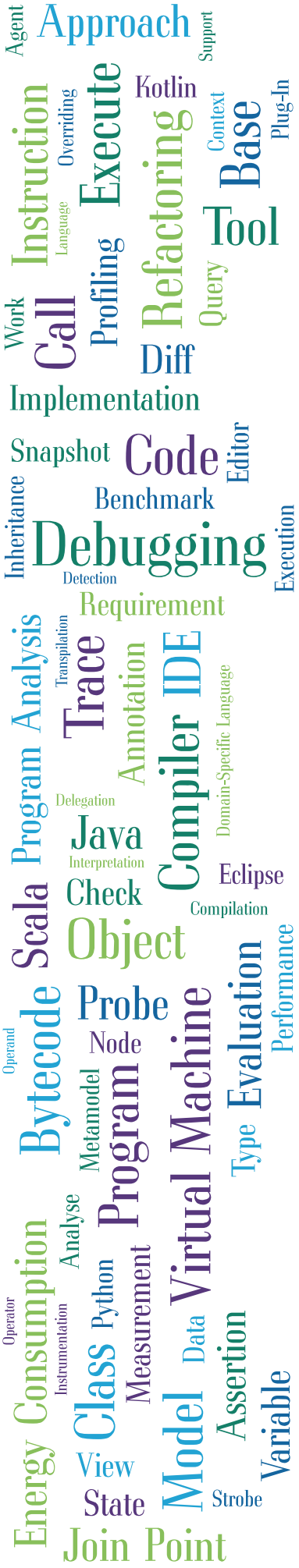


Philipps-Universität Marburg

Fachbereich Mathematik und Informatik

Analyse und Generierung von Feedback zu Programmieraufgaben für Anfänger

Omar Aji

Masterarbeit

10. März 2023

Prüfer:
Prof. Dr.-Ing. C. Bockisch
AG Programmiersprachen und -werkzeuge

Betreuer:
M.Sc. Steffen Dick
AG Programmiersprachen und -werkzeuge

Institut

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Arbeitsgruppe Programmiersprachen und -werkzeuge
Hans-Meerwein-Str. 6
35043 Marburg
Deutschland

Lizenz

Klassifikation (ACM CCS 2012)

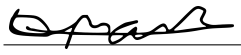
- **Applied computing Education; Document preparation;**
- *Software and its engineering Software notations and tools;*

Schlüsselwörter

E-Learning, automatisiertes Feedback, Programmieren lernen, E-Learningsystem

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Hausarbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

Marburg, den 10.3.2023 

Omar Aji

Danksagung

Ich möchte mich zuerst bei meinem Professor Prof. Dr.-Ing. Christoph M. Bockisch und bei meinem Betreuer M. Sc. Steffen Dick für die hilfreichen Feedbacks und Anleitung, die mich immer in die richtige Richtung orientiert haben, bedanken.

Außerdem möchte mich bei meiner Familie Taysier, Mahasen, Anas, Bsher, Yman und Tsnim, und bei meiner Verlobte Rama Hailam bedanken, die mich immer unterstützt und motiviert haben.

Zusammenfassung

In den letzten Jahren hat Feedback in der Lehre, insbesondere im Bereich der Programmierung, eine wachsende Bedeutung erlangt. Feedback unterstützt den Lernprozess, indem es den Lernenden Hinweise auf seine Stärken und Schwächen gibt und Verbesserungsmöglichkeiten aufzeigt. E-Learningsysteme wie Quarterfall und MASS bieten die Möglichkeit, Feedback automatisch zu generieren und somit den Lehrprozess zu optimieren.

In dieser Arbeit wird untersucht, wie Feedback durch E-Learningsysteme wie Quarterfall bzw. MASS generiert werden kann, um die Anforderungen und Bedürfnisse von Lehrern und Lernenden zu erfüllen und ein effektives Feedback zu gewährleisten. Dabei werden auch Fragen behandelt, wie Lehrer den Inhalt und Detailgrad des Feedbacks bestimmen können und ob sie ihr eigenes Feedback in Bezug auf eine Aufgabe und einen Fehler in der abgegebenen Lösung anlegen können. Darüber hinaus wird untersucht, ob das Feedback in einer gewünschten Sprache generiert werden kann.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listingsverzeichnis

1 Einleitung

Die Bedeutung von Feedback in der Lehre und insbesondere im Bereich der Programmierung hat in den letzten Jahren stark zugenommen. Feedback dient dazu, den Lernprozess zu unterstützen, indem es dem Lernenden Hinweise auf seine Stärken und Schwächen gibt und Verbesserungsmöglichkeiten aufzeigt. E-Learningsysteme wie Quarterfall bieten die Möglichkeit, Feedback automatisch zu generieren und somit den Lehrprozess zu optimieren.

Diese Arbeit beschäftigt sich mit der Frage, wie Feedback in der Lehre durch E-Learningsysteme wie Quarterfall bzw. MASS generiert werden kann. Dabei werden die Anforderungen und Bedürfnisse von Lehrern und Lernenden berücksichtigt, um ein effektives Feedback zu gewährleisten. Die Forschungsfragen, die in dieser Arbeit untersucht werden, sind die folgenden:

1. Können Lehrer den Inhalt und Detailgrad des durch eine E-Learningsystem generierten Feedbacks in Bezug auf die Schwierigkeit der Programmieraufgaben und die Programmiererfahrung des Lerners bestimmen?
2. Können Lehrer ihr eigenes Feedback in Bezug auf eine Aufgabe und einen Fehler in der abgegebenen Lösung anlegen?
3. Kann das Feedback zu einer abgegebenen Lösung in einer gewünschten Sprache generiert werden?

Zur Beantwortung dieser Fragen wird ein Feedback-Framework entwickelt, das in der Lage ist, Feedback zu generieren und individuell an die Bedürfnisse der Lernenden anzupassen. Dieses Framework wird von dem System MASS verwendet, was in Quarterfall genutzt werden kann.

Das Ziel dieser Arbeit ist es, ein effektives Feedback-System zu erweitern, das den Lehrprozess im Bereich der Programmierung optimiert und Lehrern sowie Lernenden eine wertvolle Unterstützung bietet. Durch die Implementierung und Evaluation des Feedback-Frameworks soll gezeigt werden, dass automatisch generiertes Feedback nicht nur effektiv, sondern auch individuell anpassbar ist und somit den Bedürfnissen von Lehrern und Lernenden gerecht wird.

Die Arbeit gliedert sich in mehrere Kapitel. Im ersten Kapitel werden die Grundlagen für dieser Arbeit erläutert. Das zweite Kapitel beschäftigt sich mit Feedback. Hierbei werden verschiedene Arten von Feedback vorgestellt und deren Funktion im Lernprozess erläutert. Hier legen wir den Fokus auf Programmieraufgaben in der Sprache JAVA und auf bestimmte Fehlerarten, die in den eingereichten Lösungen auftauchen können.

Im dritten Kapitel wird die Analyse und das Ziel der Arbeit genauer erläutert. Hierbei werden die Forschungsfragen und Anforderungen definiert. Im vierten Kapitel wird ein Konzept zur Verbesserung des Feedbacks in Quarterfall bzw. Das System MASS entwickelt. Im fünften Kapitel wird die Implementierung des Konzepts beschrieben und im sechsten Kapitel wird eine Evaluation durchgeführt, um die generierten Feedbacks in Bezug auf

den Anforderungen zu überprüfen.

Abschließend wird im Fazit der Arbeit ein Resümee gezogen und ein Ausblick auf weitere mögliche Entwicklungen gegeben. Die Arbeit soll dazu beitragen, das automatisch generierten Feedbacks von einem E-Learningsystem zu verbessern und somit einen wichtigen Beitrag zur Verbesserung des Lernprozesses zu leisten.

2 Grundlagen

2.1 E-Learning

Grundsätzlich existieren viele verschiedene Terminologien rund um den Begriff des E-Learnings, wobei die Nutzung nicht einheitlich ist und verschiedene Ansätze existieren. In [14] wurde E-Learning als Unterkategorie von D-Learning (Digital Learning) aufgefasst. D-Learning beschreibt dabei die Nutzung von Informations- und Kommunikationstechnologie (ICT) in der Lehre. Für diese Arbeit werden wir aber die Definition von Friederike Pfeiffer-Bohnen verwenden [21].

- "eLearning umfasst den Einsatz jeder Form elektronischer bzw. digitaler Medien oder Produkte zur Unterstützung des Lehr- oder Lernprozesses, deren Anwendung zeit- und ortsunabhängig erfolgen kann und dabei die Aufgabe des Wissenstransportes oder des wissenschaftlichen Diskurses erfüllt."

2.2 E-Learningsystem

Durch moderner Informationstechnologien wurden in den letzten zehn Jahren zahlreiche E-Learningsysteme entwickelt. Unter E-Learningsystemen oder auch Lernplattformen werden Softwaresysteme verstanden, die als Plattform zur Organisation von Lehrveranstaltung genutzt werden können. Die Funktionalität dieser Systeme reicht im Allgemeinen von der Nutzeradministration über das Bereitstellen von Lernmaterialien, Kommunikationsmöglichkeiten, Lernaufgaben und die Generierung von Feedback zu den eingereichten Lösungen [6] [21].

Solche Systeme ermöglichen, Lernprozesse besser zu gestalten. Hierfür weisen E-Learningsysteme allgemein folgenden Charakteristik auf [19] :

- Eine nicht lineare Struktur: Lernmaterialien in diesen Systemen können beliebig zugegriffen werden.
- Multimodalität: Anbieten von Informationen erfolgt auf verschiedene Art und Weisen (Text, Bild, Ton ...).
- Interaktivität: Die Lernenden können entscheiden, welche Inhalte sie wann bearbeiten.

Durch die oben genannten Merkmale ermöglichen E-Learningsysteme dadurch, den Lernprozess weitgehend zu individualisieren. Lernende können aus diesem Grund ihrer bevorzugten Lernstrategie anzuwenden und sich selbst beim Lernen zu organisieren. Daher besteht die Herausforderung bei der Gestaltung von solche Systeme, die Konstruktion und Kommunikation von Wissen anzuregen. Diese Anregung kann dadurch erfolgen, dass zu den Lernmaterialien Lernaufgaben gestellt werden.
In dieser Arbeit werden E-Learningsysteme betrachtet, die primär für die Anwendung in der Informatik, genauer in Veranstaltungen für die Programmierung entwickelt wurden.

2.3 Quarterfall

Ein Beispiel zu einem E-Learningsystem ist die Webanwendung der Firma Quarterfall. Mithilfe der Webanwendung können Lernenden ihre Lösungen einreichen und prüfen

lassen. Außerdem ermöglicht Quarterfall zu den eingereichten Lösungen automatische Generierung von Feedbacks. Hierfür bietet Quarterfall verschiedenen Arten zu Generierung von Feedbacks [1]:

- **Anzeigen der richtigen Lösung:** Lernenden erhält die Musterlösung und kann seine Lösung mit der Musterlösung vergleichen.
- **Bedingtes Feedback:** Feedback wird nur dem Lernenden gegeben, wenn ein oder mehrere Bedingungen erfüllt sind. Abbildung 1 zeigt, wie ein bedingtes Feedback im Quarterfall zu definieren.
- **Unit Test:** Quarterfall ermöglicht die Lehrende, mithilfe von Unit-Tests die Lösungen von Programmieraufgaben zu testen und ein passendes Feedback zu generieren.
- **Cloud Checks:** Für mehr Kontrolle, um eingereichten Lösung der Lernenden zu überprüfen und entsprechendes Feedback zu generieren, wird **Cloud Checks** verwendet. Cloud Check kann zum Beispiel verwendet werden, um erweiterte Komponententests für Programmieraufgaben durchzuführen. Hierfür wird Git-Repository verwendet. Wenn der Lernenden eine Lösung eingereicht hat, wird die Repository geklont und auf dem Server von Quarterfall versucht, zwei Skripte auszuführen.
  - *install.sh*: Dieses Skript sollte alle benötigten Abhängigkeiten installieren.
  - *run.sh*: Dieses Skript wird ausgeführt, wenn der Lernenden Feedbacks für seine Lösung haben möchte. Das Skript run.sh dient grundsätzlich Code aus dem Repository laufen zu lassen. Während das Skript run.sh ausführen wird, können Daten aus der Datei **qf.json** gelesen und in sie geschrieben werden. **qf.json** enthält vor der Ausführung des run.sh Skript die eingereichte Lösung, die Konfigurationen der Aufgabe, die persönlichen Daten des Lernenden. Die Abbildung 2 zeigt, wie man ein Cloud-Check zu einer Aufgabe (Assignment) hinzufügt.

Abbildung 1 Quarterfall: Bedingtes Feedback

Abbildung 2 Quarterfall: Cloud Check

3 Feedback

Der Begriff "Feedback" (engl. "to feed back") oder auch "Rückmeldung" dürfte vertraut sein. Allgemein bezeichnet er aber die Informationen, die in einem technischen, biologischen oder sozialen System nach Durchlauf eines Prozesses oder Prozess-Schrittes rückgemeldet werden, um auf künftige Prozess-Durchläufe oder –Schritte zu wirken [19]. Aufgrund der verbreiteten Begriffsverwendung sowohl im Alltagsleben als auch in verschiedenen technischen und wissenschaftlichen Bereichen kann es beim Verständnis dieser Begriff je nach Kontext stark variieren.

Der Begriff Feedback stamme aber ursprünglich aus dem technischen Bereich und ist ein Begriff der Kybernetik. Dort bezeichnet er den Messprozess in sogenannten Regelkreis und dient dafür, die Differenzen zwischen Soll-Vorgabe und dem Ist-Zustand aufzuzeigen, um entsprechenden Korrekturmaßnahmen auszulösen [16].

3.1 Feedback in Lernpsychologie

In der Lernpsychologie ist Feedback ein zentrales Prinzip, da das Ergebnis eines Verhaltens die künftigen Verhalten beeinflussen [23]. In anderen Worten, das Feedback, das eine Person über ein Ergebnis erhält, determiniert nämlich, ob diese Person seine Verhaltensweise auf künftige Prozesse oder Aufgaben verändert.

Selbst in der Lernpsychologie gibt es unterschiedliche Verständnis zu dem Begriff Feedback. In dieser Arbeit konzentrieren wir uns aber grundsätzlich auf informatives Feedback für computergestützte Lernumgebungen.

3.1.1 informatives Feedback

Susanne Narciss bezeichnet informatives Feedback als verwendete Informationen, die Lernenden während oder nach Aufgabenbearbeitung von einer externen Informationsquelle (z. B. Lernprogramm, Lehrer . . .) angeboten werden. Ziel des Feedbacks ist dabei, die korrekte Lösung derartiger Aufgabe in dieser und auch künftige Situationen zu ermöglichen [19].

3.1.2 Abgrenzung von anderen Feedbacks-Arten

Neben informativem Feedback gibt es auch zahlreiche Varianten von motivierendem Feedback. Hierzu gehören z. B. Belohnungen, Bestrafungen oder auch Feedback, das sich am individuellen Lernfortschritt orientiert. Diese Feedback-Arten sind weitgehend für die Bewertung des Lernergebnisses und dienen nicht dazu, Informationen über die Richtigkeit der Lösung einzelner Aufgaben oder Aufgabenschritte zu liefern. Daher sind sie von informativem Feedback abzugrenzen. Ein weiteres Beispiel wäre internes Feedback, wo es nicht von einer externen Informationsquelle angeboten wird, sondern von der Lernenden selbst während des Lernprozesses. Dies ist auch von dem informativen Feedback abzugrenzen [19].

3.2 Gutes Feedback in Lehre

Prinzipien für ein gutes Feedback wurden in vielen Literaturen ausführlich diskutiert. Sadler hat beispielsweise in [22] eine Theorie entwickelt, wie Lernende von Feedback profitieren. Nicol und Macfarlane-Dick haben in [20] weiterhin sieben Prinzipien für ein gutes Feedback festgestellt, die von dem Feedbackgeber (z. B. Lernprogramm, Lehrer ...) gewährleistet werden sollten. Diese sind wie folgt formuliert: Feedback sollte

- Den Lernenden helfen, eine bessere Leistung zu erreichen.
- Es ermöglichen, die Lücke zwischen aktueller und gewünschter Leistung (Lösung) zu schließen.
- Beim Lernen die Entwicklung der Selbsteinschätzung erleichtern.
- Den Lernenden qualitativ hochwertige Informationen über ihr Lernen liefern.
- Den Dialog zwischen Feedbackgeber und Lernenden rund um das Lernen fördern.
- Positive Motivation und Selbstwertgefühl sichern.
- Lehrkräften Informationen anbieten, die zur Gestaltung des Lehrprozesses genutzt werden können.

Laut der vorgestellten Definition über informativen Feedbacks werden wir den Fokus nur auf den ersten zwei Prinzipien legen, weil sie sich auf die Richtigkeit der Lösung beziehen.

3.3 Inhalte und Formen informativen Feedbacks

Abhängig von dem Bereich und Kontext unterscheidet sich das Begriffsverständnis von Feedback und entsprechend die Inhalte von Feedbacks. Selbst bei informativem Feedback gibt es verschiedenen Feedback-Inhalten. Susanne Narciss hat informative Feedbacks für computergestützte Lernumgebungen untersucht und eine inhaltsbezogene Klassifikation vorgeschlagen. Für die vorgeschlagene inhaltsbezogene Klassifikation bestehen informativen Feedbacks in der Regel aus Kombinationen von unterschiedlichen Feedback-Komponenten [19].

Susanne Narciss identifiziert zunächst zwei Arten von Feedback-Komponenten. Dies sind einfache und elaborierte Komponenten. Wir werden sie genauer in Bezug auf Computer Science und Programmieraufgaben betrachten.

3.3.1 Einfache Feedback-Komponenten

1-Knowledge of performance for a set of tasks (KP)

Nach der Bearbeitung einer oder mehrerer Aufgaben bekommt man eine quantitative Rückmeldung. Bsp.: Zu 80% wurde eine Programmieraufgabe richtig gelöst. In anderen Worten: 4 von 5 Tests dieser Aufgabe wurden bestanden.

2-Knowledge of result/response (KR)

Nach der Bearbeitung einer Aufgabe bekommt man eine Rückmeldung, ob die Aufgabe richtig oder falsch gelöst wurde. Bsp.: Eine Programmieraufgabe wird richtig gelöst, wenn sie alle relevanten Tests besteht und eine oder mehrere Einschränkungen erfüllt.

3-Knowledge of the correct result (KCR)

Nach der Bearbeitung eine Aufgabe bekommt man eine Beschreibung oder Angabe der korrekten Lösung. Bsp.: Die Lösung (Code) einer Programmieraufgabe wird vorgestellt.

Einfache Feedback-Komponenten verletzen sowohl die relevanten Prinzipien für ein gutes Feedback als auch die Definition von informativem Feedback. Ferner könnten sie eine schädliche Auswirkung auf Lernende haben. Kyrilov und Noelle [15] haben Werkzeuge untersucht, die Programmieraufgaben automatisch korrigieren und ein KR-Feedback zurückgeben. Dadurch haben sie herausgefunden, dass Lernende durch KR-Feedback häufiger plagiiert und weniger geübt haben.

3.3.2 Elaborierte Feedback-Komponenten

Da wir uns in dieser Arbeit auf informatives Feedback in Bezug auf einzelne Programmieraufgaben konzentrieren, werden wir elaborierte Feedback-Komponenten genauer brachten. Weiterhin werden wir diese Komponenten in Unter-Komponenten wie in [13] unterteilen.

1-Knowledge About Task Constraints (KTC)

Bei dieser Komponente liegt der Fokus auf der Aufgabe und es werden Hinweise diesbezüglich gegeben. KTC wird in zwei Unter-Komponenten unterteilt.

- **Hints on task requirements (TR):** Hier werden die Anforderungen für eine Programmieraufgabe gegeben. Bsp.: Die Programmieraufgabe muss rekursiv gelöst werden.
- **Hints on task-processing rules (TPR):** Hier werden Hinweise zur Herangehensweise an die Programmieraufgabe gegeben. Bsp.: Um eine Programmieraufgabe zu lösen, sollte man am Anfang nötige Felder (Variable) deklarieren und dann Methoden schreiben.

2-Knowledge About Concepts (KC)

Bei dieser Komponente liegt der Fokus auf dem Konzept (Stoff), das von einer Aufgabe geprüft werden sollte. Dafür werden Hinweise zu dem Konzept in Form eines Feedbacks gegeben.

3-Knowledge About Mistakes (KM)

Bei dieser Komponente liegt der Fokus auf dem Fehler in der abgegebenen Lösung einer Programmieraufgabe. Diesbezüglich werden Hinweise in Form eines Feedbacks gegeben. Das Feedback hier verfügt über einen Typ und einen Detailgrad. Der Detailgrad kann hoch oder niedrig sein. Anzahl der Fehler ist ein Beispiel für einen niedrigen Detailgrad. Aber durch eine Beschreibung des Fehlers Fehler kann der Detailgrad steigen. Weiterhin werden wir KM-Feedbacks in Typen (Unter-Komponenten) wie in [13] untergliedern:

- **Test Failures (TF):** Ein fehlgeschlagener Test zeigt an, dass ein Programm nicht die erwartete Ausgabe erzeugt. Bsp.: das generierte Feedback bei Unit Test in Quarterfall.
- **Compiler Errors (CE):** Kompilierfehler, die von einem Compiler erkannt werden, und dessen KM-Feedback sind hier aber nicht Aufgaben spezifisch. Bsp.: Ein “;” wurde vergessen.
- **Solution Errors (SE):** Lösungsfehler sind logische Fehler (das Programm tut nicht, was erforderlich ist) oder inakzeptabler Lösungsweg. Bsp.: Schleifen statt Rekursion wird in der Lösung verwendet.
- **Style Issues (SI):** Programmierstil ist ein wichtiger Punkt, wofür Programmieranfänger ein Feedback erhalten sollen. Bsp.: Formatierung und Dokumentation des Codes betrachtet.

- **Performance Issues (PI):** Wenn die Ausführung einer abgegebenen Lösung zu lange dauert oder mehr Ressourcen als erforderlich verbraucht, sollte ein KM-Feedback gegeben werden.

4-Knowledge About How to Proceed (KH)

Bei dieser Komponente liegt der Fokus auch auf den Fehlern in einer abgegebenen Lösung einer Programmieraufgabe. Diesbezüglich werden Hinweise in Form eines Feedbacks gegeben. Der Unterschied zwischen KH-Feedbacks und KM-Feedbacks liegt daran, dass eine Korrekturmöglichkeit angeboten wird. Das Feedback verfügt auch über einen Typ und einen Detailgrad. Detailgrad kann hoch oder niedrig sein. Weiterhin werden wir KH-Feedbacks in zwei Typen (Unter-Komponenten) untergliedern:

- **Bug-related hints for error correction (EC):** Das Feedback wäre, wie Lernende den Fehler beheben können, anstatt die Ursache des Fehlers zu zeigen.
- **Improvements (IM):** Dieses Feedback wird gegeben, wenn die abgegebene Lösung zwar richtig ist, aber sie von Lernenden optimiert werden kann.

5-Knowledge About Meta-cognition (KMC)

Hinweise auf metakognitive Strategien, Metakognitive Leitfragen. Diese Art von Feedback wird nicht betrachtet.

3.4 Relevante Feedback-Komponenten

Nachdem wir informatives Feedback und Feedback-Komponenten erläutert haben, möchten wir die für diese Arbeit relevanten Komponenten benennen. Davor stellen wir den Kontext dieser Arbeit vor. Der Kontext dieser Arbeit fassen wir durch folgende Punkte zusammen:

- Zielgruppe: Studenten im ersten Semester.
- Lernverstellung: Objektorientierte Programmierung.
- Programmiersprache: Java.
- Relevante Fehlerarten:
  - Compiler Errors (CE).
  - Solution Errors (SE).
  - Style Issues (SI).

Susanne Narciss hat in [19] den Einfluss des Feedbacks auf die Leistung der Lernende untersucht. Weiterhin ist sie zu dazu gekommen, dass die Leistung der Lernende durch informatives Feedback sich verbessert. In anderen Worten, durch informatives Feedbacks können Lernende mehr als bei KR-Feedback profitieren. Daher möchten wir den passenden informativen Feedbacks für Compiler Errors (CE), Solution Errors (SE) und Style Issues (SI) betrachten. Wir werden in diesem Abschnitt zunächst die Schwierigkeit der Programmieraufgabe und die Programmiererfahrung des Lernendes nicht betrachten. In anderen Worten, wir bestimmen die passende Feedback-Art für Lernende, der seine ersten Programmieraufgaben löst.

