class Cohesion (LCC)

Ist ein ähnliches Maß wie Tight Class Cohesion (TCC), allerdings mit dem Unterschied, dass auch Aufrufe anderer Methoden berücksichtigt werden, die sich indirekt eine Instanzvariable teilen. (Vgl. [Gl05], S.11)

Depth Of Inheritance (DIT)

Dieses Maß bezieht sich auf die Vererbungsstruktur und geht davon aus, dass es schwieriger wird, eine Klasse zu verstehen, desto stärker sie von anderen Oberklassen abhängig ist, weil dadurch eine höhere Fehleranfälligkeit entsteht. Der Wert selbst stellt die Pfadlänge von der Wurzel bis zur betrachteten Klasse dar, somit ist ein höherer Wert schlechter als ein niedriger Wert. (Vgl. [Wi18], S. 29)

Number Of Descendants (NOD)

Bezieht sich ebenfalls auf die Vererbungsstruktur eines Systems und stellt die Anzahl aller Klassen dar, die von einer beliebigen Oberklasse erben. Es ist ein ähnlicher Wert wie Depth Of Inheritance (DIT), dementsprechend ist auch hier ein höherer Wert schlechter, weil die Komplexität des Systems zunimmt. (Vgl. [Gl05], S. 11)

Reuse Ratio

Bezieht sich auch auf die Vererbungsstruktur und ist sozusagen ein Wert dafür, wie häufig auf das Konstrukt der Vererbung zurückgegriffen wird. Berechnet wird der Wert, in dem die Anzahl der Oberklassen durch die Anzahl aller restlichen Klassen dividiert wird. Laut Globke deutet ein Wert nahe 1 auf eine lineare Vererbungskette hin, während ein Wert nahe 0 auf eine flache Vererbungshierarchie hinweist. (Vgl. [Gl05], S. 11)

Specialization Ratio

Stellt das Verhältnis zwischen der Anzahl von Unterklassen zur Anzahl aller Oberklassen dar. Lässt sich dementsprechend genauso berechnen wie die Reuse Ratio, nur dass in diesem Fall die Anzahl der Unterklassen durch die Anzahl der Oberklassen dividiert wird. (Vgl. [Gl05], S. 11)

2.8 Zwischenfazit zur Softwarequalität

Um die Prüfkomponente dahingehend zu erweitern, dass sie in der Lage ist, qualitative Softwareaspekte zu messen, musste zunächst in Erfahrung gebracht werden, was überhaupt unter den Begriff Softwarequalität verstanden wird. Diesbezüglich lässt sich festhalten, dass mehrere Definitionen für diesen Begriff existieren und dass sich dieser aus verschiedenen Perspektiven betrachten lässt. Weiterhin lässt sich festhalten, dass mittels Qualitätsmodellen verschiedene Verfahren geschaffen wurden, um Qualität spezifizieren und kontrollieren zu können. Innerhalb dieser Modelle finden sich Qualitätskriterien wieder, deren Grad an Erfüllung mithilfe verschiedener Maße gemessen werden kann.

Vor allem dieser letzte Punkt ist für die Erweiterung der Prüfkomponente von Bedeutung, denn daraus lässt sich schließen, dass anhand verschiedener Maßeinheiten Rückschlüsse auf Qualitätskriterien gezogen werden können. Dementsprechend gilt es nun, die Berechnung einiger dieser Maße zu automatisieren und innerhalb der Prüfkomponente zu implementieren. Bevor sich nun mit dieser Implementierung befasst wird, werden zunächst im folgenden Kapitel verwandte Werke, Systeme und die bisherige Ruby Prüfkomponente vorgestellt, die bereits in der Lage ist, einen Aspekt von Softwarequalität zu analysieren.

3 Verwandte Werke und Systeme

Es gibt eine ähnliche Arbeit mit dem Titel „Qualitative aspects of students' programs: Can we make them measurable?" von Eliane Araujo, Dalton Serey und Jorge Figueiredo aus dem Jahr 2016. Araujo, Serey und Figueiredo verfolgen mit ihrer Arbeit ein ähnliches Ziel, das auch innerhalb dieser Bachelor-Arbeit verfolgt wird. Sie versuchen ein System zu schaffen, das in der Lage ist, qualitative Softwareaspekte studentischer Programme automatisch zu messen und für die berücksichtigten Maße Feedback zu generieren, damit Studenten ihre eigene Codequalität hinterfragen und verbessern können. Der wesentliche Unterschied dieser Bachelor-Arbeit liegt darin, dass die in dieser Arbeit entwickelte Komponente für die Programmiersprache Ruby entwickelt wurde, während das System von Araujo, Serey und Figueiredo für die Programmiersprache Python entwickelt wurde. Beide Systeme berücksichtigen allerdings die gleichen Maße und greifen für die Generierung von Feedback und für die eigentliche Beurteilung auf eine Referenzlösung zurück, dessen Ergebnisse, den Ergebnissen der Studenten gegenübergestellt werden. Innerhalb ihrer Arbeit kommen Araujo, Serey und Figueiredo zu dem Schluss, dass sich die von ihrem System berechneten Qualitätsmaßstäbe gut eignen, um Studenten aufzuzeigen, wie sie ihre Codequalität verbessern können. Außerdem konnten sie anhand eines Experiments feststellen, dass Studenten durch Hinweise auf ihre Codequalität dazu motiviert werden, bessere Ergebnisse einzureichen, auch dann, wenn die funktionalen Aspekte ihrer Programme bereits für eine korrekte Lösung ausgereicht haben. (Vgl. [ArSeFi16], S. 8)

Es gibt verschiedene E-Assessment-Systeme, die mehr oder weniger mit JACK verglichen werden können. Das E-Assessment-System EASy der Westfälischen Wilhelms-Universität Münster ist ein solches System. Das Modul für Programmieraufgaben wurde in EASy für die Programmiersprache Java entwickelt und führt dynamische als auch statische Testverfahren durch. In Bezug auf die Softwarequalität berücksichtigt das Modul bei seiner Analyse neben funktionalen Aspekten auch die Einhaltung von Programmierrichtlinien und Programmierkonventionen. Weitere Aspekte von Softwarequalität bleiben von der Komponente allerdings unberücksichtigt. (Vgl. [Gr09], S. 130 ff.) Ein weiteres E-Assessment-System, das mit JACK verglichen werden kann und gleichzeitig qualitative Softwareaspekte bei seiner Analyse berücksichtigt, trägt den Namen CourseMarker und wurde an der Universität Nottingham entwickelt. Das System unterstützt verschiedene Programmiersprachen, wie beispielsweise Java, C, C++ und Pascal und berücksichtigt bei der Analyse neben funktionalen Aspekten auch weitere qualitative Softwareaspekte, wie beispielsweise die lexikalische Struktur, Komplexität und Effizienz. (Vgl. [HHST03], S. 296 f.)

E-Assessment-Systeme, die sowohl Ruby unterstützen und gleichzeitig Maße berechnen, mit denen Rückschlüsse auf die Codequalität geschlossen werden können, konnten allerdings nicht gefunden werden.

4 Aktuelle Ruby Prüfkomponente

Für das Bachelor-Projekt aus dem Sommersemester 2019 mit dem Titel „Implementierung eines Ruby Prüfmoduls für das E-Assessment-System JACK", wurde bereits eine Ruby Prüfkomponente für das E-Assessment-System JACK implementiert. Diese Komponente ist in der Lage statische als auch dynamische Testverfahren für Quelltexte in der Programmiersprache Ruby durchzuführen. Weiterhin ist die Komponente dazu in der Lage, die Benotung durchzuführen und konstruktives Feedback zu generieren. Qualitative Softwareaspekte bleiben von dieser Prüfkomponente allerdings weitgehend unberücksichtigt.

Einige Qualitätsmerkmale werden aber bereits von der Komponente berücksichtigt. Dabei handelt es sich um die Wartbarkeit und Vollständigkeit. Die Wartbarkeit hängt mit der Codequalität zusammen, die sich mit der statischen Komponente analysieren lässt. Die Vollständigkeit hängt mit der Realisierung der gewünschten Funktionalität zusammen, die sich mit der dynamischen Komponente analysieren lässt.

4.1 E-Assessment-System JACK

JACK ist ein E-Assessment-System, das im Jahr 2006 / 2007 von Paluno veröffentlicht wurde. Seit diesem Zeitpunkt wird es vom Lehrstuhl „Spezifikation von Softwaresystemen" an der Universität Duisburg-Essen betrieben. JACK unterstützt Studenten als auch Lehrkörper bei Vorlesungen, Übungen und Prüfungen. Das System wurde so konzipiert, dass eine Vielzahl verschiedener Aufgabentypen abgedeckt werden, wie beispielsweise Multiplechoice, Lückentexte, mathematische Aufgaben und Programmieraufgaben. Für alle diese Aufgabentypen können Aufgaben generiert, automatisch korrigiert und mit Feedback versehen werden. Somit unterstützt JACK Studenten bei ihrem Lernprozess und entlastet gleichzeitig Lehrkörper bei der Durchführung von Korrekturen. (Vgl. [St16], S. 36 f.)

4.1.1 Architektur

Die JACK Architektur teilt sich in zwei wesentliche Bestandteile, dem Core- und Worker Server. Der Core Server besteht aus den hellgrauen Komponenten, die in Abbildung 8 zu sehen sind. Er stellt ein reaktives System dar, dessen Komponenten Zugriffsmöglichkeiten für Browser-Clients (Web-Frontend), integrierte Entwicklungsumgebungen (Web-Service for Eclipse) und Worker Server (Web-Services for Workers) schaffen. Weitere Aufgaben liegen in der lokalen Datenspeicherung (Persistance) und Authentifizierung externer Systeme mit eigenen Datenspeicher (Authentication Service).

