Entwicklung eines IDE-Plugins zur Prüfung von Clean Code-Regeln

Studienarbeit

Studiengang Informatik
OST – Ostschweizer Fachhochschule
Campus Rapperswil-Jona

Herbstsemester 2021

Autoren: Rafael Fuhrer, Pascal Schneider
Betreuer: Prof. Dr.-Ing. Frieder Loch

Zusammenfassung

Clean Code, also sauber geschriebener Quelltext, ist ein elementarer Bestandteil für die Softwarequalität. Unter Clean Code versteht man saubere Struktur, Lesbarkeit und einfach verständlichen Quelltext. Clean Code begünstigt vor allem die Wartbarkeit und Erweiterbarkeit und sollte bei jedem Software Projekt hohe Priorität geniessen.

Regeln, für das Schreiben von gutem Quelltext, sind viele bekannt und wurden auch schon in vielen Büchern niedergeschrieben. Diese Regeln gehen allerdings auch erfahrenen ProgrammiererInnen, beim entwickeln von Software, immer mal wieder vergessen. Für gewisse dieser Regeln gibt es verschiedene Werkzeuge die sicherstellen, dass sie eingehalten werden oder bei Verstössen die EntwicklerIn darauf aufmerksam machen. Für andere fehlt ein solches Werkzeug, insbesondere für die Benennung von Bezeichnern gibt es einige Lücken. Für angehende EntwicklerInnen wäre aber ein solches Werkzeug hilfreich, da sie sich bereits in der Phase des Erlernens der Sprachkonzepte durch sofortige Rückmeldungen an gute Namensgebung gewöhnen würden. Dieses Problem wollten wir im Rahmen dieser Arbeit, mit der Entwicklung einer Erweiterung für die Entwicklungsumgebung Visual Studio Code, adressieren.

Das erstellte Plugin prüft den geschriebenen Quelltext gegen von uns gesammelte und implementierte Regeln. Gefundene Verstösse werden in der Entwicklungsumgebung farbig markiert und die EntwicklerIn erhält Problembeschreibungen welche sie darin unterstützen soll das Problem zu lösen.

Initial haben wir Fachliteratur zum Thema Clean Code konsultiert und systematisch Regeln daraus gesammelt. Diese Regeln haben wir dann genauer analysiert, kategorisiert und auf ihre Umsetzbarkeit geprüft. Danach haben wir einen Prototyp der Erweiterung gebaut, welcher die technologische Umsetzbarkeit unserer angedachten Regelprüfung bestätigte. Basierend auf den daraus gewonnenen Erkenntnissen haben wir eine Architektur abgeleitet und darauf aufbauend die tatsächliche Erweiterung entwickelt.

Mit unserer Arbeit konnten wir aufzeigen, wie eine Erweiterung zur Prüfung von Clean Code Regeln in Visual Studio Code umgesetzt werden kann. Diese haben wir mit einigen Regeln zur Benennung von Bezeichnern und Benachrichtigung der ProgrammiererIn bei Regelverstössen umgesetzt. Dadurch ist die Grundlage gelegt für eine umfänglichere Erweiterung mit mehr Regelprüfungen und Unterstützung für mehr Programmiersprachen.

Management Summary

Kontext und Problem

Clean Code, also sauberer Quelltext ist ein elementarer Bestandteil für die Softwarequalität. Unter Clean Code versteht man sauber strukturierten und einfach verständlichen Quelltext. Clean Code begünstigt dabei vor allem die Wart- und Erweiterbarkeit und sollte bei jedem Software Projekt eine hohe Priorität haben. Die Regeln für das Schreiben von Clean Code sind weitgehend bekannt und wurden auch schon in vielen Büchern niedergeschrieben. Diese Regeln gehen auch erfahrenen Entwicklern beim Entwickeln von Software immer noch häufig vergessen. Bisher fehlt ein Werkzeug, welches Quelltexte bei ihrer Erstellung auf Verletzungen einiger dieser Regeln überprüft und den Programmierenden auf diese aufmerksam macht. Auch für angehende EntwicklerInnen wäre ein solches Werkzeug eine Entlastung, da sie bereits in der Phase des Erlernens der Sprachkonzepte sofortiges Feedback zu ihrem Quelltext bekämen. Damit könnten sie diese Regeln von Beginn an verinnerlichen, anstatt sie erst später zu erlernen und sich schlechte Angewohnheiten abgewöhnen zu müssen.

Methoden und Resultate

Wir haben dieses Problem mit einem Plugin für die Visual Studio Code IDE adressiert. Das erstellte Plugin klinkt sich in den Quelltext ein und validiert diesen gegenüber von uns erstellten Regeln. Werden Verstösse gefunden, werden diese in der IDE farbig markiert und Programmierende erhalten eine Problembeschreibung die helfen soll, das Problem zu beheben.

Zu Beginn haben wir Fachliteratur zum Thema konsultiert und systematisch Regeln daraus abgeleitet. Diese Regeln haben wir dann genauer analysiert, kategorisiert und auf ihre Umsetzbarkeit geprüft. Danach haben wir einen Prototyp einer Extension gebaut, die eine Regel validiert. Basierend auf diesen Erkenntnissen haben wir eine Architektur erstellt und schliesslich darauf aufbauend ein Plugin entwickelt.

Weitere Arbeiten

Mit unserer Arbeit konnten wir aufzeigen, wie eine Erweiterung zur Prüfung von Clean Code Regeln in Visual Studio Code umgesetzt werden kann. Diese haben wir mit einigen Regeln zur Benennung von Bezeichnern und Benachrichtigung der ProgrammiererIn bei Regelverstössen umgesetzt. Die Basis dieser Arbeit lässt mögliche Erweiterungen in folgenden Punkten zu:

- Weitere auch komplexere Regeln könnten implementiert werden
- Weitere Programmiersprachen könnten unterstützt werden
- Weitere Konfigurationsmöglichkeiten für eigene Wörter und Regeln könnten hinzugefügt werden
- Forschung und Erweiterung mit Mustererkennung, die von gutem und schlechten Beispielcode lernt und so Probleme automatisch erkennen kann

Acknowledgements

Als erstes möchten wir uns bei unserem Betreuer, Prof. Dr-Ing. Frieder Loch für seine Hilfe und Inputs zu allen Belangen dieser Arbeit bedanken.

Ausserdem möchten wir uns bei unseren Korrekturlesenden Tuija Krebs, sowie Gabriela Frei und René Schneider für ihr wertvolles Feedback bedanken.

Zuletzt möchten wir auch unserem Kommilitonen und Kollegen Christoph Scheiwiller für sein Feedback beim praktischen Testen unseres Plugins danken.

Inhaltsverzeichnis

Kapitel 1

Einführung in die Thematik

In diesem Kapitel wird eine Einführung und Einordnung in das von uns behandelte Thema gegeben. Besonders im Fokus liegen die Problembeschreibung und die Motivation hinter dem Projekt.

1.1 Softwarequalität

Clean Code ist ein elementarer Bestandteil der Softwarequalität [18].

Der Begriff Qualität kommt vom lateinischen *qualitas* und bedeutet übersetzt soviel wie Beschaffenheit, Merkmal, Eigenschaft oder Zustand und hat mehrere Bedeutungen [26]. Unter Softwarequalität versteht man die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung bezieht, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen. [1], S. 257.

Hinter dem Begriff Softwarequalität stecken viele Themen, die eigene, sich teilweise überschneidende, Informatikthemen darstellen. Diese sind gemäss ISO/IEC 25010 [10] in der Tabelle 1.1 definiert:

Clean Code begünstigt aber vor allem den Punkt Wartbarkeit und hier insbesondere die Unterpunkte Modularität, Modifizierbarkeit und Testbarkeit. Somit ist Clean Code ein elementarer Bestandteil der Softwarequalität. Leider gehören diese Prinzipien zu den am wenigsten beachteten und es existieren im Vergleich zu den anderen Punkten in der Liste noch wenige Tools, die den Programmierenden bei der Anwendung der Best Practices unterstützen.

Tabelle 1.1: Themen der Softwarequalität

Hauptthema	**Unterpunkte**	**Methoden**
Funktionale Tauglichkeit	Funktionale Vollständigkeit, Korrektheit und Angebrachtheit	User Acceptance Testing
Performance	Antwortzeit, Ressourcenverbrauch und Kapazität	Performance Testing
Kompatibilität	Interoperabilität und Ko-Existenz	Integration Testing
Benutzerfreundlichkeit	Angemessenheit, Erkennbarkeit, Lernbarkeit, Ausführbarkeit, User Error Schutz, UI Ästhetik und Zugänglichkeit	Usability & User Acceptance Testing
Verlässlichkeit	Reifegrad, Verfügbarkeit, Fehlertoleranz und Wiederherstellbarkeit	Disaster Recovery Testing
Security	Vertraulichkeit, Nichtabstreitbarkeit, Verantwortlichkeit und Authentizität	Static Code Analysis
Wartbarkeit	Modularität, Wiederverwendbarkeit, Modifizierbarkeit und Testbarkeit	Unit Testing
Portabilität	Adaptierbarkeit, Installierbarkeit und Austauschbarkeit	Integration Testing

1.2 Werkzeuge der Softwareentwicklung

Einer EntwicklerIn stehen heutzutage Werkzeuge zur Verfügung, die sie bei der Softwareentwicklung unterstützen. Es existiert eine grosse Menge dieser Tools. Einige davon, die im Zusammenhang mit Clean Code stehen, haben wir bereits im Abschnitt Softwarequalität genauer beleuchtet. Um unsere Arbeit im Kontext der Softwareentwicklung als Gesamtes einordnen zu können, gehen wir hier noch grundlegend auf die weiteren Tools der Softwareentwicklung ein. Diese lassen sich in die folgenden Kategorien, wie in Tabelle 1.2 dargestellt, gruppieren [13]:

Für dieses Projekt sind vor allem die IDEs interessant, da es am effizientesten ist, einen Fehler direkt bei seinem Ursprung zu beheben. Da die ProgrammiererIn in der IDE den Quelltext schreibt, soll bereits hier geprüft werden, dass die Clean Code Prinzipien eingehalten werden.

Tabelle 1.2: Werkzeuge der Softwareentwicklung

Werkzeugkategorie	**Zweck**	**Beispiele**
IDE	Entwicklungsumgebung in der Quelltext geschrieben wird	Visual Studio Code, IntelliJ IDEA, Atom, Visual Studio
Unit Tests	Automatische, unabhängige Testung einzelner Codebestandteile auf ihre korrekte Funktionalität	Mocha, JUnit, NUnit
Issue Tracker	Erfassen von einzelnen Teilaufgaben zur Erweiterung bzw. Fehlerbehebung des Codes	Atlassian Jira, Bugzilla, Apaches Bloodhound
Versionsverwaltung	Versionierung und Archivierung der Daten (Code und andere Projektbestandteile)	Git, svn, Mercurial
Build Tool	Erleichtert (bzw. automatisiert) die Transformation von Quelltext in ein ausführbares Programm	npm, Graddle, Ant, Maven
Continous Integration (CI)	Automatische Integration von Codeänderungen in die bestehende Codebasis	Jenkins, Atlassian Bamboo, GitLab CI/CD
Continuous Deployment (CD)	Automatisches Deployment von Codeänderungen in die Systemumgebungen	Jenkins, Atlassian Bamboo, GitLab CI/CD
Statische Code Analyse	Prüfung und Sicherstellung von Code Qualität, Sicherheit und Compliance	SonarQube, SonarLint, ESLint
Repository für Artefakte	Speicherort für Artefakte wie z.B. Cloud Credentials für das Deployment	JFrog Artifactory, Sonatype Nexus, GitLab Variables
Wiki	Erfassung von Projektinformationen und Code- bzw. Architekturinformationen	Confluence, Mediawiki, Dokuwiki, GitLabwiki

1.3 Motivation

Da es bisher noch wenige Tools gibt, die eine EntwicklerIn während dem programmieren, bei der Clean Code Thematik unterstützen, wollen wir im Rahmen dieser Arbeit ein Tool entwickeln, welches einen Teil dieser Lücke füllt. Denn auch wenn die Clean Code Regeln bestens bekannt sind, so gehen sie den meisten Programmierenden in der Anwendung häufig vergessen. Ein Tool, welches sich in die IDE einklinkt und Verletzungen dieser Clean Code Regeln direkt anzeigt, soll diesem Problem entgegenwirken. Gemäss dem Prinzip, dass es immer mehr kostet, einen Fehler bzw. ein Problem zu beheben, je später im Entwicklungsprozess dieser bzw. dieses behoben wird, macht der Einsatzort in der IDE am meisten Sinn. So kann das Problem direkt bei seiner Entstehung angegangen werden.

Kapitel 2

Identifikation relevanter Clean Code Regeln

2.1 Clean Code Regeln

Zur Identifikation von möglichen Clean Code Regeln haben wir uns auf das Buch "Clean Code" [12] von Robert C. Martin fokussiert. Es handelt sich bei diesem Buch um ein in der Branche weit bekanntes Standardwerk. Weitere nennenswerte Werke in diesem Themenbereich wären "The Pragmatic Programmer" [21], "Code Complete 2" [14] und "The Art of Readable Code" [4]. Ein grosser Vorteil dieser Vorgehensweise ist, dass wir über alle Teilbereiche hinweg eine konstante Gliederung und klare Trennung von anderen Bereichen erreichen. Denn beim Mischen von mehreren Quellen würde es Überschneidungen der Regeln geben.