3.4.1 Compiler Errors (CE)

Feedback zu einem Compiler Error kann die Standard-Compiler-Fehlermeldung sein. Diese Meldung ist für Programmieranfängern meistens nicht einfach zu verstehen [12].

Einige Automated-Assessment-Werkzeuge haben versucht, dieses Problem zu lösen, und die Standard-Compiler-Fehlermeldung durch Fehlermeldungen in verständlicher Sprache ersetzt. Ein Beispiel dafür ist das System JACK. Wir werden in dem nächsten Kapitel das generieren Feedbacks von diesen Systemen genauer betrachten.

In dieser Arbeit schlagen wir ein Feedback für Compiler Error vor, das aus KM- und KH-Komponenten besteht. Durch die KM-Komponente können Lernende die Ursache der Fehler erkennen. Außerdem durch die KH-Komponente können Lernende Hinweis zu Behebung des Fehlers bekommen und die Existenz des Fehlers bei späteren Aufgaben wahrscheinlich leichter erkennen. Die KH-Komponente kann einen Codebeispiel sein, da beispielbasierte Feedbacks Lerneffekte mit sich bringen [5]. Demzufolge wird das Ziel an einem informativen Feedback erreicht.

Beispiel
Wir nehmen an, dass eine abgegebene Lösung einen Compiler Error enthält, wo die Anweisung “Return” in einer Methode fehlt. Ein Feedback, das aus KM- und KH-Komponenten besteht, kann folgendes enthalten:

- Ort des Fehlers (KM-Komponente)
- Beschreibung des Fehlers (KM-Komponente): Bsp.: Der Compiler erwartet die Return Anweisung am Ende der Methode.
- Code Beispiel zu Behebung des Fehlers (KH-Komponente):

```
1 // Code mit Compilerfehler
2 int myMethod() {
3 }
4 // Fehlerfreier Code
5 int myMethod() {
6  return 0;
7 }
```

3.4.2 Solution Errors (SE)

Solution Errors tauchen auf, wenn ein oder mehrere Anforderungen bei einer Programmieraufgabe nicht erfüllt sind. Anforderungen können entweder die Verwendung oder das Verbot bestimmte Programmieranweisungen, Struktur oder Konzepte sein. Programmieranweisungen können beispielsweise for- oder while-Schleifen wobei Konzepte können z. B. Rekursion oder Schleifenverschachtlung sein.

Feedback-Komponenten und Inhalt des Feedbacks hängen hier somit von verschieden Punkte ab, damit ein passendes und gezieltes Feedback dem Lernenden gegeben wird. Wir werden schrittweise diese Punkte behandeln und einen Vorschlag zu einem passenden Feedback vorstellen.

Handelt es sich bei Anforderung einer Aufgabe um ein Verbot zur Verwendung bestimmter Programmieranweisungen, Struktur oder Konzepte, wird ein Feedback aus KM-Komponente ausreichen, da das Feedback kein Lerneffekt bringen soll, sodass KH-Komponente nötig ist.

Beispiel
Bei einer Programmieraufgabe dürfen keine Schleifen verwendet werden und der Lernende hat in der abgegebenen Lösung Schleifen verwendet. Somit kann das Feedback wie folgt aussehen:

- Ort des Fehlers (KM-Komponente): In welche Methode wird Schleifen verwendet.
- Beschreibung des Fehlers (KM-Komponente): Es darf in der Lösung keine Schleifen verwendet werden.

Handelt es sich bei Anforderung einer Aufgabe um die Verwendung bestimmter Programmieranweisungen, Struktur oder Konzepte, wird ein Feedback aus KM-Komponente nicht ausreichen, da Lernende (Anfänger) weitere Informationen als Feedback benötigt.

Enthalten die Anforderung der Aufgabe die Verwendung bestimmter Programmieranweisungen, schlagen wir ein Feedback vor, das aus KM-Komponente und KH-Komponente besteht. Durch die KH-Komponente wird dem Lernenden die nötigen Informationen gegeben, damit die Verwendung der Programmieranweisungen leichter und verständlicher wird. Hier stellen wir zwar die richtige Lösung nicht vor, aber liefern dem Lernenden die Möglichkeit, die Aufgabe selbst zu lösen, wodurch der Lernende ein Lerneffekt bekommt.

Beispiel
Bei einer Aufgabe soll die Split-Methode der Klasse String aufgerufen. Diese Methode wird aber in der abgegeben Lösung nicht verwendet. Somit kann ein passendes Feedback wie folgt aussehen:

- Ort des Fehlers (KM-Komponente): In welche Methode wird Split nicht verwendet.
- Beschreibung des Fehlers (KM-Komponente): Bsp.: Sie sollen die Methode “Split” verwenden.
- Code Beispiel zu Behebung des Fehlers (KH-Komponente):

```
1 void myMethod() {
2     String s = "A*B";
3     // we're splitting the String at Character "*" in Strings
4     String[] result = s.split("*");
5     // result is ["A", "B"]
6 }
```

Enthalten die Anforderung der Aufgabe die Verwendung bestimmtes Konzepts, schlagen wir ein Feedback vor, das aus KM-Komponente und KC-Komponente besteht. Durch die KC-Komponente wird dem Lernenden eine Referenz zu dem nötigen Stoff bereitgestellt, um die Aufgabe zu lösen.

Beispiel
Wir betrachten die Lösung aus Listing 2. Die Lösung enthält Solution Error, da die Aufgabe nicht rekursiv gelöst wurde. Ein Feedback, das aus KM- und KC-Komponenten besteht, kann folgendes enthalten:

- Ort des Fehlers (KM-Komponente).
- Beschreibung des Fehlers (KM-Komponente): Bsp.: Die Methode “berechneSumme” soll rekursiv sein.

- Eine Referenz zu dem Stoff, wo Rekursion erklärt wird. (KC-Komponente): Bsp.: Link zu den Folien, wo Rekursion erklärt wird.

3.4.3 Style Issues (SI)

Viele Lehrer halten das Erlernen eines guten Programmierstils für Programmieranfänger für wichtig. Wenn aber der erwartete Programmierstil nicht eingehalten wird, entstehen Stilprobleme. Zwar beziehen Stilprobleme sich nicht auf bestimmter Aufgabe, aber sie sind wichtig, um sauberer und verständlicher Code zu schreiben.

In dieser Arbeit schlagen wir ein Feedback für Style Issues vor, das aus KM- und KH-Komponenten besteht. Durch die KM-Komponente können Lernende die Ursache des Stilproblems erkennen. Außerdem durch die KH-Komponente können Lernende Hinweis zu Verbesserung ihres Programmierstils bekommen und die Existenz von Stilprobleme bei späteren Aufgaben wahrscheinlich leichter erkennen. Die KH-Komponente kann einen Codebeispiel sein, da beispielbasierte Feedbacks Lerneffekte mit sich bringen [5]. Demzufolge wird das Ziel an einem informativen Feedback erreicht.

Beispiel

Wir betrachten die Lösung aus Listing 2. Die Lösung enthält Stilproblem, da der Methodenname mit großem Buchstaben beginnt. Ein Feedback, das aus KM- und KH-Komponenten besteht, kann folgendes enthalten:

- Ort des Stilproblems (KM-Komponente)
- Beschreibung des Stilproblems (KM-Komponente): Bsp.: Sie verwenden Methodenname, die mit einem Großbuchstaben beginnen.
- Code Beispiel zu Verbesserung des Programmierstils (KH-Komponente):

```
1 // Code mit Stilproblem
2 void MyMethod() {
3 }
4 // Code mit keinem Stilproblem
5 void myMethod() {
6 }
```

4 Analyse und Ziel

Ein interessantes Feld wäre, die E-Learningsysteme für das Programmieren-Lernen anzuwenden. Dafür wurde in den 1960er-Jahren angefangen, Werkzeuge zu entwickeln, um Lernenden bei Programmieren oder Lösen von Programmieraufgaben zu unterstützen [7]. Solche Werkzeuge werden allgemein Automated-Assessment-Werkzeuge (AA-Werkzeuge) genannt. Manche Werkzeuge helfen Lernenden durch automatische Feedback-Generierung ein korrektes Programm zu schreiben. Ein Beispiel dafür wäre das System JACK [8]. Andere Werkzeuge dienen dazu, die Lösungen automatisch zu benoten. Ein Beispiel dafür ist das System INLOOP [18].

Zwei weitere wichtige Gründe für die Entwicklung von E-Learningsysteme mit AA-Werkzeuge sind folgende:

- Programmieren-Lernen ist eine Herausforderung [17] und Lernende benötigen Hilfe beim Lernen ihrer ersten Programmiersprache [5].
- Die Anzahl der Lernenden sowohl in Deutschland als auch weltweit jährlich steigt [2, 4], wodurch einen großen Aufwand bei Lehrern besteht, um den Lernenden individuell bei ihr Probleme zu helfen.

Automatisch generierte Feedback kommt somit ins Spiel und ist ein wichtiger Faktor bei Lernen [11]. Daher sollte automatisch generierte Feedback eine wichtige und zentrale Komponente bei E-Learningsysteme für das Programmieren-Lernen sein, damit Lernenden bestmöglich aus dem generierten Feedback profitieren können.

4.1 Relevante Arbeiten

Nachdem wir in dem letzten Kapitel den gewünschten Feedback-Komponenten in Bezug auf den relevanten Fehlerarten vorgestellt haben, betrachten wir das generierte Feedback von den bis jetzt existierenden E-Learningsysteme für die Programmiersprache Java. Für die drei Fehlerarten haben wir die Systeme JACK und INLOOP gefunden [18, 8].

4.1.1 INLOOP

INLOOP steht für '**In**teractive **L**earning center for **O**bject-**O**riented **P**rogramming' und wurde vom Lehrstuhl für Softwaretechnologie der TU Dresden als Unterstützungsplattform für die Veranstaltung Softwaretechnologie konzipiert und wird seit dem Sommersemester 2016 dort eingesetzt. Dabei werden den Studierenden mithilfe von INLOOP Programmieraufgaben zur Verfügung gestellt, die direkt nach der Abgabe automatisch auf Korrektheit getestet werden (Test Failures (TF)).

INLOOP liefert dem Lehrenden kein informatives Feedback, sondern ein Feedback aus KR-Komponente besteht. Weiterhin INLOOP kann leider kein Solution Errors (SE) und Style Issues (SI) identifizieren. Bei Compiler Errors (CE) liefert INLOOP ein KR-Feedback dem Lehrenden zurück. Abbildung 3 zeigt, wie ein KR-Feedback in INLOOP aussieht.

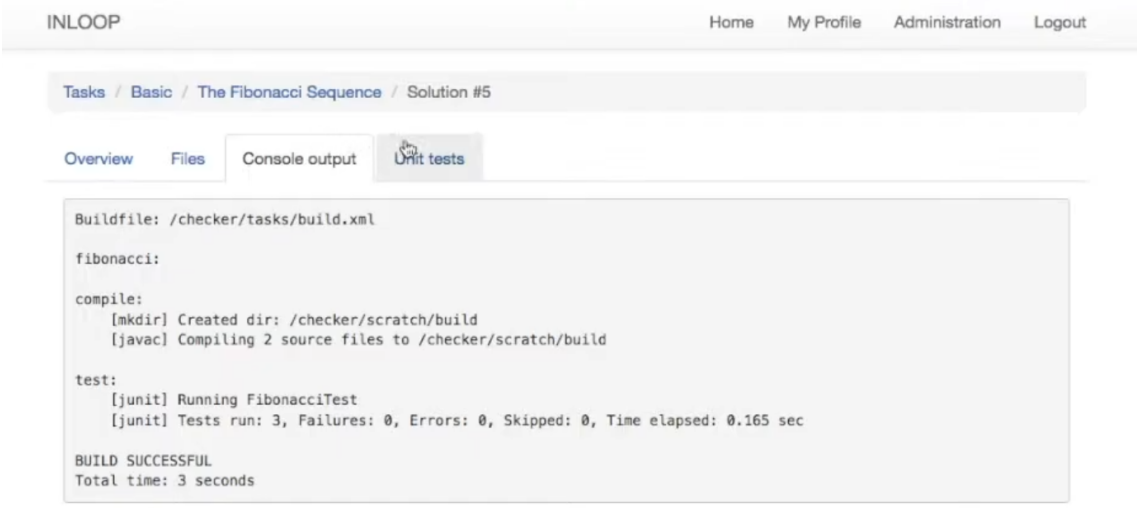


Abbildung 3 INLOOP: Konsole-Ausgabe

4.1.2 JACK

JACK ist ein E-Learningsysteme für die Durchführung von Übungen und Prüfungen mit automatischer Bewertung und Feedback-Generierung. JACK wurde im Jahr 2006 entwickelt und im Wintersemester 2006/07 für einen JAVA-Programmierkurs in Universität Duisburg-Essen verwendet.

JACK generiert Feedbacks für alle drei Fehlerarten, die wir in dieser Arbeit betrachten werden. Diese generieren Feedbacks stellen wir durch ein Beispiel vor. Danach schauen wir, ob JACK die gewünschten Feedback-Komponenten dem Lernenden liefert.

Um die generieren Feedbacks zu untersuchen, betrachten wir folgende Programmieraufgabe.

- "Implementieren Sie eine Methode, die Gaußsche Summenformel berechnet. Die Methode soll rekursiv sein."

Compiler Error (CE)

Nun werden wir folgende Lösung an JACK einreichen 1. Dies enthält ein Compiler Error, wo die Return-Anweisung der Methode in der abgegebene Lösung vergessen wurde. Das von JACK generierten Feedback ist in Abbildung 4 zu sehen.

Listing 1 Lösung mit Compiler Error.

```
1 public static int BerechneSumme(int n) {
2     int result = 0;
3     for(int i = 1 ; i < n ; i++) {
4         result += i;
5     }
6 }
```

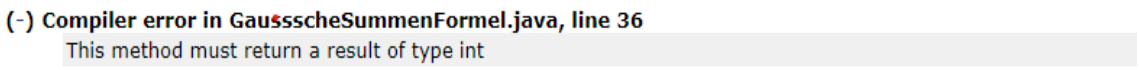


Abbildung 4 JACK: Compiler Error

Wir betrachten das Feedback aus Listing 4 nun als Beispiel für einen Compiler Error. Dieses Feedback wird im Englischen und nicht in einer gewünschten Sprache generiert. Außerdem dieses Feedback wird immer so generiert, unabhängig von der Programmiererfahrung der Lernenden.

JACK generiert zu Compiler Error ein informatives Feedback. Dieses Feedback besteht aber aus KM-Feedback-Komponente, da nur die Ursache und Ort des Fehlers erläutert wird. Für einen Lernenden mit Programmiererfahrung wird dieses Feedback über den Compiler Error vermutlich ausreichen, um den Fehler zu beheben. Aber für einen Programmieranfänger, der seine ersten Programme schreibt, kann dieses Feedback nicht sehr hilfreich sein, da kein Hinweis dem Lernenden gegeben wird. Hier fehlt somit im Vergleich zu unserem gewünschten Feedback die KH-Komponente.

Solution Errors (SE)

Nun werden wir folgende Lösung an JACK einreichen 2. Dies enthält ein Solution Error, da die Aufgabe nicht rekursiv gelöst wurde. Das von JACK generierten Feedback ist in Abbildung 5 zu sehen.

Listing 2 Lösung mit Solution Error.

```
1 public static int BerechneSumme(int n) {
2     int result = 0;
3     for(int i = 1 ; i <= n ; i++) {
4         result += i;
5     }
6     return result;
7 }
```

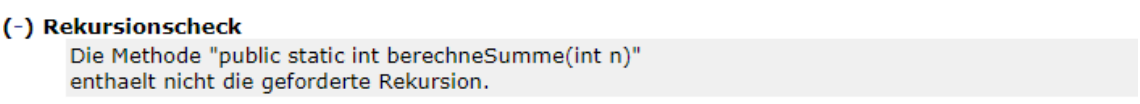


Abbildung 5 JACK: Solution Error

Wir betrachten das Feedback aus Listing 5 nun als Beispiel für einen Solution Error. Dieses Feedback wird im Deutsch und nicht in eine gewünschte Sprache generiert. Wir konnten auch leider nicht identifizieren, ob JACK weiter Solution Error berücksichtigen kann. Dies liegt daran, dass wir keinen administrativen Zugang haben.

JACK generiert zu Solution Error ein informatives Feedback. Dieses Feedback besteht aber aus KM-Feedback-Komponente, da nur die Ursache und Ort des Fehlers erläutert wird. Hier fehlt aber die KC-Komponente, sodass der Lernenden eine Referenz haben kann, um sich bei diesem Fehler weiter zu informieren. Weiterhin hat der Lehrer keine Möglichkeit, solche Referenzen im JACK einzufügen. Außerdem gibt es keine KH-Komponente, womit

Lernenden einen Hinweis zu Lösung der Aufgabe bekommen können.

Style Issues (SI)

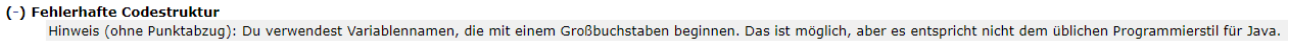


Abbildung 6 JACK: Style Issue

Wir betrachten das Feedback aus Listing 6 nun als Beispiel für einen Style Issue. Dieses Feedback wird im Deutsch und nicht in eine gewünschte Sprache generiert. Weiterhin kann der Lehrer nicht angeben, ob diese Style Issue von Lehrerenden korrigiert werden muss. Somit sind alles Style Issues, die von JACK identifiziert werden können, optional zu korrigieren. Außerdem wird der Ort des Style-Issues nicht angegeben.

JACK generiert zu Style Issue ein informatives Feedback. Dieses Feedback besteht aber aus KM-Feedback-Komponente, da nur die Ursache des Fehlers erläutert wird. Hier fehlt aber die KH-Komponente, sodass der Lernenden ein Hinweis bekommen, um Ihre Lösungen zu verbessern oder zu korrigieren. Außerdem fehlt der Ort des Style-Issues, was ein Teil der KM-Komponente ist.

4.2 Forschungsfragen und Anforderungen

In diesem Abschnitt werden wir die Forschungsfragen, die in dieser Arbeit behandelt werden, ausführlich formulieren.

4.2.1 Inhalt und Detailgrad des individuellen Feedbacks

Ein detailliertes Feedback für Programmieranfänger ist hilfreich, damit sie auf die richtige Lösung kommen können. Dies wäre z. B. ein Feedback, das die Ursache des Fehlers und ein Hinweis zu Behebung des Fehlers enthält. Im gegen Satz dazu Lernende mit Programmiererfahrung benötigen nur ein kurzes Feedback, dass die Ursache des Fehlers erläutert. Hier scheint aber ein Widerspruch zwischen dem Detailgrad des Feedbacks und Individualisierung des Feedbacks zu geben. In anderen Worten, das Feedback muss kurz wie möglich, detailliert und hilfreich auf individueller Ebene sein 3.2. Hierbei ist der Inhalt und Detailgrad des Feedbacks das Entscheidende.

Mit JACK wurden auch in [13] über 100 E-Learningsysteme mit AA-Werkzeuge für das Programmier-Lernen untersucht. Bei allen untersuchten Systeme wird ein automatisches Feedback generiert, das aus ein oder mehrere elaborierte Feedback-Komponenten besteht. Dort wurde weiterhin versucht, diese Systeme anhand ihres generierten Feedbacks zu klassifizieren. Unabhängig von Feedback-Art wird dort das Feedback immer statisch konstruiert. In anderen Worten, nach der Abgabe einer fehlerhaften Lösung wird immer ein festes Feedback generiert und kein Bezug weder die auf Programmiererfahrung der Lernenden noch auf Schwierigkeitsstufe der Aufgabe genommen.

Da Lehrer die Schwierigkeit der Programmieraufgaben schätzen können und die Programmiererfahrung der Lernenden in Ihre Veranschlagung ermitteln können, stellt sich die Frage, ob Lehrer den Inhalt und Detailgrad des gegebenen Feedbacks kontrollieren können. Somit lautet unsere erste Forschungsfrage:

- **Können Lehrer den Inhalt und Detailgrad des durch eine E-Learningsystem generierten Feedbacks in Bezug auf die Schwierigkeit der Programmieraufgaben und die Programmiererfahrung des Lerners bestimmen?**

4.2.2 Fehler oder Verbesserungsmöglichkeit

In abgegebenen Lösungen können Fehler oder Verbesserungsmöglichkeiten von dem E-Learningsystem mit AA-Werkzeug entdeckt werden. Ob es sich um einen Fehler oder eine Verbesserungsmöglichkeit handelt, wird es in den meisten E-Learningsystemen festgelegt und Lehrer haben keine Möglichkeit, dies zu ändern. Ein Beispiel dafür wäre, dass Feedbacks zu Style Issues (SI) bei einer Aufgabe als Fehler nicht als Verbesserungsmöglichkeit gesehen werden, wobei bei anderen Aufgaben Feedback zu Style Issues als Verbesserungsmöglichkeit angeboten wird. In dieser Arbeit betrachten wir das nur für Style Issues (SI), da Solution Errors (SE) und Compiler Error (CE) sind Fehler und keine Verbesserungsmöglichkeiten. Somit lautet unsere Forschungsfrage:

- **Können Lehrer festlegen, ob bei Style Issues (SI) sich um einen Fehler oder eine Verbesserungsmöglichkeit handelt?**

4.2.3 Anpassbares Feedback

Die von uns untersuchten E-Learningsysteme können nur Feedback dem Lernenden liefern, die schon im System angelegt wurden. Im anderen Worten Lehrer haben keine Möglichkeit ihr individuelles Feedback zu einer Aufgabe in Bezug auf einen Fehler zu erstellen. Ein Beispiel dafür wäre, dass der Lehrer bei einem Fehler in einer abgegebenen Lösung eine Referenz zu einem Konzept dem Lernenden als Feedback geben kann. Somit lautet unsere Forschungsfrage:

- **Können Lehrer ihr eigenes Feedback in Bezug auf eine Aufgabe und einen Fehler in der abgegebenen Lösung anlegen?**

4.2.4 Übersetzung von Feedback

Um das E-Learningsystem mit AA-Werkzeug zu internationalisieren, sollte das Feedback in eine von Lernenden gewählten Sprache generiert werden. Jack und alle anderen weiteren E-Learningsystemen generieren das Feedback aber in einer bestimmten Sprache. Weiterhin können wir aus Listing 4 und 5 feststellen, dass das Feedback sogar in zwei verschiedenen Sprachen generiert wird. Draus stellt sich die Frag:

- **Kann das Feedback zu einer abgegebenen Lösung in einer gewünschten Sprache generiert werden?**

4.2.5 Schnittstelle für Feedback-Generierung

Trotz die verschiedenen Fehlerarten und Feedback-Komponenten, die wir in dieser Arbeit betrachten, lohnt es sich eine gemeinsame Schnittstelle zu erstellen, die sich um die Gemeinsamkeiten bei Feedback-Generierung kümmert. Gründe dafür sind folgende:

- Das Feedback soll in einheitliches Template generiert werden. Dies führt dazu, dass Lernenden unabhängig von der Fehlerart immer ein strukturiertes Feedback erhalten. Und somit muss sich das System nicht darum kümmern.
- Bei Generierung des Feedbacks muss eine Formatierungsmöglichkeit der Feedback-Komponenten angeboten werden. Wir werden in dieser Arbeit Markdown-Formatierung verwenden [9]. Somit muss sich das System nur um den Inhalt des Feedbacks bzw. der Feedback-Komponenten kümmern und nicht um die Formatierung des Feedbacks.
- Die Ermittlung von in dem System gespeicherten Feedbacks in eine bestimmte Sprache kann übernommen werden. Somit kann sich die Schnittstelle darum kümmern.
- Die Schnittstelle kann das Ändern von Schlüsselwörtern in dem Feedback unterstützen. Z. B.: Das Schlüsselwort “{$Methode}” soll zu “MyMethod()” in “Schreiben Sie die {$Methode} rekursiv.” geändert werden.
- Das System kann erweitert werden und anderen Fehlerarten in den abgegebenen Lösungen entdecken. Somit kann die Funktionalität in dieser Schnittstelle direkt verwendet werden.

4.2.6 Analyse der abgegebenen Lösung konfigurieren

E-Learningsysteme wie JACK können konfiguriert werden, sodass nur bestimmte Kriterien in der abgegebenen Lösung geprüft werden können. Ein Beispiel dafür ist die Überprüfung von Rekursion in der abgegebenen Lösung. Dies möchten wir in dieser Arbeit beibehalten.