Worker Server bestehen aus den dunkelgrauen Komponenten und stellen im Gegensatz zum Core Server aktive Systeme dar, dessen Komponenten sich mit der Kommunikation zum Core Server befassen (Worker Core). Die Komponente „Worker Core" enthält zusätzlich noch weitere Komponenten zur Durchführung der eigentlichen Benotung. Die Korrektur der eingereichten Lösungen wird durch die „Asynchronous Checker Components" durchgeführt. (Vgl. [St16], S. 39)

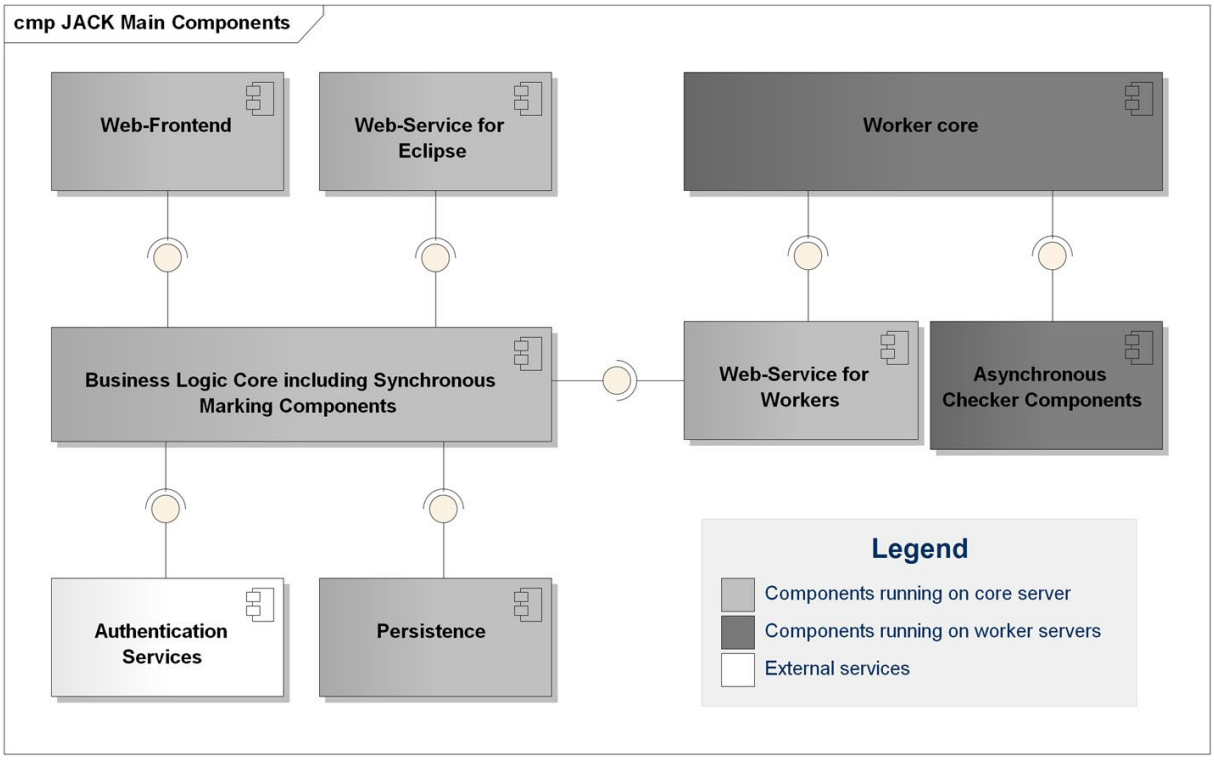

Abbildung 8 - Komponentendiagramm der JACK Architektur
Quelle: Michael Striewe: An Architecture for Modular Grading and Feedback Generation for Complex Exercises. Science of Computer Programming 129. Special issue on eLearning Software Architectures, 2016.

4.1.2 Erweiterungsmöglichkeiten

JACK lässt sich aufgrund seines Architekturstils leicht erweitern, was vor allem daran liegt, dass die Checker Komponenten als eigene Komponenten betrachtet werden. Diese Komponenten nehmen eingereichte Lösungen entgegen, führen die eigentliche Korrektur durch und leiten die Ergebnisse zurück an die „Worker Core" Komponente. Durch die Implementierung eines Interface namens „IChecker" wird ein korrekter Informationsaustausch zwischen Checker-Komponente, „Worker Core" und Core Server" gewährleistet. JACK lässt sich dementsprechend leicht erweitern, in dem diese Checker-Komponenten entwickelt und in das System eingebunden werden. (Vgl. [St16], S. 40)

4.2 Testverfahren

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

4.2.1 Dynamische Tests

Dynamische Testverfahren führen den Quelltext eines Programms aus und analysieren die Eignung des Programms hinsichtlich dessen Einsatzzwecks. Sie werden dementsprechend zur Validierung genutzt und lassen sich anhand der Prüftechnik (strukturorientiert, funktionsorientiert, diversifizierend) und des Informationsstandes (White-Box-Test, Black-Box-Text, Grey-Box-Test) weiter unterteilen. (Vgl. [Wi19], S. 84 f.)

Black-Box-Test

Bei diesen Verfahren sind den Testern die inneren Strukturen des zu testenden Objekts unbekannt und die Testfälle werden aus den Spezifikationen abgeleitet. Das Programm wird ausgeführt und mit Eingaben gespeist, dessen Ausgaben anschließend mit den erwarteten Ausgaben verglichen werden. Das Ziel eines Testers liegt darin, ein Systemverhalten zu finden, das den Spezifikationen widerspricht. (Vgl. [Wi19], S. 84)

White-Box-Test

Diese Verfahren basieren auf der Grundlage, 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

Hierbei 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)

4.2.2 Statische Tests

Diese Testverfahren analysieren den Quelltext eines Programms, ohne diesen auszuführen. Sie fokussieren sich auf die Syntax eines Programms und werden dementsprechend zur Verifikation genutzt. Geprüft wird vor allem, ob der Quelltext ausgeführt werden kann und ob gewisse Standards eingehalten werden. (Vgl. [Wi19], S. 84)

4.3 Implementierung und Vorgehensweise

Um sowohl dynamische als auch statische Testverfahren mit der Ruby Prüfkomponente abzudecken, wurden während meines Bachelor-Projekts zwei separate Checker-Komponenten entwickelt. Beide Komponenten arbeiten eng mit dem JACK Backend zusammen und greifen auf ein vom Backend angebotenes Interface namens „IChecker" zurück.

4.3.1 Statisches Ruby Prüfmodul

Diese Komponente analysiert den Quelltext der eingereichten Lösungen auf Verstöße gegen Konventionen und Syntax, ohne den Quelltext dabei auszuführen. Geprüft wird einerseits, ob die Regeln der Programmiersprache (Syntax) eingehalten werden und andererseits, ob die Programmierrichtlinien der Ruby Community (Konventionen) eingehalten werden. Die statische Ruby Checker Komponente greift für die Analyse des Quelltexts auf ein externes Werkzeug namens RuboCop zurück. Nach dem die eingereichten Lösungen lokal zwischengespeichert wurden, wird RuboCop mittels Kommandozeilenbefehl gestartet. Das Programm analysiert anschließend den Quelltext der Lösungen und exportiert die Ergebnisse in einer JSON-Datei, dessen Dateiname zufällig erzeugt wird. Diese JSON-Datei wird letztendlich von der Komponente eingelesen. Die darin gespeicherten Informationen werden zur Generierung der Fehlermeldungen genutzt. Zuletzt führt die Komponente die Benotung durch, wobei die Gesamtpunktzahl von der Anzahl der gefundenen Verstöße gegen Syntax und Konventionen und den zuvor getätigten Optionen abhängig ist, siehe Abbildung 9.

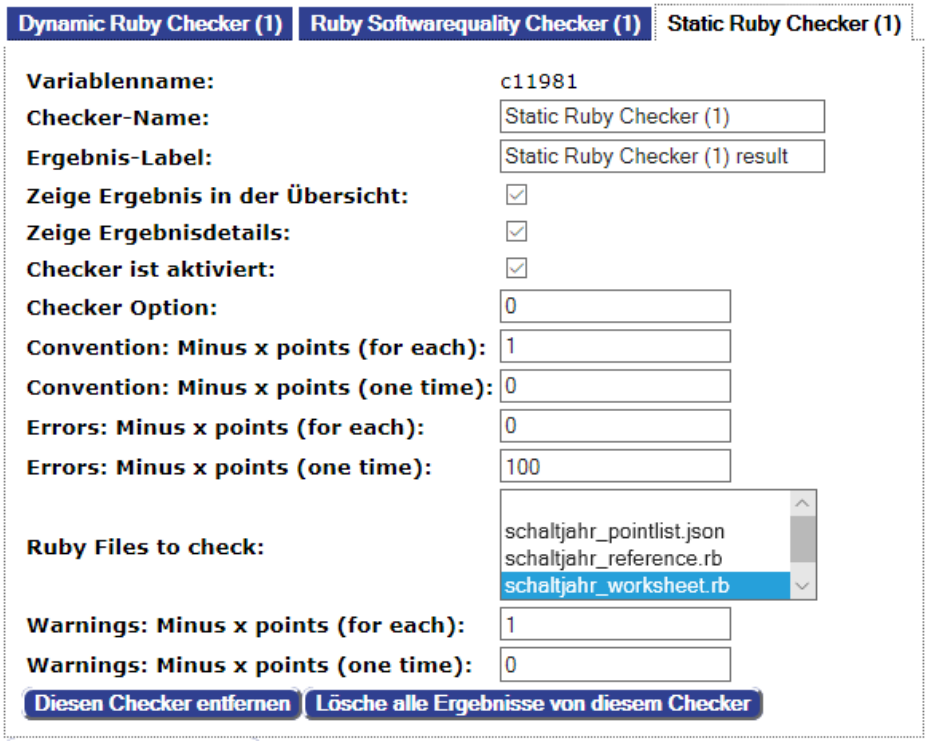

Abbildung 9 - Statische Ruby Prüfmodul

Über die „Checker Option" können Lehrkörper bestimmen, welche Formen von Verstößen bei der Analyse berücksichtigt werden sollen. Bei einer 0 werden sowohl Verstöße gegen Konventionen als auch Warnungen und Fehlermeldungen berücksichtigt. Durch die Eingabe einer 1 fließen nur noch Warnungen und Fehlermeldungen in die Analyse ein. Wird eine 2 als Wert verwendet, dann werden ausschließlich Fehlermeldungen von der Komponente berücksichtigt. Für jede Art von Verstoß können Punkte abgezogen werden, dies kann einmalig (one time) oder mehrmalig (for each) erfolgen. Einmalig bedeutet, dass die Punktzahl nur einmal abgezogen wird, sobald 1 Verstoß erkannt wurde, auch wenn es sich um mehrere Verstöße handelt. Mehrmalig bedeutet, dass immer wieder die eingetragene Punktzahl abgezogen wird, sobald ein weiterer Verstoß erkannt wurde.

Somit bietet die Komponente bereits die Möglichkeit, einen Aspekt von Softwarequalität zu überprüfen und zu benoten. Bei diesem Aspekt handelt es sich um die Codequalität. Die Codequalität zählt zur Produktqualität als auch zur inneren Qualität. Eine hohe Codequalität führt dazu, dass ein Programm leichter modifiziert werden kann. Auch die Generierung von Testfällen wird dadurch erleichtert. Somit hat die Codequalität auch einen Einfluss auf das Qualitätsmerkmal der Wartbarkeit. (Vgl. [ObRe16], S. 48)

Als Maßstab für Verstöße gegen Konventionen wird der Ruby Style Guide verwendet. Dabei handelt es sich um bewährte Verfahren, mit denen Ruby Programmierer sich einen Programmierstil aneignen, der von anderen Ruby Programmierern leicht interpretiert und modifiziert werden kann. Die Regeln dieses Styleguides basieren wiederum auf beruflichen Erfahrungen und hoch angesehenen, literarischen Werken, wie beispielsweise „The Ruby Programming Language", von David Flanagan und Yukihiro Matsumoto, dem Entwickler von Ruby. Weitere Anregungen, die in diesem Style-Guide mit einfließen, kommen von den Mitgliedern der Ruby Community selbst. Zuletzt lässt sich erwähnen, dass dieser Styleguide über die Zeit stetig weiterentwickelt und somit auch für zukünftige Ruby Versionen Bestand haben wird. (Vgl. [Ba20], Introduction)