Nachfolgend werden alle Clean Code Regeln, welche wir während der Literaturrecherche identifiziert haben, tabellarisch aufgeführt. Wir haben für die Regeln, basierend auf Gemeinsamkeiten, übergeordnete Bezeichnungen definiert und die Regeln entsprechend gruppiert.

2.1.1 Benennung von Variablen und Funktionen

In diesem Block geht es um Benennungsregeln von Variablen und Funktionen. Ziel dieser Regeln ist es, schematisch festzulegen, wie die Namen von Variablen und Funktionen aufgebaut sein sollten und was in den Namen vermieden werden soll.

Ein Beispiel für eine solche Regel wäre: "zweckbeschreibende Namen wählen". Nach dieser Regel sollten Variablennamen wie:

```
int d = 10   //maximale Login Versuche
```

vermieden werden. Besser wäre ein zweckbeschreibender Name wie z.B.:

```
int maximumLoginAttempts = 10
```

Wir haben alle Regeln, die diesem Block zugeordnet werden können, in der Tabelle 2.1 zusammengefasst.

Tabelle 2.1: Benennungsregeln von Variablen und Funktionen

Regel	**Bezeichnung**	**Quelle [12] (Kapitel)**
BR01	Zweckbeschreibende Namen wählen	2.2
BR02	Fehlinformationen im Namen vermeiden	2.3
BR03	Unterschiede deutlich machen	2.4
BR04	Aussprechbare Namen verwenden	2.5
BR05	Suchbare Namen verwenden	2.6
BR06	Codierungen vermeiden	2.7
BR07	Mentale Mappings vermeiden	2.8
BR08	Klassen- und Objektnamen	2.9
BR09	Methodennamen	2.10
BR10	Vermeiden von humorigen Namen	2.11
BR11	Ein Wort pro Konzept	2.12
BR12	Keine Wortspiele	2.13
BR13	Namen der Lösungsdomäne verwenden	2.14
BR14	Namen der Problemdomäne verwenden	2.15
BR15	Bedeutungsvollen Kontext hinzufügen	2.16
BR16	Keinen überflüssigen Kontext hinzufügen	2.17
BR17	Beschreibende Funktionsnamen verwenden	3.5

2.1.2 Funktionen

In diesem Block geht es um alle Regeln für Funktionen, ausgenommen der Benennungsregeln. Ein Beispiel für eine solche Regel wäre “Anweisung und Abfrage trennen”. Nach dieser Regel soll eine Funktion entweder etwas tun oder etwas antworten, aber niemals beides zusammen. Im folgenden Beispiel:

```
public bool doTransaction (int amount, Account senderAccount, Account receiverAccount){
    ... // do the transaction
    return (receiverAccount.getBalance() == receiverAccount.getLastBalance() + amount)
}
```

führt die Funktion eine Transaktion aus, prüft gleichzeitig ob diese auch korrekt erfolgt ist und gibt das Resultat

direkt als Boolean zurück. Eine typische Verletzung dieser Regel. Besser wäre es, zwei Funktionen zu implementieren. Eine Funktion welche die Transaktion durchführt und eine Funktion, welche überprüft ob die Transaktion korrekt durchgeführt wurde.

Wir haben alle Regeln, die diesem Block zugeordnet werden können, in der Tabelle 2.2 zusammengefasst.

Tabelle 2.2: Regeln für Funktionen

Regel	**Bezeichnung**	**Quelle [12] (Kapitel)**
FU01	Funktionen sollen kurz sein	3.1
FU02	Eine Aufgabe (pro Funktion) erfüllen	3.2
FU03	Eine Abstraktionsebene pro Funktion	3.3
FU04	Switch Anweisungen (wenn möglich) vermeiden	3.4
FU05	Funktionsargumente (max. 3)	3.6
FU06	Nebeneffekte vermeiden	3.7
FU07	Anweisung und Abfrage trennen	3.8
FU08	Ausnahmen sind besser als Fehlercodes	3.9
FU09	Don't Repeat Yourself (DRY)	3.10
FU10	Strukturierte Programmierung	3.11

2.1.3 Kommentare

In diesem Block geht es um Regeln, die Kommentare betreffen. Der Fokus liegt hier darauf zu spezifizieren, wann Kommentare eingesetzt werden sollen und wie ein guter Kommentar aussehen soll. Ein Beispiel wäre die Regel "Kommentare wo möglich vermeiden". Diese Regel besagt, dass Quelltexte mit sinnvollen Benennungen von Variablen und Funktionen nur selten Kommentare brauchen. Ein Beispiel für einen schlechten und unnötigen Kommentar wäre:

```
//This function tests if a specified Account is active or not
bool isActiveAccount(Account account){
    return account.isActive()
}
```

In diesem Beispiel gibt der Kommentar nicht mehr Preis als man bereits aus dem Namen der Funktion ableiten kann. Der Kommentar ist also überflüssig.

Beispiel für sinnvolle Kommentare können rechtliche Hinweise, Angabe zum Autor des Quelltextes oder eine Dokumentation einer unerwarteten / kontraintuitiven Funktionalität sein.

Wir haben alle Regeln, die diesem Block zugeordnet werden können, in der Tabelle 2.3 zusammengefasst.

Tabelle 2.3: Regeln für Kommentare

Regel	**Bezeichnung**	**Quelle [12] (Kapitel)**
RK01	Kommentare wo möglich vermeiden	4.1 & 4.2
RK02	Gute Kommentare schreiben	4.3

2.1.4 Formatierung

Dieses Kapitel beschreibt was für Formatierungsregeln verwendet werden sollten und wie man diese Formatierungsregeln richtig anwendet. Das Ziel ist, dass sich eine Quelldatei mit möglichst geringem Aufwand lesen lässt. Dies geschieht mit dem Hintergrund, dass Quelltext häufiger gelesen als geschrieben wird.

Ein gutes Beispiel zur Verdeutlichung ist, dass man Formatierungsregeln konsistent anwenden soll:

```
if (amount1 == 0)
{
    amount1 = 100;
}
if (amount2 == 0)
    amount2 = 150;
```

Wenn Quelltext wie im oberen Beispiel nicht einheitlich formatiert ist, ist es für den Lesenden schwieriger schnell zu erfassen, was der Code macht bzw. wie die einzelnen Elemente zueinander gehören.

Wir haben alle Regeln, die diesem Block zugeordnet werden können, in der Tabelle 2.4 zusammengefasst.

Tabelle 2.4: Regeln für Formatierung

Regel	Bezeichnungen	Quelle [12] (Kapitel)
FO01	Formatierungsregeln konsistent anwenden	5.1
FO02	Dokumente maximal ein paar hundert Zeilen	5.2
FO03	Zusammengehöriger Code vertikal beieinander	5.2
FO04	Variablendeklaration wo sie gebraucht wird	5.2
FO05	Instanzvariablen zuoberst in Klassen	5.2
FO06	Aufrufer oberhalb von Aufgerufenen	5.2
FO07	Zeilen nicht länger als 80–120 Zeichen	5.3
FO08	Hierarchien durch Zeileneinzug trennen	5.3
FO09	Alle Teammitglieder selbe Formatierungsregeln	5.4

2.1.5 Objekte und Datenstrukturen

In diesem Kapitel geht es um die Unterschiede zwischen Objekten und Datenstrukturen. Insbesondere wird darauf eingegangen, wie sich die Regeln für die Beiden unterscheiden und welche Regeln wo gelten.

Als Beispiel wäre das "Law of Demeter", welches besagt, dass ein Modul nichts über die Internas der Objekte kennen sollte die es manipuliert.

```
var address = account.getAccountHolder().getAddress();
```

Dieser Quelltext widerspricht dieser Regel. Allerdings nur, wenn es sich bei Account und AccountHolder um Objekte handelt. Sollten diese Klassen Datenstrukturen sein, wäre die Regel nicht verletzt.

Wir haben alle Regeln, die diesem Block zugeordnet werden können, in der Tabelle 2.5 zusammengefasst.

Tabelle 2.5: Regeln für Objekte und Datenstrukturen

Regel	Bezeichnung	Quelle [12] (Kapitel)
OD01	Law of Demeter	6.3
OD02	Objekt / Datenstruktur Hybriden vermeiden	6.3

2.1.6 Fehlerbehandlung

Bei den Regeln zur Fehlerbehandlung geht es darum, wie sich eine Anwendung im Fehlerfall verhalten soll. Hier geht es vor allem darum, wie Informationen zu Fehlverhalten sowohl innerhalb vom Quelltext weitergereicht und weiterverarbeitet werden sollen, als auch in welcher Form diese Informationen an die Nutzenden des Programms gelangen sollen. Weiter gibt es einige Regeln dazu wie der Quelltext geschrieben werden soll um robuster gegen Fehler zu sein.

Ein Beispiel wäre die Regel “Nicht NULL aus Methoden zurückgeben”:

```
public void addToAccount(int accountId, int amount) {$
    Account account = accountRepository.getAccount(accountId);

    if(account != null) {
        account.add(amount);
    }
}
```

In diesem Fall verletzt die Methode getAccount der Klasse AccountRepository diese Regel. Weiss der Programmierende nicht, dass aus dieser Methode NULL zurück kommen kann, wird der darunter folgende NULL-Check vermutlich nicht gemacht und wir riskieren eine NullPointerException zur Laufzeit.

Wir haben alle Regeln, die diesem Block zugeordnet werden können, in der Tabelle 2.6 zusammengefasst.

Tabelle 2.6: Regeln für die Fehlerbehandlung

Regel	**Bezeichnung**	**Quelle [12] (Kapitel)**
FE01	Exception statt Error Codes	7.1
FE02	Keine checked Exceptions	7.3
FE03	Fehlerbeschreibungen hoher Informationsgehalt	7.4
FE04	Eine Exception-Klasse für zusammengehörige Fehlertypen	7.5
FE05	Nicht NULL aus Methoden zurückgeben	7.7
FE06	Nicht NULL in Methoden hineingeben	7.8

2.2 Vergleich mit existierenden Tools

In diesem Abschnitt geht es darum zu erfassen, welche Regeln bereits durch existierende Tools abgedeckt werden. Dafür beschreiben wir die verwendete Methodik zum Überprüfen der vorhandenen Tools und prüfen für jede im vorherigen Kapitel identifizierte Regel, ob sie bereits abgedeckt wird.

2.2.1 Methodik

Um zu testen ob eine Regel bereits durch existierende Tools abgedeckt ist, sind wir nach der folgenden Methode vorgegangen. Wir haben die in den Kapiteln vorhandenen Negativbeispiele genommen und in den Editor Visual Studio Code [16] (Version 1.61.0) eingefügt.

Ausserdem haben wir im Editor die folgenden Plugins, wie in Tabelle 2.7 dargestellt, installiert:

Tabelle 2.7: Installierte Visual Studio Code Plugins

Plugin Name	**Version**	**Funktion**
SonarLint	v3.0.0	Linter
Prettier	v9.0.0	Formater

Beim Einfügen des problematischen Quelltextes, wurde dann untersucht, ob die problematischen Stellen grafisch in der IDE markiert oder im Log des Tools eine entsprechende Warnung ausgegeben wurde. War dies der Fall, wurde das Ergebnis entsprechend erfasst.

Konnte kein Indikator des Problems ausgemacht werden, wurde der problematische Quelltext zusätzlich einem SonarQube (Version 9.1.0) Scan unterzogen, um festzustellen ob das Problem hier erkannt wird.

Als Programmiersprache wird Java (OpenJDK 17) verwendet, da die Beispiele im Clean Code Buch ebenfalls in Java sind.

Da es sein kann, dass eine Regel nur teilweise von einem bestehenden Tool abgedeckt wird, definieren wir dafür, wie in der Tabell 2.8 dargestellt, neben dem Wert Ja und Nein noch den Wert Tlw. (Teilweise). Ein Beispiel für eine teilweise abgedeckte Regel wäre die Regel “Kommentare wo möglich vermeiden”. Ein Tool wie SonarLint kann zwar anzeigen, dass ein Kommentar wo möglich vermieden werden soll, da es aber den Kontext nicht versteht, zeigt es diese Meldung auch bei sinnvollen Kommentaren an. Diese Regel wird von SonarLint also nur teilweise abgedeckt.

Tabelle 2.8: Abdeckungswerte

Wert	**Bedeutung**
Ja (Tool X)	Die Regel wird vollständig abgedeckt
Nein	Die Regel wird durch keines der untersuchten Tools abgedeckt
Tlw. (Tool X)	Die Regel wird zum Teil abgedeckt

2.2.2 Abdeckung durch bestehende Tools

In diesem Abschnitt fassen wir die Ergebnisse der Untersuchung tabellarisch zusammen. Jede Regel wurde nach dem im Abschnitt Methodik beschriebenen Vorgehen getestet.