4.2.7 Feedback-Generierung garantieren

E-Learningsysteme müssen garantieren, dass für jede entdeckte Fehler oder Verbesserungsmöglichkeit ein Feedback dem Lernenden geliefert wird.

5 Konzept

Nachdem wir in dem letzten Kapitel unsere Forschungsfragen und die Anforderungen an das gewünschte System vorgestellt haben, werden wir in diesem Kapitel ein Konzept entwerfen, um diese Forschungsfragen zu beantworten und die Anforderungen zu erfüllen. Weiterhin werden wir die benötigten Technologien und Anwendungen benennen und ihre Verwendung begründen.

5.1 Anwendungen

5.1.1 Quarterfall

In dem Kapitel 2.3 haben wir das E-Learningsystem "Quarterfall" vorgestellt. Wir werden in dieser Arbeit Quarterfall als E-Learningsystem verwenden und es mit einem AA-Werkzeug verbinden. Wir verwenden Quarterfall, weil Quarterfall bereits die Standardfunktionalitäten für das Erstellen und Bearbeiten von Programmieraufgaben unterstützt. Weiterhin Quarterfall gibt uns durch Cloud Checks die Möglichkeit, eingereichten Lösung der Lernenden zu überprüfen und entsprechendes Feedback zu generieren. Das AA-Werkzeug nennen wir Marburg university Auto Assess System (MASS) und es dient dazu, Feedbacks für Style Issues, Solution Errors und Compiler Errors automatisch zu generieren.

5.1.2 MASS

MASS ist ein AA-Werkzeug und wurde in den Arbeiten [3, 10] zum ersten Mal eingeführt. Dort wurden Style Issues, Solution Errors und Compiler Errors in einer eingereichten Lösung entdeckt und dafür ein Feedback generiert. Das generierte Feedback entspricht aber nicht das gewünschte Feedback in Bezug auf den entdeckten Fehler bzw. die Verbesserungsmöglichkeit. Dies werden wir im Laufe dieser Arbeit verbessern.

5.1.3 Feedback-Generierungsprozess

Nachdem wir MASS und Quarterfall vorgestellt haben, zeigen wir den Feedback-Generierungsprozess. Dies ist in Abbildung 7 dargestellt. Hier reicht der Lernenden zunächst seine Lösung an Quarterfall ein. Dies kann eine Zip-Datei oder eine in Quarterfall Code-Editor geschriebenen Lösung. Abbildung 8 zeigt die beiden Möglichkeiten, um eine Lösung einzureichen. Die Zip-Datei sollte Dateien mit der Endung ". Java" beinhalten, die überprüft werden sollen. Nachdem Einreichen wird Cloud-Checks-Action ausgeführt und die Lösung am MASS weitergeleitet. Dafür wird die Git-Repository für "MASS" geklont und auf dem Quarterfall Server die run.sh Datei ausgeführt. Inhalt der **run.sh** Datei ist in Abbildung 9 dargestellt.

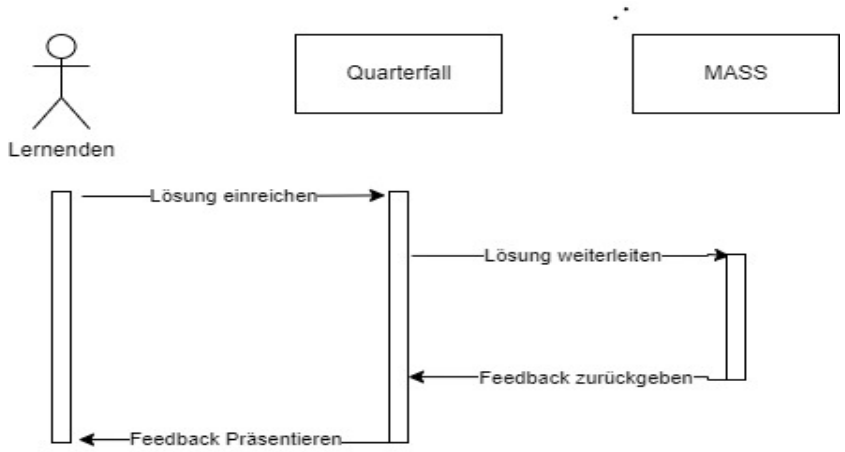


Abbildung 7 Feedback: Generierungsprozess

(a) Lösung in Code-Editor **(b)** Lösung als Zip-Datei

Abbildung 8 Quarterfall: Lösung einreichen

```
mvn --batch-mode -X -e compile  exec:java@CheckerRunner > output
```

Abbildung 9 Quarterfall: run.sh Inhalt

Als Eingabe für MASS wird ein sogenanntes qf-Objekt eingereicht. qf-Objekt ist ein JSON-Objekt. In dem qf-Objekt befindet sich MASS-Konfigurationen sowie die von Lernenden eingereichten Lösung. Falls es sich bei der eingereichten Lösung um eine Zip-Datei handelt, muss MASS dies durch eine URL in dem qf-Objekt herunterladen. Sonst ist die Lösung in dem qf-Objekt enthalten. Abbildung 10 zeigt das qf-Objekt, falls es sich um eine Zip-Datei handelt. Abbildung 10 zeigt die eingereichte Lösung in dem qf-Objekt, falls die Lösung in dem Code-Editor geschrieben wurde. MASS-Konfigurationen werden wir in späteren Abschnitt genauer betrachten.

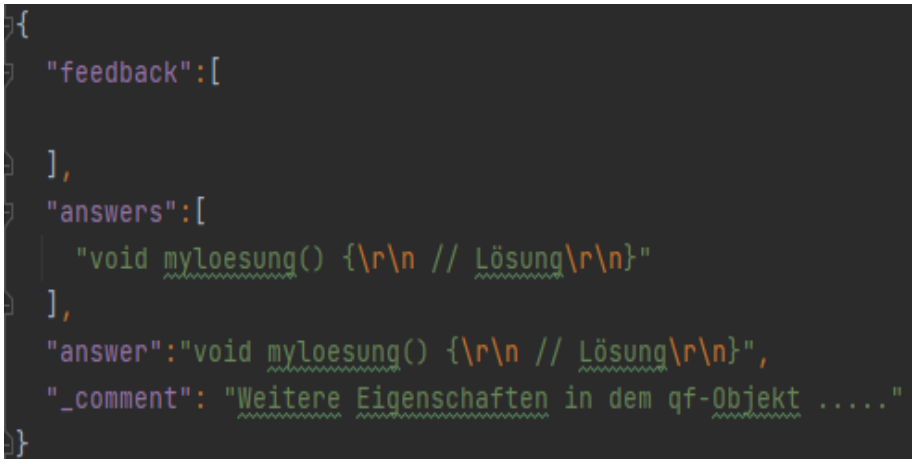


Abbildung 10 Quarterfall: qf-Objekt bei in Code-Editor geschriebenen Lösung

Abbildung 11 Quarterfall: qf-Objekt bei Zip-Datei Lösung

Danach wird MASS ausgeführt und sein Feedback zu der abgegebenen Lösung in das qf-Objekt geschrieben. Abbildung 12 zeigt das qf-Objekt nach der Ausführung von MASS. Das Feedback wird dem Lernenden präsentiert und der Lernenden hat die Möglichkeit seine Lösung anzupassen.

```
{
  "feedback": [
    "# MASS Feedbacks\n"
  ],
  "answers":[
    "void myloesung() {\r\n // Lösung\r\n}"
  ],
  "answer":"void myloesung() {\r\n // Lösung\r\n}",
  "_comment": "Weitere Eigenschaften in dem qf-Objekt ....."
}
```

Abbildung 12 Quarterfall: qf-Objekt bei Zip-Datei Lösung

5.2 MASS-Komponenten und -Architektur

Nachdem wir MASS und den Feedback-Generierungsprozess vorgestellt haben, möchten wir in diesem Abschnitt ein Stück tiefer ins MASS bzw. MASS-Komponenten gehen. Abbildung 13 zeigt, wie MASS aufgebaut ist und über welche Komponenten MASS verfügt.

Zunächst verfügt MASS über mehrere Komponenten für CE, SI und SE. Diese Komponenten nennen wir Syntax Checker, Style Checker und Solution Approach Checker. Syntax Checker such nach CE und generiert dafür entsprechendes Feedback. Style Checker sucht nach SI und generiert dafür entsprechendes Feedback, wobei Solution Approach Checker ist für SE verantwortlich. Wir werden in dem nächsten Abschnitt auf diese Checker genauer eingehen. Außerdem verfügt MASS über ein sogenannten MASS Executor. Durch MASS Executor werden die Checker ausgeführt. Überdies verfügt MASS über Settings, wodurch entschieden werden kann, welche Checker ausgeführt werden sollen. Der Grund liegt darin, dass nicht bei allen Programmieraufgaben nach SI und SE gesucht werden soll.

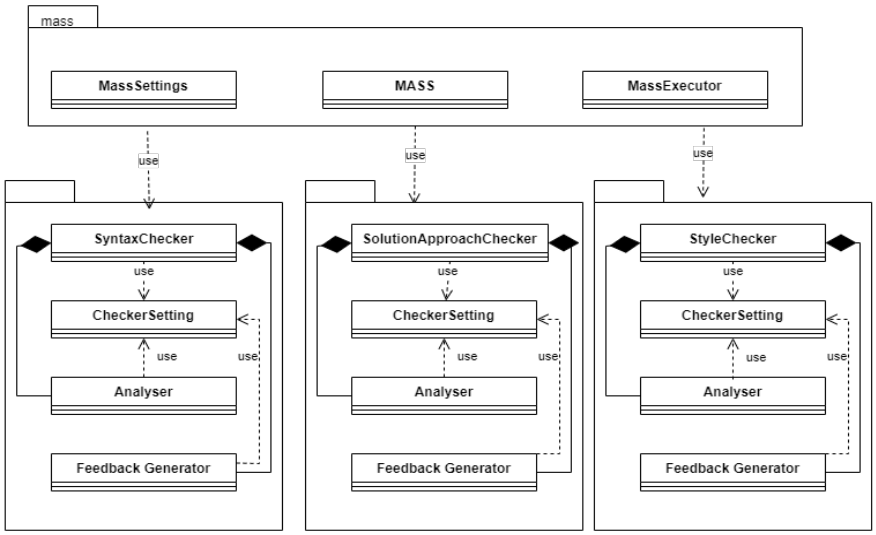


Abbildung 13 MASS: Architektur

5.2.1 Compiler Errors (CE)

Syntax Checker dient dazu, nach CE in der eingereichten Lösung zu suchen und dementsprechend Feedback an MASS bzw. dem Lernenden zurückzuliefern. Für das Analysieren und die Suche nach CE verfügt der Syntax Checker über ein sogenannten Analyser. Der Analyser bereitet der Lösung und schickt sie weiter an den JAVA-Compiler [24]. JAVA-Compiler versucht die Lösung zu kompilieren und falls ein CE in der eingereichten Lösung existiert, erhält der Analyser eine Liste von gefundenen CE.

Bei einem CE sind wir an den wichtigsten Daten interessiert, was JAVA-Compiler für uns generiert. Diese Daten sind Folgendes:

- Compiler Error Code. Dadurch erkennen wir, worum es sich bei diesem CE handelt
- Compiler Error Message. Bessere Beschreibung zu dem Compiler Error Code
- Dateiname, wo diesen Compiler Error auftritt.
- Die Zeile, wo diesen Compiler Error auftritt.

Aus der Liste von gefundenen CE wird ein Bericht "Syntax Analysis Report" erstellt und an den Syntax Checker weitergeleitet. Abbildung 14 zeigt die Beziehung zwischen die obengenannten Komponenten des Syntax-Checkers.

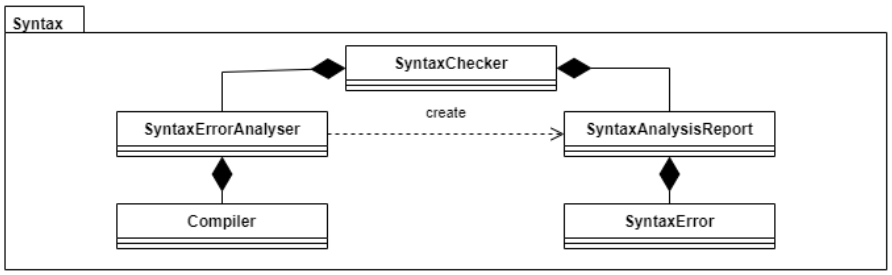


Abbildung 14 MASS: Syntax Checker und Syntax Analyser

Falls in der eingereichten Lösung ein oder mehrere CE existieren, ruft der Syntax Checker den sogenannten Syntax Feedback Generator auf, um Feedbacks in Bezug auf die gefundenen CE zu erstellen. Wie die Feedbacks generiert werden und welche Feedback-Komponenten in den Feedbacks enthalten sind, werden wir im späteren Abschnitt behandeln. Die generieren Feedbacks werden am Ende an MASS weitergeleitet und MASS schreibt sie in das qf-Objekt.

5.2.2 Solution Errors (SE)

MASS verfügt über einen Checker, der Solution Approach Checker heißt und für SE verantwortlich ist. Hier leitet MASS zunächst die von Lernenden eingereichten Lösung an Solution Approach Checker weiter, falls die Lösung keine CE enthält. Solution Approach Checker überprüft, ob die eingereichte Lösung die Aufgabenspezifikation erfüllt, die von dem Lehrer voraussetzt wurde. Falls dies nicht der Fall ist, generiert der Solution Approach Checker passendes Feedback.

Aufgabenspezifikation

Solution Approach Checker überprüft die Aufgabenspezifikation in Bezug auf ein oder mehreren von Lehrer gewählten Methoden. Die Aufgabenspezifikation, die von Solution Approach Checker überprüft werden können, sind Folgendes:

1. Die erlaubte Anzahl von Schleifen in einer Methode. Folgende Schleifen-Typen werden berücksichtigt: for, forEach, while und doWhile.
2. Die erlaubte Anzahl von IfElse-Anweisungen in eine Methode.
3. Der Rückgabetyp einer Methode.
4. Ob eine Methode, ein Rekursive-Aufruf hat.

Damit Lehrer die Aufgabenspezifikation für eine oder mehrere Methoden definieren können, muss für den Solution Approach Checker eine Liste von Einstellungen ermittelt werden. Diese Einstellungen können Lehrer in dem qf-Objekt definieren und somit kann der Checker konfiguriert werden. Jede Einstellung ist für eine Methode zuständig, obwohl innerhalb einer Methode mehrere SEs gefunden werden können. Dies definieren wir so, damit Lehrer nicht für jede Methode mehrere Einstellungen definieren muss. Jede Einstellung heißt Solution Approach Setting Item.

Ablauf

Für die Überprüfung von der Aufgabenspezifikation verfügt dieser Checker über einen sogenannten Analyser. Der Analyser überprüft die eingereichte Lösung und erstellt eine Liste von den gefundenen SEs in Bezug auf alle Solution Approach Setting Item. Jeden Eintrag in diese List nennen wir Solution Approach Report Item und ist genau für einen SE zuständig. Duplikate in dieser Liste können später eliminieren und vermeiden wir somit doubelte Feedback zu einem SE.

Der Analyser verfügt über zwei Komponenten. Die erste Komponente ist für Rekursive-Aufrufe, und Methoden-Aufrufe, Methoden-Parameter und Rückgabe-Typ der Methode zuständig. Die zweite Komponente ist für Schleifen und IfElse-Anweisungen verantwortlich. Abbildung 15 zeigt die Beziehung zwischen die obengenannten Komponenten des Solution Approach Checkers.

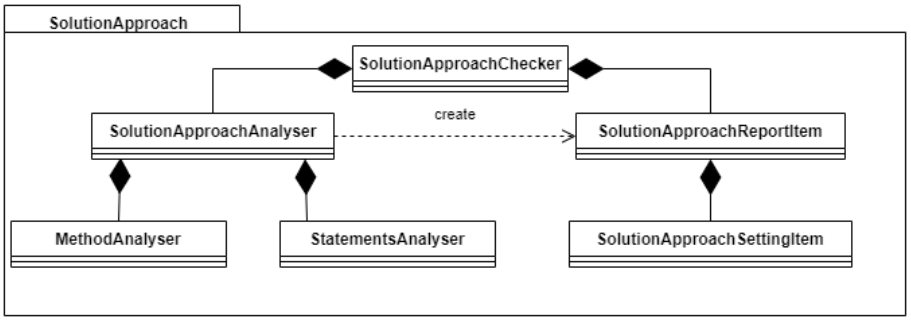


Abbildung 15 MASS: Solution Approach Checker und Solution Approach Checker Analyser

Die Liste der Solution Approach Report Item wird zu dem Solution Approach Checker weitergeleitet und er ruft einen sogenannten Feedback-Generator auf, um passenden Feedbacks in Bezug auf die gefundenen SEs zu generieren. Wie die Feedbacks generiert werden und welche Feedback-Komponenten in den Feedbacks enthalten sind, werden wir

im späteren Abschnitt behandeln. Die generieren Feedbacks werden am Ende an MASS weitergeleitet und MASS schreibt sie in das qf-Objekt.

5.2.3 Style Issues (SI)

MASS verfügt außerdem über einen Checker, der Style Checker heißt und für SI verantwortlich ist. Hier leitet MASS zunächst die von Lernenden eingereichten Lösung an Style Checker weiter, falls die Lösung keine CE enthält. Style Checker überprüft, ob die eingereichte Lösung SI enthält. Falls dies der Fall ist, generiert der Style Checker passendes Feedback.

Wie wir es oben erwähnt haben, sucht Style Checker nach SIs in der eingereichten Lösung. In [3] wurden die SIs auf drei Teile aufgeteilt. Diese sind die Komplexität des Codes, Benennungen und andere SIs. Beispiele zu der Komplexität des Codes sind die Anzahl der Methodenparameter, Länge einer Methode und Vereinfachung von logische Ausdrücke. Bei Benennungen handelt es sich z. B. um Namenskonventionen für Variablen, Methoden und Klassen. Beispiele zu anderen SIs sind Methodenkommentare und Klassen Kommentare.

Jede diese drei Teile ist in 3 Stufen aufgeteilt. Dies sind BEGINNER, INTERMEDIATE und ADVANCED. In [3] wurde diese Aufteilung so gemacht, damit Lehrer das Überprüfen von SIs stufenweise im Laufe der Veranstaltung erhöhen können. Ein Beispiel dafür ist die Länge von Variabel-Name. Sie muss in den späteren abgegebenen Lösungen länger als ein Buchstabe sein, wobei sie in den ersten eingereichten Lösungen bzw. Semesterwochen die Länge 1 haben darf. Diese Stufen werden von dem Lehrer in dem qf-Objekt eingesetzt und an MASS weitergeleitet.

Außerdem kann der Style Checker weiter konfiguriert werden. Hierfür werden die Konfigurationen in dem qf-Objekt von Lehrer definiert und an MASS weitergegeben. Diese Konfigurationen sind folgende:

- Die maximale erlaubte Klasse-Länge
- Die maximale Methodenlänge
- Die maximale Anzahl von Klassenfeldern
- Regulärer Ausdruck für Variabel-Name
- Regulärer Ausdruck für Methoden-Name
- Regulärer Ausdruck für Klassenname

Ablauf

Der Style Checker verfügt über einen Analyser, der die eingereichte Lösung überprüft und eine Liste von gefundenen Stilisierungsfehlern (SIs) erstellt. Jeder Eintrag in dieser Liste, den wir als Violation bezeichnen, ist genau für einen SI zuständig. Duplikate in dieser Liste können später eliminiert werden, um doppeltes Feedback zu einem SI zu vermeiden. Abbildung 16 zeigt die Beziehung zwischen den oben genannten Komponenten des Style Checkers.

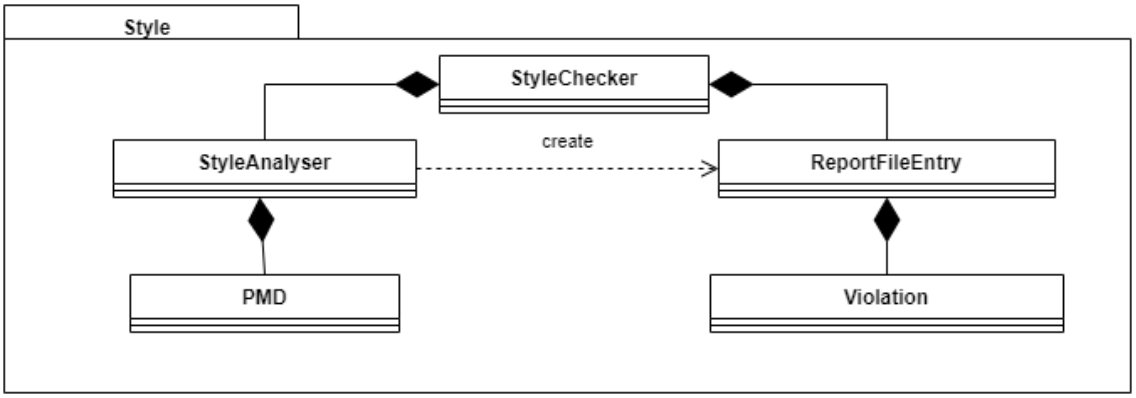


Abbildung 16 MASS: Style Checker und Style Analyser

Die Liste der Violations wird an den Style Checker weitergeleitet, der dann einen Feedback-Generator aufruft, um passende Feedbacks in Bezug auf die gefundenen SIs zu erstellen. Wir werden später besprechen, wie die Feedbacks generiert werden und welche Feedback-Komponenten enthalten sind. Die generierten Feedbacks werden schließlich an MASS weitergeleitet, das sie in das qf-Objekt schreibt.

5.3 Generierung von Feedback

Im Kapitel 3 haben wir die gewünschten Feedback-Komponenten für CE, SI und SE vorgestellt. Wir haben auch im Abschnitt 4.2 erläutert, dass wir eine gemeinsame Schnittstelle für die Generierung von Feedbacks anstreben. Dafür werden wir im vorliegenden Kapitel zunächst einen Entwurf für das Feedback-Modell vorstellen. Anschließend werden wir den Feedback-Generierungsprozess, die Feedback-Formatierung und die Integration von Feedback in ein einheitliches Template behandeln.

5.3.1 Feedback-Modell

Beim Konstruieren des Feedback-Modells werden wir zunächst die benötigten Feedback-Komponenten für CE, SI und SE betrachten. Anschließend werden wir das Modell für technische Zwecke erweitern und anpassen.

Feedback-Komponenten

Um ein informatives Feedback für die eingereichten Lösungen mit CEs, SIs und SE zu generieren, benötigen wir die KM-, KH- und KC-Komponenten. Dies haben wir detailliert in Kapitel 3 behandelt.

Die KM-Komponente eines Feedbacks enthält eine Beschreibung des entdeckten Fehlers und den Orts des Fehlers. Um den Fehler dem Lernenden zu beschreiben, enthält die KM-Komponente einen im Markdown-Format formatierten Text, der bei allen Feedbacks unabhängig von der Art des Fehlers enthalten ist. Um den Ort des Fehlers speichern zu können, muss das Modell mehrere Informationen enthalten, die je nach Art des entdeckten Fehlers unterschiedlich sind. Daher verfügt das Modell über alle möglichen Optionen für CEs, SIs und SEs. Wir beschreiben den Ort des Fehlers durch folgende Informationen: 1)

Den Dateinamen, 2) Den Methodennamen, 3) Die Startzeile und 4) Die Endzeile.

Die KH-Komponente ist etwas komplexer, da sie entweder einen im Markdown-Format formatierten Text (Hinweis) zur Behebung des Fehlers oder ein Codebeispiel enthalten kann. Das Codebeispiel wollen wir ebenfalls im Markdown-Format generieren. Auch können mehrere Hinweise in einem Feedback gegeben werden. Deshalb verfügt das Feedback über eine Liste von Hinweisen. Ein Beispiel wäre, dass man einen Tipp zur Behebung des Fehlers gibt und anschließend ein Codebeispiel dafür, wie dieser Fehler bei einer anderen Aufgabe behoben wurde.

Bei der KC-Komponente liegt der Fokus auf dem Konzept (Stoff), das von einer Aufgabe geprüft werden sollte. Eine Möglichkeit, dieses Konzept zu erklären, wäre, dass die KC-Komponente einen Text enthält, in dem dieses Konzept erläutert wird. Wir möchten dies jedoch vermeiden, da jeder Lehrer das Konzept auf seine eigene Art und Weise beschreibt. Daher sollte die KC-Komponente eine Referenz auf das Konzept in den Lernmaterialien enthalten. Die Referenz bzw. die KC-Komponente enthält folgende Informationen: 1) Den Namen des Lernmaterials, 2) Einen Link zur Referenz, 3) Die Seitenzahl im genannten Lernmaterial und 4) Die Abschnittsnamen im Lernmaterial.