4.3.2 Dynamisches Ruby Prüfmodul

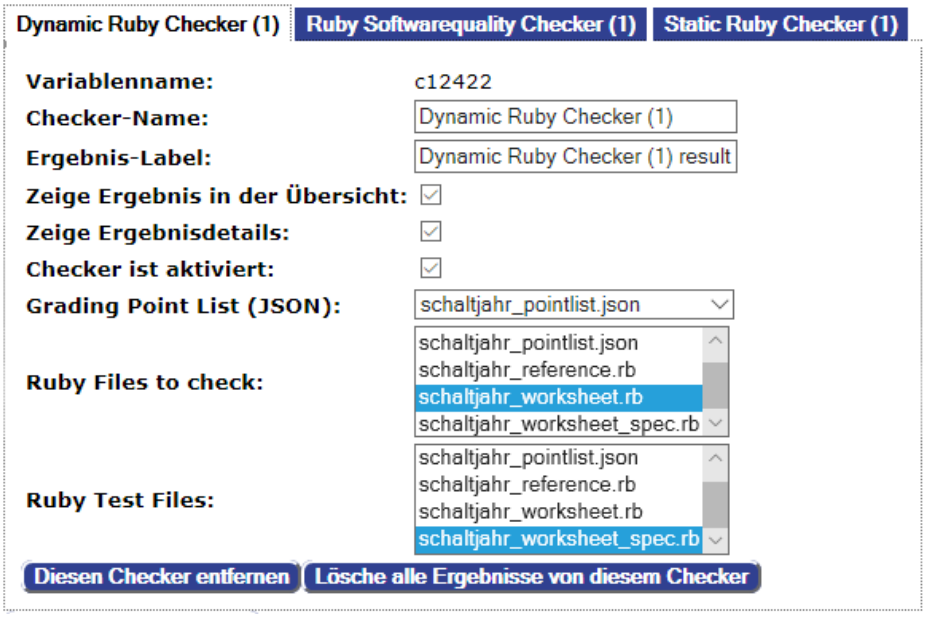

Abbildung 10 - Dynamisches Ruby Prüfmodul

Diese Komponente ist ähnlich aufgebaut, wie die statische Komponente, mit dem Unterschied, dass sie den Quelltext ausführt und das Programm hinsichtlich seines Einsatzzwecks analysiert. Auch in diesem Fall wird auf ein externes Werkzeug zurückgegriffen. Dabei handelt es sich um RSpec, einem Test-Framework, mit dem das Verhalten eines Ruby Programms spezifiziert und kontrolliert werden kann.

Nach dem die eingereichten Lösungen zwischengespeichert wurden, wird RSpec mittels Kommandozeilenbefehl gestartet. Das Programm analysiert den Quelltext anschließend mit einem zuvor definierten Testdokument, das in einem RSpec kompatiblen Format verfasst wurde, siehe Abbildung 11. Danach werden die Ergebnisse in einer JSON-Datei exportiert, die anschließend von der dynamischen Komponente eingelesen wird. Die Informationen aus dieser Datei werden zur Generierung von Fehlermeldungen und zur Durchführung der Benotung verwendet. Für die Benotung greift die Komponente auf eine sogenannte Grading Point List im JSON-Format zurück, siehe Abbildung 12. Diese Punkteliste enthält die jeweiligen Teilpunkte, die für korrekt gelöste Aufgaben zu vergeben sind.

Es lässt sich festhalten, dass die dynamische Komponente den Quelltext hinsichtlich seiner korrekten Funktionalität überprüft. Aus Sicht des Qualitätsmerkmals der Vollständigkeit, deckt diese Komponente somit auch einen Teil von Softwarequalität ab, denn sie kann überprüfen, ob die in den Spezifikationen geforderten Funktionen realisiert wurden. Voraussetzung dafür ist, dass die Spezifikationen in einem RSpec kompatiblen Format überführt wurden, mit denen sich die Funktionalität überprüfen lässt.

schaltjahr_worksheet_spec.rb

```ruby
require_relative 'schaltjahr_worksheet'

describe Schaltjahr do
  let!(:testObject) {Schaltjahr.new}
  it "Error: Operation schaltjahr(jahr) should return true for 2020." do
    expect(testObject.schaltjahr(2020)).to eq(true)
  end
  it "Error: Operation schaltjahr(jahr) should return true for 2012." do
  end
  it "Error: Operation schaltjahr(jahr) should return true for 1996." do
  end
  it "Error: Operation schaltjahr(jahr) should return false for 1805." do
  end
  it "Error: Operation schaltjahr(jahr) should return false for 1979." do
  end
  it "Error: Operation schaltjahr(jahr) should return true for 2041." do
    expect(testObject.schaltjahr(2041)).to eq(false)
  end
end
```

Abbildung 11 - Ruby Testdokument im RSpec Format

schaltjahr_pointlist.json

```json
{
  "lesson":
  {
    "name": "Schaltjahr",
    "author": "Patrick Czarnetzki",
    "exercise":
    [
      {
        "id": "0",
        "name": "Schaltjahr 2020 True",
        "points": "20"
      },
      {},
      {},
      {},
      {},
      {
        "id": "5",
        "name": "Schaltjahr 2041 False",
        "points": "15"
      }
    ]
  }
}
```

Abbildung 12 - Grading Point List

4.4 Zwischenfazit zur aktuellen Ruby Prüfkomponente

In diesem Kapitel wurde die Komponente, die für diese Bachelor-Arbeit erweitert werden soll, genauer vorgestellt. Neben der Architektur von JACK wurden Aufbau und bereitgestellte Funktionalitäten der Komponente erläutert. Es lässt sich feststellen, dass die Ruby Prüfkomponente aus zwei separaten Komponenten besteht, die statische als auch dynamische Testverfahren durchführen können. Weiterhin konnten wir feststellen, dass beide Komponenten bei ihrer Analyse bereits verschiedene Aspekte von Softwarequalität berücksichtigen. Bei diesen Aspekten handelt es sich um die Qualitätsmerkmale Wartbarkeit und Vollständigkeit. Das ist eine wichtige Erkenntnis für diese Arbeit, die Redundanzen verhindert und Zeit für die Entwicklung reduziert, weil bereits realisierte Funktionalitäten nicht erneut implementiert werden müssen.

5 Erweiterung der Ruby Prüfkomponente

Um die Komponente dahingehend zu erweitern, dass sie weitere qualitative Softwareaspekte bei ihrer Analyse berücksichtigen kann, wurde eine dritte Prüfkomponente entwickelt. Diese Komponente befasst sich ausschließlich mit qualitativen Softwareaspekten und berechnet verschiedene Maße für eine Referenzlösung, die anschließend mit den Maßen der studentischen Lösung verglichen werden.

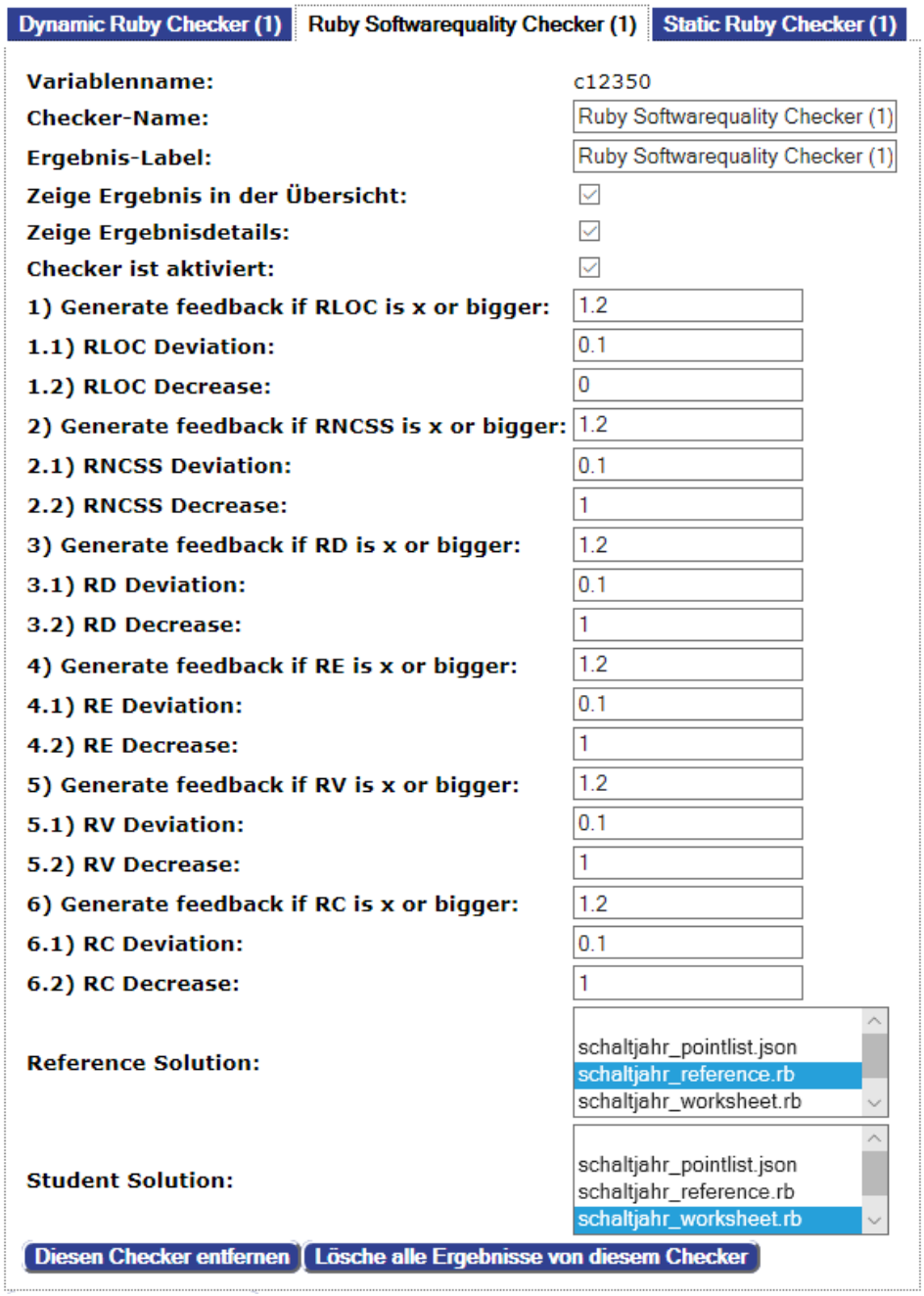

Abbildung 13 - Ruby Softwarequality Checker

5.1 Unterstützte Maße

Horst Zuse macht darauf aufmerksam, dass mindestens 500 verschiedene Metriken existieren, mit denen Qualitätseigenschaften von Software gemessen werden können. Der Versuch, alle diese Metriken innerhalb dieser Arbeit zu erfassen und in die Komponente zu implementieren wäre dementsprechend ein utopisches Vorhaben. Aus diesem Grund musste eine Auswahl getroffen werden, auf die im Folgenden genauer eingegangen wird. (Vgl. [Zu94])

5.1.1 Lines Of Code (LOC)

Wie bereits in Kapitel 2.7.3 erwähnt, ist nicht genau definiert, ob bei diesem Maß Leer- und Kommentarzeilen berücksichtigt werden. In diesem Fall legt die Komponente dieses Maß so aus, dass sowohl Leer- als auch Kommentarzeilen mitgezählt werden. Das liegt darin begründet, dass die Komponente ein weiteres Maß (NCSS) berechnet, in dem bereits Leer- und Kommentarzeilen unberücksichtigt bleiben. Bei dieser Auslegung stellt Lines of Code allerdings kein gutes Maß zur Bewertung von Softwarequalität dar, weil es impliziert, dass auf Kommentare und Leerzeilen verzichtet werden sollte. Durch einen Verzicht auf Kommentare und Leerzeilen werden automatisch weniger Zeilen benötigt und dementsprechend entsteht ein kleinerer und somit besser LOC-Wert. Kommentare sind jedoch ein wichtiger Bestandteil eines Programms und tragen zu einer besseren Wartbarkeit bei, weil der Quelltext dadurch vor allem für andere Programmierer verständlicher wird. (Vgl. [ScWi74], S. 1 ff.) Das Maß wurde trotzdem implementiert, weil das Maß Non Commented Source Statements (NCSS) definitiv implementiert werden sollte und die Berechnung beider Maße sehr ähnlich ist. Die Berechnung erfolgt, in dem der Quelltext Zeile für Zeile eingelesen und für jede nicht leere Zeile inkrementiert wird, wobei mit einem Ausgangswert von 0 gestartet wird.