Benennung von Variablen und Funktionen

Tabelle 2.9: Bestehende Abdeckung Benennungsregeln

Regel	**Bezeichnung**	**Abgedeckt?**
BR01	Zweckbeschreibende Namen wählen	Nein
BR02	Fehlinformationen im Namen vermeiden	Nein
BR03	Unterschiede deutlich machen	Nein
BR04	Aussprechbare Namen verwenden	Nein
BR05	Suchbare Namen verwenden	Nein
BR06	Codierungen vermeiden	Nein
BR07	Mentale Mappings vermeiden	Nein
BR08	Klassen- und Objektnamen	Nein
BR09	Methodennamen	Nein
BR10	Vermeiden von humorigen Namen	Nein
BR11	Ein Wort pro Konzept	Nein
BR12	Keine Wortspiele	Nein
BR13	Namen der Lösungsdomäne verwenden	Nein
BR14	Namen der Problemdomäne verwenden	Nein
BR15	Bedeutungsvollen Kontext hinzufügen	Nein
BR16	Keinen überflüssigen Kontext hinzufügen	Nein
BR17	Beschreibende Funktionsnamen verwenden	Nein

Funktionen

Tabelle 2.10: Bestehende Abdeckung von Funktionsregeln

Regel	**Bezeichnung**	**Abgedeckt?**
FU01	Funktionen sollen kurz sein	Ja (SonarQube)
FU02	Eine Aufgabe (pro Funktion) erfüllen	Nein
FU03	Eine Abstraktionsebene pro Funktion	Nein
FU04	Switch Anweisungen (wenn möglich) vermeiden	Ja (Sonarlint)
FU05	Funktionsargumente (max. 3)	Nein
FU06	Nebeneffekte vermeiden	Nein
FU07	Anweisung und Abfrage trennen	Nein
FU08	Ausnahmen sind besser als Fehlercodes	Nein
FU09	Don't Repeat Yourself (DRY)	Ja (Sonarlint)
FU10	Strukturierte Programmierung	Ja (Sonarlint)

Kommentare

Tabelle 2.11: Bestehende Abdeckung von Kommentarregeln

Regel	**Bezeichnung**	**Abgedeckt?**
RK01	Kommentare wo möglich vermeiden	Tlw. (SonarLint)
RK02	Gute Kommentare schreiben	Nein

Formatierung

Objekte und Datenstrukturen

Fehlerbehandlung

Tabelle 2.12: Bestehende Abdeckung von Formatierungsregeln

Nr.	**Bezeichnungen**	**Abgedeckt?**
FO01	Formatierungsregeln konsistent anwenden	Ja (Linter)
FO02	Dokumente maximal ein paar hundert Zeilen	Ja (Sonarlint)
FO03	Zusammengehöriger Code vertikal beieinander	Nein
FO04	Variablendeklaration wo sie gebraucht werden	Ja (Visual Studio Refactoring)
FO05	Instanzvariablen zuoberst in Klassen	Ja (ESLint)
FO06	Aufrufer oberhalb von Aufgerufenen	Nein
FO07	Zeilen nicht länger als 80–120 Zeichen	Ja (Sonarlint)
FO08	Hierarchien durch Zeileneinzug trennen	Ja (Sonarlint)
FO09	Alle Teammitglieder selbe Formatierungsregeln	Ja (SonarQube)

Tabelle 2.13: Bestehende Abdeckung von Objekt- und Datenstrukturregeln

Regel	**Bezeichnungen**	**Abgedeckt?**
OD01	Law of Demeter	Nein
OD02	Objekt / Datenstruktur Hybriden vermeiden	Nein

Tabelle 2.14: Bestehende Abdeckung von Fehlerbehandlungsregeln

Regel	**Bezeichnung**	**Abgedeckt?**
FE01	Exception statt Error Codes	Nein
FE02	Keine checked Exceptions	Ja (SonarQube)
FE03	Fehlerbeschreibungen hoher Informationsgehalt	Nein
FE04	Eine Exception-Klasse für zusammengehörige Fehlertypen	Nein
FE05	Nicht NULL aus Methoden zurückgeben	Nein
FE06	Nicht NULL in Methoden hineingeben	Nein

2.3 Diskussion

Die Prüfung ergab, dass es für viele Clean Code Regeln noch keine Hilfsmittel gibt. Vor allem im Bereich der Benennung von Variablen und Funktionen, welcher für verständlichen und einfach zu lesenden Quelltext so wichtig ist, gibt es noch gar keine Hilfsmittel. Aber auch in den anderen Themenfeldern gibt es noch Lücken bei einzelnen Regeln. Diese sind je nach Themenfeld unterschiedlich stark ausgeprägt. So sind zum Beispiel beim Thema Formatierung schon fast alle Regeln abgedeckt. Dies dürfte am breiten Einsatz von verschiedenen Lintern liegen.Bei den Regeln in den Kategorien Kommentare, Fehlerbehandlung und Funktionen ist eine teilweise Abdeckung zu beobachten. Hier sind vorwiegend die simpleren Regeln durch Linter und Static Code Analysis Werkzeuge abgedeckt. Dies könnte daran liegen, dass es sehr aufwändig ist, diese zu prüfen und viele dieser Regeln vom Kontext des Einsatzes abhängen.

Die in diesem Kapitel gewonnenen Informationen haben wir genutzt um die noch nicht abgedeckten Regeln zu analysieren und auf dieser Basis zu entscheiden, welche Regeln wir für unsere Arbeit umsetzen möchten. Da in den vier Themenfeldern “Benennung”, “Funktionen”, “Objekte und Datenstrukturen” und “Fehlerbehandlung” die grössten Lücken in Bezug auf die Abdeckung zu finden waren, haben wir in den folgenden Kapiteln einen besonderen Fokus auf diese Bereiche gelegt.

Kapitel 3

Detailanalyse der Regeln

In diesem Kapitel wollen wir untersuchen, welche der Regeln aus dem vorherigen Kapitel, die nicht bereits von einem anderen Tool oder Ansatz abgedeckt werden, für diese Arbeit konkret in Frage kommen.

3.1 Kontextabhängige Regeln

Viele der identifizierten Regeln erfordern ein Verständnis der Problemdomäne bzw. können nur unter Berücksichtigung des Kontexts angewendet werden. Ein Beispiel für eine solche Regel wäre "Vermeiden von humorigen Name". Eine Verletzung dieser Regel würde folgendermassen aussehen:

```
// check if element is flagged for ''destruction''
element.isBomb()
```

Dieses Beispiel zeigt, dass man diese Regelverletzung nicht mit einem Wörterbuchabgleich oder ähnlichem detektieren kann. Um hier feststellen zu können, dass es sich um einen humorigen Namen handelt, ist sowohl ein Verständnis des Gesamtkontextes, wie auch des Konzepts von Humor nötig.

IT-Systemen fehlt aber dieses Verständnis des Kontext oder Humorkonzepts. Es gab zwar in der jüngerer Zeit einige Fortschritte in diesem Bereich. So kann das GPT-3 Modell von Openai beispielsweise Quelltext aufgrund der wörtlichen Beschreibung generieren [19]. Wir haben aber keinen Zugriff auf diese Technologien, da solche Modelle nicht öffentlich zugänglich sind. Ausserdem fehlen uns die benötigte Masse an Daten und das Know-How um ein eigenes Modell zu entwickeln. Ausserdem hätte ein solches Vorgehen den Rahmen dieses Projekts auch deutlich gesprengt. Deshalb haben wir uns einige Regeln herausgesucht, welche sich am ehesten mit unseren Mitteln umsetzen liessen und haben diese als KANN Ziele für das Projekt festlegt. Die anderen Regeln schliessen wir aufgrund der zu hohen Komplexität aus.

Die folgenden Regeln, wie in Tabelle 3.1 dargestellt, haben wir als Kontextabhängig identifiziert:

Tabelle 3.1: Kontextabhängige Regeln

Kategorie	**Regel Nr.**
Benennungsregeln	BR01, BR02, BR03, BR07, BR10, BR11, BR12, BR13, BR14, BR15, BR16, BR17
Funktionen	FU02, FU03, FU06, FU07, FU08
Kommentare	RK02
Formatierung	FO03, FO04, FO06, FO08, FO09
Objekte und Datenstrukturen	OD01
Fehlerbehandlung	FE01, FE03, FE04, FE05, FE06

Wir haben aber beschlossen, dass wir einige einfache kontextabhängigen Regeln trotzdem als KANN Ziele (nicht obligatorisch zu erreichen) in den Projektumfang aufnehmen. Die Vorarbeiten in diesem Bereich könnten als Grundlage genutzt werden, um diese Regeln im Rahmen von weiterführenden Arbeiten umzusetzen.

Die folgenden Regeln kommen für diese Arbeit in Frage:

3.1.1 Kommentare

RK02 Gute Kommentare schreiben

Auch wenn Kommentare in den meisten Fällen vermieden werden sollten, gibt es diverse Gründe wo ein Kommentar trotzdem angebracht oder notwendig ist. Ein Beispiel eines solchen "guten Kommentar" wäre:

```
// Gives back an Instance of the tested Responder
Responder responderInstance();
```

Dieser Kommentar gibt zusätzlichen Kontext, welcher aus der Benennung der Variable und Funktion nicht hervorgehen kann.

Umsetzbarkeit

Die Prüfung des Kontext liegt, wie im vorgängigen Abschnitt beschrieben, ausserhalb des Fokus den wir in diesem Projekt setzen wollen. Die Regel könnte aber trotzdem teilweise umgesetzt werden, indem man prüft ob im Kommentar primär Nomen und Verben der Benennungen im darunterliegenden Quelltext vorkommen. Ist dies der Fall handelt es sich mit grosser Wahrscheinlichkeit um einen "schlechten Kommentar".

3.1.2 Objekte und Datenstrukturen

OD01 Law of Demeter

Die Regel "Law of Demeter" besagt, dass ein Modul nichts über die Internas von Objekten kennen sollte, die es manipuliert.

```
var address = account.getAccountHolder().getAddress();
```

Handelt es sich beim obigen Account und AccountHolder um Objekte, ist die Regel verletzt. Handelt es sich hingegen um Datenstrukturen, greift die Regel gar nicht erst.

Umsetzbarkeit

Für eine Umsetzung müssten in einem ersten Schritt verkettete Methodenaufrufe erkannt werden. Danach müsste für die Klassen, auf welchen die Methoden aufgerufen werden, geprüft werden ob es sich bei ihnen um Objekte oder Datenstrukturen handelt.

3.2 Kontextunabhängige Regeln

In diesem Abschnitt sollen die verbleibenden Regeln analysiert und – anhand eines Beispiels – erläutert werden. Ausserdem soll entschieden und begründet werden, auf welche Regeln wir uns fokussieren.

3.2.1 Benennungsregeln

BR04 Aussprechbare Namen verwenden

Namen sollten aussprechbar sein, damit unser Gehirn einfacher mit ihnen umgehen kann. Damit ist insbesondere die Vermeidung von wenig bekannten Akronymen & Abkürzungen aber auch das Weglassen einzelner Buchstaben eines Wortes gemeint. Denn diese Gegebenheiten führen dazu, dass unser Gehirn mehr Schwierigkeiten hat den Quelltext "flüssig" zu verarbeiten.

Beispiel (Ausschnitt aus [12] Kapitel 2.5):

```
class DtaRcrd102 {
    ...
}
```

Ist sowohl schwer lesbar (muss zuerst entziffert werden) und aus dem Namen heraus ist nicht intuitiv klar, was der Kontext dieser Klasse sein könnte.

Besser wäre daher:

```
class Customer {
    ...
}
```

Umsetzbarkeit

Um einen aussprechbaren Namen zu erhalten müssen vollständige Wörter verwendet werden. Wird dies sichergestellt, folgen daraus grösstenteils aussprechbare Benennungen. Die technische Umsetzung könnte mit einem Abgleich der Variablen und Funktionsnamen mit einer umfangreichen, englischen Wörterliste bewerkstelligt werden.

BR05 Suchbare Namen verwenden

Es kann beim Programmieren (vor allem bei grösseren Projekten) nötig sein, dass man nach einer bestimmten Variablen oder Funktion suchen muss, deren Namen man nicht genau kennt. Im Grossen und Ganzen liegen hier die selben Ansätze vor wie bei Regel #4 (Aussprechbare Namen verwenden.).

Dies zeigt auch das folgende Beispiel (Ausschnitt aus [12], Kapitel 2.6):

```
e = 7
```

Wenn man nach der maximalen Anzahl zu belegender Klassen eines Studenten suchen muss, wird man kaum intuitiv nach "e" suchen. Ausserdem würde die Suche, selbst wenn man nach "e" suchen würde, keine brauchbaren Ergebnisse liefern, da alle Namen mit einem "e" als Treffer zurückkommen würden.

Besser wäre daher:

```
MAX_CLASSES_PER_STUDENT = 7
```

Umsetzbarkeit

Ein suchbarer Name besteht fast immer aus vollständigen Wörtern. Beispiele in denen das nicht der Fall ist, werden wir vernachlässigen da der zusätzliche Aufwand im Vergleich zum Zugewinn in keinem Verhältnis steht. Daher kann der selbe Ansatz für die Umsetzung wie bei Regeln #4 angewendet werden. Die Wörter können natürlich sinnlos verwendet oder kombiniert werden. Dies zu erkennen würde aber ein Kontextverständnis seitens des Programms erfordern. Dies liegt, wie im Abschnitt Kontextabhängige Regeln beschrieben, aber nicht im Fokus dieser Arbeit.