Erweiterung des Modells

Nachdem wir die Feedback-Komponenten behandelt haben, wollen wir nun die technischen relevanten Punkte für das Feedback-Modell erläutern.

Da wir den Lehrern die Entscheidung geben möchten, ob es sich bei SIs in einer abgegebenen Lösung um einen Fehler oder eine Verbesserungsmöglichkeit handelt, benötigt das Feedback-Modell ein Feld, um diese Information zu speichern. Dieses Feld ist der Typ des Feedbacks und kann entweder einen Fehler oder eine Verbesserungsmöglichkeit (Enum) sein.

Um den Lernenden über die Art des Fehlers zu informieren, erweitern wir das Modell um den Checker-Namen. Der Checker-Name macht es eindeutig, ob es sich bei einem Feedback um einen CE, SI oder SE handelt. Weiterhin wird Jade Feedback dadurch generiert, dass der Analyser eines Checkers einen Fehler oder eine Verbesserungsmöglichkeit in der eingereichten Lösung gefunden hat. Dieser Fehler oder Verbesserungsmöglichkeit hat einen eindeutigen Namen und ist systemspezifisch. Somit erweitern wir das Feedback-Modell um ein Feld für diesen systemspezifischen Namen.

Außerdem wollen wir wissen, ob das Feedback schon formatiert wurde oder muss noch formatiert werden. Dafür erweitern wir das Modell um das Boolean-Feld “isFormatted”.

Abbildung 17 zeigt das Feedback-Modell. Hier präsentieren RelatedLocation und readableCause die KM-Komponente. Die Liste von Hint ist für die KH-Komponente verantwortlich und ConceptReference stellt die KC-Komponente dar.

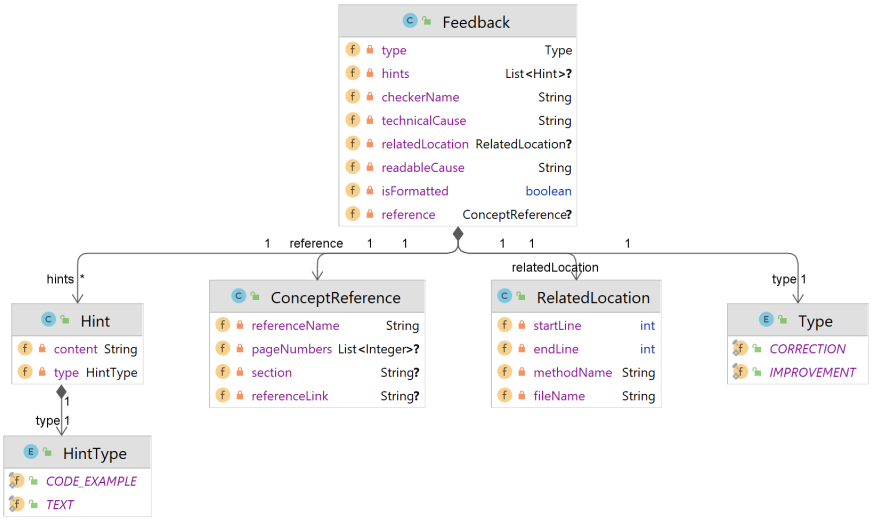


Abbildung 17 Feedback-Modell

5.3.2 Ermitteln von Feedback-Inhalt

Das System soll dem Lehrer die Möglichkeit geben, Inhalt des Feedbacks in Bezug auf einen Fehler zu definieren. Es muss jedoch nicht für jeden möglichen Fehler ein Feedback definiert werden. Daher soll das System vorab über den Inhalt von Feedbacks verfügen. Diese Inhalte können jedoch jederzeit überschrieben werden. Die Frage, die sich hier stellt, ist, welche Feedback-Komponenten bzw. welcher Inhalt vorher im System vorhanden sein muss.

Um diese Frage zu beantworten, schauen wir uns das oben definierte Feedback-Modell an. Daraus können wir erkennen, welche Inhalte wir im System speichern können. Dafür gehen wir jede Feedback-Komponente einzeln durch.

Die KM-Komponente eines Feedbacks enthält eine Beschreibung des entdeckten Fehlers und den Ort des Fehlers. Der Ort des Fehlers ist aufgabenspezifische Information und kann daher nicht vorher gespeichert werden. Die Beschreibung des entdeckten Fehlers kann jedoch eine allgemeine Information sein, die im System gespeichert werden kann. Die KH-Komponente kann entweder einen Text (Hinweis) zur Behebung des Fehlers oder ein Codebeispiel sein. Beide sind allgemeine Informationen, die im System vorher gespeichert werden können. Die KC-Komponente ist eine Referenz auf ein Konzept in den Lernmaterialien und kann daher nicht vorher im System gespeichert werden. Sie muss von Lehrern angelegt werden. Andere Feedback-Felder können während der Feedback-Generierung ermittelt werden, da sie von der Aufgabe und dem entdeckten Fehler abhängig sind.

Um den gespeicherten Inhalt des Feedbacks eindeutig zu identifizieren, verwenden wir den systemspezifischen Namen. Somit kann der Checker durch den systemspezifischen Namen den Inhalt des Feedbacks ermitteln. Die Feedback-Inhalte für jeder Checker (Fehlerart) werden wir in einer separaten JSON-Datei anlegen. Hierbei werden wir aber das Problem haben, dass der Feedback-Inhalt in einer bestimmten Sprache ist. Daher wird eine JSON-Datei für Feedback-Inhalte für jede unterstützte Sprache angelegt. Abbildung

18 zeigt ein Beispiel zu dem Inhalt eines Feedbacks in der JSON-Datei.

```
{
  "technicalCause": "KEY_1",
  "readableCause": "Beschreibung zum Fehler (KM-Komponente)",
  "hints": [
    {
      "type": "CODE_EXAMPLE",
      "content": "int beispiel = 7;"
    }
  ]
}
```

Abbildung 18 Inhalt eines Feedbacks in der JSON-Datei

5.3.3 Feedback anpassen und formatieren

Feedback anpassen

Im letzten Abschnitt haben wir uns mit dem Thema der gespeicherten Feedback-Inhalte beschäftigt. Es wurde erwähnt, dass diese Inhalte von Lehrern überschrieben werden können. Eine Herausforderung besteht darin, diese Inhalte dynamisch zu gestalten. Das können wir anhand eines Beispiels genauer erläutern.

In einer Programmieraufgabe fordert die Aufgabenstellung den Lernenden auf, eine Methode X zu schreiben, die einen Rückgabetyp boolean besitzen muss. Wir nehmen an, dass die Implementierung der Methode X durch den Lernenden von den Erwartungen abweicht, indem sie einen Rückgabetyp int statt boolean verwendet. Um diese Abweichung zu korrigieren, soll das System Feedback bereitstellen. Ein Beispiel für eine KM-Komponente in diesem Feedback könnte wie folgt aussehen:

- **Die Methode wurde mit dem Rückgabetyp int statt dem vorgesehenen Rückgabetyp boolean implementiert.**

Da wir Feedback-Inhalte im System vorher speichern möchten, ist es wichtig, dass die Typen int und boolean im Voraus bekannt sind. Eine Möglichkeit wäre, Feedback-Inhalte für alle möglichen Typen zu speichern, um die KM-Komponenten für jede Abweichung in Bezug auf den Rückgabetyp zu generieren. Dies ist jedoch nicht nur unpraktisch, sondern würde auch bei neuen Typen (Klassen) nicht funktionieren. Um dieses Problem zu lösen, schlagen wir eine Lösung vor, die auf dynamischer Text-Anpassung basiert. Stattdessen speichert das System folgenden Text:

- **Die Methode wurde mit dem Rückgabetyp {$istTyp} statt dem vorgesehenen Rückgabetyp {$sollTyp} implementiert.**

Die sogenannten Schlüsselwörter ({$istTyp} und {$sollTyp}) werden automatisch vom System angepasst, um einen passenden Feedback-Inhalt zu generieren. Um dies zu ermöglichen, erweitern wir die Feedback-Schnittstelle um einen KeyWordReplacer. Der KeyWordReplacer erhält die Feedback-Komponente, die angepasst werden soll, sowie die

Schlüsselwörter und deren Werte. Anschließend wird die Feedback-Komponente entsprechend der übergebenen Werte angepasst und in das Feedback eingefügt. Der KeyWordReplacer sucht nach dem Muster "{$keyword_Name}", um die Feedback-Komponenten anzupassen. Dieses Muster wurde ausgewählt, um sicherzustellen, dass nur die Schlüsselwörter und nicht andere Teile der Feedback-Komponenten versehentlich geändert werden.

Feedback formatieren

In den vorherigen Abschnitten haben wir erwähnt, dass Feedback im Markdown-Format generiert werden soll. Die Formatierung soll durch die Feedback-Schnittstelle übernommen werden. Um dies zu ermöglichen, ist es notwendig, eine Klasse zu implementieren, die Texte in ein gewünschtes Markdown-Format bringt, sowie eine Formatter-Klasse, die gewisse Teile des Feedbacks direkt formatieren kann.

Für die Formatierung in Markdown möchten wir lediglich eine begrenzte Anzahl von Formaten unterstützen, da das System "Quarterfall" nicht aller Markdown-Formate unterstüzen kann. Folgende Formate werden unterstützt:

- Heading 1, 2, 3 und 4
- Bold
- Italic
- CodeBlock
- CodeLine
- Link

Außerdem ist die Klasse eine Utility-Klasse, weshalb es nicht erforderlich ist, ein Objekt dieser Klasse zu instanziieren, um die Funktionalitäten zu nutzen.

Die Formatter-Klasse hat die Aufgabe, Feedback-Komponenten, die noch nicht im Markdown-Format vorliegen, automatisch zu formatieren. Ein Beispiel hierfür sind Codebeispiele aus KH-Komponente.

Als Beispiel betrachten wir den gespeicherten Feedback-Inhalt aus Abbildung 18. In diesem Beispiel enthält die KH-Komponente folgendes Codebeispiel "int beispiel = 7;". Dies soll durch die Formatter-Klasse in den folgenden Markdown-Text umgewandelt werden 3:

Listing 3 formatiertes Codebeispiel.

```
1 ``` java
2 int beispiel = 7;
3 ```
```

5.3.4 Feedback Template

Um das Feedback einheitlich zu gestalten, erweitern wir unsere Feedback-Schnittstelle um eine Klasse namens "TemplateBuilder". Die Klasse "TemplateBuilder" integriert das Feedback in ein Markdown-formatierte Template für eine bestimmte Sprache und gibt es

als String zurück.

Als Template für Feedback haben wir folgende Template ausgewählt 4.

Listing 4 Feedback Template

```
#### Art des Fehlers (Verbesserungsmöglichkeit/Fehler) (KM-Komponente)
In: Dateiname. Methode: MethodenName. Zeilen: X bis Y (KM-Komponente)
Beschreibung zu dem entdeckten Fehler (KM-Komponente)
Beispiel, um dieser Fehler zu beheben (KH-Komponente)
Mehr erfahren: Folie(link) in Sektion. Seiten: X,Y,Z. (KC-Komponente)
```

Dadurch werden alle Feedback-Komponenten in das vorab definierte Template integriert. Ein Beispiel hierfür ist in Abbildung 5 dargestellt, welche die Integration des Feedbacks in das Template veranschaulicht.

Listing 5 Beispiel zu Feedback Template.

````
#### Syntaktische Fehler 01 (muss korrrigiert werden):
In: TestClass.java. Methode: getName. Zeile 2.
du Hast Simcolen vergessen.
```Java
// Alte Code
int i = 0
// Neue Code
int i = 0;
```
Mehr erfahren: **[folie 1](www.google.com)** in **Sektion 1** in Seiten 1,2,11
---
````

5.3.5 Generierung von Feedback für CE

In den vorherigen Abschnitten wurde der Syntax-Checker und seine Komponenten vorgestellt. Im Folgenden möchten wir uns nun auf den sogenannten Feedback-Generator und den Feedback-Generierungsprozess konzentrieren.

Falls in der eingereichten Lösung ein oder mehrere CEs vorliegen, ruft der Syntax-Checker den sogenannten Syntax-Feedback-Generator auf, um Feedbacks in Bezug auf die festgestellten CEs zu erstellen. Der Feedback-Generator erhält zunächst eine Liste von CEs, für die Feedbacks erzeugt werden sollen. Im nächsten Schritt werden die CEs iteriert und Feedback mithilfe unseres Feedback-Modells erstellt. Die Felder des Feedbacks werden wie folgt ausgefüllt.

- Typ des Feedbacks: Fehler.
- Der Checker-Name: Syntax Checker.
- Systemspezifischen Namen: Compiler Error Code.
- Ort des Fehlers: Zeilennummer und Klassenname werden aus dem CE ermittelt. Dies ist ein Teil der KM-Komponente.
- Beschreibung des entdeckten Fehlers (KM-Komponente): Falls die Fehlerbeschreibung bereits im System gespeichert ist, wird diese aus der JSON-Datei (FeedbackStore) in der gewünschten Sprache extrahiert. Andernfalls wird die Compiler Error Message verwendet.

- Hinweise zur Behebung des Fehlers (KH-Komponente): Falls das System über Hinweise im Zusammenhang mit diesem Fehler verfügt, wird die KH-Komponente aus der JSON-Datei (FeedbackStore) in der gewünschten Sprache ermittelt.

Die erstellten Feedbacks werden anschließend in einem sogenannten Feedback-Manager eingefügt. Der Feedback-Manager formatiert diese Feedbacks und integriert sie in Feedback-Template. Abbildung 19 veranschaulicht die Beziehung zwischen den oben genannten Klassen. Abschließend werden die generierten Feedbacks an den Syntax-Checker bzw. an MASS übergeben und MASS speichert sie in das qf-Objekt.

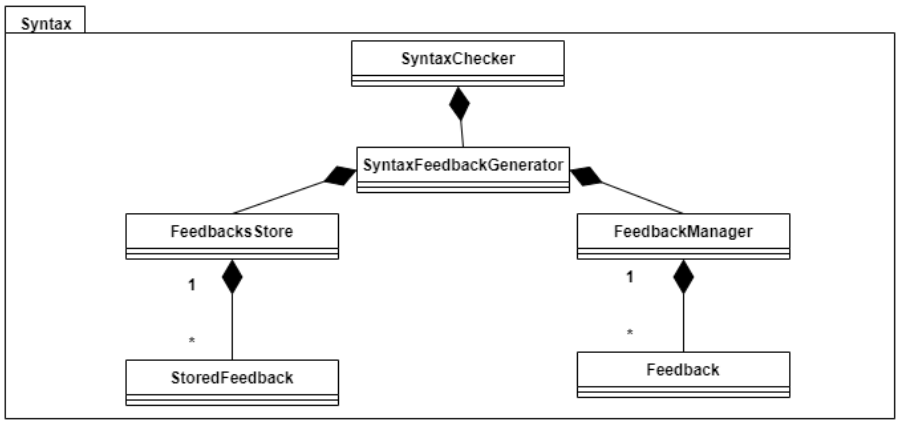


Abbildung 19 Syntax Feedback Generator

Ein Feedback zu einem CE kann aus der KM-Komponente bestehen oder durch die Erweiterung um die KH-Komponente ergänzt werden. Dies hängt von der Programmierkompetenz des Lernenden ab. In Abschnitt 2 haben wir diesen Aspekt detaillierter behandelt. Welche Komponenten dem Lernenden zur Verfügung gestellt werden, kann der Lehrer festlegen. Dies erfolgt durch die Angabe der Feedback-Stufe für CE im qf-Objekt. Es gibt hierbei zwei Stufen: ADVANCED und BEGINNER. Bei ADVANCED enthält das Feedback lediglich die KM-Komponente. Bei BEGINNER wird das Feedback durch die Erweiterung um die KH-Komponente ergänzt. Abbildung 20 zeigt das qf-Objekt mit der Feedback-Stufe BEGINNER.

```
{
  "syntax": {
    "level": "BEGINNER"
  }
}
```

Abbildung 20 Feedback-Stufe bei Syntax-Chacker.

5.3.6 Generierung von Feedback für SE

In den vorangegangenen Abschnitten wurde der Ansatz des Solution Approach Checkers und seiner Komponenten präsentiert. In diesem Abschnitt möchten wir uns nun auf den

Feedback Generator und den Prozess der Feedback-Generierung konzentrieren.

Der Feedback Generator im Solution Approach Checker funktioniert ähnlich wie der Feedback Generator im Syntax Checker. Hierbei erhält der Solution Approach Checker eine Liste von identifizierten SEs, für die Feedback generiert werden soll. Der Unterschied besteht darin, dass Lehrer Feedback bereitstellen können, wenn sie Aufgaben erstellen. Dies liegt daran, dass Lehrer ihre eigenen Lernmaterialien als Referenz verwenden können (KC-Komponente) und die anderen Feedback-Komponenten entsprechend anpassen können. Dadurch werden die im System festgelegten Feedbacks überschrieben.

Hierfür können Lehrer Feedback-Definitionen im qf-Objekt vornehmen. Abbildung 21 zeigt ein Beispiel für ein vom Lehrer definiertes Feedback bezüglich einer Methode, falls diese nicht rekursiv implementiert wurde. Dies ist ein Konfigurationseintrag für diese Checker (SolutionApproachSettingItem). Der Solution Approach Checker prüft die Methode “rekursivMethod”, ob sie in der eingereichten Lösung rekursiv gelöst wurde. Falls dies nicht der Fall ist, wird das aufgabenspezifische Feedback (TaskSpecificFeedback) anstatt des im System angelegten Feedbacks verwendet. “conceptReference” enthält die KC-Komponente und “readableCause” ist die Beschreibung des identifizierten Lösungsansatzes (KM-Komponente).

```
{
  "semanticSelected": true,
  "semantic": {
    "semantics": [
      {
        "recursive": true,
        "methodName": "rekursivMethod",
        "taskSpecificFeedbacks" : [
          {
            "technicalCause" : "solutionMustHaveRecursion",
            "readableCause" : "Die Aufgabe enthält keine Rekursion. Bitte Lösen Sie die Aufgabe rekursiv.",
            "conceptReference" : {
              "referenceName" : "Lernmaterialien",
              "referenceLink" : "https://www.Lernmaterialien.de/"
            }
          }
        ]
      }
    ]
  }
}
```

Abbildung 21 qf-Objekt mit dem von Lehrer definierten Feedback.

Die Felder des Feedbacks werden wie folgt beschrieben:

- Typ des Feedbacks: Fehler.
- Name des Checkers: Solution Approach Checker.
- Systemspezifischer Name: Name des identifizierten SE.
- Relevanter Ort: Der Name der Methode und der Klasse werden aus dem SE ermittelt. Dies ist ein Teil der KM-Komponente.
- Beschreibung des entdeckten Fehlers (KM-Komponente): Falls dies nicht von Lehrer überschrieben wird, wird die Beschreibung aus der JSON-Datei (FeedbackStore) in der

gewünschten Sprache ermittelt (StoredFeedback). Andernfalls wird sie aus den vom Lehrer definierten Feedbacks ermittelt.

- Hinweis zur Behebung des Fehlers (KH-Komponente): Falls das System über einen Hinweis zu diesem Fehler verfügt, wird die KH-Komponente aus der JSON-Datei (Feedback-Store) in der gewünschten Sprache ermittelt. Dies kann auch von Lehrer überschrieben werden.
- Referenz zu den Lernmaterialien (KH-Komponente): Falls der Lehrer dies definiert hat, wird dies verwendet.

Die generierten Feedbacks werden anschließend in einem sogenannten Feedback-Manager gespeichert. Der Feedback-Manager formatiert diese Feedbacks und integriert sie in das Feedback-Template. Abbildung 22 veranschaulicht die Beziehung zwischen den oben genannten Klassen. Am Ende werden die generierten Feedbacks an den Solution Approach Checker bzw. an MASS weitergeleitet und MASS speichert sie in das qf-Objekt.

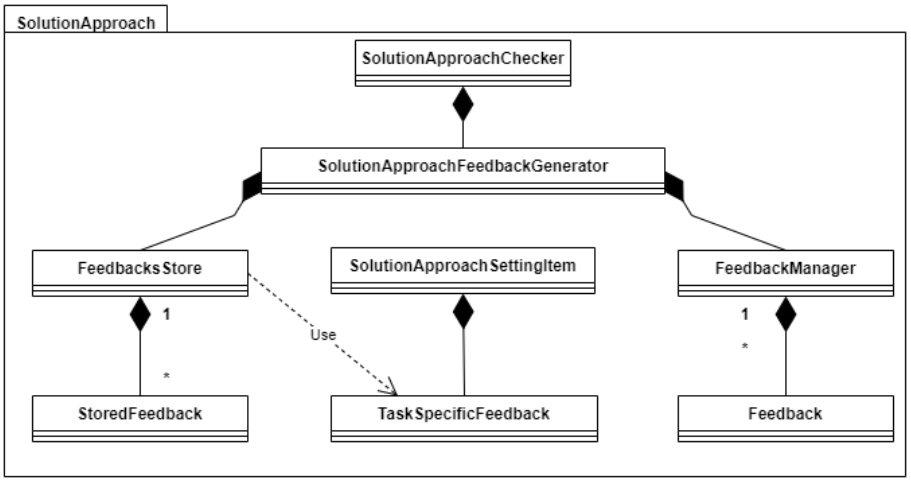


Abbildung 22 Solution Approach Feedback Generator.

5.3.7 Generierung von Feedback für SI

In den vorherigen Abschnitten wurde der Ansatz des Style Checkers und seiner Bestandteile vorgestellt. Im Folgenden konzentrieren wir uns nun auf den Feedback-Generator und den Prozess der Feedback-Erstellung.

Der Feedback-Generator im Style Checker arbeitet ähnlich wie der Feedback-Generator im Solution Approach Checker. Dabei erhält der Solution Approach Checker eine Liste von identifizierten SIs, für die Feedback generiert werden soll. Zusätzlich kann der Lehrer festlegen, ob es sich bei den identifizierten SIs um Fehler oder Verbesserungsmöglichkeit handelt.

Für diesen Zweck können Lehrer im qf-Objekt spezifizieren, ob es sich um einen Fehler oder eine Möglichkeit zur Verbesserung handelt. Abbildung X veranschaulicht, wie dies im qf-Objekt festgelegt wird.

```
{
  "styleSelected": true,
  "style": {
    "improvement": false
  }
}
```

Abbildung 23 qf-Objekt für Style Checker.

Die Felder des Feedbacks werden ebenso wie beim Solution Approach Checker ausgefüllt. Der Unterschied besteht lediglich beim Typ des Feedbacks, der entweder als Fehler oder als Verbesserungsmöglichkeit klassifiziert werden kann. Abbildung 23 verdeutlicht die Beziehung zwischen den genannten Klassen.

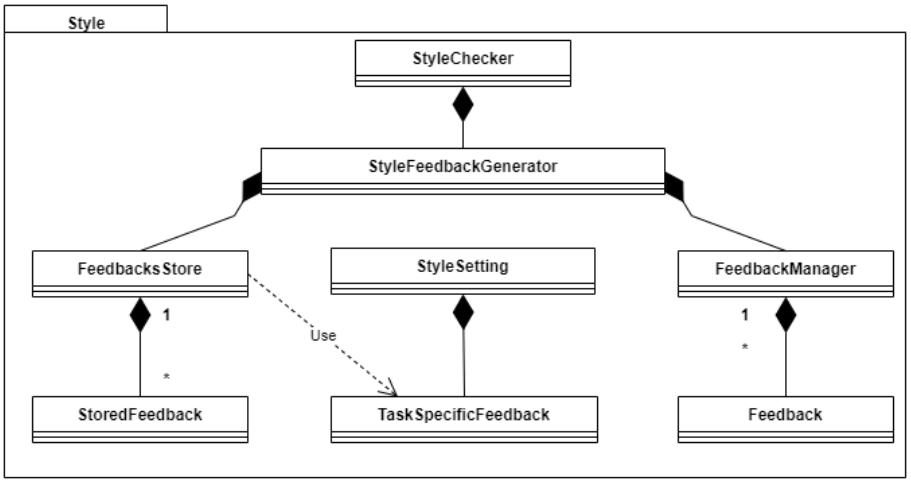


Abbildung 24 qf-Objekt mit Style Checker Konfiguration für Feedback Typ.