5.1.2 Non Commented Source Statements (NCSS)

Dieses Maß lässt sich ähnlich ermitteln, wie das vorherige Maß Lines of Code (LOC). Die Komponente geht genauso vor wie bei dem vorherigen Maß (LOC), mit dem Unterschied, dass sie vor dem Zählen der Zeilen auf Leer- und Kommentarzeilen prüft. Gezählt wird nur, wenn es sich um keine Leer- oder Kommentarzeile handelt. In Ruby gibt es mehrzeilige und einzeilige Kommentare. Mehrzeilige Kommentare starten mit dem Schlüsselwort „=begin“ und enden mit dem Schlüsselwort „=end“. Durch diese Konstellation lassen sie sich beim Einlesen leicht erkennen. Einzeilige Kommentare starten mit einer Raute (#) und haben kein Tag, mit dem sie geschlossen werden. Alles, was sich hinter der Raute befindet, wird innerhalb dieser Codezeile als Kommentar gewertet. Dementsprechend lassen sich auch diese Kommentare beim Einlesen der Zeilen leicht erkennen. Leerzeilen lassen sich ermitteln, indem zuvor potenziell mögliche Leerzei-

chen gelöscht werden und anschließend auf eine leere Zeile geprüft wird. Sollte es sich bei der eingelesenen Codezeile nicht um eine Leerzeile, einen einzeiligen Kommentar oder einen mehrzeiligen Kommentar handeln, wird die Komponente inkrementieren, wobei wiederum mit einem Ausgangswert von 0 gestartet wird. Das Maß wurde implementiert, weil es sich leicht erfassen lässt und weil sich damit Rückschlüsse auf die Effizienz eines Programms ziehen lassen. Erfahrene Entwickler lösen Probleme mit weniger Codezeilen als unerfahrene Entwickler, was zur Folge hat, dass deren Programme effizienter sind, weil sie weniger Ressourcen verbrauchen und sich in der Regel schneller ausführen lassen. Durch die Implementierung dieses Maß werden Studenten also ermutigt, ihre Programme kompakter und effizienter zu gestalten. (Vgl. [BhTaPa12], S. 150 f.)

5.1.3 Halstead Maße

Die Halstead Maße wurden in die Prüfkomponente implementiert, weil mit ihnen detaillierte Aussagen über die Größe und Komplexität eines Programms getroffen werden können, als das mit den Maßen Lines Of Code (LOC) und Non Commented Source Statements (NCSS) möglich ist. (Vgl. [Mü97], S. 47 f.) Außerdem handelt es sich bei den Halstead Maßen um gut untersuchte und weit verbreitete Maße, die vor allem zur Schätzung von Wartungsmaßnahmen herangezogen werden. (Vgl. [Wo03], S. 29) Laut Müller sinkt die Wartbarkeit eines Systems, desto größer und komplexer dieses System ist, weshalb beispielsweise das Halstead Volumen dazu geeignet sei, Rückschlüsse auf die Wartbarkeit eines Systems zu schließen. (Vgl. [Mü97], S. 61 f.)

Berücksichtigt werden Schwierigkeit (Difficulty), Umfang (Volume) als auch Aufwand (Efffort). Um die jeweiligen Maße zu berechnen, wird der Quelltext zunächst mittels Ruby-Parse in ein JSON Format überführt, dass sich anschließend besser analysieren lässt. Der Dateiname dieser JSON-Datei wird zufällig erzeugt, um mögliche Manipulationen von außen zu erschweren. Die erzeugte JSON-Datei wird anschließend Zeile für Zeile eingelesen. Beim Einlesen der Zeilen wird für jede Zeile überprüft, ob Operatoren und Operanden enthalten sind, die zuvor in Arrays definiert wurden. Gefundene Operatoren und Operanden werden in separaten Listen gespeichert. Mithilfe dieser Listen können anschließend 2 weitere Listen generiert werden, in denen Duplikate ausgeschlossen wurden. Das Ergebnis sind 4 Listen, in den Operatoren und Operanden des gesamten Quelltexts enthalten sind, einmal mit Duplikaten und einmal ohne Duplikate. Mit diesen Listen werden die eigentlichen Maße berechnet, gemäß ihren mathematischen Definitionen.

5.1.4 McCabe Maße

Die zyklomatische Komplexität wurde in die Prüfkomponente implementiert, weil sie ein gutes Maß für verschiedene Qualitätsmerkmale darstellt. Sie ist ein Maßstab für Verständlichkeit und steht in Bezug zur Testbarkeit und Wartbarkeit eines Programms. (Vgl. [Wi18], S. 26) Außerdem handelt es sich bei diesem Maß um das meist referenzierte Maß in Bezug auf die Komplexität eines Programms. Es wird häufig innerhalb der Industrie und akademischen Lehre verwendet, weil McCabe exakt definiert, was er unter Komplexität versteht und somit eine genaue Bewertung erfolgen kann. Diese Metrik basiert zudem auf mathematischen Theorien der Graphentheorie und wurde durch verschiedene Arbeiten aus Industrie und Wissenschaft empirisch bewiesen. (Vgl. [TAAL08], S. 3) In Kapitel 2.7.3 wurde bereits erwähnt, dass sich die zyklomatische Komplexität erfassen lässt, in dem die Anzahl der Entscheidungen abgezählt wird, die während des Programmablaufs getroffen werden können. Dementsprechend lässt sich der Wert ähnlich ermitteln, wie auch die Halstead Maße und wird auch innerhalb der gleichen Methode berechnet. Hierzu wird in der zuvor erzeugten JSON-Datei nach Programmieranweisungen gesucht, bei denen das Programm eine Entscheidung treffen muss. Bei diesen Anweisungen handelt es sich beispielsweise um if- und while-Anweisungen. Gefundene Anweisungen werden in einem Array gespeichert und gleichzeitig wird ein Wert inkrementiert, wobei bei einem Ausgangswert von 1 gestartet wird. Der Wert, der bei dieser Berechnung letztendlich herauskommt, stellt die zyklomatische Komplexität dar.

5.1.5 Weitere Maße

Auf die Implementierung weiterer Maße wurde aus verschiedenen Gründen verzichtet. Einer dieser Gründe liegt in der Anzahl der existierenden Maße, die so groß ist, dass die Implementierung aller dieser Maße den zeitlichen Rahmen dieser Arbeit sprengen würde. Ein weiterer Grund liegt darin, dass bestimmte Maße für den üblichen Lehrbetrieb ungeeignet sind, hierzu zählen beispielsweise die Maße Mean Time Between Failure (MTBF) und Mean Time To Failure (MTTF), die sich auf die Ausfallzeiten austauschbarer bzw. nicht austauschbarer Einheiten beziehen, die im Lehrbetrieb in dieser Form nicht existieren. Im Lehrbetrieb ist es normalerweise üblich, den Studenten ein Aufgabengerüst zur Verfügung zu stellen. Bei Programmieraufgaben handelt es sich dabei in der Regel um Dateien, in denen Methoden und Klassen bereits vorgegeben wurden. Aus diesem Grund ist die Verwendung von Maßen ungeeignet, die sich auf Beziehungen zwischen Methoden und Klassen beziehen, wie beispielsweise die Reuse Ratio oder Specialization Ratio. Selbst wenn davon ausgegangen wird, dass den Studenten ein solches Gerüst nicht zur Verfügung gestellt wird, kann angenommen werden, dass die Struktur des gesamten Systems nur aus wenigen einzelnen Klassen bestehst und dementsprechend sowieso keine komplexe Vererbungsstruktur entstehen kann, die stark von der Vererbungsstruktur einer Referenzlösung abweicht.

5.2 Bewertungsverfahren

Die Komponente ist nun in der Lage, verschiedene Maße zu berechnen, allerdings fehlt noch ein für die Studenten nachvollziehbares Bewertungsverfahren. Damit die berechneten Maße für die Studenten einen Mehrwert bieten, müssen sie verständlich und vergleichbar sein. Ala-Mutka weist darauf hin, dass es keinen Sinn macht, von Studenten zu verlangen, dass sie ein Programm einreichen, das einen bestimmten Komplexitätswert oder eine bestimmte Anzahl von Zeilen besitzt. (Vgl. [Al05], S. 91 f.)

Dementsprechend wurde ein Bewertungsverfahren implementiert, dass zunächst die Maße für eine Referenzlösung berechnet und diese den berechneten Maßen der studentischen Lösungen gegenüberstellt. Hierdurch entsteht ein faires, verständliches und vergleichbares Bewertungsverfahren, bei der die Leistung des Dozenten mit den Leistungen der Studenten verglichen wird. Diese Vorgehensweise verhindert beispielsweise, dass Ergebnisse von Studenten verlangt werden, die vielleicht von den Dozenten selbst nicht eingehalten werden können. Laut Araujo, Serey und Figueiredo stellt das Verfahren mit einer Referenzlösung die bestmögliche Lösung dar, um die zuvor genannten Probleme zu lösen. (Vgl. [ArSeFi16], S. 3)

Aus der Gegenüberstellung der jeweiligen Werte entstehen neue Werte, die für die eigentliche Bewertung herangezogen werden. Diese Werte geben das Verhältnis wieder, zwischen den erzielten Werten der Referenzlösung und den erzielten Werten der studentischen Lösungen. Ein RLOC Wert von 1,3 würde beispielsweise bedeuten, dass der Quelltext der studentischen Lösung 30 % größer ist als der Quelltext der Referenzlösung.

Tabelle 1 - Messungen zur Bewertung von Codequalität

Bez.	Beschreibung	Formel
RLOC	Verhältnis zwischen LOC der Referenzlösung und dem LOC der studentischen Lösung.	$\frac{LOC\ studentischer\ Lösung}{LOC\ Referenzlösung}$
RNCSS	Verhältnis zwischen NCSS der Referenzlösung und dem NCSS der studentischen Lösung.	$\frac{NCSS\ studentischer\ Lösung}{NCSS\ Referenzlösung}$
RD	Verhältnis zwischen D der Referenzlösung und dem D der studentischen Lösung.	$\frac{D\ studentischer\ Lösung}{D\ Referenzlösung}$
RE	Verhältnis zwischen E der Referenzlösung und dem E der studentischen Lösung.	$\frac{E\ studentischer\ Lösung}{E\ Referenzlösung}$
RV	Verhältnis zwischen V der Referenzlösung und dem V der studentischen Lösung.	$\frac{V\ studentischer\ Lösung}{V\ Referenzlösung}$
RC	Verhältnis zwischen C der Referenzlösung und dem C der studentischen Lösung.	$\frac{C\ studentischer\ Lösung}{C\ Referenzlösung}$

5.2.1 Optionen beim Bewertungsverfahren

In den Optionen des Ruby Softwarequality Checkers (Abbildung 13) finden Lehrkörper für jedes unterstützte Maß drei Einstellungsmöglichkeiten, mit denen sie die Analyse und Bewertung des Checkers ihren Bedürfnissen entsprechend anpassen können.