BR06 Codierungen vermeiden

Bei dieser Regel handelt es sich im wesentlichen um eine Präzisierung von Regel #4 (Aussprechbare Namen verwenden). Sie beschreibt, dass bei der Benennung von Variablen und Funktionen keine (eigenen) Codierungen verwendet werden sollen, da das zusätzliche Entschlüsseln dieser Codierung das Lesen und Interpretieren weiter erschwert.

Beispiel (Ausschnitt aus [12] 2.7)

```
public class Part {
    private String m_dsc: \\die textliche Beschreibung
    void setName(String name) {
        m_dsc = name;
    }
}
```

Dieser Codeausschnitt ist nicht schnell und intuitiv verständlich.

Besser wäre daher:

```
public class Part {
    String description;
    void setDescription(String description) {
        this.description = description;
    }
}
```

Dieser Codeausschnitt lässt sich einfach & schnell lesen und der Sinn und Zweck geht intuitiv daraus hervor.

Umsetzbarkeit

Die Aussagen der Regel #5 (Suchbare Namen verwenden) trifft hier ebenfalls 1:1 zu (wie auch die Aussagen zu Regel #4). Das gleiche gilt für die technische Umsetzung und die Einschränkungen.

Wir werden diese 3 Regeln daher aufgrund der gemeinsamen technischen Umsetzung ab sofort unter dem gemeinsamen Namen "Aussprechbare, suchbare und nicht codierte Namen verwenden" weiter behandeln.

BR08 Klassen- und Objektnamen

Den Klassen und Objekten sollten Namen gegeben werden, welche aus einem Substantiv oder einem substantivischen Ausdruck bestehen.

Zum Beispiel (Ausschnitt aus [12] Kapitel 2.9)

```
Customer, WikiPage, Account oder AdressParser
```

Hingegen sollten Wörter wie:

```
Manager, Processor, Data oder Info
```

vermieden werden. Ein Klassenname sollte ausserdem kein Verb sein.

Umsetzung

Zur Umsetzung ist ein Datensatz mit möglichst allen englischen Wörtern kategorisiert nach "Wortart" nötig. Damit kann die Einhaltung dieser Regeln mit einem Abgleich des gegebenen Namens geprüft werden. Ausserdem kann gegen eine Blacklist von Wörtern geprüft werden, welche nach den Clean Code Regeln nicht im Namen verwendet werden sollten.

BR09 Methodennamen

Methodennamen sollten aus einem Verb oder einem Ausdruck mit einem Verb bestehen (in der Reihenfolge: Verb + Ausdruck).

Zum Beispiel (Ausschnitt aus [12] Kapitel 2.10):

```
postPayment, deletePage, oder save
```

Mutatoren und Prädikate sollten weiter nach ihrem Wert benannt werden und gemäss dem JavaBean Standard [17] einen Präfix wie "get", "set" oder "is" haben.

Umsetzung

Diese Regel kann technisch 1:1 wie die Regel #8 (Klassen- und Objektnamen) umgesetzt werden, allerdings mit einer abweichenden Wort-Kategorie und Blacklist.

Da die Regeln #8 und #9 im Bezug auf ihre Umsetzung identisch sind, werden diese beiden Regeln zusammengefasst und ab sofort unter dem gemeinsamen Namen "Klassen-, Objekt- und Methodennamen" weiter behandelt.

3.2.2 Funktionsregeln

FU05 Funktionsargumente (max. 3)

Die optimale Anzahl von Argumenten für eine Funktion ist gemäss Robert C. Martin 0 [12] (S71). Auch 1 und 2 Argumente sind noch in Ordnung. Ein drittes Argument sollte wenn möglich vermieden werden. Mehr als drei Argumente sollten niemals benutzt werden. Begründet wird dies mit der grossen konzeptionellen Vorstellungskraft die zum Verständnis jedes zusätzlichen Arguments erforderlich ist. Ausserdem wird es mit der steigenden Anzahl von Argumenten immer schwieriger, die Funktion sinnvoll zu testen.

Umsetzung

Diese Regel kann einfach umgesetzt werden. Dafür brauchen wir lediglich die Anzahl der Argumente einer Funktion zu zählen und bei 3 oder mehr Argumenten eine entsprechende Warnung anzuzeigen. Da für eine automatische Reduktion der Parameteranzahl aber wiederum Kontextwissen der Software vorausgesetzt wäre, würde dies ausserhalb des Projektfokus liegen. Alternativ könnte im Plugin auf eine Website verlinkt werden, wo der Sachverhalt erläutert wird.

3.2.3 Objekte und Datenstrukturregeln

OD02 Objekt / Datenstruktur Hybriden vermeiden

Objekte verstecken ihre internen Daten und exponieren Verhaltensweisen. Dies macht es einfach den Aufbau der zugrundeliegenden Daten zu verändern, da die BenutzerIn des Objekts nichts über dessen Struktur weiss. Datenstrukturen auf der anderen Seite haben kein Verhalten sondern beinhalten nur Daten deren Aufbau der BenutzerIn in diesem Fall bekannt sind. Dies macht das Ändern der Funktionsweise einfach, dafür ist der Aufwand hoch, wenn man die Struktur der Daten ändern will.

Wenn man diese beiden Konzepte mischt, resultieren daraus Hybriden welche sowohl das Ändern der Funktionsweise als auch der Struktur der Daten schwierig macht, was schlecht für die Veränderbarkeit einer Software ist.

Umsetzung

Diese Regel könnte umgesetzt werden, indem man analysiert ob eine Klasse sowohl Zugriff auf die zugrunde liegenden Felder exponiert, als auch zusätzliche Funktionsweisen über Methoden zur Verfügung stellt. Ein automatischer Lösungsvorschlag würde Kontextwissen der Software voraussetzen, es könnte jedoch im Plugin auf eine externe Hilfe verwiesen werden, die das Problem erklärt.

3.3 Diskussion

In diesem Abschnitt geht es darum, mit Hilfe der Erkenntnisse aus der Detailanalyse die Regeln zu identifizieren, welche für diese Arbeit am interessantesten sind und welche davon in welcher Priorität umgesetzt werden sollen. Das Ergebnis haben wir in der Tabelle 3.2 dargestellt.

Bereits in der Aufgabenstellung der Studienarbeit wird der Fokus auf die Benennungsregeln von Variablen und Funktionen gelegt. Entsprechend haben wir die beiden identifizierten Regeln (a & b) aus diesem Themenbereich auch am höchsten priorisiert. Beide Regeln können ausserdem auf technischer Ebene ähnlich umgesetzt werden. Zusätzlich spricht für diese beiden Regeln, dass sie beide durch uns aus je 3 ähnlichen Regeln zusammengesetzt wurden und daher 6 Regeln in einem abdecken.

Bei den Regeln c & e handelt es sich um kontextabhängige Regeln. Aufgrund der Schwierigkeit im Zusammenhang mit der Prüfung vom Kontext bei solchen Regeln, werden wir diese Regeln vereinfacht umsetzen. Da diese Regeln aber vor allem für auf dieser Arbeit aufbaunde Projekte interessant sind, wurden sie entsprechend als nächstes priorisiert. Hierbei müsste allerdings die Regel d noch vor der Regel e umgesetzt werden, da die Unterscheidung von Objekten und Datenstrukturen für die Prüfung der Regel e relevant ist.

Tabelle 3.2: Gewählte Regeln

Regel	**Regel Bezeichnung**	**Priorität**	**MUSS / KANN**
a)	Aussprechbare, suchbare und nicht codierte Namen verwenden	1.	MUSS
b)	Klassen-, Objekt- und Methodennamen	2.	MUSS
c)	Gute Kommentare schreiben	3.	KANN
d)	Objekt / Datenstruktur Hybriden vermeiden	4.	KANN
e)	Law of Demeter	5.	KANN
f)	Funktionsargumente (max. 3)	6.	KANN

Die letzte Regel (f) kommt aus dem Themenbereich der Funktionsregeln. Auf diesem Bereich liegt gemäss der Aufgabenstellung weniger der Fokus, weshalb diese die tiefste Priorität erhält.

Der Implementierungsaufwand liess sich für alle Regeln zum Zeitpunkt der Evaluation nur schwer abschätzen. Daher haben wir nur die ersten beiden Regeln als “MUSS-Ziele” (zwingend zu erreichen) für die Arbeit definiert. Die anderen Regeln wurden alle als “KANN-Ziele” definiert, welche von uns verfolgt werden, wenn die “MUSS-Ziele” erfüllt wären und noch Zeit für die Arbeit übrig bleibt.

Kapitel 4

Technische Umsetzung

In diesem Kapitel gehen wir auf die Details der technischen Umsetzung und die entsprechenden Entscheidungsfindungen ein. Ausserdem wollen wir in diesem Kapitel eine Zusammenfassung der Softwarearchitektur geben. Eine detaillierte Architekturdokumentation befindet sich im Anhang.

4.1 Vorgehen

Bei der Umsetzung sind wir gemäss unserem zu Projektbeginn definierten Projektplan vorgegangen. Daher haben wir zuerst einen Prototyp erstellt um sicherzustellen, dass alle angedachten Technologien für dieses Projekt funktionieren und wie erwartet zusammenspielen. Danach haben wir, basierend auf den Erfahrungen mit dem Prototypen, die Architektur des Plugins abschliessend geplant und in einer Architekturdokumentation festgehalten. Zum Abschluss haben wir das Plugin gemäss unserer Architekturplanung umgesetzt und getestet.

4.2 Umsetzung

Die einzelnen Schritte der Umsetzung, sowie die Engineering Probleme die wir dabei lösen mussten haben wir nachfolgend genauer beschrieben.

4.2.1 Eingesetzte Werkzeuge im Projekt

Für dieses Projekt haben wir diverse Werkzeuge eingesetzt. Darunter die Visual Studio Code IDE oder YouTrack als unseren Issue Tracker. Als Versionsverwaltung ist bei uns Git zum Einsatz gekommen. Microsoft setzt Visual Studio Code Extension Projekte direkt mit npm als Build Tool auf, daher kam dieses bei uns für diese Aufgabe zur Verwendung. Da unsere Code Repositories auf einer GitLab Instanz gehostet wurden, haben wir sowohl für die Continous Integration, wie auch für das Artefakte Repository, die von GitLab bereitgestellten Tools (GitLab CI/CD & GitLab Variables) verwendet. Für die Statische Code Analyse haben wir ESLint und SonarQube verwendet. Weitere Informationen zu den angewandten Methodiken können dem im Anhang beigefügten Projektplan entnommen werden.

4.2.2 Prototyp

Da wir mit der Entwicklung von Visual Studio Code Extensions vor dieser Arbeit noch keine Erfahrungen hatten und überdies nur rudimentäre TypeScript Kenntnisse vorweisen konnten, haben wir uns zum Bau eines Prototyps vor der effektiven Umsetzung entschieden. Dieser sollte sowohl die grundsätzliche Machbarkeit unserer Projektidee bestätigen als auch die Möglichkeit geben verschiedene Architekturansätze auszuprobieren.

Auf eine grobe Architektur konnten wir uns relativ schnell festlegen, da diese nicht von der verwendeten Technologie abhängt. Wir mussten für die effektive Umsetzung an dieser nur kleinere Refactorings machen. Grössere Probleme bereitete uns die Integration unserer Prüflogik in eine Visual Studio Code Extension. Es kam mehrfach vor, dass wir einen Lösungsansatz andachten und nach dem Ausprogrammieren feststellten, dass dieser so nicht mit den APIs der IDE funktioniert. Wir wollten zum Beispiel den Befehl zum Aufrufen der Prüfung beim Speichern nur bei Visual Studio Code hinterlegen, wenn die ProgrammiererIn auch diesen Modus eingestellt hatte. Dies führte zu vielen kryptischen Fehlermeldungen bis wir unseren Ansatz dahingehend abänderten, dass der Befehl beim Aufstarten immer hinterlegt

wird und beim Speichern geprüft wird, ob der richtige Modus eingestellt ist, bevor die Prüfung angestossen wird. Viele Probleme bereitete uns auch das grundsätzliche Registrieren von Befehlen an der Entwicklungsumgebung, da diese deklarativ im Manifest der Erweiterung (`package.json`) geschehen muss. Zusätzlich zu den Befehlen müssen hier auch Aktivierungs-Ereignisse definiert werden und es ist nicht immer gänzlich ersichtlich, wie die beiden Konfigurationen zusammenspielen. Erschwert wird dies noch dadurch, dass man nicht mit Quelltext arbeitet und deshalb auch nicht richtig Debuggen kann. All dies führt dazu, dass wir unsere Lösungsansätze nicht mehr von der Codeseite her andachten, sondern uns zuerst informierten, welche APIs für die konkreten Aufgaben zur Verfügung stehen und auf diesen aufbauend die Lösung entwickelten.

Das Resultat der Prototyp-Phase war ein funktionaler Durchstich aller Architekturebenen mit stark vereinfachter Logik in den einzelnen Modulen. Aufbauend auf den gewonnen Kenntnissen entwarfen wir dann das effektive Plugin.