6 Implementierung

Im vorherigen Kapitel haben wir ein Konzept entwickelt, um die Forschungsfragen zu beantworten und die Anforderungen zu erfüllen. In diesem Kapitel stellen wir die Umsetzung dieses Konzeptes vor.

6.1 Analyse eingereichter Lösungen

In diesem Abschnitt untersuchen wir den Analyseprozess von Syntax-Checker, Solution Approach Checker und Style Checker. Zuvor stellen wir das MASS und den Lösungsvorbereitungsprozess innerhalb von MASS vor.

6.1.1 MASS

Wie bereits im letzten Kapitel erwähnt, wird für die Durchführung von MASS eine Cloud-Checks-Aktion benötigt. Diese Aktion führt den Inhalt der Datei run.sh aus. Durch diese Datei wird die Main-Methode in der Klasse CheckerRunner ausgeführt, was letztendlich MASS startet. In der Main-Methode wird die Check-Methode der Klasse Mass ausgeführt.

Die Check-Methode nimmt das qf-Objekt als Eingabe und ist das Kernstück des Systems. Das qf-Objekt enthält wichtige Informationen über die eingereichte Lösung, wie die Lösung selbst, User-Informationen wie Sprache und User-Id, die in der Klasse QfObject beschrieben und in Abbildung 25 dargestellt sind. In der Check-Methode wird zunächst die Lösung aus dem qf-Objekt ausgelesen und vorbereitet. Falls sie im Quarterfall Code-Editor geschrieben wurde, wird sie in eine temporäre JAVA-Datei überführt, andernfalls wird sie aus einer Zip-Datei heruntergeladen und temporär gespeichert. Hierfür wird die Klasse QpedQfFilesUtility verwendet.

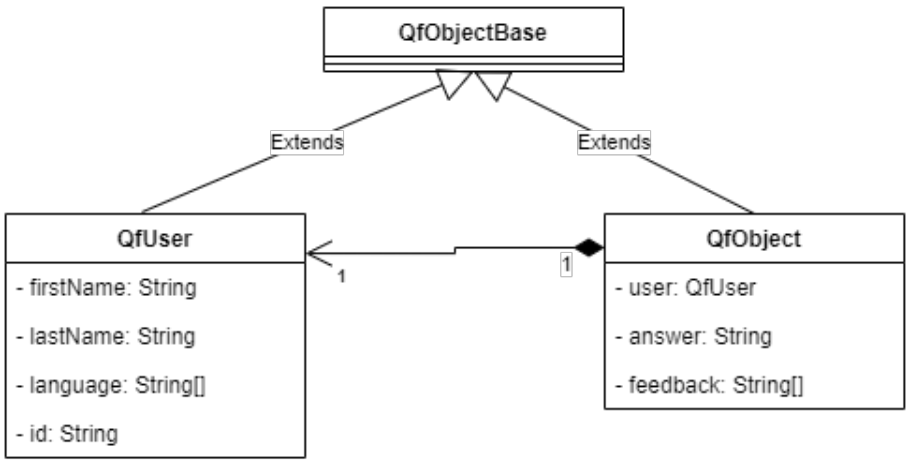


Abbildung 25 Klassen des qf-Objektes

Die Klasse Mass enthält auch Konfigurationen für MASS und die Checker. Diese Konfigurationen werden in der Klasse QfMass dargestellt und als Feld in der Klasse Mass gespeichert. In QfMass gibt es Felder wie styleSelected und semanticSelected, um die

Aktivierung bzw. Deaktivierung von Style- und Solution-Approach-Checkern zu steuern. Die Felder QfSyntaxSettings, QfStyleSettings und QfSemanticSettings beinhalten die Konfigurationen der Checker. Wir werden diese Klassen später näher betrachten.

In der Check-Methode der Klasse Mass, nach Vorbereitung der eingereichten Lösung, werden Instanzen für die aktivierten Checker erzeugt und dem MassExecutor übergeben. Der MassExecutor führt die Checker aus und gibt die generierte Feedbacks zurück an die Check-Methode in Mass. Am Ende werden Feedbacks im qf-Objekt gespeichert. Die Abbildung 26 zeigt die Architektur von MASS. Im Listing 6 wird die Check-Methode detailliert beschrieben.

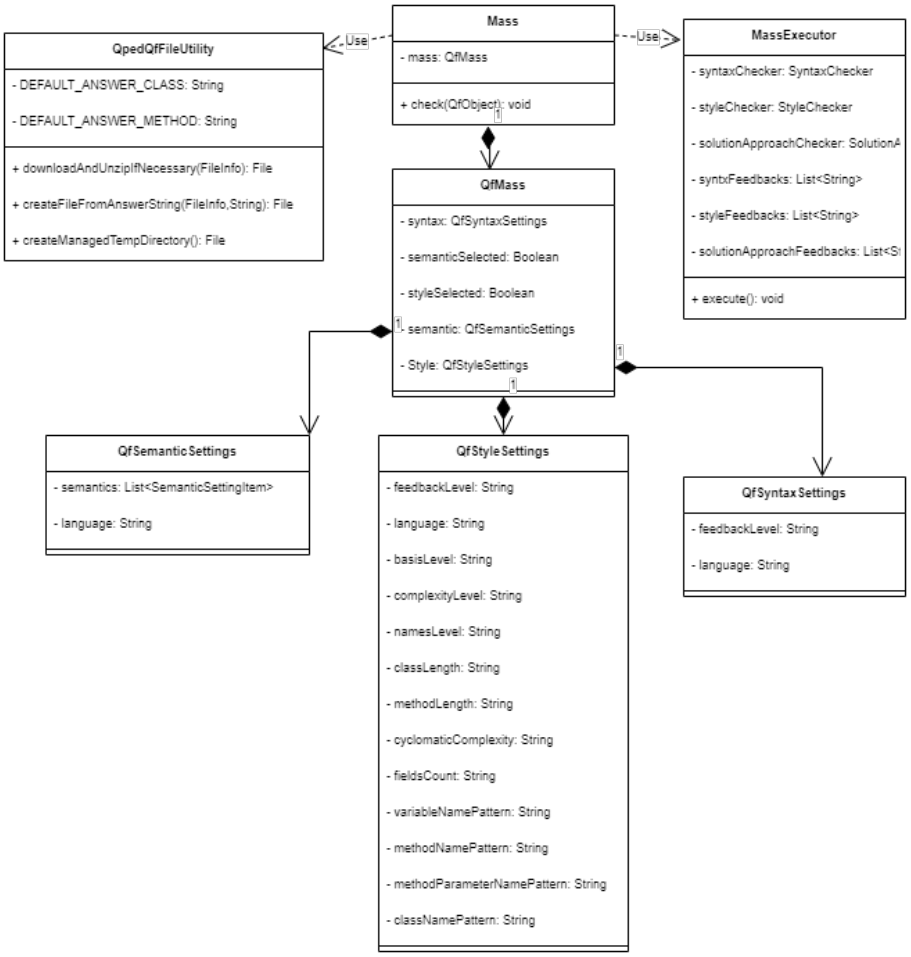


Abbildung 26 MASS Klassen

Listing 6 Check Methode in Mass.

```
1 public void check(QfObject qfObject) throws Exception {
2     // Lösung vorbereiten
3     File solutionRoot;
4     if (file != null) {
5         solutionRoot = QpedQfFilesUtility.downloadAndUnzipIfNecessary(file);
6     } else {
7         solutionRoot = QpedQfFilesUtility.createManagedTempDirectory();
```

```
        CreatedAnswerFileSummary summary = QpedQfFilesUtility.createFileFromAnswerString(
            ↪ solutionRoot, qfObject.getAnswer());
    }
    // Checker erzeugen
    MassExecutor.MassExecutorBuilder massExecutorBuilder = MassExecutor.builder();
    SyntaxChecker syntaxChecker = SyntaxChecker.builder().targetProject(solutionRoot).build();
    massExecutorBuilder.syntaxChecker(syntaxChecker);
    if (mass.isStyleSelected()) {
        StyleChecker styleChecker = StyleChecker.builder().qfStyleSettings(mass.getStyle()).build();
        massExecutorBuilder.styleChecker(styleChecker);
    }
    if (mass.isSemanticSelected()) {
        SolutionApproachChecker solutionApproachChecker = SolutionApproachChecker.builder().
            ↪ qfSemanticSettings(mass.getSemantic()).build();
        massExecutorBuilder.solutionApproachChecker(solutionApproachChecker);
    }
    MassExecutor massExecutor = massExecutorBuilder.build();
    // massExecutor ausführen und feedback generieren
    massExecutor.execute();
    var syntaxFeedbacks = massExecutor.getSyntaxFeedbacks();
    var styleFeedbacks = massExecutor.getStyleFeedbacks();
    var solutionApproachFeedbacks = massExecutor.getSolutionApproachFeedbacks();
    var resultArray = mergeFeedbacks(
            syntaxFeedbacks,
            styleFeedbacks,
            solutionApproachFeedbacks
    );
   qfObject.setFeedback(resultArray);
}
```

Der MassExecutor dient dazu, aktivierte Checker auszuführen und die erzeugten Feedbacks zu sammeln. Dies findet in der Execute-Methode innerhalb dieser Klasse statt.

Die Methode beginnt mit der Durchführung der Syntax-Checker. Dies geschieht, indem die Methode Check innerhalb der Syntax-Checker-Klasse aufgerufen wird. Der Syntax-Checker wird immer ausgeführt und kann nicht deaktiviert werden, da es ohne ihn keine weiteren Überprüfungen von SIs oder SEs möglich ist, sofern die Lösung CEs aufweist. Zudem müssen Lösungen immer kompilierbar sein. Falls CEs in der eingereichten Lösung vorliegen, werden die entsprechenden Feedbacks zu den gefundenen CEs im Feld syntaxFeedbacks gespeichert. Falls keine CEs vorliegen, werden gegebenenfalls die anderen Checker ausgeführt. Hierfür werden die Methoden Check in den Style- und Solution-Approach-Checkern aufgerufen. Feedbacks von diesen Checkern werden in den Feldern styleFeedbacks und solutionApproachFeedbacks gespeichert. Listing 7 zeigt die Methode execute in der Klasse MassExecutor.

Listing 7 execute Methode in MassExecutor.

```
public void execute() {
    syntaxFeedbacks = syntaxChecker.check();
    var syntaxAnalyseReport = syntaxChecker.getAnalyseReport();
    if (syntaxAnalyseReport.isCompilable()) {
        if (settings.isStyleNeeded()) {
            styleFeedbacks = styleChecker.check();
        }
        if (settings.isSemanticNeeded()) {
```

```
9                 solutionApproachFeedbacks = solutionApproachChecker.check();
10            }
11        }
12    }
```

6.1.2 Syntax-Checker

Im vorhergehenden Kapitel wurden Syntax-Checker und SyntaxErrorAnalyser eingeführt. In diesem Abschnitt werden wir uns die Implementierung dieser Klassen in Detail ansehen und den Prozess beschreiben.

Nachdem Mass die eingereichte Lösung im temporären Verzeichnis angelegt hat und die Check-Methode des Syntax-Checkers von MassExecuter aufgerufen wurde, wird zunächst eine Instanz des SyntaxErrorAnalysers innerhalb der Check-Methode erstellt. Der SyntaxErrorAnalyser erhält das temporäre Verzeichnis als Wurzel, in dem die eingereichte Lösung gespeichert ist. Anschließend wird die Analyse-Methode des SyntaxErrorAnalysers aufgerufen.

In dieser Methode wird zunächst die Konfiguration des Java Compilers vorgenommen. Hierfür werden folgende Konfigurationen angegeben:

- Aktivierung von Warnungen durch -Xlint.
- Generierung von Debugging-Informationen, einschließlich Zeilennummern und Quelldateiinformationen, durch -g.

Es gibt weitere Konfigurationen, die für lokale Tests und Quarterfall eingerichtet werden können. Diese Konfigurationen werden in dieser Arbeit jedoch nicht betrachtet, da sie für die Feedback-Generierung irrelevant sind. Nach der Konfiguration wird der Java-Compiler ausgeführt. Das Ergebnis des Kompilierprozesses ist eine Liste von Diagnostic, falls es Warnungen oder Kompilierfehler in der Lösung gibt. Warnungen weisen nicht auf CEs hin, sondern deuten auf Möglichkeiten zur Verbesserung des Codes und Vermeidung von Fehlern. Darüber hinaus erhält man einen booleschen Wert, der angibt, ob das Kompilieren erfolgreich war. Die Liste von Diagnostic wird in eine Liste von SyntaxError-Objekten umgewandelt. Die Felder dieser Klasse sind in Abbildung 27 dargestellt und enthalten ErrorCode, ErrorMessage, FileName, ErrorKind und Line. Aus dieser Liste von SyntaxError, dem booleschen Wert und dem Wurzelverzeichnis der Lösung wird ein SyntaxAnalysis-Report erstellt, der von der Analyse-Methode zurückgegeben wird. Listing 8 zeigt die Analyse-Methode. Der Syntax-Checker verwaltet den generierten Check-Report, indem er die SyntaxFeedbackGenerator aufruft, was in einem späteren Abschnitt näher betrachtet wird. Listing 9 zeigt die Check-Methode des Syntax-Checkers.

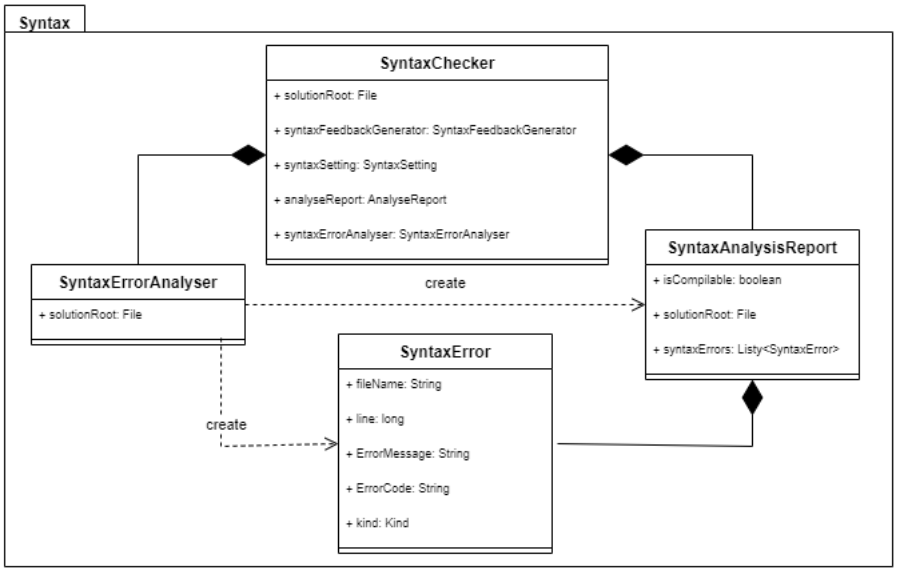


Abbildung 27 Syntax-Checker und Syntax Error Analyser

Listing 8 Analyse Methode in SyntaxErrorAnalyser.

```
public SyntaxAnalysisReport analyse() {
    SyntaxAnalysisReport.SyntaxAnalysisReportBuilder resultBuilder = SyntaxAnalysisReport.builder
        ↪ ();
    DiagnosticCollector<JavaFileObject> diagnosticsCollector = new DiagnosticCollector<>();
    boolean compileResult = compile(diagnosticsCollector); // Compiler konfigurieren und Lösung
        ↪ kompilieren.
    resultBuilder.isCompilable(compileResult);
    List<SyntaxError> collectedErrors = analyseDiagnostics(diagnosticsCollector.getDiagnostics());
    resultBuilder.syntaxErrors(collectedErrors);
    resultBuilder.path(solutionRoot);
    return resultBuilder.build();
}
```

Listing 9 Check Methode in Syntax-Checker.

```
public List<String> check() {
    if (syntaxErrorAnalyser == null) {
        syntaxErrorAnalyser = SyntaxErrorAnalyser
                .builder()
                .solutionRoot(targetProject)
                .build();
    }
    analyseReport = syntaxErrorAnalyser.analyse();
    if (syntaxFeedbackGenerator == null) {
        syntaxFeedbackGenerator = SyntaxFeedbackGenerator.builder().build();
    }
    return syntaxFeedbackGenerator.generateFeedbacks(analyseReport.getSyntaxErrors(),
        ↪ syntaxSetting);
}
```

6.1.3 Solution Approach Checker

Im vorherigen Kapitel wurden die Klassen Solution Approach Checker und Solution Approach Analyser eingeführt. In diesem Abschnitt werden wir uns die detaillierte Imple-

mentierung dieser Klassen ansehen und den entsprechenden Prozess beschreiben.

Bevor aber der Checker und seiner Analyser vorgestellt werden, betrachten wir die Konfiguration dieser Checker genauer. Dazu verwenden wir Abbildung 28. Die Klasse QfSemanticSettings wird für die Konfiguration des Checkers verwendet. Hier werden die Sprache und die Konfigurationseinträge aus dem qf-Objekt ermittelt. Jeder Konfigurationseintrag ist für eine Methode in einer Java-Datei verantwortlich und heißt SemanticSettingItem. In diesen werden die erwarteten Werte für eingereichte Lösungen definiert. Ein Beispiel hierfür ist der Rückgabetyp der Methode “returnType”.

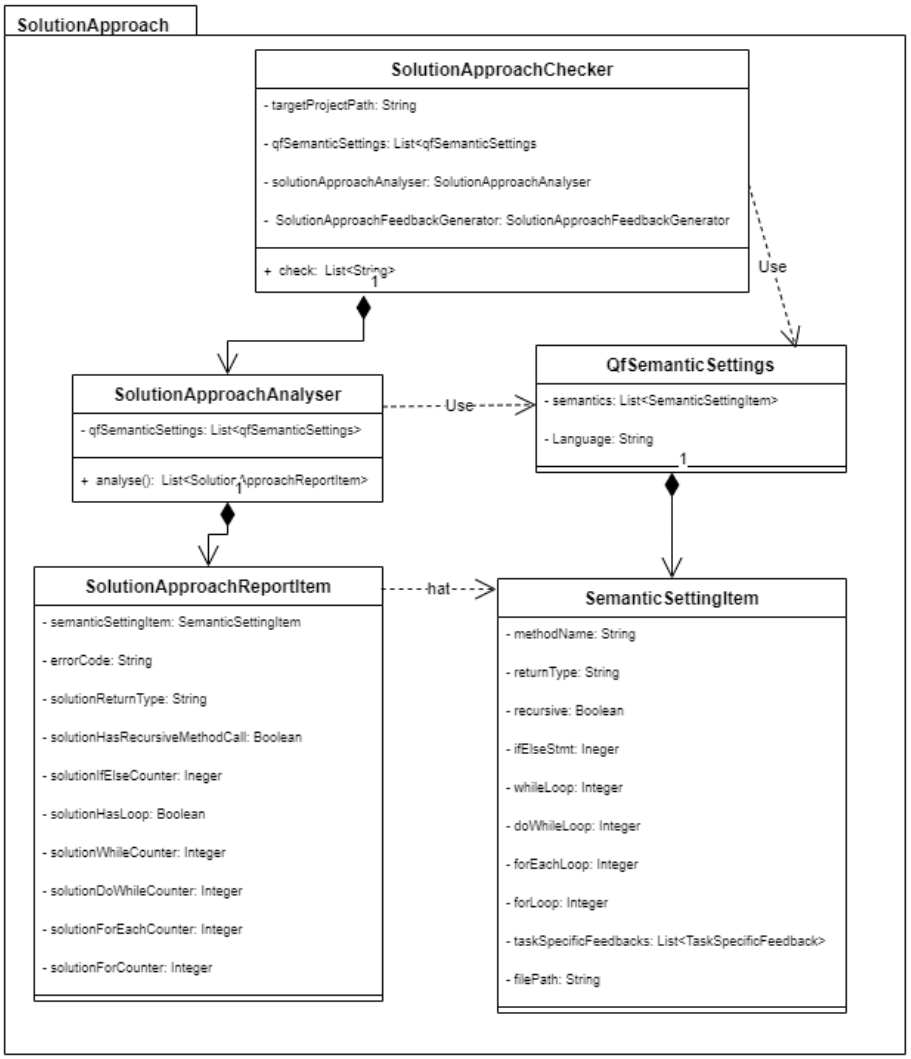


Abbildung 28 Solution Approach Checker und Solution Approach Analyser

Nachdem der MassExecuter den Syntax-Checker aufgerufen hat und keine Compiler-Fehler (CEs) in der eingereichten Lösung gefunden wurden, kann der Solution Approach Checker, falls erforderlich, aufgerufen werden, indem die Check-Methode aufgerufen wird. Diese Methode ist in Abbildung 10 dargestellt. Hierbei wird zunächst eine Instanz der Klasse SolutionApproachAnalyser erstellt. Der Analyser erhält das temporäre Verzeichnis der Lösung und die Checker-Konfigurationen. Anschließend wird die Analyse-Methode des

Analysers aufgerufen, die eine Liste von SolutionApproachReportItems zurückgibt. Über diese Liste werden Feedback-Informationen generiert, die in einem späteren Abschnitt behandelt werden.

Listing 10 Check Methode in Solution Approach Checker.

```
public List<String> check() {
    if (solutionApproachAnalyser == null) {
        solutionApproachAnalyser = SolutionApproachAnalyser
                .builder()
                .qfSemanticSettings(qfSemanticSettings)
                .targetProjectPath(targetProjectPath)
                .build();
    }
    var solutionApproachReportEntries = solutionApproachAnalyser.analyse();

    if (solutionApproachFeedbackGenerator == null) {
        solutionApproachFeedbackGenerator = SolutionApproachFeedbackGenerator.builder().build
            ↪ ();
    }
    return solutionApproachFeedbackGenerator.generateFeedbacks(solutionApproachReportEntries,
        ↪ qfSemanticSettings);
}
```

In der Analyse-Methode werden die Konfigurationseinträge iteriert und mit der eingereichten Lösung verglichen. Falls die eingereichte Lösung ein SE aufweist, wird ein SolutionApproachReportItem erstellt. Das SolutionApproachReportItem enthält die tatsächlichen Werte, wie z. B. den tatsächlichen Rückgabetyp der Methode, sowie eine Referenz zu dem Konfigurationseintrag, in dem der erwartete Wert angegeben ist, z. B. der gewünschte Rückgabetyp der Methode. Darüber hinaus enthält das SolutionApproachReportItem einen Error-Code, mit dessen Hilfe der entdeckten SE identifiziert werden kann. Die detailliertere Betrachtung des Analysers wird in der Arbeit [3] vorgenommen. Folgende Error-Codes werden vom Analyser generiert:

- **DifferentReturnTypeThanExpected**: Die eingereichte Lösung verwendet einen anderen Rückgabetyp als erwartet.
- **SolutionMustHaveRecursion**: Die Methode muss einen rekursiven Aufruf enthalten.
- **SolutionMustHaveRecursionInsteadLoop**: Die Methode muss einen rekursiven Aufruf statt einer Schleife enthalten.
- **SolutionMustNotHaveRecursion**: Die Methode darf keinen rekursiven Aufruf enthalten.
- **MoreThanExpectedWhileLoops**: Es werden mehr While-Schleifen verwendet als erwartet.
- **MoreThanExpectedForLoops**: Es werden mehr For-Schleifen verwendet als erwartet.
- **MoreThanExpectedForEachLoops**: Es werden mehr ForEach-Schleifen verwendet als erwartet.
- **MoreThanExpectedIfElseStatements**: Es werden mehr IfElse-Anweisungen verwendet als erwartet.

- **MoreThanExpectedDoWhileLoops**: Es werden mehr DoWhile-Schleifen verwendet als erwartet.

6.1.4 Style Checker

Im vorangegangenen Kapitel wurden der Style Checker und der Style Analyser eingeführt. In diesem Abschnitt werden wir uns mit der Implementierung dieser Klassen befassen und den Prozess beschreiben.

Die Konfiguration des Style-Checkers wurde in Abbildung 26 dargestellt. Hier werden Einstellungen wie die Check-Stufe für die Komplexität des Codes, Benennungen und andere SIs festgelegt. Eine detailliertere Beschreibung findet sich in der Arbeit Y; daher wird in dieser Arbeit nicht näher darauf eingegangen.