Generate feedback if [Value] is x or bigger

Der Wert, der an dieser Stelle eingetragen wird, stellt einen Grenzwert dar. Wird dieser Grenzwert erreicht bzw. überschritten, werden Punkte von den 100 erreichbaren Punkten abgezogen und es wird Feedback für das jeweilige Maß generiert. Eine Ausnahme stellt ein Wert kleiner null dar. Wird ein solcher Wert an dieser Stelle eingetragen, dann wird das Maß bei der Analyse und dementsprechend auch bei der Bewertung nicht berücksichtigt. Lehrkörper können somit selbst entscheiden, welche Maße sie für ihre Analyse als sinnvoll erachten.

[Value] Deviation

Dieser Wert stellt die erlaubte Abweichung zwischen Referenzlösung und studentischer Lösung dar. Nehmen wir beispielsweise an, dass für die Einstellungsmöglichkeit „Generate feedback if RLOC is x or bigger" ein Wert von 1,2 gewählt wurde. Nehmen wir weiterhin an, dass der RLOC einer studentischen Lösung 1,5 beträgt, dann hätten wir nun eine Gesamtabweichung von 0,3. Wenn für RLOC Deviation nun der Wert 0,1 gewählt wurde, dann würde das dazu führen, dass drei Abweichungen berechnet werden, nämlich 0,3 / 0,1 = 3. Für diese Abweichungen werden anschließend Punkte abgezogen. Außerdem wurden zuvor einmalig Punkte abgezogen, weil die 1,2 erreicht bzw. überschritten wurden.

[Value] Decrease

An dieser Stelle wird die Punktzahl eingetragen, die für jede gefundene Abweichung abgezogen werden soll. Wenn wir bei dem vorherigen Beispiel bleiben, müssen einmalig Punkte für das Erreichen bzw. überschreiten des Grenzwerts von 1,2 abgezogen werden. Zusätzlich müssen Punkte für die drei weiteren Abweichungen vom Grenzwert abgezogen werden. Wenn an dieser Stelle nun eine 5 eingetragen wird, würde das dazu führen, dass (0,3 / 0,1) * 5 + 5 = 3 * 5 + 5 = 15 + 5 = 20 Punkte abgezogen werden. Wird an dieser Stelle eine null eingetragen, dann hat das zur Folge, dass für die jeweiligen Maße noch Feedback generiert wird, jedoch keinerlei Punkte abgezogen werden. Das ist sinnvoll, wenn Lehrkörper lediglich die Informationen zu einem Maß an ihre Studenten weitergeben möchten, ohne ihre jeweiligen Leistungen in der Bewertung mit einfließen zu lassen.

5.2.2 Feedback Generierung

Immer wenn ein Grenzwert erreicht bzw. überschritten wird, wird für das jeweilige Maß Feedback generiert. In diesem Feedback wird den Studenten erklärt, wie sich ihr erzielter Wert berechnet und in welchen Zusammenhang dieses Maß mit Softwarequalität steht. Außerdem werden sie darüber informiert, welchen Wert die Referenzlösung erzielte, um wie viel Prozent ihr Wert von dieser Lösung abweicht und wie sie ihr Ergebnis zukünftig verbessern können.

5.2.3 Testfälle

Bei Programmieraufgaben ist es üblich, kleine Testfälle innerhalb des zu bearbeitenden Quelltexts zu definieren, mit denen die zu implementierende Logik schnell überprüft werden kann. Dabei handelt es sich in der Regel um Methodenaufrufe mit bestimmten Parametern, für die bereits die dazugehörigen Ausgaben bekannt sind, weshalb sie als Vergleich herangezogen werden können. Die Anzahl solcher Testfälle kann stark variieren, einige Menschen verzichten völlig darauf, andere Menschen definieren sich Dutzende solcher kleinen Testfälle, um ihre Programme bereits während der Entwicklung grob testen zu können. Dieser Umstand hat zur Folge, dass die Ergebnisse der jeweiligen Maße verfälscht werden können.

Ein Beispiel hierfür wäre das NCSS Maß, das von der Anzahl echter Codezeilen abhängig ist, die wiederum mit der Anzahl definierter Testfälle zunimmt. Um zu verhindern, dass diese Testfälle in die Analyse mit einfließen, wurde ein Tag definiert. Dieser Tag lautet „# $$ Test $$" und veranlasst die Komponente dazu, alles was nach diesem Tag folgt von der Analyse auszuschließen, einschließlich des Tags selbst. In Abbildung 14 ist ein solcher Tag zu erkennen, in diesem Fall wird alles ab Zeile 23 von der Analyse ausgeschlossen.

6 Tests für die erweiterte Prüfkomponente

Um die neue Prüfkomponente zu testen, wurden verschiedene Programmieraufgaben verfasst. Bei diesen Aufgaben handelt es sich um typische Programmieraufgaben aus dem Übungsbetrieb. Alle diese Aufgaben bestehen aus einer Aufgabenstellung im PDF-Format, Punkteliste im JSON-Format, Referenzlösung, Worksheet-Datei im Ruby-Format und einem Testdokument in einem RSpec kompatiblen Format. Eine dieser Aufgaben wird in diesem Kapitel als Beispiel genutzt. Bei dieser Aufgabe müssen die Studenten die Logik einer Ruby Methode definieren. Dieser Methode wird eine Jahreszahl als Parameter übergeben, anschließend soll sie prüfen, ob es sich bei dieser übergebenen Jahreszahl um ein Schaltjahr handelt.

6.1 Verwendete Einstellungen

6.1.1 Dynamische Ruby Checker Einstellungen

Um den dynamischen Ruby Checker betriebsfähig zu machen, wurde den Ressourcen eine Worksheet-Datei (Abbildung 14), ein Testdokument (Abbildung 11) und eine Punkteliste (Abbildung 12) hinzugefügt. Die Punkteliste enthält die zu vergebenen Punkte für korrekt gelöste Aufgaben. Das Worksheet stellt die Datei dar, die von den Studenten bearbeitet werden muss und die anschließend von der Komponente analysiert wird. Im Testdokument finden sich verschiedene Testfälle in einem RSpec kompatiblen Format wieder, mit denen die korrekte Funktionalität der zu lösenden Aufgabe überprüft wird.

6.1.2 Statische Ruby Checker Einstellungen

Die statische Komponente (Abbildung 9) wurde so eingestellt, dass sie sowohl Verstöße gegen Konventionen als auch Warnungen und Fehlermeldungen berücksichtigt. Für Fehlermeldungen werden 100 Punkte abgezogen, weil das Programm in diesem Fall nicht kompilierbar ist. Für Warnungen und Verstöße gegen Konventionen wird jeweils 1 Punkt abgezogen. Beim Worksheet handelt es sich um das gleiche Worksheet, was auch bereits beim dynamischen Checker verwendet wurde.

6.1.3 Ruby Softwarequality Checker Einstellungen

Diese Komponente (Abbildung 13) wurde so konfiguriert, dass alle unterstützten Metriken bei der Analyse berücksichtigt werden. Für alle Maße gilt ein Grenzwert von 1,2. Für jede Abweichung von 0,1 wird ein Punkt abgezogen. Hiervon ausgeschlossen ist der RLOC Wert, Feedback wird zwar erzeugt, allerdings keine Punkte abgezogen, weil bereits bei RNCSS Punkte abgezogen werden und es sich hierbei um ein ähnliches Maß handelt. Weiterhin wurde eine Referenzlösung (Abbildung 15) angegeben, die für den Vergleich mit den studentischen Lösungen genutzt wird.

6.2 Worksheet

Bei dem Worksheet aus Abbildung 14 handelt es sich um die eigentliche Aufgabenstellung. Diese Datei muss von den Studenten heruntergeladen werden und enthält bereits ein Gerüst, in dem Bezeichnungen für Klassen und Methoden vorgegeben wurden. Im Lehrbetrieb ist es üblich, ein solches Gerüst vorzugeben, sodass die Studenten nur noch die eigentliche Logik implementieren müssen.

In diesem Beispiel muss die Logik der Methode schaltjahr(jahr) implementiert werden, wobei für den Parameter „jahr" eine Jahreszahl übergeben wird. Die Methode soll anschließend einen booleschen Wert „true" zurückgeben, sofern es sich um ein Schaltjahr handelt, andernfalls soll sie „false" zurückgeben. Sobald diese Logik implementiert wurde, muss sie innerhalb der JACK Plattform als Lösung für diese Aufgabe eingereicht werden. Anschließend erfolgt die Analyse durch jeweiligen Prüfkomponenten.

schaltjahr_worksheet.rb

```ruby
# Dies Methode schaltjahr(jahr) soll zurueckgeben, ob es sich bei dem gegebenen Jahr um ein Schaltjahr handelt.
# Die Erde braucht fuer die Umrundung der Sonne etwa 365 Tage, 5 Stunden, 49 Minuten und 12 Sekunden.
# 1 Jahr besteht dementsprechend nicht aus exakt 365 Tagen, wird aber trotzdem so berechnet.
# Das fuehrt zu einer Verschiebung, die durch Schaltjahre mit 366 Tagen ausgeglichen wird.

# Jedes Jahr, das sich durch 4 teilen laesst, ist ein Schaltjahr.
# Alle 100 Jahre faellt ein Schaltjahr aus, dementsprechend ist ein Schaltjahr kein Schaltjahr,
# wenn es sich durch 100 teilen laesst. Alle 400 Jahre faellt ein Schaltjahr allerdings statt, ohne auszufallen.

# Beispiele:
# 2016 ist ein Schaltjahr
# 2004 ist ein Schaltjahr
# 2000 ist ein Schaltjahr
# 2015 ist kein Schaltjahr
# 2003 ist kein Schaltjahr
# 1900 ist kein Schaltjahr

class Schaltjahr
  def schaltjahr(jahr)
  end
end

# $$ Test $$
object = Schaltjahr.new
puts object.schaltjahr(2016)
puts object.schaltjahr(2004)
puts object.schaltjahr(2000)
puts object.schaltjahr(2015)
puts object.schaltjahr(2003)
puts object.schaltjahr(1900)
```

Abbildung 14 - Schaltjahr Worksheet

6.3 Referenzlösung und studentische Lösung

Die Komponente ermittelt anhand folgender Referenzlösung aus Abbildung 15 die Maße, die den studentischen Maßen gegenübergestellt werden.

```
18  class Schaltjahr
19    def schaltjahr(jahr)
20      if jahr%4==0 and jahr%100!=0 or jahr%400==0
21        return true
22      else
23        return false
24      end
25    end
26  end
27
```

Abbildung 15 - Schaltjahr Referenzlösung

Der Quelltext aus Abbildung 16 stellt eine fiktive studentische Lösung für die Schaltjahraufgabe dar. In diesem Fall wurde die gesamte Logik ausschließlich mit if-Anweisungen realisiert.

```
18  class Schaltjahr
19    def schaltjahr(jahr)
20      if jahr%4==0
21        if jahr%100==0
22          if jahr%400==0
23            return true
24          end
25          if jahr%400!=0
26            return false
27          end
28        end
29        if jahr%100!=0
30          return true
31        end
32      end
33      if jahr%4!=0
34        return false
35      end
36    end
37  end
38
```

Abbildung 16 - Schaltjahr Studentische Lösung

6.4 Ergebnisse der Analyse

6.4.1 Ergebnisse des dynamischen Checkers

Die Lösung des Studenten erfüllt die gewünschte Funktionalität, weshalb die dynamische Prüfkomponente keine Fehler feststellen konnte, was in Abbildung 17 ersichtlich wird.