4.2.3 Infrastruktur

Wir haben für das Projekt 3 Git Repositories aufgesetzt. In einem haben wir den Quelltext des Plugins verwaltet. In einem weiteren wurde die Dokumentation, also die Architekturdokumentation, dieser Bericht und alle weiteren Anhänge verwaltet. Das letzte Repository dient zur Unterstützung des Projekts. Darin haben wir Renovate, einen Bot für automatisches Dependency Management für npm verwaltet. Die 3 Git Repositories wurden auf einer selbst gehosteten GitLab Instanz betrieben. Wir haben beim Code Repository ausserdem von der GitLab CI/CD Pipeline Automatisierung Gebrauch gemacht.

Pipeline

Unsere Pipeline besteht aus 3 Schritten:

- 1. Static Code Analysis mit SonarQube
- 2. Linting Checks mit ESLint
- 3. Automatische Ausführung der Mocha basierten Unit Tests

Alle drei Schritte werden automatisch bei jedem Commit in den Master Branch angestossen. Sobald das Plugin einmal einen Status erreicht, dass es auf dem offiziellen Visual Studio Code Marketplace veröffentlicht werden könnte, kann diese Pipeline um einen vierten Schritt erweitert werden, welcher das Plugin automatisch dort veröffentlicht.

4.2.4 Architektur

Das Plugin bestehet aus mehreren Untersystemen, welche, voneinander unabhängig, Teile der Prüflogik ausführen. Diese werden von einem übergeordneten Hauptsystem koordiniert und angestossen.

Der `ParseManager` ist verantwortlich um aus dem Quelltext die Elemente, die zur Prüfung der Clean Code Regeln benötigt werden, zu isolieren und zurückzugeben. Er schaut ob für die jeweilige Programmiersprache ein `Parser` implementiert ist und übergibt diesem den Text. Der `Parser` hat, definiert durch das `Parser`-Interface, eine Methode, welche ein Visual Studio Code `TextDocument` entgegen nimmt und ein Array von `Nodes` zurückgibt. Abgesehen von dieser Interface-Methode ist für die Implementierung eines neuen `Parsers` nichts vorgegeben. Dadurch sind Entwickler flexibel darin ob sie bestehende Parsing-Bibliotheken verwenden oder selbst einen Lexer/Parser implementieren wollen. Die `Nodes` leiten von einer abstrakten `Node`-Klasse ab und erweitern diese um spezifische Felder, welche zusätzlich gebraucht werden.

Der `ValidationManager` ist das Kernmodul und befasst sich mit der Anwendung der Clean Code Regeln auf die vorher isolierten Elemente. Auf dem Manager sind alle vorhandenen `Validator`-Implementationen registriert. Jede dieser Implementationen kümmert sich um die Überprüfung einer einzelnen Regel. Dazu benutzt ein `Validator` ein oder mehrere Rule-Impelmentationen. Schlägt eine Regelprüfung fehl, wird dies in einer Liste von Regelverletzungen vermerkt. Sobald alle Prüfungen abgeschlossen sind, retourniert der `ValidationManager` diese Liste der Regelverletzungen an den Aufrufenden.

Dem `NotificationManager` kann eine Liste von Regelverletzungen übergeben werden, welche dieser der ProgrammiererIn visuell sowohl direkt im Quelltext, als auch im Problem-Fenster von Visual Studio Code anzeigt.

Das Zusammenspiel der drei beschriebenen Untersysteme, wird in der Abbildung 4.1 aufgezeigt.

In der Umsetzung konnten wir viele an der OST gelernte Konzepte praktisch anwenden. Durch die Verwendung von Interfaces (`Parser`, `Validator`, `Rule`, ...) und abstrakten Klassen (`Node`, `RuleViolation`, ...) konnten wir den

Quelltext durch den Einsatz von Typ-Polymorphismen vereinfachen. Die Implementierung des Java-`Parsers` nutzt ausserdem das Visitor-Pattern um den konkreten Syntaxbaum, welcher von der verwendeten Bibliothek zurückgeliefert wird, zu traversieren.

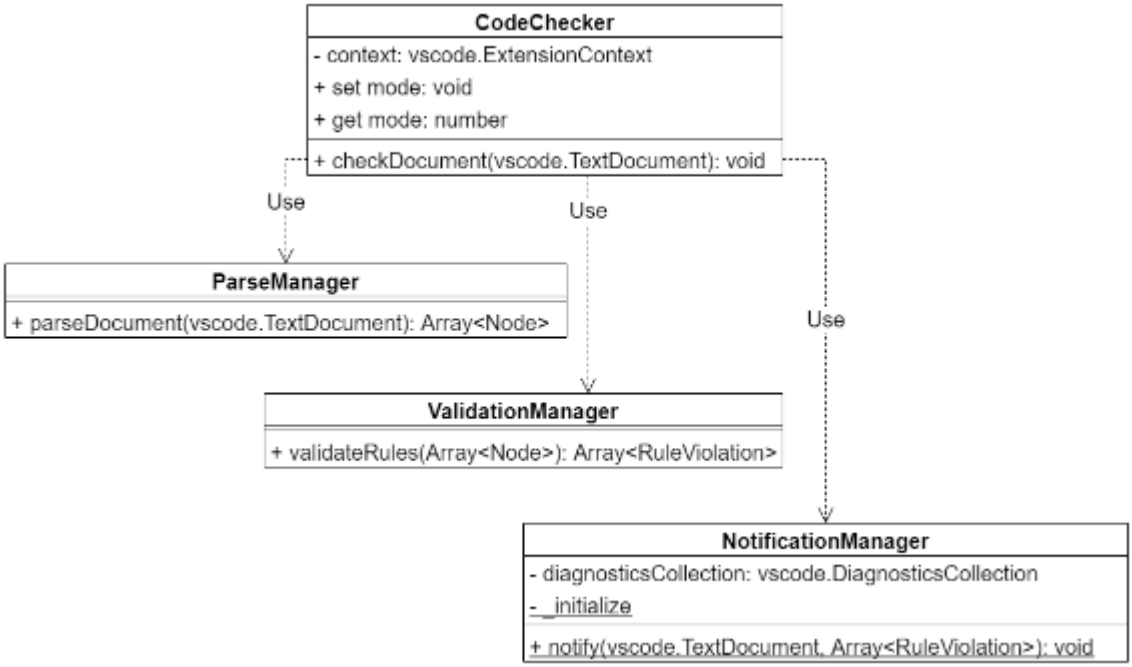


Abbildung 4.1: Bausteinsicht Gesamtsystem

4.2.5 Umsetzung der Regelanalyse

Die einzelnen Regelprüfungen werden von jeweils eigenen `Rule`-Implementationen abgebildet. Einzelne `Rule`-Implementationen können von `Validator`-Implementationen verwendet werden, um die benötigten Prüfungen auf ihren `Nodes` durchzuführen.

Der Hauptfokus der Arbeit liegt auf der Prüfung der Benennungsregeln. Die Umsetzung dieser werden wir hier noch genauer beleuchten. Die Regelprüfungen unterscheiden sich in ihrer Implementation allerdings sehr stark, da diese von der zu prüfenden Regel abhängen.

Um zu prüfen, ob ein Bezeichner sinnvoll benannt ist, müssen wir diesen zuerst einmal in seine Bestandteile zerlegen. Im einfachsten Fall von sauberem camelCase ist dies einfach zu bewerkstelligen. Nach einigen Tests stellten wir allerdings fest, dass es wesentlich komplexere Regular Expressions benötigt, um verschiedene Abweichungen von normalem camelCase auch verarbeiten zu können. In den meisten Programmiersprachen können zum Beispiel Zahlen in Bezeichnern eingesetzt werden. Auch schreiben viele ProgrammiererInnen Akronyme gross, was ebenfalls speziell behandelt werden muss. Es dauerte daher einige Iterationen bis wir zu einer Lösung kamen mit der wir zufrieden waren.

Nachdem die Wörter getrennt sind, muss für jedes Wort geprüft werden ob dieses tatsächlich ein gültiges Wort darstellt. Für diesen Teil der Prüfung vergleichen wir das Wort mit einem Wörterbuch (mehr zum Wörterbuch im nächsten Kapitel). Je nachdem um was für einen Bezeichner es sich handelt, muss diese Prüfung noch konkretisiert werden. Bei einer Methode zum Beispiel sollte das erste Wort ein Verb sein. Auch diese Information stellt das Wörterbuch bereit.

Für jedes Wort welches nicht im Wörterbuch gefunden wird, erstellen wir ein Regelverletzungs-Objekt und informieren den `ValidationManager` darüber.

4.3 Wörterbuch

Für die Umsetzung der Benennungsregeln, welchen wir höchste Priorität eingeräumt hatten, aber auch für die Umsetzung von weiteren potentiellen Clean Code Regeln benötigten wir ein Wörterbuch. Dieses Wörterbuch sollte möglichst

viele englische Wörter inklusive der Wortart (Nomen, Verb, Adjektiv etc...) enthalten, damit wir unsere Regeln überhaupt validieren können. Für weitere Clean Code Regeln, die wir zwar nicht während dieser Arbeit umsetzen wollten, die aber eventuell später umgesetzt werden könnten, wäre ausserdem eine oder mehrere Definitionen zu jedem Wort im Wörterbuch wünschenswert. Trotz intensiver Recherche haben wir weder im Internet noch in den uns von der Hochschule zur Verfügung gestellten Ressourcen ein solches Wörterbuch gefunden. Auch keines welches unter einer kommerziellen Lizenz steht. Wir mussten daher selbst, mit den öffentlich zugänglichen Informationen, ein solches Wörterbuch erstellen.

Die erste Herausforderung die es bei der Umsetzung zu bewältigen gab, war die Beschaffung einer Wortliste die einfach eine grosse Menge an englischen Wörtern enthält. Auf Github wurden wir schliesslich fündig [3]. Die Liste enthält über 460'000 Wörter und war bereits im einfach zu verarbeitenden JSON Format erhältlich. Diese Wortliste stand ausserdem unter der public Domain und konnte von uns daher ohne Einschränkungen genutzt werden. Wir haben die Qualität der Liste mit Stichproben geprüft und haben dabei feststellen müssen, dass die Qualität vieler Wörter auf der Liste mangelhaft war. So waren auch viele Wörter vorhanden, die lediglich Abkürzungen für andere Begriffe waren, wie z.B. *aa* das für *an associate degree in arts* stehen könnte. Oder *xi* das für die Zahl elf auf römisch stehen könnte. Einige dieser Probleme wie z.B. jeden einzelnen Buchstabe des Alphabets, welche sich auf der Liste befanden, konnten wir manuell beseitigen. Die gesamte Liste mit über 460'000 Wörtern auf solche Wörter zu prüfen wäre aber in der Projektzeit weder möglich noch effizient gewesen. Wir mussten das Problem also anderweitig lösen.

Die Lösung fanden wir schliesslich im weiteren Vorgehen. Da wir keine Liste mit Worttypen oder Definitionen gefunden hatten, mussten wir eine Web API, die einen solchen Service anbietet verwenden. Ein solcher API Service kennt meistens deutlich weniger als die 460'000 Wörter in der Liste und so war es uns möglich die ungewünschten Wörter herauszufiltern, wenn die API keine Definition für das Wort zurückliefern konnte. Nach einer kurzen Recherche haben wir die folgenden Anbieter, wie in Tabelle 4.1 dargestellt, identifiziert. Kriterium für die Auswahl war die Anzahl erlaubten Abfragen und ein Gratis Tarif oder ein günstiger Tarif der weniger als fünfzig Schweizerfranken im Monat kostet.

Tabelle 4.1: API Anbieter

Anbieter	**Erlaubte Abfragen**	**Anzahl Wörter**	**Kosten pro Monat**
Oxford Dictionary	1000 pro Monat	>170'000	Free
Merriam Webster	1000 pro Monat	>250'000	Free
WordsAPI	25'000 pro Tag	>150'000	10$

Wir haben uns schliesslich für WordsAPI [29] entschieden, da sie uns als einziger Provider genug Abfragen pro Tag erlaubten, damit wir bis zum Ende des Projekts die mehr als 460'000 Wörter abarbeiten konnten.

Zur Umsetzung haben wir ein Python Script erstellt, welches alle Wörer der Wörterliste einzeln durchgeht und eine Abfrage an die WordsAPI sendet. Kannte die WordsAPI das Wort nicht oder die Antwort enthielt keine Wortdefinition, wurde das Wort verworfen. Ansosten wurde das Wort samt Definition und `PartOfSpeech` (Worttyp) im JSON Format abgespeichert. Dies wurde für alle Wörter in der Wortliste gemacht. Am Ende blieben über 98'000 Wörter übrig die ins Wörterbuch gespeichert wurden. Das finale Wörterbuch wird nun vom Plugin verwendet. Das Format haben wir in der Abbildung 4.2 dargestellt.

```
  "word":"professor",
"definitions":[{"definition": "someone who is a member of the faculty at a college or university", "partOfSpeech": "noun"}]
```

Abbildung 4.2: Beispiel Wörterbucheintrag

4.4 Testing

Neben Unit Tests sollte unser Plugin auch auf seine Usability getestet werden. Dafür haben wir es einem uns bekannten Programmierer gegeben, damit er das Plugin vorab testen konnte. Ziel dieses Tests war auch, sonstiges Feedback

jeglicher Art zu sammeln. Ein Testprotokoll liegt dem Anhang bei.