Nach erfolgreichem Abschluss des Syntax-Checks durch den MassExecuter kann, falls gewünscht, der Style-Checker aufgeführt werden, indem die Check-Methode aufgerufen wird. Der Aufbau des Style-Checkers und die verwendeten Klassen sind in Abbildung 29 dargestellt. Darüber hinaus ist die Check-Methode in Listing 11 gezeigt. Zu Beginn wird eine Instanz der Klasse StyleAnalyser generiert. Der Analyser erhält das temporäre Verzeichnis der Lösung sowie die Checker-Konfiguration. Anschließend wird die Analyse-Methode des Analysers aufgerufen, die ein Objekt der Klasse StyleAnalysisReport zurückgibt. Dieses Objekt stellt einen Bericht über die identifizierten SIs dar. Eine SI wird durch die Klasse Violation repräsentiert und durch die Klasse ReportEntryFile nach dem zugehörigen Dateinamen in der eingereichten Lösung gruppiert. Eine Verletzung enthält Informationen wie die Start- und End-Zeile, den Namen der Regel (PMD Rule Name) und den Namen des Regelsets. Diese Informationen können genutzt werden, um Feedback zu generieren, das in späteren Kapiteln näher betrachtet wird.

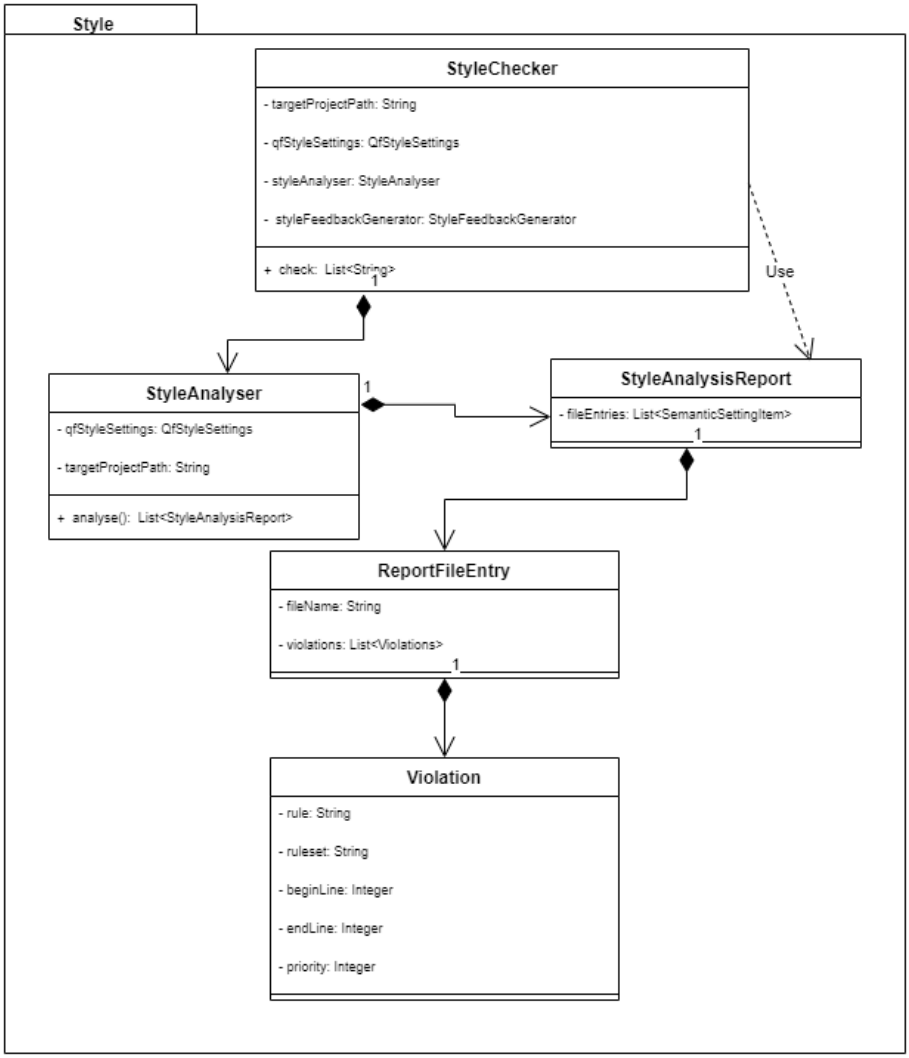


Abbildung 29 Style Checker und Style Analyser

Listing 11 Check Methode in Style Checker.

```
public List<String> check() {
    if (styleAnalyser == null) {
        styleAnalyser = StyleAnalyser.builder()
                .styleSettings(styleSettings)
                .targetPath(targetPath)
                .build()
        ;
    }
    var styleReport = styleAnalyser.analyse();
    if (styleFeedbackGenerator == null) {
        styleFeedbackGenerator = StyleFeedbackGenerator.builder().build();
    }
    return styleFeedbackGenerator.generateFeedbacks(styleReport, styleSettings);
}
```

6.2 Feedback Framework

Im vorangegangenen Kapitel wurde ein Modell für Feedback und die Komponenten der Feedback-Schnittstelle entworfen. Im vorliegenden Kapitel wird dieses Modell vertieft und dessen Implementierung vorgestellt. Das Kapitel gliedert sich in zwei Teile. Der erste Teil befasst sich mit dem Feedback-Modell, dem Formatieren des Feedbacks und der Einrichtung von Feedback-Templates. Im zweiten Teil geht es um Feedback-Informationen, die im System gespeichert sind, sowie um solche, die vom Lehrer selbst definiert werden.

6.2.1 Feedback-Modell

Das Feedback Modell heben wir in dem letzten Kapitel entworfen. Nun werden wir die Implementierung des Modells vorstellen, was in Abbildung 17 vorgestellt wurde. Das Feedback haben wir durch die Klasse Feedback implementiert, welche die KM-, KH- und KC-Komponenten für CE, SE und SI enthält.

Die KM-Komponente besteht aus Informationen über die Beschreibung des aufgetretenen Fehlers sowie dessen Position. Zur Beschreibung des Fehlers wird das Feld “readableCause” vom Typ “String” verwendet. Dieses Feld darf nicht Null sein, da es dem Lernenden ermöglicht, über entdeckte Fehler oder Verbesserungsmöglichkeiten zu informieren. Um dies sicherzustellen, wurde die Lombok-Annotation “@NonNull” verwendet. Dadurch wird beim Erstellen eines Objekts der Klasse “Feedback” die Erstellung des “readableCause”-Feldes erzwungen, um ein “NullPointerException” zu vermeiden. Zusätzlich kann das Feld mit Markdown formatiert werden, was in einem späteren Abschnitt erläutert wird.

Um den Ort des Fehlers anzugeben, wurde das Feld “relatedLocation” vom Typ “RelatedLocation” zur Klasse “Feedback” hinzugefügt. Dieses Feld kann Null sein, da nicht alle Feedbacks einen bestimmten Ort haben müssen. Beispielsweise kann dies der Fall sein, wenn der Lernende keine Java-Dateien in seiner Lösung eingereicht hat. Obwohl dies keine CE, SI oder SE ist, muss dieser Fall von der Klasse “Feedback” behandelt werden. Die Klasse "RelatedLocationënthält Informationen wie den Dateinamen (vom Typ “String”), den Methodennamen (vom Typ “String”), die Startzeile (vom Typ “int”) und die Endzeile (vom Typ “int”). Die Felder können entweder Null oder mit Werten belegt sein.

Die KH-Komponente setzt sich aus einer Liste von Hinweisen zusammen, die entweder als Markdown-formatierter Text oder als Codebeispiel vorliegen können. Hierfür verfügt die Klasse “Feedback” über ein “hints”-Feld, das eine Liste von Objekten des Typs “Hint” enthält. Die Klasse “Hint” besteht aus dem Inhalt “content” vom Typ “String” sowie dem Typ des Hinweises, der entweder “CODE_EXAMPLE” oder “TEXT” sein kann. Um diese Entscheidung abzubilden, wurde der Aufzählungstyp “HintType” implementiert. Dies ermöglicht eine Erweiterung des Aufzählungstyps, um weitere Formatierungsmöglichkeiten für die KH-Komponente hinzuzufügen. Es ist möglich, dass die KH-Komponente leer ist, indem die “hints”-Liste eines Feedbacks leer ist.

Die KC-Komponente hat den Fokus auf dem zu prüfenden Konzept oder Stoff einer Aufgabe. Da diese Komponente als Referenz für ein Konzept betrachtet wird, enthält sie die folgenden Informationen: den Namen des Lernmaterials vom Typ “String”, einen

Link zur Referenz vom Typ SString", die Seitenzahl im genannten Lernmaterial vom Typ “List<Integer>” und die Abschnittsnamen im Lernmaterial vom Typ “String”. Diese Informationen wurden durch die Implementierung der Klasse “ConceptReference” umgesetzt. Das Feedback-Objekt verfügt nun über ein Feld namens “reference”, das vom Typ “ConceptReference” ist.

Um festzulegen, ob es sich bei einem Feedback um eine Verbesserungsmöglichkeit oder einen Fehler handelt, haben wir ein Feld namens “type” in der Klasse Feedback hinzugefügt. Dieses Feld ist ein Aufzählungstyp und die möglichen Werte sind “IMPROVEMENT” oder “CORRECTION”.

Der Name des Checkers wird in der Klasse Feedback durch das Feld “checkerName” gespeichert, das vom Typ “String” ist.

Um den systemspezifischen Namen des entdeckten Fehlers oder der Verbesserungsmöglichkeit zu speichern, verfügt die Klasse Feedback über das Feld “technicalCause” vom Typ “String”. Dieses Feld enthält den eindeutigen Namen des Fehlers oder der Verbesserungsmöglichkeit im System, um eine klare Identifikation zu ermöglichen.

Alle oben genannten Felder sind private Felder und können durch Getter- und Setter-Methoden von außerhalb des Frameworks zugegriffen werden.

Um zu überprüfen, ob das Feedback bereits formatiert wurde, verfügt die Klasse Feedback über ein privates boolesches Feld namens “isFormatted”, das nur innerhalb des Frameworks geändert werden kann. Der Grund hierfür liegt darin, dass der Formatierungsprozess ausschließlich vom Framework durchgeführt werden sollte und nicht von externen Quellen. Aus diesem Grund haben wir die Setter-Methode als “protected” deklariert, um den Zugriff auf das Feld von außerhalb des Frameworks zu beschränken.

Das Erstellen einer Instanz der Klasse Feedback wurde mithilfe des Builder-Patterns realisiert. Hierbei können zunächst die erforderlichen Felder eines Feedbacks ermittelt und anschließend über den FeedbackBuilder eingefügt werden. Die eigentliche Erstellung des Feedback-Objekts erfolgt durch Aufruf der build-Methode des FeedbackBuilders.

6.2.2 Feedback-Formatierer

Die Klasse MarkdownFeedbackFormatter in Abbildung 30 ist dafür zuständig, Feedbacks in Markdown-Format zu formatieren, insbesondere die Codebeispiele der KH-Komponente. Dadurch können sich die anderen Komponenten des Systems auf ihre jeweiligen Aufgaben konzentrieren, ohne sich um die Formatierung kümmern zu müssen. Darüber hinaus besteht die Möglichkeit, Feedback-Informationen in Markdown-Format zu bringen. Hierfür haben wir die Klasse MarkdownFormatterUtility implementiert, die statische Methoden zur Formatierung von Texten wie readableCause bereitstellt. Diese Klasse umfasst folgende Methoden:

- asHeading4(String toFormat): String
- asBold(String toFormat): String
- asItalic(String toFormat): String
- asCodeBlock(String toFormat, String language): String

- asCodeLine(String toFormat): String
- asLink(String toFormat, String url): String

Die Klasse Feedback verfügt über eine protected Methode namens format(), die das Feedback formatiert. Der Aufruf dieser Methode ist auf den FeedbackManager beschränkt, welcher in einem späteren Abschnitt vorgestellt wird.

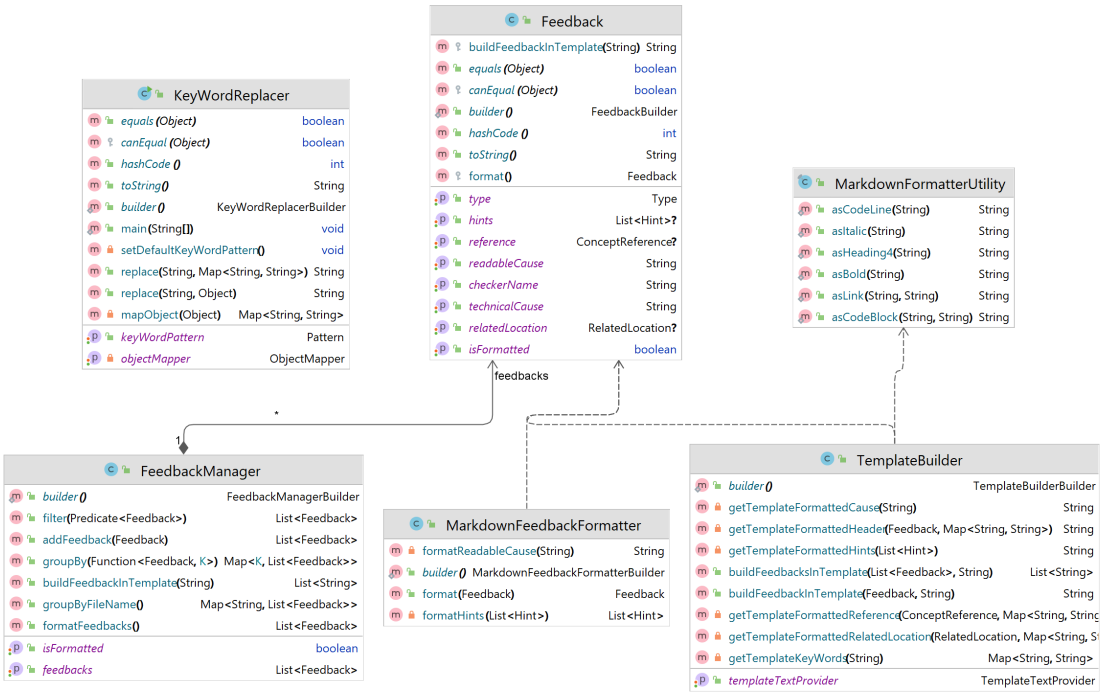


Abbildung 30 Klassen der Feedback-Frameworks

6.2.3 Schlüsselwörter

Ein weiterer wichtiger Aspekt bei der Generierung von Feedback ist die Ersetzung von Schlüsselwörtern. Im Rahmen des Frameworks ist die Implementierung dieses Prozesses von Bedeutung, da sie anderen Systemkomponenten, wie beispielsweise dem Checker, bei der Ersetzung von Schlüsselwörtern unterstützt.

Das Ersetzen von Schlüsselwörtern wird mithilfe der KeyWordReplacer-Klasse aus Abbildung 30 durchgeführt. Diese Klasse bietet verschiedene Funktionen an, die für das Ersetzen von Schlüsselwörtern zuständig sind. Zu diesen Funktionen gehören:

- replace(String target, String toReplace, String replacement): Ersetzt ein bestimmtes Schlüsselwort in einem Text (String) durch einen bestimmten Wert.
- replace(String target, Map<String, String> replacements): Ersetzt mehrere Schlüsselwörter in einem Text durch ihre entsprechenden Werte, die in einer Map gespeichert sind.
- replace(String target, Object object): Ersetzt Felder eines Objekts durch ihre Werte. Hierbei werden auch Objekte berücksichtigt, auf die das ursprüngliche Objekt verweist.

Standardmäßig sollten die Schlüsselwörter im Text das Pattern “{$*}” aufweisen, wobei “*” das eigentliche Schlüsselwort darstellt. Dieses Muster kann jedoch nach Bedarf geän-

dert werden, indem das Feld “keyWordPattern” des KeyWordReplacer-Objekts angepasst wird.

Es ist außerdem zu beachten, dass bei Verwendung der “replace”-Methoden das Muster für das Schlüsselwort nicht explizit angegeben werden muss. Stattdessen wird lediglich das Schlüsselwort als Parameter übergeben. Ein Beispiel dafür wäre die Verwendung des Schlüsselworts “Filename” anstelle des Musters “{$Filename}”.

6.2.4 Feedback Template

Zur Formatierung des Feedbacks in einem Template wurde die Klasse TemplateBuilder in Abbildung 30 entwickelt. Sie verfügt über die Methode "buildFeedbackInTemplate", die ein Feedback-Objekt und die gewünschte Sprache als Parameter erhält. Die Methode wandelt das Feedback-Objekt in einen String um, der das Feedback in einem Template und in der ausgewählten Sprache enthält.

6.2.5 Feedback-Manager

Der FeedbackManager aus Abbildung 30 ist eine Klasse, die sich mit der Verwaltung von Feedbacks beschäftigt. Sie beinhaltet eine Liste von Feedback-Objekten, auf welche verschiedene Funktionen angewandt werden können. Dazu gehören:

- formatFeedbacks(): Diese Funktion dient dazu, existierende Feedbacks im Manager zu formatieren.
- addFeedback(Feedback feedback): Hiermit kann ein neues Feedback zum Manager hinzugefügt werden.
- buildFeedbackInTemplate(@NonNull String language): Mithilfe dieser Funktion können Feedbacks des Managers in einem Template in einer bestimmten Sprache generiert werden.
- filter(Predicate<Feedback> filter): Diese Funktion ermöglicht das Filtern von Feedbacks im Manager anhand eines übergebenen Prädikats.
- groupBy(Function<Feedback, K> groupByFunction): Hiermit können Feedbacks im Manager anhand einer übergebenen Funktion gruppiert werden. Das Ergebnis ist eine Map mit den gruppierten Feedbacks.
- groupByFileName(): Diese Funktion gruppiert die Feedbacks im Manager nach Dateinamen. Das Ergebnis ist eine Map mit den gruppierten Feedbacks.

6.2.6 Feedback-Inhalte

Im vorherigen Kapitel wurde festgelegt, welche Feedback-Komponenten bzw. -Felder im System gespeichert werden sollten oder von Lehrern definiert werden können, um ein qualitativ hochwertigeres Feedback im Vergleich zum Standard-Feedback zu generieren. In diesem Abschnitt wird die Implementierung dieser Komponenten bzw. Felder behandelt.

Die Feedback-Inhalte, die im System gespeichert werden, sind in mehrere JSON-Dateien unterteilt. Diese Unterteilung erfolgt anhand von zwei Kriterien: die Sprache und der Art des Fehlers (CE, SI oder SE). In Abbildung 31 sind die erstellten JSON-Dateien dargestellt. Die Begründung für diese Aufteilung liegt darin, dass nicht relevante Feedback-Inhalte nicht jedes Mal gelesen werden müssen, um Feedback zu generieren. Zum Beispiel muss

der Syntax-Checker die in der JSON-Datei gespeicherten Feedback-Inhalte für SI in Deutsch nicht lesen. Durch diese Aufteilung können somit nur relevante Feedback-Inhalte in das Feedback einbezogen werden, was die Effizienz des Systems erhöht.

Jede JSON-Datei enthält eine Liste von gespeicherten Feedbacks, die durch die Klasse StoredFeedback in Abbildung 32 repräsentiert wird. Die Felder dieser Klasse sind ein Teil des Feedback-Modells und umfassen technicalCause, readableCause, und hints.

Um den Pfad zur entsprechenden JSON-Datei zu ermitteln, kann die Methode provideStoredFeedbackDirectory aus der Klasse StoredFeedbackDirectoryProvider verwendet werden. Diese Methode benötigt lediglich den Checker als Parameter, um den Pfad zur relevanten JSON-Datei zu liefern.

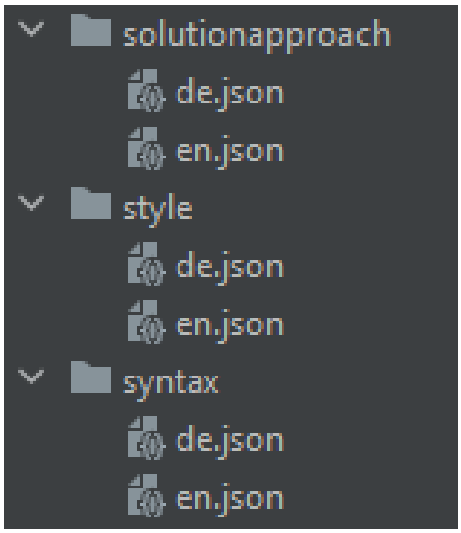


Abbildung 31 JSON-Datein für Feedback-Inhalte

Die Klasse FeedbacksStore ist zuständig für das Ermitteln und Zwischenspeichern der im System gespeicherten Feedback-Inhalte und verfügt daher über eine Liste von StoredFeedback-Objekten. Um die Feedback-Inhalte aus der entsprechenden JSON-Datei zu lesen, benötigt diese Klasse im Konstruktor sowohl die gewünschte Sprache als auch den Pfad zur entsprechenden JSON-Datei.

Feedback-Inhalte, die von Lehrern definiert werden, können im qf-Objekt an beliebiger Stelle angelegt werden, da die genaue Position für das Framework nicht relevant ist und vom Checker festgelegt werden kann. Das Framework ist jedoch in der Lage, diese Feedback-Inhalte darzustellen. Hierzu dient die Klasse TaskSpecificFeedback aus Abbildung 32, welche ein Teil des Feedback-Modells ist und folgende Felder enthält: technicalCause, readableCause, hints und conceptReference. Die Klasse FeedbacksStore ist auch für die Verwaltung der von Lehrern definierten Feedback-Inhalte zuständig und verfügt daher über eine Liste von TaskSpecificFeedback-Objekten.

Um ein oder mehrere TaskSpecificFeedback-Objekte dem FeedbacksStore hinzuzufügen, wird die Methode customizeStore(List<TaskSpecificFeedback> taskSpecificFeedbacks) aufgerufen. Über die Methode getRelatedFeedbackByTechnicalCause(String technicalCause) kann man dann Feedback-Inhalt erhalten, die zu einem gegebenen systemspezifischen Namen passt. Die Feedback-Inhalte werden standardmäßig aus den gespeicherten Feedbacks

ermittelt. Jedoch kann ein TaskSpecificFeedback-Objekt Teile der Feedback-Inhalte überschreiben. Beispielsweise wird die KC-Komponente aus dem TaskSpecificFeedback-Objekt ausgelesen, während die restlichen Feedback-Inhalte aus den gespeicherten Feedbacks ausgelesen werden. Um die zwischengespeicherten TaskSpecificFeedback-Objekte zu löschen, wird die Methode rebuildStore() aufgerufen.

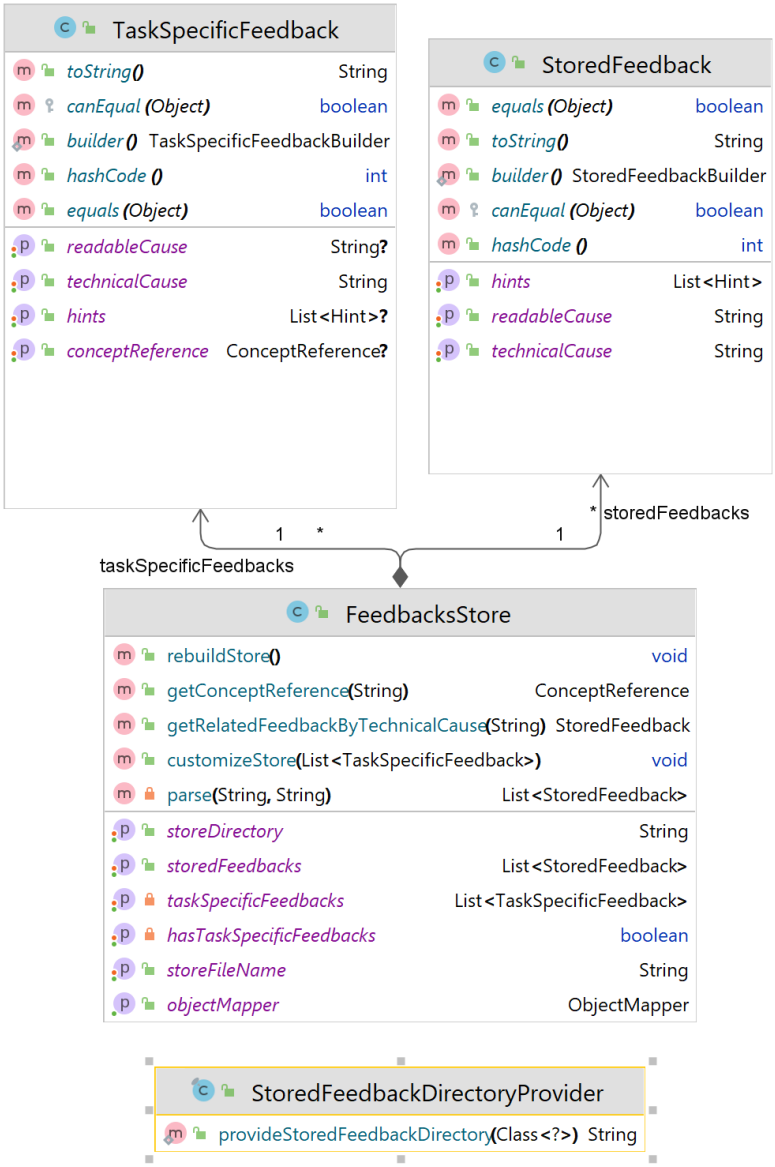


Abbildung 32 Feedback-Inhalte und FeedbackStore

6.3 Generierung von Feedback

In den vorherigen Kapiteln haben wir den Feedback-Generierungsprozess für CE, SI und SE beschrieben. In diesem Kapitel werden wir uns nun genauer mit der Implementierung der Feedback-Generierung für CE, SI und SE befassen.