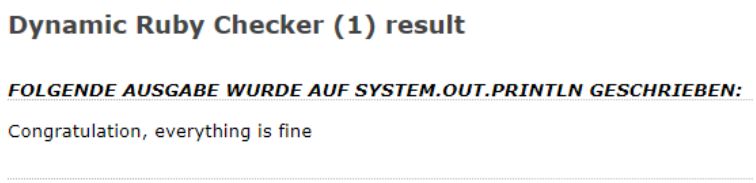

Abbildung 17 - Ergebnisse dynamischer Checker

6.4.2 Ergebnisse des statischen Checkers

Die statische Prüfkomponente konnte keine Fehlermeldungen oder Warnungen feststellen, jedoch Verstöße gegen die Ruby Konventionen. Bei diesen Verstößen handelt es sich beispielsweise um fehlende Leerzeichen zwischen verschiedenen Operatoren und Codezeilen mit zu vielen Zeichen. Einige dieser Verstöße sind in Abbildung 18 ersichtlich, allerdings handelt es sich dabei nur um eine Teilmenge aller gefundenen Verstöße.

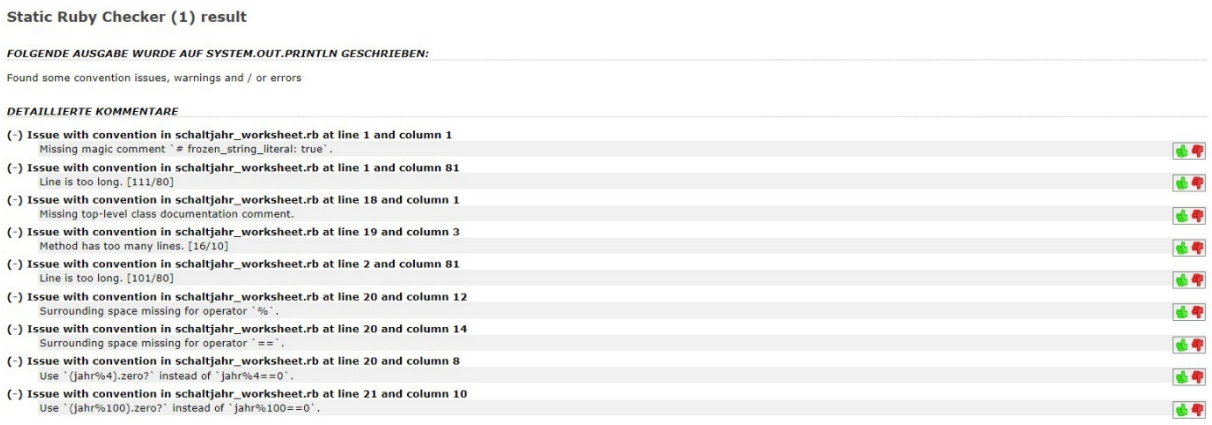

Abbildung 18 - Ergebnisse statischer Checker

6.4.3 Ergebnisse des Ruby-Softwarequality-Checkers

Der Ruby-Softwarequality-Checker konnte feststellen, dass die studentische Lösung bei allen Maßen schlechter abschneidet als die Referenzlösung. Die studentische Lösung besteht aus insgesamt 47 Codezeilen, wobei die ersten 17 Codezeilen aus Kommentaren und die letzten 10 Codezeilen aus einer Leerzeile und 9 Testzeilen bestehen. Für die Maße LOC und NCSS ergeben sich somit folgende Werte:

$$Lines\ Of\ Code\ (LOC) = 47 - 9 = 38$$
$$Non\ Commented\ Source\ Statements(NCSS) = 47 - 17 - 10 = 20$$

Die Referenzlösung hingegen besteht aus insgesamt 36 Codezeilen, wobei die ersten 17 Codezeilen aus Kommentaren und die letzten 10 Codezeilen aus einer Leerzeile und 9 Testzeilen bestehen. Für die Maße LOC und NCSS ergeben sich somit folgende Werte:

$$Lines\ Of\ Code\ (LOC) = 36 - 9 = 27$$
$$Non\ Commented\ Source\ Statements\ (NCSS) = 36 - 17 - 10 = 9$$

Die Referenzlösung löst das Problem dementsprechend mit weniger Codezeilen und erzielt dadurch bessere Werte bei LOC und NCSS. Werden die Ergebnisse gegenübergestellt, ergeben sich folgende RLOC und RNCSS Werte:

$$RLOC = \frac{38}{27} = 1{,}41\ ;\ RNCSS = \frac{20}{9} = 2{,}22$$

Daraus folgt, dass die studentische Lösung im Vergleich zur Referenzlösung einen 41 % größeren LOC Wert und einen 122 % größeren NCSS Wert vorweist. Somit wurde der zuvor für beide Maße eingestellte Grenzwert von 1,2 überschritten, was dazu führt, dass einerseits Feedback produziert und andererseits Punkte abgezogen werden. Daraus ergeben sich folgende Punktabzüge:

$$Anzahl\ Punktabz\ddot{u}ge\ RLOC = \frac{RLOC - RLOC\ Grenzwert}{RLOC\ Deviation} = \frac{1{,}41 - 1{,}2}{0{,}1} = 2{,}1 = 2$$
$$Punktabz\ddot{u}ge\ RLOC = RLOC\ Decrease + Anzahl\ Punktabz\ddot{u}ge\ RLOC * RLOC\ Decrease$$
$$Punktabz\ddot{u}ge\ RLOC = 0 + 2 * 0 = 0$$

$$Anzahl\ Punktabz\ddot{u}ge\ RNCSS = \frac{RNCSS - RNCSS\ Grenzwert}{RNCSS\ Deviation} = \frac{2{,}22 - 1{,}2}{0{,}1} = 10{,}2 = 10$$
$$Punktabz\ddot{u}ge\ RNCSS = RNCSS\ Decrease + Anzahl\ Punktabz\ddot{u}ge\ RNCSS * RNCSS\ Decrease$$
$$Punktabz\ddot{u}ge\ RNCSS = 1 + 10 * 1 = 11$$

Weiterhin besteht die studentische Lösung aus 6 verschiedenen Operatoren (n1) und Operanden (n2). Insgesamt wurden 23 Operatoren (N1) und 20 Operanden (N2) verwendet. N ist die Summe aller verwendeten Operatoren und Operanden, dementsprechend ist N = 23 + 20 = 43. Weiterhin ist n die Summe aller verschiedener Operatoren und Operanden, dementsprechend ist n = 6 + 6 = 12. Für die Halstead-Maße ergeben sich somit folgende Werte:

$$Halstead\ Volume\ (V) = N * \log_2(n) = 43 * \log_2(12) = 154{,}15$$
$$Halstead\ Difficulty\ (D) = \left(\frac{n1}{2}\right) * \left(\frac{N2}{n2}\right) = \frac{6}{2} * \frac{20}{6} = 10$$
$$Halstead\ Effort\ (E) = D * V = 10 * 154{,}15 = 1541{,}5$$

Die Referenzlösung löst das Problem ebenfalls mit 6 verschiedenen Operatoren (n1) und Operanden (n2). Insgesamt greift die Referenzlösung allerdings auf nur 10 Operatoren (N1) und 11 Operanden (N2) zurück. Für die Halstead-Maße ergeben sich somit folgende Werte:

$$Halstead\ Volume\ (V) = N * \log_2(n) = 21 * \log_2(12) = 75{,}28$$
$$Halstead\ Difficulty\ (D) = \left(\frac{n1}{2}\right) * \left(\frac{N2}{n2}\right) = \frac{6}{2} * \frac{11}{6} = 5{,}5$$
$$Halstead\ Effort\ (E) = D * V = 5{,}5 * 75{,}28 = 414{,}04$$

Werden die Ergebnisse gegenübergestellt, ergeben sich somit folgende RV, RD und RE Werte:

$$RV = \frac{154{,}15}{75{,}28} = 2{,}05\ ; RD = \frac{10}{5{,}5} = 1{,}82\ ; RE = \frac{1541{,}5}{414{,}04} = 3{,}72$$

Daraus folgt, dass die studentische Lösung im Vergleich zur Referenzlösung einen 105 % größeren V-Wert, 82 % größeren D-Wert und 272 % größeren E-Wert erreicht. Dementsprechend wurde auch hier der Grenzwert für die jeweiligen Maße von 1,2 überschritten, was wiederum dazu führt, dass Feedback produziert und Punkte abgezogen werden. Die Punktabzüge ergeben sich wie folgt:

$$Anzahl\ Punktabz\ddot{u}ge\ RV = \frac{RV - RV\ Grenzwert}{RV\ Deviation} = \frac{2{,}05 - 1{,}2}{0{,}1} = 8{,}5 = 8$$
$$Punktabz\ddot{u}ge\ RV = RV\ Decrease + Anzahl\ Punktabz\ddot{u}ge\ RV * RV\ Decrease$$
$$Punktabz\ddot{u}ge\ RV = 1 + 8 * 1 = 1 + 8 = 9$$

$$Anzahl\ Punktabz\ddot{u}ge\ RD = \frac{RD - RD\ Grenzwert}{RD\ Deviation} = \frac{1{,}82 - 1{,}2}{0{,}1} = 6{,}2 = 6$$
$$Punktabz\ddot{u}ge\ RD = RD\ Decrease + Anzahl\ Punktabz\ddot{u}ge\ RD * RD\ Decrease$$
$$Punktabz\ddot{u}ge\ RV = 1 + 6 * 1 = 1 + 6 = 7$$

$$Anzahl\ Punktabz\ddot{u}ge\ RE = \frac{RE - RE\ Grenzwert}{RE\ Deviation} = \frac{3{,}72 - 1{,}2}{0{,}1} = 25{,}2 = 25$$
$$Punktabz\ddot{u}ge\ RE = RE\ Decrease + Anzahl\ Punktabz\ddot{u}ge\ RE * RE\ Decrease$$
$$Punktabz\ddot{u}ge\ RV = 1 + 25 * 1 = 1 + 25 = 26$$

Die zyklomatische Komplexität ergibt sich aus der Anzahl verwendeter Anweisungen, bei denen Entscheidungen getroffen werden müssen. Bei der studentischen Lösung wurden 6 solcher Anweisungen verwendet, wobei es sich bei allen dieser Anweisungen um If-Anweisungen handelt, dementsprechend beträgt die zyklomatische Komplexität der studentischen Lösung 6. Die Referenzlösung verwendet hingegen 3 solcher Anweisungen, dabei handelt es sich um If, And und Or. Somit beträgt die zyklomatische Komplexität der Referenzlösung 3. Werden diese Werte gegenübergestellt, ergibt sich folgender RC-Wert:

$$RC = \frac{6}{3} = 2$$

Das bedeutet, dass die zyklomatische Komplexität der studentischen Lösung doppelt so groß ist, wie die zyklomatische Komplexität der Referenzlösung. Somit wurde auch hier der Grenzwert von 1,2 überschritten, weshalb Feedback generiert und Punkte abgezogen werden. Die Punktabzüge ergeben sich in diesem Fall wie folgt:

$$Anzahl\ Punktabz\ddot{u}ge\ RC = \frac{RC - RC\ Grenzwert}{RC\ Deviation} = \frac{2 - 1{,}2}{0{,}1} = 8$$
$$Punktabz\ddot{u}ge\ RC = RC\ Decrease + Anzahl\ Punktabz\ddot{u}ge\ RC * RC\ Decrease$$
$$Punktabz\ddot{u}ge\ RV = 1 + 8 * 1 = 1 + 8 = 9$$

Die Gesamtpunktzahl des Ruby-Softwarequality-Checkers ergibt sich aus einer Höchstpunktzahl von 100 Punkten, abzüglich der summierten Fehlerpunkte. Die Gesamtpunktzahl ergibt sich also wie folgt:

$$Gesamtpunktzahl = 100 - 0\ - 11 - 9 - 7 - 26 - 9 = 100 - 62 = 38$$

Es lässt sich festhalten, dass die Komponente die Lösung korrekt analysieren und für die jeweiligen Maße die passenden Werte berechnen konnte. Auch die Generierung von Feedback und die eigentliche Beurteilung durch Punktabzüge ist korrekt erfolgt. Die genauen Ausgaben für die jeweiligen Maße sind in Abbildung 19 zu erkennen.