4.4.1 Auswertung

Die Auswertung haben wir in der Tabelle 4.2 zusammengefasst.

Tabelle 4.2: Auswertung Testing

Kritikpunkt	**Lösung**
Spezifischere Fehlermeldungen bzw. Hintergrundinformationen zum Problem.	Wurde auf die Liste weiterer Implementierungsmöglichkeiten für die Zukunft aufgenommen.
Die Überprüfung, ob ein Klassennamen mit einem Nomen startet, schlägt fehl.	Die Überprüfung funktionierte korrekt. Der Fehler entstand beim testen mit dem Wort "get", welches jedoch eine Definition als Nomen hat.
Die englischen Wörter "is" und "a" existieren nicht im Wörterbuch.	Das Wörterbuch wurde manuell um diese fehlenden Wörter, inklusive deren Definitionen, erweitert.
Das fehlen einer Whitelist für eigene Wortdefinitionen.	Wurde auf die Liste weiterer Implementierungsmöglichkeiten für die Zukunft aufgenommen.
In den Fehlermeldungen soll (graphisch) hervorgehoben werden welches Wort fehlerhaft ist.	Wurde auf die Liste weiterer Implementierungsmöglichkeiten für die Zukunft aufgenommen.
Zahlen sollen generell nicht als "fehlerhafte Wörter" hervorgehoben werden.	Die Logik zur Auftrennung von Bezeichnern in einzelne Worte wurde überarbeitet und liefert nun Zahlen gar nicht mehr zur Prüfung zurück.
Die Whitelist sollte zwischen mehreren Teammitgliedern gesynct werden können bzw. sollte austauschbar sein.	Wurde auf die Liste weiterer Implementierungsmöglichkeiten für die Zukunft aufgenommen.

Kleinere Kritikpunkte aus dem Feedback haben wir noch während der Arbeit direkt umgesetzt. Grössere Punkte wurden schriftlich festgehalten, um in Zukunft durch Nachfolgeprojekt(e) umgesetzt zu werden.

4.5 Ergebnis

Das Ergebnis des Projekts ist eine Visual Studio Code Extension, welche einige der identifizierten Regeln prüft und Verstösse der ProgrammiererIn anzeigt. Die Frequenz in welcher die Prüfung durchgeführt wird, kann über Visual Studio Code Commands angepasst werden. So kann der Programmierende die Prüfung manuell ausführen, automatisiert nach jedem Speichervorgang oder nach jeder Codeänderung.

Da wir uns dagegen entschieden haben, die Extension im Visual Studio Marketplace zu veröffentlichen, muss die vsix-Datei manuell installiert werden.

Die Extension ist vom Aufbau her in der Lage beliebig viele Programmiersprachen zu unterstützen. Umgesetzt haben wir die Prüfung von Quelltext in der objektorientierten Sprache Java.

Die Tabelle 4.3 zeigt welche Regeln im Zuge der Arbeit umgesetzt wurden:

Durch die offene Architektur ist es im Anschluss einfach weitere Regelprüfungen zu implementieren.

Das fertige Plugin, welches die Auswertung auf einer Testdatei vornimmt und diese Regelverletzungen anzeigt, ist in der Abbildung 4.3 ersichtlich.

Tabelle 4.3: Umgesetzte Regeln

Regel	**Regel Bezeichnung**	**Priorität**	**Status**
a)	Aussprechbare, suchbare und nicht codierte Namen verwenden	1.	Umgesetzt
b)	Klassen-, Objekt- und Methodennamen	2.	Umgesetzt
c)	Gute Kommentare schreiben	3.	Nicht umgesetzt
d)	Objekt / Datenstruktur Hybriden vermeiden	4.	Nicht umgesetzt
e)	Law of Demeter	5.	Nicht umgesetzt
f)	Funktionsargumente (max. 3)	6.	Umgesetzt

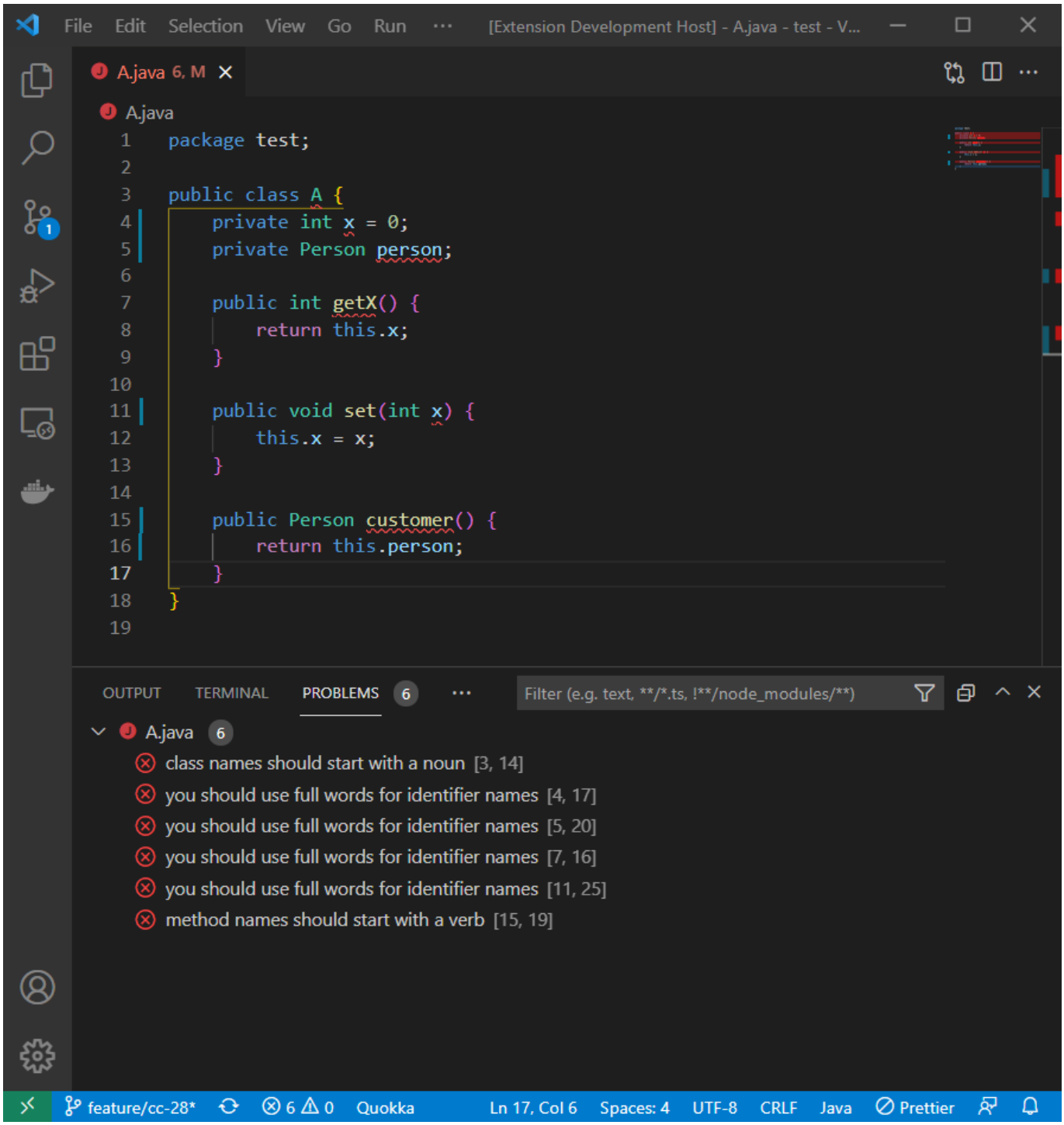


Abbildung 4.3: Beispiel des fertigen Plugins

4.6 Diskussion

In diesem Kapitel beschreiben wir die konkrete Implementation unserer Arbeit. Wir mussten auf dem Weg einige Probleme überwinden.

Die nennenswertesten sind die Ausarbeitung einer funktionierenden Architektur und die Erstellung einer passenden Wörterliste, da diese den grössten Aufwand dargestellt haben. Die Architektur mussten wir mehrmals überdenken und auch während der Umsetzung hin und wieder nachjustieren. Bei der Umsetzung der Arbeit mussten wir diverse Engineering Probleme lösen, was viel Zeit in Anspruch genommen hat. Während die eigentliche Entwicklung des Plugins uns in der Umsetzung weniger Probleme bereitet hat, standen wir öfters vor dem Problem, dass wir beim Suchen nach Lösungen von kleineren Problemen nicht fündig wurden. Das konkrete Problem war hier, dass es sehr viele Resultate zu Problemen mit Visual Studio Code von den Benutzern der Software gab, aber gleichzeitig nur wenige Resultate von Entwicklern zu Problemen der Entwicklung von Extensions an sich. Beim Erstellen der Wörterliste war die grösste Herausforderung an passende Daten zu kommen. Die frei verfügbaren Daten die unter entsprechend freier Lizenz standen, waren jeweils von unzureichender Qualität. Durch den ohnehin nötigen Abgleich mit einer kostenpflichtigen API für die weiter benötigten Informationen wie den Worttyp, konnten wir die Datenqualität erheblich steigern. Im Zusammenhang mit der Erstellung des Wörterbuchs haben wir auch mehr über das JSON Format bzw. das Manipulieren von JSON gelernt.

Die Infrastruktur hat uns während des gesamten Projekts keine Probleme bereitet und immer wie gewünscht funktioniert. Das gleiche gilt für die Verfügbarkeit der Infrastruktur die stets gegeben war. Dies dürfte daran gelegen haben, dass wir sie komplett selbst gehostet und aufgesetzt hatten. Entsprechend hatten wir in der ursprünglichen Projektplanung zu viel Zeit für die Infrastruktur eingeplant. Die freigewordene Zeit konnten wir dann dafür in das Lösen von Problemen und die Entwicklung eines weiteren, optionalen Features investieren. Insgesamt hatten wir während der Arbeit immer wieder viele kleinere Probleme zu bewältigen. Auf grössere, unlösbare Probleme sind wir aber während der gesamten Arbeit nicht gestossen.

Besonders wichtig war die Durchführung einer Nutzertestung, die von einem unserer Kommilitonen durchgeführt wurde. Diese hat uns nochmal einige Probleme mit unserer Extension aufzeigen können, die uns vorher so nicht bewusst waren. Weitere Informationen zu den Resultaten dieses Tests kann der Tabelle 4.2 entnommen werden.

Wir sind mit dem Ergebnis zufrieden. Es ist uns gelungen, ein funktionierendes Plugin zu programmieren und so eine Grundlage zu schaffen, auf welcher weiter aufgebaut werden kann. Mit dem Abschluss der Implementation war diese Studienarbeit so gut wie vollendet. Als letztes haben wir nun noch diesen Bericht finalisiert und eine Projektauswertung vorgenommen in der wir Gelerntes herausgearbeitet und festgehalten haben.

Kapitel 5

Zusammenfassung

In der Schlussfolgerung fassen wir die Studienarbeit nochmals zusammen und evaluieren die erhaltenen Resultate. Ausserdem geben wir einen Ausblick darauf, wie man unser Plugin in der Zukunft weiterentwickeln könnte und welche Themenbereiche dafür noch erforscht werden könnte.

5.1 Gewonnene Erkenntnisse

Wir haben im Rahmen dieser Arbeit unglaublich viel gelernt. Im folgenden wollen wir einige dieser Erkenntnisse diskutieren.

5.1.1 Architektur

Im Bereich der Architektur hatten wir bereits einiges an Vorwissen. Zum einen wegen der theoretischen Ausbildung an der OST und zum anderen durch unsere berufliche Tätigkeit neben dem Studium. Entsprechend waren wir mit diversen Konzepten, die wir eingesetzt haben, bereits vertraut. Dennoch gab es vereinzelt Konzepte, die wir so noch nie praktisch angewendet haben. Ein konkretes Beispiel dafür wäre der praktische Einsatz des Visitor Patterns zum besuchen der einzelnen Parser Nodes. Weitere Details zur Architektur bzw. deren Umsetzung können der Architekturdokumentation im Anhang entnommen werden.

5.1.2 Clean Code

Auch wenn wir beide schon Erfahrung mit dem Programmieren und dem Thema Clean Code hatten, haben wir durch die detaillierten Analysen der verschiedenen Regeln und der Implementierung von Prüfmechanismen für ebendiese, unser Wissen in diesem Bereich weiter ausbauen und vertiefen können.

Aus diesem Punkt nehmen wir vor allem mit, wie umfangreich und komplex gewisse Clean Code Regeln in der Prüfung sein können, auch wenn sie zuerst simpel anmuten. Hier wird noch viel Forschung investiert werden müssen, bis man alle diese Regeln einmal automatisch prüfen können wird.