6.3.1 Generierung von Feedback für CE

Das Generieren von Feedbacks im Zusammenhang mit CE erfolgt durch die Verwendung der Methode generateFeedbacks des SyntaxFeedbackGenerator-Objekts. Diese Methode wird von der Check-Methode eines SyntaxChecker-Objekts aufgerufen, wie in Abschnitt 6.1.2 beschrieben. Die Eingabe der Methode besteht aus SyntaxSetting und einer Liste von gefundenen SyntaxError, für die Feedbacks generiert werden sollen. Listing 12 zeigt die Implementierung dieser Methode.

Listing 12 Generieren von Feedbacks für CEs.

```
1 public List<String> generateFeedbacks(List<SyntaxError> errors, SyntaxSetting syntaxSetting) {
2     List<Feedback> nakedFeedbacks = generateNakedFeedbacks(errors, syntaxSetting);
3     nakedFeedbacks = adaptFeedbackByCheckLevel(syntaxSetting, nakedFeedbacks);
4     if (feedbackManager == null) feedbackManager = FeedbackManager.builder().build();
5     feedbackManager.setFeedbacks(nakedFeedbacks);
6     return feedbackManager.buildFeedbackInTemplate(syntaxSetting.getLanguage());
7 }
```

Zunächst wird eine Liste von nicht formatierten Feedbacks generiert, indem die Methode generateNakedFeedbacks aus Listing 13 aufgerufen wird. Diese Methode erstellt zunächst eine Instanz der Klasse FeedbacksStore und iteriert dann über die SyntaxError-Liste. In jeder Iteration wird ein Feedback für jeden gefundenen SyntaxError durch die FeedbackBuilder-Klasse generiert. Die KM-Komponente und die KH-Komponente werden vom FeedbacksStore ermittelt. Falls jedoch kein entsprechender Feedback-Inhalt im FeedbackStore vorhanden ist, wird die ErrorMessage aus dem SyntaxError als technicalCause und readableCause verwendet.

Nach der Iteration wird eine Feedback-Liste zurückgegeben, die anhand des Feedback-Levels angepasst wird. Falls das SyntaxSetting das Scheck-Level "ADVANCED" besitzt, werden die KH-Komponenten aus den generierten Feedbacks entfernt, da sie für Programmiererfahrene nicht benötigt werden. Andernfalls bleiben die generierten Feedbacks unverändert. Die generierten Feedbacks werden in einen FeedbackManager eingefügt, der die Feedbacks am Ende formatiert und in ein Template einfügt. Das Ergebnis ist eine Liste von Strings, die an den Checker weitergeleitet wird.

Zusammenfassend wird deutlich, dass das Generieren von Feedbacks im CE-Kontext eine Reihe von Schritten erfordert, darunter das Erstellen von FeedbacksStore- und FeedbackBuilder-Instanzen, das Iterieren über SyntaxFehler-Listen und das Anpassen der generierten Feedbacks anhand des Feedback-Levels. All diese Schritte dienen dazu, den Lernenden angemessenes Feedback zu geben und ihnen bei der Korrektur ihres Codes zu helfen.

Listing 13 Generierung von unformatierten Feedbacks

```
1 private List<Feedback> generateNakedFeedbacks(List<SyntaxError> errors, SyntaxSetting
      ↪ syntaxSetting) {
2     List<Feedback> result = new ArrayList<>();
3     if (feedbacksStore == null) {
4         feedbacksStore = new FeedbacksStore(
5                 provideStoredFeedbackDirectory(SyntaxChecker.class)
```

```
                , syntaxSetting.getLanguage() + FileExtensions.JSON
        );
    }
    for (SyntaxError error : errors) {
        var feedbackBuilder = Feedback.builder();
        feedbackBuilder.type(Type.CORRECTION);
        feedbackBuilder.checkerName(SyntaxChecker.class.getSimpleName());
        feedbackBuilder.relatedLocation(RelatedLocation.builder()
                .startLine((int) error.getLine())
                .methodName("")
                .fileName(
                        error.getFileName() != null && error.getFileName().equals("TestClass.java") ?
                            ↪ "" : error.getFileName()
                )
                .build()
        );
        if (feedbacksStore.getRelatedFeedbackByTechnicalCause(error.getErrorMessage()) != null) {
            feedbackBuilder.updateFeedback(feedbacksStore.getRelatedFeedbackByTechnicalCause(
                ↪ error.getErrorMessage()));
        } else if (feedbacksStore.getRelatedFeedbackByTechnicalCause(error.getErrorCode()) != null) {
            ↪
            feedbackBuilder.updateFeedback(feedbacksStore.getRelatedFeedbackByTechnicalCause(
                ↪ error.getErrorCode()));
        } else {
            feedbackBuilder.technicalCause(error.getErrorMessage());
            feedbackBuilder.readableCause(error.getErrorMessage());
        }
        result.add(feedbackBuilder.build());
    }
    return result;
}
```

6.3.2 Generierung von Feedback für SE

In Bezug auf SE wird die Generierung von Feedbacks durch die Verwendung der Methode "generateFeedbacks" des "SolutionApproachFeedbackGenerator"-Objekts realisiert. Der "SolutionApproachFeedbackGenerator" ist ähnlich aufgebaut wie die Klasse "SyntaxFeedbackGenerator". Diese Methode wird von der "Check"-Methode eines "SolutionApproachChecker"-Objekts aufgerufen, wie im Abschnitt 6.1.3 beschrieben. Die Eingabe für diese Methode besteht aus "QfSemanticSettings" und einer Liste von "SolutionApproachReportItem", für die Feedbacks generiert werden sollen. Die Implementierung dieser Methode ist in Listing 14 dargestellt.

Listing 14 Generieren von Feedbacks für SEs.

```
public List<String> generateFeedbacks(List<SolutionApproachReportItem> reportEntries,
    ↪ QfSemanticSettings checkerSetting) {
    List<Feedback> nakedFeedbacks = generateNakedFeedbacks(reportEntries, checkerSetting);
    if (feedbackManager == null) feedbackManager = FeedbackManager.builder().build();
    feedbackManager.setFeedbacks(nakedFeedbacks);
    return feedbackManager.buildFeedbackInTemplate(checkerSetting.getLanguage());
}
```

Zunächst erfolgt die Generierung von unformatierten Feedbacks durch Aufruf der Methode “generateNakedFeedbacks” aus Listing 15. Dabei wird eine Instanz der Klasse “FeedbacksStore” erstellt und über die Liste “SolutionApproachReportItem” iteriert. In jeder Iteration wird überprüft, ob das zugehörige “SemanticSettingItem” über “TaskSpecificFeedbacks” verfügt. Falls dies der Fall ist, wird der “FeedbacksStore” temporär angepasst, indem Feedback-Inhalte überschrieben werden. Anschließend wird ein Feedback für das jeweilige “SolutionApproachReportItem” mithilfe der Klasse “FeedbackBuilder” generiert. Die Komponenten KM, KH und KC werden dabei über den “FeedbacksStore” ermittelt. Zudem werden Schlüsselwörter durch die Methode “replaceKeyWords” aus dem “FeedbackBuilder” ersetzt, wobei die Werte aus dem “SolutionApproachReportItem” herangezogen werden.

Nach der Iteration wird eine Feedback-Liste zurückgegeben, werden die generierten Feedbacks in einen “FeedbackManager” eingefügt, der die Feedbacks am Ende formatiert und in ein Template einfügt. Das Ergebnis ist eine Liste von Strings, die an den Checker weitergeleitet wird.

Listing 15 Generierung von unformatierten Feedbacks

```
1  private List<Feedback> generateNakedFeedbacks(List<SolutionApproachReportItem> reportItems,
       ↪ QfSemanticSettings checkerSetting) {
2      List<Feedback> result = new ArrayList<>();
3      if (feedbacksStore == null) {
4          feedbacksStore = new FeedbacksStore(
5                  provideStoredFeedbackDirectory(SolutionApproachChecker.class)
6                  , checkerSetting.getLanguage() + FileExtensions.JSON
7          );
8      }
9      for (SolutionApproachReportItem reportItem : reportItems) {
10         if (reportItem.getRelatedSemanticSettingItem().getTaskSpecificFeedbacks() != null) {
11             feedbacksStore.customizeStore(reportItem.getRelatedSemanticSettingItem().
                   ↪ getTaskSpecificFeedbacks());
12         }
13         var defaultFeedback = feedbacksStore.getRelatedFeedbackByTechnicalCause(reportItem.
               ↪ getErrorCode());
14         if (defaultFeedback != null) {
15             var feedbackBuilder = Feedback.builder();
16             feedbackBuilder.type(Type.CORRECTION);
17             feedbackBuilder.checkerName(SolutionApproachChecker.class.getSimpleName());
18             feedbackBuilder.technicalCause(reportItem.getErrorCode());
19             feedbackBuilder.readableCause(defaultFeedback.getReadableCause());
20             feedbackBuilder.hints(defaultFeedback.getHints());
21             feedbackBuilder.relatedLocation(RelatedLocation.builder()
22                     .fileName(
23                             reportItem.getRelatedSemanticSettingItem().getFilePath()
24                     )
25                     .methodName(reportItem.getRelatedSemanticSettingItem().getMethodName())
26                     .build()
27             );
28             feedbackBuilder.reference( feedbacksStore.getConceptReference(reportItem.
                   ↪ getErrorCode()));
29             feedbackBuilder.replaceKeyWords(reportItem);
30             result.add(feedbackBuilder.build());
31         }
32         feedbacksStore.rebuildStore();
```

```
33      }
34      return result;
35  }
```

6.3.3 Generierung von Feedback für SI

Die Methode zur Generierung von Feedbacks für SIs erfolgt durch die Verwendung der Methode “generateFeedbacks” eines Objekts der Klasse “StyleFeedbackGenerator”. StyleFeedbackGenerator ist so ähnlich wie die Klasse SyntaxFeedbackGenerator gebaut. Die Methode “generateFeedbacks” wird von der “Check”-Methode eines Objekts der Klasse “StyleChecker” aufgerufen, wie in Abschnitt 6.1.4 beschrieben. Die Eingabe der Methode “generateFeedbacks” besteht aus den StyleSettings und einem StyleAnalysisReport-Objekt, für das Feedbacks generiert werden sollen. Die Implementierung dieser Methode wird in Listing 16 dargestellt.

Listing 16 Generieren von Feedbacks für SIs.

```
1  public List<String> generateFeedbacks(StyleAnalysisReport report, StyleSettings styleSettings) {
2      List<Feedback> nakedFeedbacks = generateNakedFeedback(report, styleSettings);
3      if (feedbackManager == null) feedbackManager = FeedbackManager.builder().build();
4      feedbackManager.setFeedbacks(nakedFeedbacks);
5      return feedbackManager.buildFeedbackInTemplate(styleSettings.getLanguage());
6  }
```

Zu Beginn wird eine Liste von nicht formatierten Feedbacks generiert, indem die Methode “generateNakedFeedbacks” aus Listing 17 aufgerufen wird. Diese Methode erstellt zunächst eine Instanz der Klasse “FeedbacksStore” und iteriert dann über die “violations”-Liste aus dem “StyleAnalysisReport”. Bei jeder Iteration wird geprüft, ob die Einstellungen (Settings) über TaskSpecificFeedback für das gefundene Violation verfügt. Falls dies der Fall ist, wird der “FeedbacksStore” angepasst, indem Feedback-Inhalte überschrieben werden. Anschließend wird ein Feedback für die SI durch die “FeedbackBuilder”-Klasse generiert. Dabei werden die Komponenten KM und KH durch den “FeedbacksStore” ermittelt. Wenn das FeedbacksStore-Objekt über keine entsprechende KM-Komponente verfügt, wird die Standardbeschreibung aus PMD verwendet. Somit garantieren wir, dass immer ein Feedback für jede gefundene Violation generiert wird. Außerdem wird durch die Einstellung des Checkers entschieden, ob es sich bei dem Feedback um einen Fehler oder eine Verbesserungsmöglichkeit handelt.

Nach Abschluss der Iteration wird eine Liste von Feedbacks zurückgegeben. Diese generierten Feedbacks werden in einen “FeedbackManager” eingefügt, der am Ende das Feedback formatiert und in ein Template einfügt. Das Ergebnis ist eine Liste von Strings, welche dann an den Checker weitergeleitet wird.

Listing 17 Generierung von unformatierten Feedbacks

```
1  public List<Feedback> generateNakedFeedback(StyleAnalysisReport report, StyleSettings settings) {
2      List<Feedback> result = new ArrayList<>();
3      if (feedbacksStore == null) {
4          feedbacksStore = new FeedbacksStore(
5                  provideStoredFeedbackDirectory(StyleChecker.class)
6                  , settings.getLanguage() + FileExtensions.JSON
```

```
7           );
8       }
9       for (Violation violation : report.getViolations()) {
10          var feedbackBuilder = Feedback.builder();
11          feedbackBuilder.type((!settings.getIsCorrection()) ? Type.IMPROVEMENT : Type.CORRECTION);
12          feedbackBuilder.checkerName(StyleChecker.class.getSimpleName());
13          File file = new File(violation.getFileName());
14          feedbackBuilder.relatedLocation(
15                  RelatedLocation.builder()
16                          .fileName(file.getName())
17                          .startLine(violation.getBeginLine())
18                          .endLine(violation.getEndLine())
19                          .build()
20          );
21          var defaultFeedback = feedbacksStore.getRelatedFeedbackByTechnicalCause(violation.
                ↪ getRule());
22          if (defaultFeedback != null) {
23              feedbackBuilder.updateFeedback(defaultFeedback);
24          } else {
25              feedbackBuilder.readableCause( violation.getDescription() );
26              feedbackBuilder.technicalCause(violation.getRule());
27          }
28          result.add(feedbackBuilder.build());
29      }
30      return result;
31  }
```

7 Evaluation

In diesem Abschnitt erfolgt eine Evaluierung des Konzepts und der Implementierung in Hinblick auf die zu Beginn der Arbeit festgelegten Anforderungen und Forschungsfragen. Da der Umfang dieser Arbeit keine empirische Analyse zulässt, beschränkt sich die Analyse auf eine technische Betrachtung. Dabei wird untersucht, inwiefern das System die gestellten Forschungsfragen und Anforderungen erfüllt. Hierzu werden konkrete Aufgaben in Quarterfall definiert und das Verhalten von MASS sowie des Checker- und Feedback-Frameworks näher betrachtet.

Für jede Forschungsfrage wird ein Szenario definiert und am Ende überprüft, ob das System die erwarteten Ergebnisse liefert. Falls nicht, werden die Gründe hierfür begründet.

7.1 Feedback in MASS

Vor der Betrachtung des Systems im Hinblick auf unsere Forschungsfragen, wird das vom System generierte Feedback für Compiler Error (CE), Style Issues (SI) und Solution Approach Error (SE) untersucht. Hierzu wird die folgende Programmieraufgabe verwendet:

- **Implementieren Sie eine Methode, die Gaußsche Summenformel berechnet. Die Methode soll rekursiv sein.**

Wir haben uns entschieden, diese spezielle Programmieraufgabe zu nutzen, um das Verhalten des Systems JACK zu testen. Aus diesem Grund werden wir die Aufgabe in Quarterfall implementieren, um eventuelle Unterschiede zwischen JACK und MASS zu identifizieren. Der Lernende muss seine Lösung als Zip-Datei in Quarterfall hochladen und vom System prüfen lassen. Eine beispielhafte Darstellung der Aufgabe in Quarterfall ist in Abbildung 33 dargestellt.

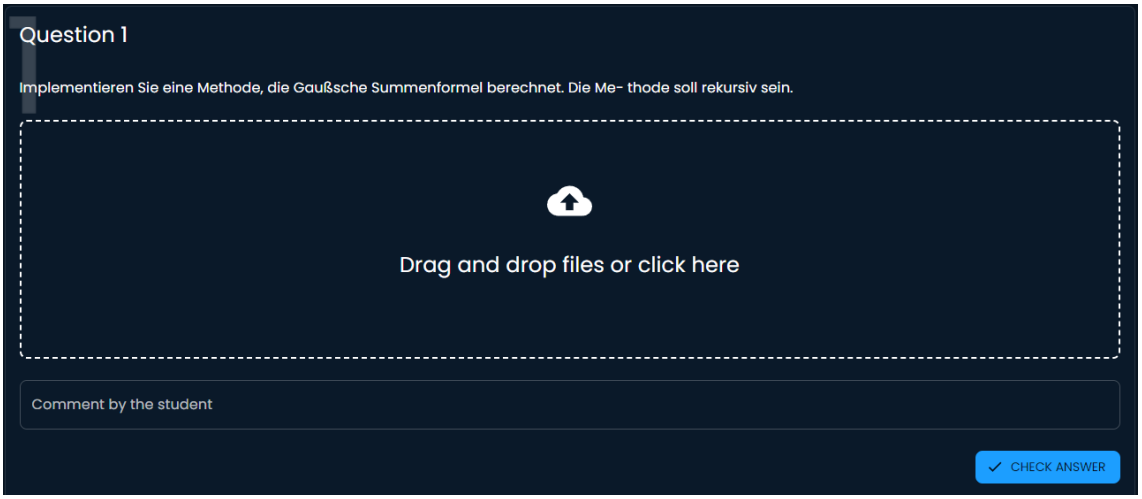


Abbildung 33 Aufgabe in Quarterfall

7.1.1 Compiler Error (CE)

Zunächst werden wir die Lösung aus Listing 18 hochladen. Diese Lösung enthält einen CE im Code, da die "Return"-Anweisung nicht implementiert wurde. Zusätzlich zeigt Listing 19 die Konfiguration des MAS-Systems, wobei wir die Syntax-Level auf "BEGINNER" festgelegt haben. Diese Einstellung ist relevant, da die Aufgabe für Programmieranfänger gedacht ist und das Zeigen der KH-Komponente somit von großer Bedeutung ist.

Listing 18 MASS: Lösung mit Compiler Error.

```
public class Sum {

        public static int BerechneSumme(int n) {
                int result = 0;
                for(int i = 1 ; i < n ; i++) {
                        result += i;
                }
        }
}
```

Listing 19 MASS-Konfiguration für CE

```
qf.mass = {
  "syntax": {
    "level": "BEGINNER"
  }
}
```

Durch die Betrachtung von Abbildung 34 können wir das generierte Feedback anschauen. Dabei wird der Lernende darüber informiert, dass ein Syntax-Fehler in seiner Lösung vorliegt und dieser korrigiert werden muss. Das Feedback enthält auch eine KM-Komponente, die den Ort und die Beschreibung des Fehlers angibt. Zusätzlich wird die KH-Komponente angezeigt, welche ein Codebeispiel zur Behebung des Fehlers bereitstellt. In Vergleich zu JACK wird in Quarterfall die KM-Komponente angezeigt.

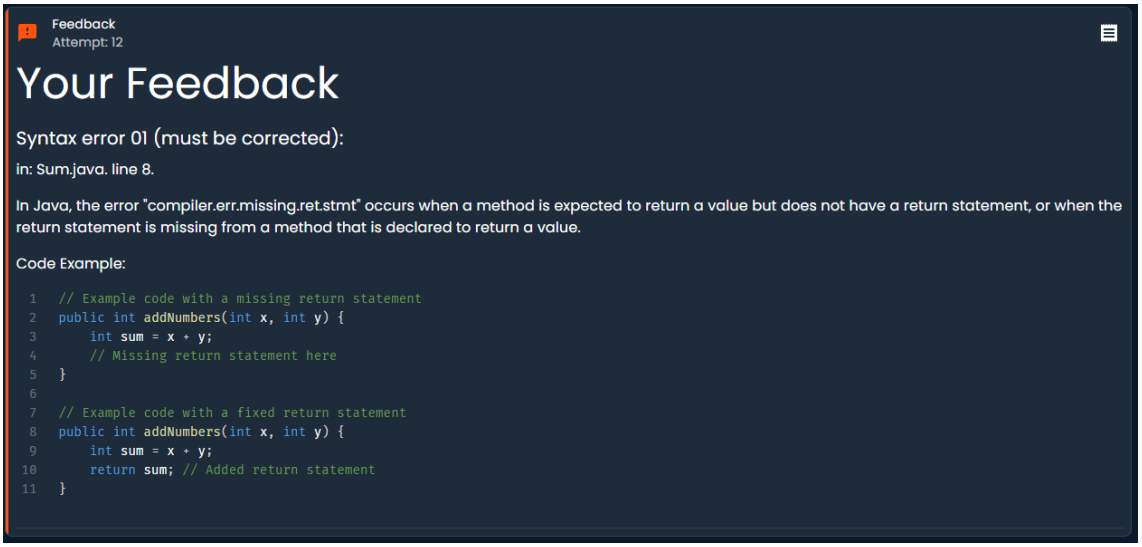


Abbildung 34 Feedbacks von MASS für CE

7.1.2 Solution Approach Error (SE)

Nun laden wir die Lösung aus Listing 20 hoch, welche zwei SEs enthält, da die Aufgabe nicht rekursiv, sondern mit Schleifen gelöst wurde. Da seitens des Lehrers kein spezifisches Feedback zu diesem Fehler definiert wurde, wird das in System gespeicherte Feedback verwendet. Die MASS-Konfiguration in Listing 21 enthält einen Eintrag, der die Methode "BerechneSumme" in der Klasse "Sum" überprüft. Durch diesen Eintrag wird sichergestellt, dass keine Schleifen in der Lösung verwendet werden und die Methode einen rekursiven Aufruf enthält.

Listing 20 MASS: Lösung mit SEs.

```
public class Sum {
        public static int BerechneSumme(int n) {
                int result = 0;
                for(int i = 1 ; i <= n ; i++) {
                        result += i;
                }
                return result;
        }
}
```

Listing 21 MASS-Konfiguration für SE

```
qf.mass = {
  "semanticSelected": true,
  "syntax": {
    "level": "BEGINNER"
  },
  "semantic": {
    "semantics": [
      {
        "filePath": "Sum.java",
        "recursive": true,
        "whileLoop": 0,
        "forLoop": 0,
        "forEachLoop": 0,
        "doWhileLoop": 0,
        "ifElseStmt": -1,
        "methodName": "BerechneSumme",
        "returnType": "int"
      }
    ]
  }
}
```

Durch die Betrachtung von Abbildung 35 können wir das generierte Feedback untersuchen. Wie erwartet, werden dem Lernenden zwei Feedbacks präsentiert. Das erste Feedback informiert den Lernenden darüber, dass in der Lösung eine For-Schleife verwendet wurde, obwohl dies nicht zulässig ist. Das zweite Feedback fordert den Lernenden auf, die Aufgabe durch Rekursion zu lösen. Beide Feedbacks verfügen über eine KM-Komponente, welche den Ort des Fehlers sowie eine detaillierte Beschreibung des Fehlers enthält. JACK und MASS sind in diesem Fall ähnlich, obwohl MASS die Klasse, wo dieser Fehler auftritt, anzeigt und ein Feedback für die Verwendung von Schleifen generiert.