Ruby Softwarequality Checker (1) result

FOLGENDE AUSGABE WURDE AUF SYSTEM.OUT.PRINTLN GESCHRIEBEN:

The following metric values are worse than those of the reference solution: Lines Of Code (LOC), Non Commented Source Statements (NCSS), Halstead Difficulty (D), Halstead Volume (V), Halstead Effort (E), Cyclomatic Complexity (C),

DETAILLIERTE KOMMENTARE

(-) Cyclomatic Complexity Information for schaltjahr_worksheet.rb

Cyclomatic complexity (C) is a McCabe metric value which focus on the structure complexity of a program.
We count decision points like if statements to calculate this metric value.

Your C: 6.0 [if, if, if, if, if, if]
Reference C: 3.0 [if, or, and]

Feedback:
Your C value is 100% bigger compared with reference solution.
So it appears that your program has too many conditional structures or loops.

(-) Halstead Difficulty Information for schaltjahr_worksheet.rb

Halstead Difficulty (D) is a halstead metric value which stands for the difficulty to understand a program.
We calculated the following values to calculate the halstead difficulty metric:
n1 = number of different operators used.
n2 = number of different operands used.
N2 = number of all operands used.
D = (n1 / 2) * (N2 / n2).

Your solution:
Different operators used (n1): 6 [def, if, %, ==, return, !=]
Different operands used (n2): 6 [schaltjahr, jahr, 4, 0, 100, 400]
All operands used (N2): 20 [schaltjahr, jahr, jahr, 4, 0, jahr, 100, 0, jahr, 400, 0, jahr, 400, 0, jahr, 100, 0, jahr, 4, 0]
Your D: (6 / 2) * (20 / 6) = 10.0

Reference Solution:
Different operators used (n1): 6 [def, if, %, ==, !=, return]
Different operands used (n2): 6 ([schaltjahr, jahr, 4, 0, 100, 400])
All operands used: 11 [schaltjahr, jahr, jahr, 4, 0, jahr, 100, 0, jahr, 400, 0]
Reference D: (6 / 2) * (11 / 6) = 5.5

Feedback:
Your D value is 82% bigger compared with D value of reference solution.
You can reduce Halstead Difficulty by using fewer operators and operands.

(-) Halstead Effort Information for schaltjahr_worksheet.rb

Halstead Effort (E) is a Halstead metric value which stands for the effort to understand a program.
We calculated the following values to calculate the halstead effort metric:
n1 = number of different operators used.
n2 = number of different operands used.
n = n1 + n2.
N1 = number of all used operators.
N2 = number of all used opeprands.
N = N1 + N2.
D = (n1 / 2) * (N2 / n2).
V = N * log2(n).
E = D * V.

Your solution:
Different operators used (n1): 6 [def, if, %, ==, return, !=]
Different operands used (n2): 6 [schaltjahr, jahr, 4, 0, 100, 400]
n = 6 + 6 = 12
All operators used (N1): 23 [def, if, %, ==, if, %, ==, if, %, ==, return, if, %, !=, return, if, %, !=, return, if, %, !=, return]
All operands used (N2): 20 [schaltjahr, jahr, jahr, 4, 0, jahr, 100, 0, jahr, 400, 0, jahr, 400, 0, jahr, 100, 0, jahr, 4, 0]
N = 23 + 20 = 43
D = (6 / 2) * (20 / 6) = 10.0
V = 43 * log2(12) = 154.15
Your E: 10.0 * 154.15 = 1541.5
Your Time to implement (T) is E / 18 = 85.63888888888889

Reference solution:
Different operators used (n1): 6 [def, if, %, ==, !=, return]
Different operands used (n2): 6 [schaltjahr, jahr, 4, 0, 100, 400]
n = 6 + 6 = 12
All operators used (N1): 10 [def, if, %, ==, %, !=, %, ==, return, return]
All operands used (N2): 11 [schaltjahr, jahr, jahr, 4, 0, jahr, 100, 0, jahr, 400, 0]
N =10 + 11 = 21
D = (6 / 2) * (11 / 6) = 5.5
V = 21 * log2(12) = 75.28
Reference E: 5.5 * 75.28 = 414.04
Reference time to implement (T) is E / 18 = 23.002222222222223

Feedback:
Your E value is 272% bigger compared with E value of reference solution.
You can reduce Halstead Effort by using fewer operators and operands.

(-) Halstead Volume Information for schaltjahr_worksheet.rb

Halstead Volume (V) is a halstead metric value which stands for the scope of a program.
You can also say it is a value which stands for the size of the algorithm in bits.
We calculated the following values to calculate the halstead volume metric:

n1 = number of different operators used.
n2 = number of different operands used.
n = n1 + n2
N1 = number of all used operators.
N2 = number of all used operands.
N = N1 + N2.
V = N * log2(n).

Your solution:
Different operators used (n1): 6 [def, if, %, ==, return, !=]
Different operands used (n2): 6 [schaltjahr, jahr, 4, 0, 100, 400]
n = 6 + 6 = 12
All operators used (N1): 23 [def, if, %, ==, if, %, ==, if, %, ==, return, if, %, !=, return, if, %, !=, return, if, %, !=, return]
All operands used (N2): 20 [schaltjahr, jahr, jahr, 4, 0, jahr, 100, 0, jahr, 400, 0, jahr, 400, 0, jahr, 100, 0, jahr, 4, 0]
N = 23 + 20 = 43
Your V: 43 * log2(12) = 154.15

Reference solution:
Different operators used (n1): 6 [def, if, %, ==, !=, return]
Different operands used (n2): 6 [schaltjahr, jahr, 4, 0, 100, 400]
n = 6 + 6 = 12
All operators used (N1): 10 [def, if, %, ==, %, !=, %, ==, return, return]
All operands used (N2): 11 [schaltjahr, jahr, jahr, 4, 0, jahr, 100, 0, jahr, 400, 0]
N = 10 + 11 = 21
Reference V: 21 * log2(12) = 75.28

Feedback:
Your V value is 105% bigger compared with V value of reference solution.
So it appears that your program has too many operations.

(-) LOC Information for schaltjahr_worksheet.rb

Lines of Code (LOC) is a value which stands for the number of lines used in your solution include comments and blank lines. We calculated the LOC value of your solution and compared it with the LOC value of a reference solution.

Your LOC: 38
Reference LOC: 27

Feedback:
Your LOC is 41% bigger compared with LOC of reference solutions.
So it appears that your program has to many lines of code. Try to remove not necessary lines

(-) NCSS Information for schaltjahr_worksheet.rb

Non-commented source statements (NCSS) is a value which stands for the number of lines used in your solution exclude comments and blank lines.
So it is the number of lines of your real code statements.
We calculated the NCSS value of your solution and compared it with the NCSS value of a reference solution.

Your NCSS: 20
Reference NCSS: 9

Feedback:
Your NCSS is 122% bigger compared with NCSS of reference solution.
So it appears that your program has too many lines of code. Try to remove not necessary lines

Abbildung 19 - Ergebnisse des Ruby Softwarequality Checkers

6.4.4 Gesamtergebnis und Bewertung

Die Gesamtpunktzahl eines Studenten ergibt sich aus der Gewichtung der einzelnen Checker Komponenten. Diese kann zuvor innerhalb der JACK Plattform bei der Aufgabenerstellung eingestellt werden. In diesem Beispiel wurde der dynamischen Komponente eine Gewichtung von 60 % zugeteilt, der statischen Komponente 10 % und dem Ruby Softwarequality Checker 30 %, siehe Abbildung 20.

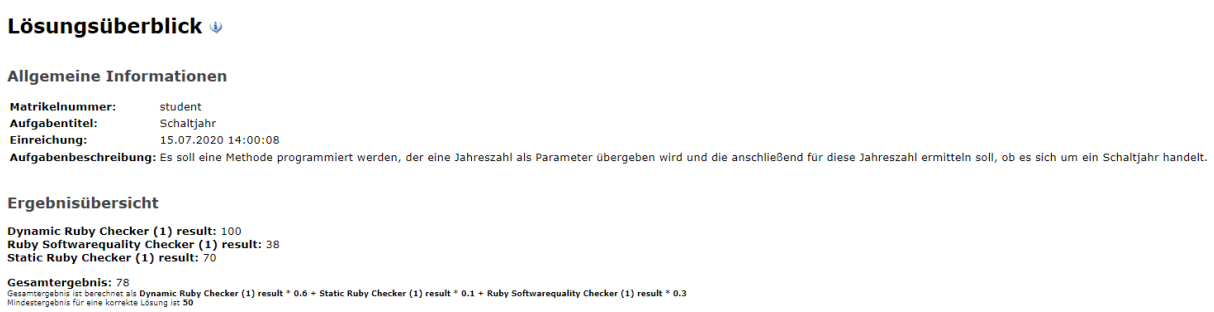

Abbildung 20 – Gesamtergebnis

6.5 Alternative Lösung

Die folgende Lösung aus Abbildung 21 löst das Problem mit nur einer einzigen Codezeile. In diesem Fall wurden alle Anweisungen in einer Zeile definiert und alle Kommentare entfernt, was dazu führt, dass diese Lösung bessere LOC- und NCSS-Werte erreicht als die Referenzlösung. Das zeigt auch zugleich die Schwäche von LOC und NCSS. Durch das Entfernen der Kommentare und durch die Nutzung einer eher unleserlichen Struktur, wird die Interpretation des Quelltexts und somit die Wartbarkeit eigentlich erschwert, dementsprechend stellt die Lösung eigentlich keine bessere Lösung als die Referenzlösung dar, wird aber trotzdem so behandelt.