5.1.3 LaTeX

Zu Beginn der Arbeit haben wir uns, aufgrund unserer schlechten Erfahrungen mit Word, für die Umsetzung des Berichts mit LaTeX entschieden. Wir hatten bereits grundlegende LaTeX Kenntnisse, aber für die Umsetzung einer Arbeit in diesem Umfang sind wir hier auf Themengebiete gestossen, die wir vorher noch nie gebraucht haben. So hat uns z.B. das Erstellen des Glossars einige Probleme bereitet.

Als wichtigstes Learning in diesem Punkt werden wir die Erkenntnis mitnehmen, dass LaTeX zwar sehr mächtig bzw. viel mächtiger als Word ist was die Möglichkeiten angeht, aber auch mehr Aufwand bei der Umsetzung in Anspruch nimmt. Sobald man die Grundlagen aber beherrscht empfinden wir LaTeX als besser geeignet für wissenschaftliche Arbeiten.

5.1.4 Literaturrecherche

Auch wenn wir an der Hochschule und zum Teil auch bereits davor Literaturrecherchen durchgeführt hatten, war dies das erste Mal in einem solchen Umfang. Da es sich mit Clean Code um ein uns Beiden vertrautes Thema handelte, war die Recherche an sich zwar zeitaufwändig, hat uns aber nicht vor grundlegende Probleme gestellt. Was uns allerdings Probleme gemacht hat, war die Darstellung der daraus abgeleiteten Ergebnisse. Da wir beide damit keine Erfahrung hatten, waren wir froh, dass unser Betreuer uns immer wieder mit viel nützlichem Feedback unterstützen konnte.

Aus den vielen gelernten Inhalten, nehmen wir vor allem die Nützlichkeit einer strukturierten Darstellung (z.B. mit Tabellen) bei komplexen Sachverhalten für die Zukunft mit.

5.1.5 Technologien

Zu Beginn des Projekts verfügten wir nur über rudimentäre Kenntnisse zur Sprache Typescript. Im Laufe der Arbeit haben wir uns aber immer weiter in diese Technologie vertiefen und unser Wissen erweitern können. So haben wir beispielsweise im Laufe des Projektes gelernt, wie man die Imports mit Typescript korrekt verwendet und wie man ein Typescript Projekt organisieren sollte. Ein weiteres Beispiel wäre die Verwendung von Properties in Typescript. Diese haben wir so vor dieser Arbeit noch nicht gekannt, haben sie aber als sehr nützlich bei der Implementierung empfunden.

Über das Testing Framework Mocha, welches wir vor diesem Projekt ebenfalls noch nie verwendet hatten, haben wir ebenfalls viel gelernt. Trotz Ähnlichkeiten mit uns bereits bekannten Frameworks wie JUnit, mussten wir uns zuerst an durch Typescript verursachten Sprachunterschiede (im direkten Vergleich zu Java) gewöhnen.

Da wir unser Wörterbuch selbst, mit Hilfe der Daten einer externen API, erstellen mussten, haben wir auch viel über das Datenformat JSON gelernt. Vor allem das Zusammensetzen der einzelnen JSON Fragmente der unterschiedlichen Abfragen, war in dieser Form neu für uns.

5.1.6 Visual Studio Code Plugin Entwicklung

Durch die Entwicklung eines Plugins für die IDE, die wir selbst schon seit Jahren einsetzen, haben wir viel über den Aufbau und die einzelnen Komponenten der IDE gelernt. Die Entwicklung für die Visual Studio Code Plattform haben wir dabei als angenehm empfunden, da alle Aspekte welche wir benötigten von Microsoft dokumentiert wurden. Lediglich im Bereich der API Dokumentation gibt es noch Verbesserungspotential, da in der Dokumentation auf Beispiele verzichtet wurde. Daher mussten wir zum Teil selbst ergründen, wie eine API genau funktioniert bzw. wie sie genau eingesetzt werden soll.

Eine besondere Herausforderung bei der Implementierung waren auch die Themengebiete des Lexen und Parsen einer Programmiersprache. Dabei handelt es sich um ein sehr interessantes Gebiet, das man im Alltag leider eher selten antrifft. Entsprechend konnten wir auch hier viel Neues lernen.

5.2 Weitere Schritte

Mit dem Abschluss unserer Arbeit haben wir die Grundlage eines Plugins geschaffen, einfach weiterentwickelt werden kann. Wir haben uns zum Abschluss der Arbeit dazu Gedanken gemacht, wie eine Weiterentwicklung konkret aussehen könnten. Dabei unterscheiden wir zwischen konkreten Implementierungsmöglichkeiten und Feldern der Forschung die in diesem Zusammenhang vertieft werden könnten.

Um die Arbeit an dieser Extension fortführen zu können, sollte man deshalb unbedingt Technologien wie Typescript, Mocha und JSON beherrschen. Ausserdem ist ein tiefgreifendes Verständnis über den Aufbau von Programmiersprachen, sowie der Funktionsweise von Werkzeugen wie Lexer und Parser unabdinglich. Weiter sollte man ein grundlegendes Verständnis über CI/CD Prozesse haben, um das Deployment angemessen automatisieren zu können. Zuletzt ist natürlich auch ein vertieftes Wissen rund um die Clean Code Prinzipien wichtig.

5.2.1 Erweiterungsmöglichkeiten

Während der Arbeit mussten wir bei vielen Themen entscheiden, worauf wir den Fokus legen wollten. Dabei wurden automatisch gewisse Punkte vernachlässigt. Ausserdem haben wir bei der Nutzertestung Feedback erhalten, wie wir das Plugin in der Zukunft weiter verbessern könnten. Wir möchten daher diese Punkte hier festhalten:

- Veröffentlichung des Plugins im Visual Studio Code Marketplace für die freie Nutzung

- Spezifischere Darstellung des Problems im Editor (Welches Wort in der Bezeichnung ist das Problem?)
- Spezifische Regeln sollen mittels einer Konfigurationsdatei deaktiviert werden können
- Ausweitung des Sprachsupports über Java hinaus (z.B. Support für Python, Typescript, Go etc. . .)
- Möglichkeit das Wörterbuch durch eigene Domänen- bzw. Projektspezifische Wörter zu erweitern
- Möglichkeit eigene Wörterbücher zwischen Teammitgliedern zu verteilen bzw. zu synchronisieren
- Die Möglichkeit eine Problemmeldung zu akzeptieren bzw. zu unterdrücken
- Implementation eines eigenen Language Servers
- Die Möglichkeit einfach selbst weitere Regeln zu definieren, die dann vom Plugin geprüft werden
- Hinzufügen von Weblinks auf Seiten welche die Regel mit typischen Problemen genauer erklären und typische Lösungsansätze aufzeigen (Spezifischere Fehlermeldungen)
- Die Möglichkeit Clean Code Metriken auszuwerten und mit SCA Tools wie z.B. SonarQube zu analysieren & auszuwerten

5.2.2 Forschungsfelder

Einige der Clean Code Regeln, vor allem Regeln die ein Kontextverständnis zur Erkennung eines Verstosses voraussetzen, können mit den heutigen Mitteln leider noch nicht implementiert werden. Wir haben uns daher im Rahmen der Reflexion über diese Arbeit auch Gedanken gemacht, wo es noch Forschungsarbeit braucht um dies in Zukunft zu ermöglichen. Dabei wollen wir die folgenden Punkte festhalten:

- Trainieren eines Modells auf schlechtem Quelltext, um Verstösse gegen die Clean Code Prinzipien automatisiert mit Mustererkennung zu finden.
- Erkennung von Problemen im Zusammenspiel mit weiteren in der Praxis geläufigen Code Analyse Tools wie z.B. Software Cities. Durch einen austausch von Metriken könnten die Analysen auf beiden Seiten verbessert werden.
- Analyse von klassenspezifischen Abhängigkeiten, um weitere kontextabhängige Regeln im Bereich der Objekte und Datenstrukturen umsetzen zu können.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Helmut Balzert. *Lehrbuch der Softwaretechnik. Band 2 Softwaremanagement, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag, 1998.

[2] Mozilla MDN Web Docs. Arbeit mit json. `https://developer.mozilla.org/de/docs/Learn/JavaScript/Objects/JSON`, Dezember 2021.

[3] dwyl. english words. `https://github.com/dwyl/english-words`, November 2021.

[4] Dustin Boswell & Trevor Foucher. *The Art of Readable Code*. O'Reilly Media, Inc., 2012.

[5] Martin Fowler. Refactoring. `https://www.refactoring.com/`, Dezember 2021.

[6] Red Had. Was versteht man unter ci/cd? `https://www.redhat.com/de/topics/devops/what-is-ci-cd`, April 2018.

[7] Red Had. Was ist eine ide? `https://www.redhat.com/de/topics/middleware/what-is-ide`, Januar 2021.

[8] Red Had. Was ist eine java runtime-umgebung (jre)? `https://www.redhat.com/de/topics/cloud-native-apps/what-is-a-Java-runtime-environment`, Dezember 2021.

[9] Dev Insider. Was ist ein parser? `https://www.dev-insider.de/was-ist-ein-parser-a-756662/`, Oktober 2018.

[10] iso25000.com. Iso/iec 25010. `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010`, Oktober 2021.

[11] Kulturbanause. Was ist camelcase? `https://kulturbanause.de/faq/camelcase/`, Dezember 2021.

[12] Robert C. Martin. *Clean Code*. mitp Verlag, 2009.

[13] Boris Mayer. Ohne passende werkzeuge geht das nicht. `https://www.golem.de/news/softwareentwicklung-ohne-passende-werkzeuge-geht-das-nicht-2104-153931.html`, April 2021.

[14] Steve McConnel. *Code Complete 2*. Microsoft Press, 2004.

[15] Microsoft. Typescript is javascript with syntax for types. `https://www.typescriptlang.org/`, Dezember 2021.

[16] Microsoft. Visual studio code. `"https://code.visualstudio.com/`, Oktober 2021.

[17] Oracle. Java se desktop technologies. `https://www.oracle.com/java/technologies/javase/desktop.html`, Oktober 2021.

[18] Arkadius Roczniewski. Schluss mit frust: Clean code hilft bei der softwarequalität. `https://www.heise.de/hintergrund/Schluss-mit-Frust-Clean-Code-hilft-bei-der-Softwarequalitaet-6181456.html`, September 2021.

[19] RAM SAGAR. Openai's gpt-3 can now generate the code for you. `https://analyticsindiamag.com/open-ai-gpt-3-code-generator-app-building/`, Juli 2020.

[20] Talend. Api (application programming interface). `https://www.talend.com/de/resources/was-ist-eine-api/`, Dezember 2021.

[21] Andrew Hunt & David Thomas. *The Pragmatic Programmer*. Addison Wesley Longman, 1999.

[22] Wikipedia. Git. `https://de.wikipedia.org/wiki/Git`, Juni 2021.

[23] Wikipedia. Künstliche intelligenz. https://de.wikipedia.org/wiki/K%C3%BCnstliche_Intelligenz, Dezember 2021.

[24] Wikipedia. Lint (programmierwerkzeug). https://de.wikipedia.org/wiki/Lint_(Programmierwerkzeug), September 2021.

[25] Wikipedia. Modultest. https://de.wikipedia.org/wiki/Modultest, September 2021.

[26] Wikipedia. Qualität. https://de.wikipedia.org/wiki/Qualit%C3%A4t, Januar 2021.

[27] Wikipedia. Statische code-analyse. https://de.wikipedia.org/wiki/Statische_Code-Analyse, März 2021.

[28] Wikipedia. Visual studio code. https://de.wikipedia.org/wiki/Visual_Studio_Code, November 2021.

[29] WordsAPI. Wordsapi. https://www.wordsapi.com/, November 2021.