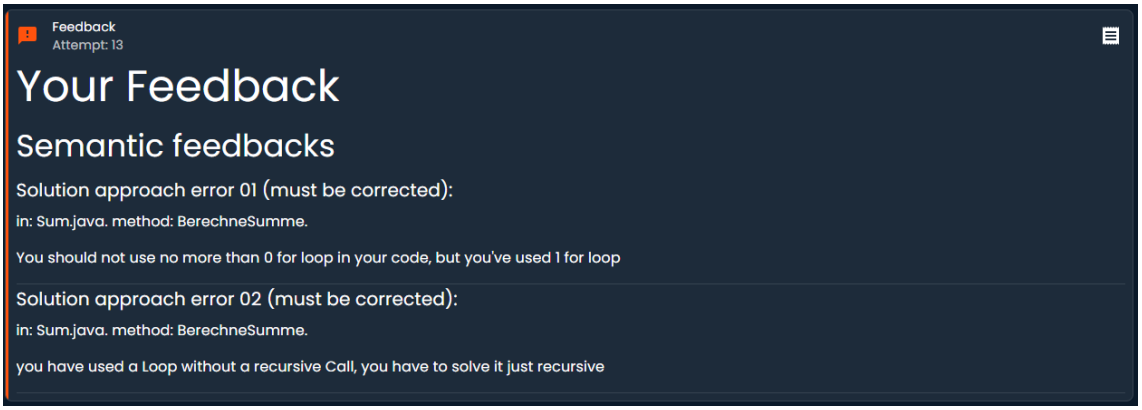


Abbildung 35 Feedbacks von MASS für SE

7.1.3 Style Issues (SI)

Wir werden zunächst die Lösung aus Listing 22 hochladen, die zwei SI enthält, da die Methode nicht kommentiert wurde und die Namenskonventionen von Methoden verletzt werden. Listing 23 zeigt die MASS-Konfiguration, in der wir die Konfiguration für einen Programmieranfänger verwendet haben. SIs werden als Verbesserungsvorschläge betrachtet und nicht als Fehler, die korrigiert werden müssen.

Listing 22 MASS: Lösung mit SEs.

```
public class Sum {
        public static int berechneSumme(int n) {
                int Result = 0;
                for(int i = 1 ; i <= n ; i++) {
                        Result += i;
                }
                return Result;
        }
}
```

Listing 23 MASS-Konfiguration für SI

```
qf.mass = {
  "styleSelected": true,
  "semanticSelected": false,
  "syntax": {
    "level": "BEGINNER"
  },
  "style": {
    "basisLevel": "BEGINNER",
    "complexityLevel": "BEGINNER",
    "namesLevel": "BEGINNER",
    "classLength": -1,
    "methodLength": -1,
    "cyclomaticComplexity": -1,
    "fieldsCount": -1,
    "variableNamePattern": "[a-z][a-zA-Z0-9]*",
    "methodNamePattern": "[a-z][a-zA-Z0-9]*",
    "classNamePattern": "[A-Z][a-zA-Z0-9_]*",
    "isCorrection" : false,
    "checkLevel" : "BEGINNER"
```

```
20   }
21 }
```

Das generierte Feedback aus Abbildung 36 zeigt wie erwartet zwei Feedbacks an. Das erste Feedback informiert den Lernenden darüber, dass die Methode "berechneSumme"nicht kommentiert wurde. Das zweite Feedback wird angezeigt, da der Variablenname gegen die Namenskonventionen verstößt und mit einem großen Buchstaben beginnt. Das erste Feedback besteht nur aus der KM-Komponente, da diese ausreichend ist, um den Ort und die Beschreibung des Fehlers anzuzeigen. Das zweite Feedback enthält sowohl die KM- als auch die KH-Komponente, um dem Lernenden ein Beispiel zu geben, wie der Fehler behoben werden kann. MASS Feedbacks sind besser als JACK Feedbacks in Bezug auf SI, da KH-Komponente generiert wird und Ort des Fehlers angezeigt wird.

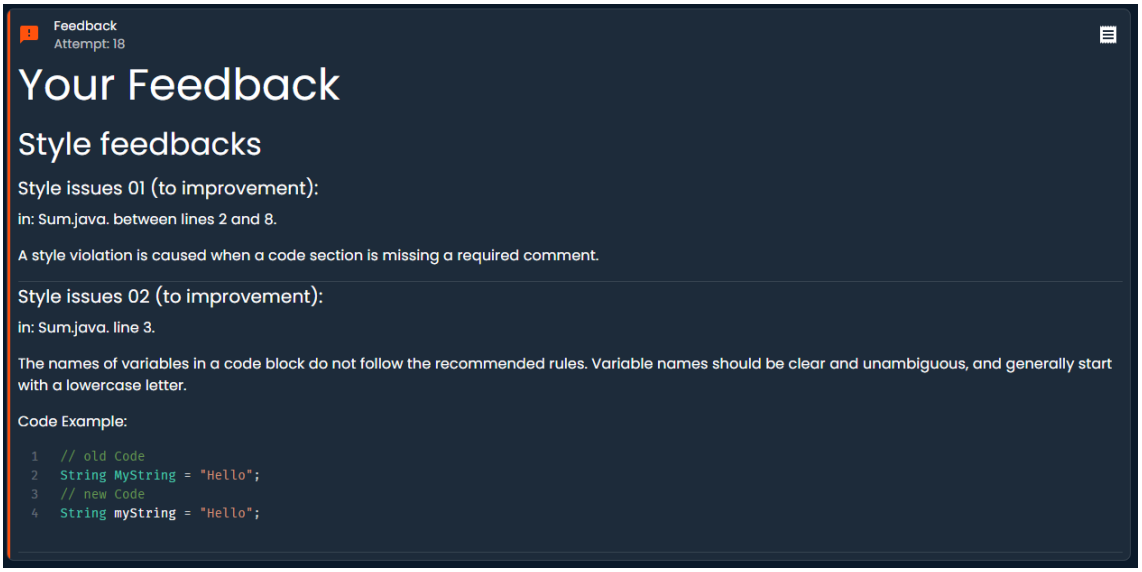


Abbildung 36 Feedbacks von MASS für SI

7.2 Individualisierung von Inhalt eines Feedbacks

Es besteht die Möglichkeit, den Inhalt des Feedbacks in MASS anzupassen. Unter der Annahme, dass die oben vorgestellte Aufgabe von einem erfahrenen Programmierer gelöst wird, ist es unwahrscheinlich, dass die KH-Komponente dem Lernenden zusätzliche Unterstützung bietet. Infolgedessen kann die KH-Komponente aus dem Feedback eliminiert werden und es kann stattdessen ausschließlich die KM-Komponente angezeigt werden.

Die Lösung aus Listing 18 wird nun in Quarterfall hochgeladen, welches eine CE beinhaltet, da die Return-Anweisung noch nicht implementiert wurde. Die Konfiguration von MASS wird in Listing 24 dargestellt, wobei das Check-Level auf ADVANCED eingestellt ist.

Listing 24 MASS-Konfiguration für erfahrenen Programmierer

```
1 qf.mass = {
2   "syntax": {
3     "level": "ADVANCED"
```

```
4     }
5 }
```

Abbildung 37 veranschaulicht das Feedback, welches von MASS generiert wurde. Das Feedback enthält ausschließlich die KM-Komponente. Somit erfüllen wir die Anforderung, dass Inhalt eines Feedbacks individualisiert werden kann.

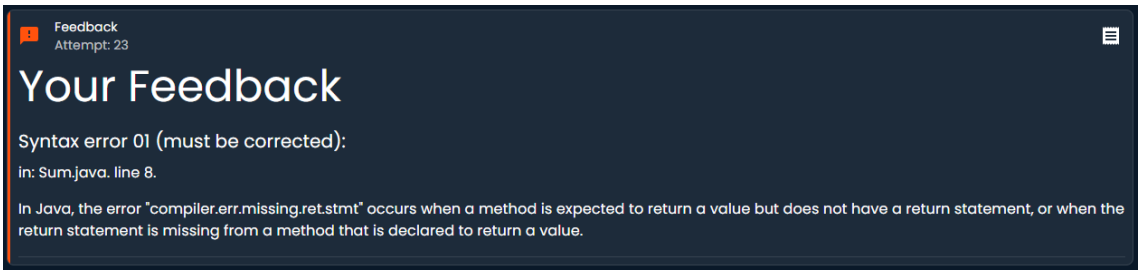


Abbildung 37 Feedbacks von MASS für CE ohne KH-Komponente

Darüber hinaus kann der Inhalt eines Feedbacks in Bezug auf SI durch Anpassung des "checkLevel" in den Style-Checker-Settings auf "ADVANCED" eingesetzt wird. Abbildung 38 zeigt das resultierende Feedback, bei dem die KH-Komponente ebenfalls eliminiert wird.

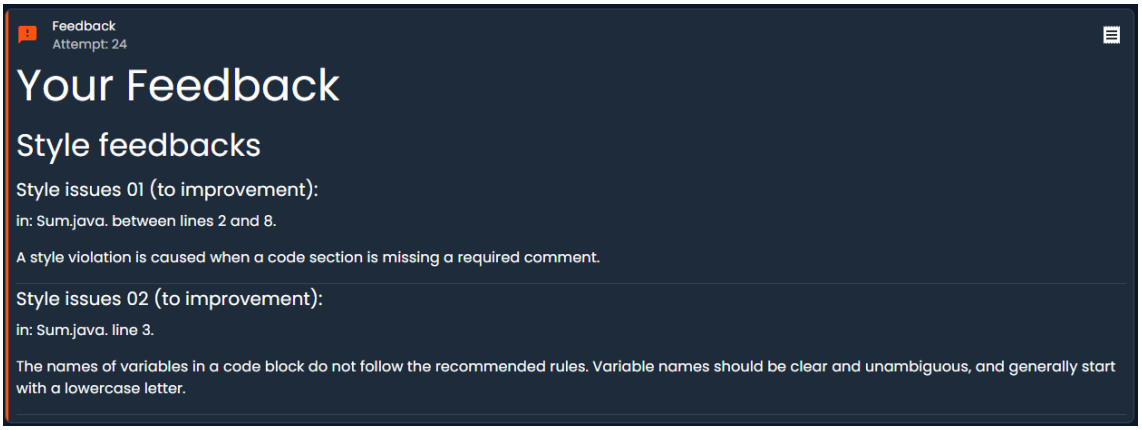


Abbildung 38 Feedbacks von MASS für SI ohne KH-Komponente

7.3 Fehler oder Verbesserungsvorschlag

Manche Programmieraufgaben beinhalten die Überprüfung des Programmierstils, wobei Stil-Verletzungen als Fehler und nicht als Verbesserungsvorschläge betrachtet werden. Das MASS-System kann bei solchen Aufgaben unterstützen, indem der Style-Checker so konfiguriert wird, dass Stil-Verletzungen als Fehler eingestuft werden und von Lernenden korrigiert werden müssen. Um dies anzuwenden, muss das Feld "isCorrection" in der Style-Checker-Konfiguration (siehe Listing 23) auf "True" gesetzt werden.

Ein Beispiel zur Veranschaulichung wäre, dieselbe Programmieraufgabe aus Abschnitt 7.1 zu verwenden und die Lösung aus Listing 22 in Quarterfall hochzuladen. Abbildung 39 zeigt das automatisch generierte Feedback für die eingereichte Lösung. Dabei wird

der Lernende direkt darauf hingewiesen, dass die Lösung korrigiert werden muss. Im Gegensatz zu Abbildung 36, in der das Feedback als Verbesserungsvorschlag formuliert wird, dient es in diesen Feedbacks hier der Identifikation von Fehlern.

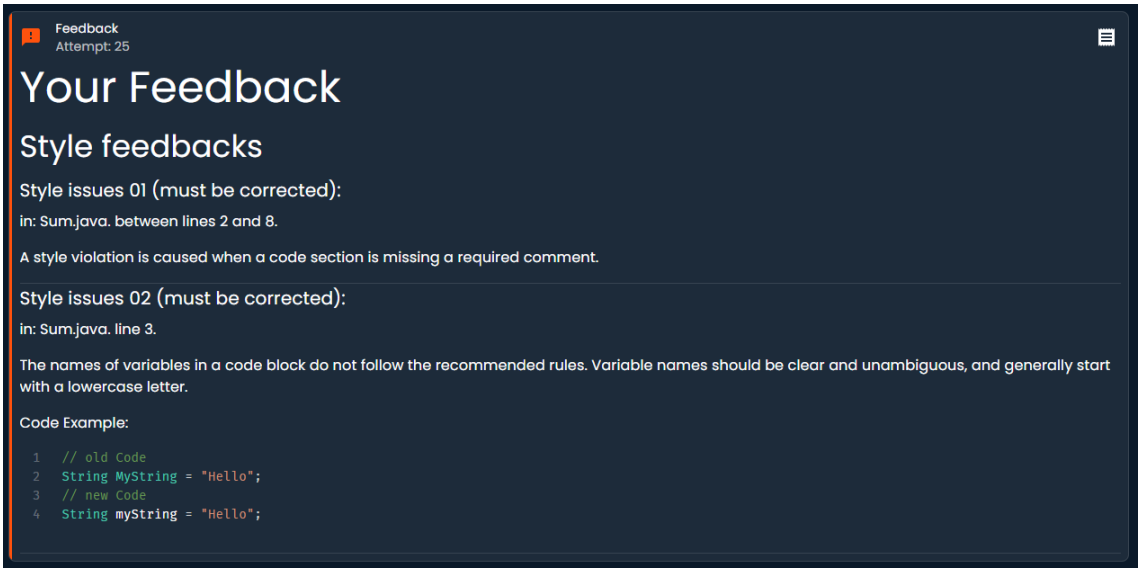


Abbildung 39 Feedbacks von MASS für Style-Fehler

7.4 Anpassbares Feedback

Anstatt die im System gespeicherten Feedbacks zu nutzen, haben Lehrer die Möglichkeit, ihr eigenes Feedback in Bezug auf SI oder SE zu definieren. MASS kann entsprechend konfiguriert werden, sodass das vom Lehrer definierte Feedback dem Lernenden zur Verfügung gestellt wird, falls in der abgegebenen Lösung SI oder SE erkannt werden. Darüber hinaus kann MASS Feedback-Komponenten automatisch kombinieren, wobei Lehrer nur einen Teil des Feedbacks definieren müssen und andere Feedback-Komponenten von MASS automatisch generiert werden.

Um dies zu veranschaulichen, nehmen wir die Aufgabe aus Abschnitt 7.1 und die Lösung aus Abschnitt 35 als Beispiel. Die Lösung enthält zwei SEs, da die Aufgabe nicht rekursiv, sondern mit Schleifen gelöst wurde. Angenommen, der Lehrer möchte dem Lernenden eine Referenz zur Vorlesung über Rekursion geben, falls die Aufgabe nicht rekursiv gelöst wurde. MASS kann in solchen Fällen angepasst werden, indem die MASS-Konfiguration modifiziert wird. Listing 25 zeigt die aktualisierte MASS-Konfiguration, welche eine Überprüfung der Methode “BerechneSumme” in der Klasse “Sum” enthält. Falls keine Rekursion gefunden wird, wird das Feedback, welches im System gespeichert ist, mit der vom Lehrer definierten Referenz kombiniert.

Listing 25 MASS-Konfiguration für SE und selbst definiertes Feedback.

```
qf.mass = {
  "semanticSelected": true,
  "syntax": {
    "level": "BEGINNER"
```

```
  },
  "semantic": {
    "semantics": [
      {
        "filePath": "Sum.java",
        "recursive": true,
        "whileLoop": 0,
        "forLoop": 0,
        "forEachLoop": 0,
        "doWhileLoop": 0,
        "ifElseStmt": -1,
        "methodName": "berechneSumme",
        "returnType": "int",
        "taskSpecificFeedbacks" : [
          {
            "technicalCause" : "solutionMustHaveRecursionInsteadLoop",
            "conceptReference" : {
              "referenceName" : "Lecture for recursion",
              "referenceLink" : "https://www.google.de/",
              "section" : "Recursion example",
              "pageNumbers" : [3,4,5]
            }
          }
        ]
      }
    ]
  }
}
```


Abbildung 40 zeigt das Feedback, das von MASS generiert wurde. Im Vergleich zu Abbildung 35 erhält der Lernende hier zusätzlich eine Referenz zur Vorlesung über Rekursion (KC-Komponente), während das Feedback in Abbildung 35 lediglich die KM-Komponente enthält.

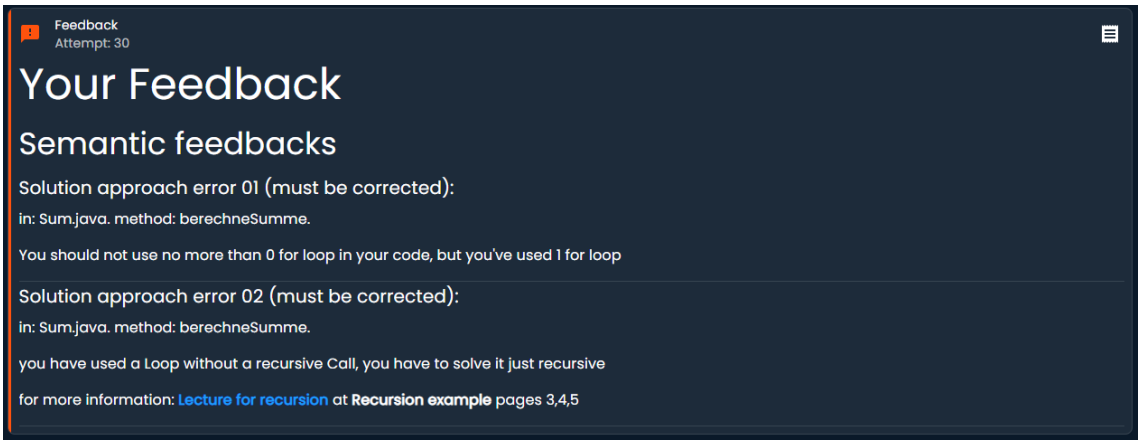


Abbildung 40 Feedbacks von MASS für SE

Es scheint ein Problem zu bestehen, bei dem das vom Lehrer bereitgestellte Feedback in einer bestimmten Sprache verfasst wird. Dadurch wird das Feedback dem Lernenden in dieser Sprache präsentiert, obwohl der Lernende es möglicherweise in einer anderen

Sprache erhalten möchte.

7.5 Feedback in eine gewünschte Sprache

MASS ist in der Lage, das Feedback in die vom Quarterfall-Benutzer gewünschte Sprache zu generieren, indem die ausgewählte Sprache verwendet wird. Derzeit können Benutzer in Quarterfall nur zwischen zwei Sprachen wählen, nämlich Englisch und Niederländisch. Es ist derzeit nicht möglich, Deutsch auszuwählen. MASS kann jedoch derzeit mit Deutsch und Englisch umgehen und kann auf andere Sprachen erweitert werden, indem Feedbacks für diese Sprachen in den JSON-Dateien erstellt werden.

8 Fazit

Im Rahmen dieser Arbeit wurde das System MASS erweitert, welches in den Arbeiten [3, 10] entwickelt wurde. Die Neuerungen haben das System nun für sowohl Programmieranfänger als auch für Studierende in höheren Semestern geeignet gemacht. MASS ist in der Lage, eingereichte JAVA-Lösungen im Quarterfall zu überprüfen und dabei Compilerfehler (CE), Style Issues (SI) und Solution Errors (SE) zu erkennen. Falls solche Fehler in den eingereichten Lösungen gefunden werden, generiert das System informative Feedbacks, um den Lernenden bei der Verbesserung ihrer Fähigkeiten zu unterstützen. Das Hauptziel von MASS ist es, den Lernprozess zu fördern und nicht, eine Bewertung vorzunehmen. Im Rahmen dieser Arbeit wurden die generierten Feedbacks strukturiert und es wurde besonderes Augenmerk auf informatives Feedback gelegt, um den Lerneffekt zu maximieren. Hierbei wurden unterschiedliche Feedback-Komponenten identifiziert, darunter Knowledge About Mistakes (KM), Knowledge About How to Proceed (KH) und Knowledge About Concepts (KC). Zusätzlich verfügt MASS nun über ein zentrales Feedback-Framework, das von allen Checkern verwendet wird. Dieses Framework ist für folgende Aufgaben zuständig:

- Feedbacks werden durch dieses Framework formatiert.
- Feedbacks werden durch dieses Framework in einheitliches Template eingefügt.
- Feedback-Komponenten werden in einer zentralen Stelle im System gespeichert und durch das Framework ermittelt.
- Feedback-Komponenten können von Lehrer überschrieben werden und durch das Framework leicht kombiniert.
- Feedback-Komponenten können auf Deutsch oder Englisch ermittelt werden. Hier kann aber eine neue Sprache leicht eingefügt werden.
- Feedbacks können für bestimmte Zielgruppe von Lehrer angepasst werden, indem manche generierten Feedback-Komponenten angezeigt oder nicht angezeigt werden.

Auf Basis der modellierten Anforderungen wurde das System weiterentwickelt und anschließend einer Evaluation unterzogen. Es konnte festgestellt werden, dass MASS in der Lage ist, das gewünschte Feedback innerhalb eines angemessenen Zeitraums zu generieren und den Anforderungen entspricht. Allerdings besteht weiterhin Verbesserungspotential. Im Folgenden werden Vorschläge zur Erweiterung von MASS vorgestellt:

- Neben den in dieser Arbeit genannten Checkern verfügt MASS auch über weitere Checker, die nun das Feedback-Framework nutzen können.
- Das Framework bietet die Möglichkeit, Feedback zu filtern und zu gruppieren, sodass Lernenden das Feedback gruppiert angezeigt bekommen. Hierfür sind entsprechende gruppierte Feedback-Templates erforderlich.

9 Literatur

[1] Feedback overview in quarterfall. https://help.quarterfall.com/article/57-feedback-overview. Accessed: 2022-08-11.

[2] informatikDeutschland. https://de.statista.com/statistik/daten/studie/732331/umfrage/studierende-im-fach-informatik-in-deutschland-nach-geschlecht/. Accessed: 2022-07-27.

[3] Basel Alaktaa. Automatisierte generierung von feedback für programmieranfänger: Stilverletzungen. Bachelor's thesis, Philipps-Universität Marburg, November 2021.

[4] Jens Bennedsen and Michael E Caspersen. Failure rates in introductory programming. *AcM SIGcSE Bulletin*, 39(2):32–36, 2007.

[5] Michael E Caspersen and Jens Bennedsen. Instructional design of a programming course: a learning theoretic approach. In *Proceedings of the third international workshop on Computing education research*, pages 111–122, 2007.

[6] Justin CW Debuse, Meredith Lawley, and Rania Shibl. Educators' perceptions of automated feedback systems. *Australasian Journal of Educational Technology*, 24(4), 2008.

[7] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3):4–es, 2005.

[8] Michael Goedicke, Michael Striewe, and Moritz Balz. Computer aided assessments and programming exercises with jack. Technical report, ICB-Research Report, 2008.

[9] John Gruber. Markdown: Syntax. *URL http://daringfireball. net/projects/markdown/syntax. Retrieved on June*, 24:640, 2012.

[10] MHD Mayar Hamdash. Automatisierte generierung von feedback für programmieranfänger: Syntax- und semantikfehler. Bachelor's thesis, Philipps-Universität Marburg, November 2021.

[11] JT Hattie Helen, J Hattie, and H Timperley. The power of feedback.[references]. *Rev Educ Res*, 77(1):16–17, 2007.

[12] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153–156, 2003.

[13] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, 19(1):1–43, 2018.

[14] Sujit Kumar Basak, Marguerite Wotto, and Paul Belanger. E-learning, m-learning and d-learning: Conceptual definition and comparative analysis. *E-learning and Digital Media*, 15(4):191–216, 2018.

[15] Angelo Kyrilov and David C Noelle. Binary instant feedback on programming exercises can reduce student engagement and promote cheating. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pages 122–126, 2015.

[16] Norbert Landwehr. Grundlagen zum aufbau einer feedback-kultur. *Konzepte, Verfahren und Instrumente zur Einführung von lernwirksamen Feedbackprozessen*, 2, 2003.

[17] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A

multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 125–180. 2001.

[18] Martin Morgenstern and Birgit Demuth. Continuous publishing of online programming assignments with inloop. In *Software Engineering (Workshops)*, pages 32–33, 2018.

[19] Susanne Narciss. *Informatives tutorielles Feedback: Ableitung und empirische Überprüfung von Entwicklungs-und Evaluationsprinzipien auf der Basis instruktionspsychologischer Erkenntnisse*. Waxmann Verlag, 2005.

[20] David J Nicol and Debra Macfarlane-Dick. Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. *Studies in higher education*, 31(2):199–218, 2006.

[21] Friederike Pfeiffer-Bohnen. Vom lehren zum lernen. In *Vom Lehren zum Lernen*. De Gruyter Oldenbourg, 2017.

[22] D Royce Sadler. Formative assessment and the design of instructional systems. *Instructional science*, 18(2):119–144, 1989.

[23] Edward L Thorndike. The law of effect. *The American journal of psychology*, 39(1/4):212–222, 1927.

[24] John Zukowski. The java compiler api. *Java™ 6 Platform Revealed*, pages 155–169, 2006.