Es lässt sich also festhalten, dass kompaktere Lösungen nicht immer die besseren Lösungen darstellen, auch wenn sie bessere Werte bei verschiedenen Maßen erreichen. Weiterhin lässt sich festhalten, dass sich die Werte der Halstead Maße und der zyklomatischen Komplexität nicht durch eine andere Schreibweise manipulieren lassen, weil sie sich auf die generelle Struktur eines Programms beziehen. Sie stellen somit definitiv bessere Maße für den Lehrbetrieb dar.

```
1  class Schaltjahr def schaltjahr(jahr) if jahr%4==0 and jahr%100!=0 or jahr%400==0; return true else return false end end end
```

Abbildung 21 - Alternative Lösung

7 Ausblick

In dieser Bachelor-Arbeit wurde die zuvor entwickelte Ruby Prüfkomponente, die sich bisher aus zwei separaten Checker-Komponenten zusammensetzte, um eine dritte Checker-Komponente erweitert. Mithilfe dieser neuen Komponente können nun auch qualitative Softwareaspekte bei der Analyse eines Ruby Quelltexts berücksichtigt werden. Diese Aspekte werden durch bestimmte Maße definiert, die einen Bezug zu Qualitätsmerkmalen herstellen. Aufgrund der Vielzahl von Maßeinheiten musste innerhalb dieser Arbeit allerdings eine Auswahl getroffen werden, was dazu führt, dass bisher noch nicht alle Maße von der Komponente unterstützt werden. Das bedeutet auch, dass noch zahlreiche Qualitätsmerkmale existieren, die ebenfalls unberücksichtigt bleiben. Durch die Implementierung weiterer Maße kann dementsprechend ein Bezug zu den bisher unberücksichtigten Qualitätsmerkmalen hergestellt werden.

Bisher werden die Ergebnisse ausschließlich in schriftlicher Form repräsentiert. Auch an dieser Stelle kann für eine Erweiterung angesetzt werden, indem Visualisierungen für die Ergebnisse implementiert werden. Beispielsweise könnten das verschiedene Formen von Diagrammen sein, in denen die Ergebnisse der Referenzlösungen und der studentischen Lösungen visuell gegenübergestellt werden. Auch UML-Diagramme wären eine Möglichkeit, mit denen die Komponente erweitert werden kann. So könnten beispielsweise Referenzlösung als auch studentische Lösung in Aktivitätsdiagramme konvertiert werden.

Eine weitere sinnvolle Erweiterung wäre die Implementierung einer graphbasierten Analyse, mit der sich Aufgabenstellungen dynamischer gestalten lassen. Mithilfe einer solchen Analyse lassen sich spezielle Lösungsansätze voraussetzen und überprüfen. So könnte beispielsweise vorausgesetzt werden, dass eine Aufgabe ausschließlich mittels Rekursion gelöst werden soll. Durch die graphbasierte Analyse wäre es möglich zu überprüfen, ob diese Aufgabe auch tatsächlich mittels Rekursion gelöst wurde. Eine solche Komponente wurde auch bereits in JACK implementiert und trägt den Titel GReQLJavaChecker. Wie der Name bereits vermuten lässt, bezieht sich diese Komponente allerdings ausschließlich auf die Programmiersprache Java und ist dementsprechend für Ruby Quelltexte ungeeignet.

Das Verfahren könnte auch auf andere Programmiersprachen übertragen werden, weil die Maße auf Elementen basieren, die sich in jeder Programmiersprache wiederfinden, wie beispielsweise Codezeilen, die für die Maße LOC und NCSS verwendet werden. Die Erweiterungsmöglichkeiten dieser Prüfkomponente sind dementsprechend noch lange nicht ausgeschöpft. Die Komponente reicht allerdings bereits aus, um neben dynamischen Testverfahren auch statische Testverfahren durchzuführen, bei denen insbesondere auch Softwarequalitätsaspekte berücksichtigt werden.

8 Fazit

In dieser Bachelor-Arbeit konnte die Ruby Prüfkomponente für das E-Assessment-System JACK erfolgreich erweitert werden, dahingehend, dass die Komponente nun auch qualitative Softwareaspekte bei ihrer Analyse und Bewertung berücksichtigen kann. Zunächst wurde auf grundlegende Themen der Softwarequalität eingegangen. Hierbei stellte sich heraus, dass mittels verschiedener Qualitätsmodelle, Qualitätsmerkmale und Maße versucht wird, den abstrakten Begriff der Qualität fassbar und messbar zu machen.

Anschließend wurden Aufbau und Funktionalität der zu erweiternden Ruby Prüfkomponente erläutert. Hierbei stellte sich heraus, dass die Komponente mit ihren statischen und dynamischen Testverfahren bereits einige Aspekte der Softwarequalität bei ihrer Analyse berücksichtigt. Bei diesen Aspekten handelt es sich einerseits um die Wartbarkeit, die durch die statische Komponente überprüft wird und andererseits um die Vollständigkeit, die durch die dynamische Komponente überprüft wird.

Im Anschluss daran wurde sich mit der Implementierung der Erweiterung befasst. Es wurde eine weitere Prüfkomponente entwickelt, die sich ausschließlich mit der Erfassung von Maßen befasst, mit denen Rückschlüsse auf die Softwarequalität geschlossen werden können. Bei dem Bewertungsverfahren dieser neuen Komponente stellte sich heraus, dass in diesem Fall eine faire Bewertung nur dann erfolgen kann, wenn die berechneten Maße der Studenten, den berechneten Maßen einer Referenzlösung gegenübergestellt werden. Weiterhin wurde erläutert, welche Einstellungsmöglichkeiten die Komponente bietet und wie sich diese generell bedienen lässt.

Zuletzt wurde die Prüfkomponente erfolgreich getestet. Hierzu wurde ein Ruby Einstiegskurs mit Aufgaben aus verschiedenen Themengebieten genutzt. Das Ergebnis einer solchen Aufgabe wurde anschließend innerhalb dieser Arbeit schriftlich festgehalten und erläutert. Es lässt sich letztendlich festhalten, dass die Ruby Prüfkomponente erfolgreich erweitert werden konnte, dahingehend, dass sie nun auch qualitative Softwareaspekte bei ihrer Analyse berücksichtigen kann. Ermöglicht wurde das vor allem durch die Erkenntnis, dass mittels verschiedener Maße Rückschlüsse auf Qualitätsmerkmale gezogen werden können und dass sich die Berechnung dieser Maße automatisieren lässt. Wir konnten außerdem feststellen, dass sich das in dieser Prüfkomponente verwendete Verfahren auf andere Programmiersprachen übertragen lässt, weil viele dieser Maße auf Elementen basieren, die sich in anderen Programmiersprachen wiederfinden.

Literatur

[Al05] Kirsti M. Ala-Mutka: A Survey of Automated Assessment Approaches for Programming Assignments. Computer science education, vol. 15, 2005.

[Al10] Rafa E. Al-Qutaish: Quality Models in Software Engineering Literature: An Analytical and Comparative Study. Journal of American Science, 2010.

[AMA13] Thamer A. Alrawashdeh, Mohammad Muhairat, Ahmad Althunibat: Evaluating the Quality of Software in ERP Systems Using the ISO 9126 Model. International Journal of Ambient Systems and Applications (IJASA) Vol. 1, No. 1, März 2013.

[ArSeFi16] Eliane Araujo, Dalton Serey, Jorge Figueiredo: Qualitative aspects of students' programs: Can we make them measurable? IEEE Frontiers in Education Conference (FIE), Oktober 2016.

[Ba20] Bozhidar Batsov: The Ruby Style Guide.
https://rubystyle.guide/
Zuletzt aktualisiert am 31. Mai 2020.
Abgerufen am 03. Juli 2020.

[BhTaPa14] Kaushal Bhatt, Vinit Tarey, Pushpraj Patel: Analysis Of Source Lines Of Code (SLOC) Metric. International Journal of Emerging Technology and Advanced Engineering, ISSN 2250-2459, Volume 2, Issue 5, Mai 2012.

[BrSc07] Manfred Broy, Bernhard Schätz: Softwarearchitekturen und modellgetriebene Softwareentwicklung – Qualitätsmodelle. Technische Universität München, Fakultät für Informatik, 2007.

[Gl05] Wolfgang Globke: Software-Metriken. Seminar "Moderne Softwareentwicklung", Karlsruher Institut für Technologie, 2005.

[Gr06] Hans-Gert Gräbe: Software-Qualitätsmanagement – Kernfach Angewandte Informatik. Universität Leipzig, Institut für Informatik, Betriebliche Informationssysteme, 2006.

[Gr09] Susanne Johanna Gruttmann: Formatives E-Assessment in der Hochschullehre – Computerunterstützte Lernfortschrittskontrollen im Informatikstudium. Inaugural-Dissertation zur Erlangung des Doktorgrades der Naturwissenschaften im Fachbereich Mathematik und Informatik der Mathematisch-Naturwissenschaftlichen Fakultät der Westfälischen Wilhelms-Universität Münster, 2009.

[HHST03] Colin Higgins, Tarek Hegazy, Pavlos Symeonidis, Athanasios Tsintsifas: The CourseMarker CBA System: Improvements over Ceilidh. Education and Information Technologies, 8(3):287-304, 2003.

[IE98] Institute of Electrical and Electronics Engineers: Std 1061-1998. IEEE Standard for a Software Quality Metrics Methodology. IEEE, New York, 1998.

[La10] Clara Lange: Softwarequalitätsmodelle. Technische Universität München, Fakultät für Informatik, 2010.

[Li09] Peter Liggesmeyer: Software-Qualität – Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag Heidelberg, 2. Auflage, 2009.

[Mü97] Bernd Müller: Reengineering – Eine Einführung. B. G. Teubner, Stuttgart, 1997.

[ObRe16] Andreas Oberweis, Ralf Reussner: Modellierung 2016. Lecture Note in Informatics (LNI), Gesellschaft für Informatik, Bonn, 2016.

[Pa13] Paluno – The Ruhr Institute for Software Technology: Grundlagen des Software Engineering – Kapitel 2: Eigenschaften und Prinzipien von Software. Universität Duisburg-Essen, 2013.

[Schn12] Kurt Schneider: Abenteuer Software Qualität – Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement. dpunkt.verlag GmbH, 2. überarbeitete und erweiterte Auflage, 2012.

[ScHo14] Markus Schmees, Janine Horn: E-Assessments an Hochschulen: Ein Überblick. Szenarien. Praxis. E-Klausur-Recht. Waxmann Verlag GmbH, Münster 2014.

[ScWi74] R. S. Scowen, B. A. Wichmann: The Definition of Comments in Programming Languages. National Physical Laboratory, Teddington, Middlesex TW11 OLW, England, Juni 1974.

[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.

[TAAL08] Aline Lopes Timóteo, Alexandre Alvaro, Eduardo Santana de Almeida, Silvio Romero de Lemos Meira: Software Metrics: A Survey. Centro de Informatica – Universidade Federal de Pernambuco (UFPE) and Recife Center for Advanced Studies and Sstems (C.E.S.A.R), Brasilien, 2008.

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

[Wi18] Frank Witte: Metriken für das Testreporting – Analyse und Reporting für wirkungsvolles Testmanagement. Springer Vieweg, Springer Fachmedien Wiesbaden GmbH, 2018.

[Wo03] Björn Wolle: Statische Analyse von Java-Anwendungen – Eignen sich Lines-of-Code-Metrik und Halstead-Länge? Wirtschaftsinformatik 45, 2003.

[Ze02] Andreas Zeller: Software-Metriken. Lehrstuhl Softwaretechnik, Universität des Saarlandes, Saarbrücken, 2002.

[Zu94] Horst Zuse: Complexity Metrics / Analysis. In John J. Marciniak: Encyclopedia of Software Engineering. John Wiley & Sons, 1994.