Glossar

Begriff	Definition
AI	*Artificial Intelligence, auf deutsch künstliche Intelligenz (KI) beschreibt ein Teilgebiet der Informatik, in dem Aspekte des menschlichen Denkens & Handelns im Computer nachgebildet werden. [23]*
Application Programming Interface (API)	*Eine API ist eine Programmierschnittstelle die von einem Softwaresystem für andere Systeme zur Verfügung gestellt wird. [20]*
camelCase	*camelCase ist eine Notation bei der ein Name aus mehreren Wörtern gebildet wird. Die Wörter werden dabei direkt aneinander gereiht. Das erste Wort wird klein geschrieben und jedes neue Wort beginnt mit einem Grossbuchstaben. [11]*
Continuous Integration & Continuous Deployment (CI/CD)	*Es handelt sich hierbei um Konzepte mit denen alle Phasen der Anwendungsentwicklung vom programmieren bis hin zum Deployment automatisiert werden. [6]*
Clean Code	*Sauberer, einfach und intuitiv verständlicher Quelltext mit klarer Struktur und angewandten Konzepten. [12] (S34)*
Git	*Git ist eine freie Software zur verteilten Versionsverwaltung von Dateien. [22]*
Integrated Development Environment (IDE)	*Integrierte Entwicklungsumgebung, wird von einem Programmierenden für das Entwickeln von Software verwendet. [7]*
Lexer	*Ein Programm zum auftrennen von Quelltext in kleinstmögliche, zusammengehörige, distinkte Teile zur weiteren Verwendung durch einen Parser. [9]*
Linter	*Ein Programm zur Durchführung von Statischer Code Analyse (Siehe SCA). [24]*
Java	*Java ist eine 1995 entwickelte objektorientierte Programmiersprache. [8]*
JSON	*Die JavaScript Object Notation (JSON) ist ein standardisiertes, textbasiertes Format, um strukturierte Daten auf Basis eines JavaScript Objekts darzustellen. Es wird häufig für die Übertragung von Daten in Webanwendungen verwendet. [2]*

Begriff	**Definition**
Pipeline	*In der Softwareentwicklung ist eine Pipeline ein System automatisierter Prozesse, welche Aktualisierungen beim Code schnell von der Versionskontrolle in die Produktion übertragen. [6]*
Parser	*Ein Programm welches eine syntaktische Analyse auf der Rückgabe eines Lexers durchführt. Bringt die Informationen in eine hierarchische Struktur. [9]*
Refactoring	*Umarbeitung von Quelltext für verbesserte Verständlichkeit und Wartbarkeit ohne die Funktion zu ändern. [5]*
Static Code Analysis (SCA)	*Static Code Analyse, ist eine Methode zur Fehlersuche, bei der der Quelltext untersucht wird, bevor ein Programm ausgeführt wird. [27]*
Typescript	*TypeScript ist eine stark typisierte Programmiersprache, die auf JavaScript aufbaut und bessere Werkzeuge für die Anwendung bietet. [15]*
Unit Test	*Ein Test der ein einzelnes abgegrenztes Modul (Unit) der Software testet. [25]*
Visual Studio Code (VS Code)	*Eine in Typescript geschriebene IDE von Microsoft. [28]*
Wörterbuch	*Eine Wortliste, die in unserem Fall eine Liste mit englischen Wörtern inklusive ihrer Definition und ihrem Worttyp enthält.*

Anhang A

Projektplan

A.1 Zweck

Dieses Dokument zeigt auf, wie die Umsetzung des Clean Code Plugins zeitlich und inhaltlich geplant ist.

A.2 Gültigkeitsbereich

Dieses Dokument ist gültig für die Studienarbeit "CleanCode Plugin" im Herbstsemester 2021/2022 an der Fachhochschule OST Rapperswil-Jona. Es ist für die Betreuer und EntwicklerInnen des Projekts ausgelegt.

A.3 Projektübersicht

A.3.1 Problembeschrieb

Es gibt viele Empfehlungen wie gut geschriebener Quelltext aussieht. Viele EntwicklerInnen beachten einige oder auch alle diese Empfehlungen nicht wenn sie programmieren. Um EntwicklerInnen direkt beim schreiben von Quelltext auf Regelverletzungen hinzuweisen, soll ein IDE Plugin oder eine Erweiterung eines Linters entworfen werden. Dies erfordert eine genauere Analyse des Quelltextes, da viele dieser Regeln (zum Beispiel zur Namensgebung) nicht mit einer oberflächlichen statischen Codeanalyse überprüfbar sind.

A.3.2 Zweck und Ziel

ProgrammiererInnen sollen beim entwickeln von Quelltext durch Hinweise auf Regelverletzungen bezüglich Clean Code hingewiesen werden. Dadurch sollte qualitativ hochwertiger Quelltext entstehen und gleichzeitig lernen die EntwicklerInnen zukünftig besseren Quelltext zu schreiben.

Das Ziel ist ein Plugin welches die Arbeit der EntwicklerInnen durch sinnvolle Regelprüfungen und daraus resultierenden Hinweisen und Verbesserungsvorschlägen vereinfacht.

A.3.3 Lieferumfang

- Visual Studio Code Plugin
- Dokumentation

A.3.4 Annahmen und Einschränkungen

Die Idee des Projekts ist es, die bestehende Landschaft an Clean Code Plugins und Linter Definitionen zu erweitern. Insofern sind die Regeln, welche zur Umsetzung in Frage kommen dadurch eingeschränkt was bereits durch andere Software abgedeckt ist.

A.4 Projektorganisation

Das Projekt wird durch zwei Software-Engineering Studenten der Ostschweizer Fachhochschule am Campus Rapperswil-Jona durchgeführt. Beide Projektmitarbeiter studieren im berufsbegleitenden Studienmodell und arbeiten nebenher bis zu 60 Prozent. Betreut wird das Projekt durch Prof. Dr.-Ing. Frieder Loch.

A.4.1 Tools

- Microsoft Teams für die Kommunikation
- GitLab für Dokumentation und Quelltext
- Wiki für Notizen und sonstige Informationssammlung
- YouTrack für Issue Tracking und Zeiterfassung

A.4.2 Organisationsstruktur

- Rafael Fuhrer ist verantwortlich für Infrastruktur und CI/CD
- Pascal Schneider ist verantwortlich für Code Architektur und Qualität

A.4.3 Externe Schnittstellen

Wir werden unterstützt und betreut durch Prof. Dr.-Ing. Frieder Loch. Für die Evaluation des Projektergebnisses werden nicht am Projekt beteiligte Personen hinzugezogen.

A.5 Management Abläufe

A.5.1 Kostenvoranschlag

Unser Projekt wird auf 14 Semesterwochen aufgeteilt. Dies bedeutet ein durchschnittliches Arbeitspensum von ungefähr 17 Stunden pro Person pro Woche oder 240 Stunden in Total. Je nach anstehendem Arbeitsaufwand und vorhandener Zeit kann dieser Wert pro Woche variieren.

A.5.2 Zeitliche Planung

RUP Phasen	Inception	Elaboration			Construction								Transition	
Teilschritte aus Aufgabenbeschrieb	Literaturrecherche	Konzeption			Implementation							Evaluation		
Studiumswoche	**1** 20.09.-26.09.	**2** 27.09-03.10.	**3** 04.10.-10.10.	**4** 11.10.-17.10.	**5** 18.10.-24.10.	**6** 25.10.-31.10.	**7** 01.11.-07.11.	**8** 08.11.-14.11.	**9** 15.11.-21.11.	**10** 22.11.-28.11.	**11** 29.11.-05.12.	**12** 06.12.-12.12.	**13** 13.12.-19.12.	**14** 20.12.26.12.
Projektmanagement & Dokumentation	10	10	5	5	10	5	5	10	5	5	5	5	10	10
Recherche	10	10	10	5	5	5	3	2	3	2	3	2	0	0
Implementation	0	0	0	10	15	15	15	20	20	20	20	20	15	10
Testing	0	0	0	0	0	0	0	5	5	5	5	10	5	5
Infrastruktur	2	2	2	2	2	2	1	1	1	1	1	1	1	1
Reserve	6	6	6	6	6	6	6	6	6	6	6	6	6	6
Total	**28**	**28**	**23**	**28**	**38**	**33**	**30**	**44**	**40**	**39**	**40**	**44**	**37**	**32**
Milestones		M1			M2			M3				M4		M5

M1: Projektplan
M2: Prototyp
M3: Architektur
M4: Plugin
M5: Abgabe

A.5.3 Phasen

Inception

In der ersten Phase wird grob bestimmt auf welche Clean Code Regeln sich die Arbeit fokussieren soll.

Elaboration

Während der Elaboration wird detailliert beschrieben, welchen Umfang das Plugin haben soll und in welcher Form die Regeln geprüft werden sollen. Es entsteht ein Prototyp welcher aufzeigt, dass der Lösungsansatz technisch umsetzbar ist.

Construction

Während dieser Phase entwickeln wir das Clean Code Plugin und testen dieses fortlaufend.

Transition

In der letzten Phase wird das Plugin und die Dokumentation finalisiert. Das Plugin ist in der ersten Version zur Installation bereit. Überdies wird das Git Repository auf einen Stand gebracht auf welchem erweiternde Arbeiten am Plugin möglich sind.

A.5.4 Meilensteine

M1 Projektplan

- Literaturrecherche abgeschlossen
- Clean Code Regeln für Plugin identifiziert
- Zeitplanung gemacht
- Meilensteine definiert
- Git Repository eingerichtet
- Dokumentationsvorlage vorbereitet

M2 Prototyp

- Funktionale Anforderungen dokumentiert
- Nichtfunktionale Anforderungen dokumentiert
- Optionale Anforderungen dokumentiert
- Konzept des Systems zur Regelprüfung
- Konzept zur Notifizierung der verletzen Regeln
- Prototyp des Plugins

M3 Architektur

- Architektur des Plugins finalisiert
- Architektur dokumentiert

M4 Plugin

- Plugin fertig implementiert
- Eigene Tests abgeschlossen
- Vorhandene Bugs identifiziert

M5 Abgabe

- Dokumentation abgeschlossen
- Git Repositories in abgabefähigem Zustand
- Bugs behoben
- Rückmeldungen aus Evaluation eingearbeitet

A.5.5 Besprechungen

Als Fixpunkt gibt es eine wöchentliche Besprechung mit dem Betreuer. Diese ist auf Mittwoch um 17 Uhr terminiert. Des weiteren ist jeden Montag Abend von 19 bis 22 Uhr ein Block für Besprechungen und gemeinsame Arbeiten der Projektmitarbeiter reserviert. Alle weiteren Besprechungen die bilateral zwischen den beiden Projektmitarbeitern nötig sind werden nach Bedarf vereinbart.

A.6 Anforderungen

In diesem Abschnitt haben wir die Anforderungen an unser Projekt, aufgeteilt nach funktionalen und nicht-funktionalen Anforderungen, aufgeführt.

A.6.1 Funktionale Anforderungen

Wir haben uns aufgrund des einfachen Einsatzzweckes in Form eines Plugins dagegen entschieden Use Cases einzusetzen, da diese einen hohen Mehraufwand mit sich bringen würden. Wir setzen dafür auf einen schlanken Mix aus User Stories und MUSS/KANN Anforderungen. Diese sind in der Tabelle A.1 aufgelistet.

ProgrammiererIn

Als ProgrammiererIn möchte ich das Plugin in meiner IDE aktivieren können und alle problematischen Codestellen automatisch markiert bekommen.

Tabelle A.1: Funktionale Anforderungen

Anforderungs-ID	Kategorie	Priorität
FA-1	MUSS	Das Plugin soll beim Speichern den geöffneten Quelltext scannen und alle Regelverstösse grafisch hervorheben.
FA-2	KANN	Alle Regelverstösse sollen im Problem Log, inklusive ihrer Position in der Datei, in der IDE ausgegeben werden.
FA-3	KANN	Als ProgrammiererIn kann ich das Plugin bequem über den Marketplace meiner IDE installieren (keine manuelle Installation nötig).

A.6.2 Nicht Funktionale Anforderungen

Die Nicht Funktionalen Anforderungen, aufgeteilt in MUSS und KANN Anforderungen, werden in der Tabelle A.2 aufgelistet.

Tabelle A.2: Nicht Funktionale Anforderungen

Anforderungs-ID	Kategorie	Priorität
NF-1	KANN	Die Prüfung der Regeln durch das Plugin soll nicht mehr als 5s Zeit pro Durchlauf in Anspruch nehmen.
NF-2	MUSS	Das Plugin funktioniert auch bei grossen Dateien (>1000 Zeilen Code) noch wie vorgesehen.
NF-3	MUSS	Das Plugin kann von der Architektur her einfach um weitere Clean Code Regeln erweitert werden.
NF-4	MUSS	Die Code Qualität gemäss SonarQube Scans ist mit der Gesamtnote A gegeben.
NF-5	KANN	Das Projekt erreicht eine Unit Test Abdeckung von mindestens 80%.
NF-6	KANN	EntwicklerInnen die mit dem Projekt nicht vertraut sind, sollen bei einem einfachen Problem die betroffene Codestelle innerhalb von 30 Minuten finden können.

A.7 Auswertung

In diesem Abschnitt möchten wir die Arbeit rückblickend auswerten. Wir fokussieren uns hier aber nur auf die für das Projektmanagement relevanten Punkte. Die Ergebnisse der Auswertung basieren auf der Auswertung des Berichts (inklusive Anhang), der Architekturdokumentation und auf der durchgeführten Nutzertestung.

A.7.1 Auswertung der Anforderungen

Zu Beginn der Arbeit haben wir diverse funktionale und nicht funktionale Anforderungen definiert. Die Auswertung der Funktionalen Anforderungen haben wir in der Tabelle A.3 dargestellt.

Tabelle A.3: Auswertung funktionale Anforderungen

Anforderungs-ID	Kategorie	Erfüllt?	Begründung
FA-1	MUSS	Ja	Siehe T01 im Testbericht (Anhang)
FA-2	KANN	Ja	Siehe T02 im Testbericht (Anhang)
FA-3	KANN	Nein	Siehe Architekturdokumentation (Anhang)

Wir haben unser MUSS-Kriterium, sowie ein weiteres KANN-Kriterium erfüllt. Einzig die optionale Anforderung zur Veröffentlichung im Marketplace haben wir nicht erfüllen können, da das Plugin noch nicht bereit für den grossflächigen Einsatz ist.

Die Auswertung der nicht funktionalen Anforderungen haben wir in der Tabelle A.4 dargestellt.

Alle Nicht Funktionalen Anforderungen ausser der optionalen Anforderung an die Testabdeckung wurden erfüllt. Die Testabdeckung konnten wir während des Projektes, wegen Problemen mit dem Testframework Mocha nicht zum laufen bekommen. Daher konnten wir keine exakte Auswertung vornehmen und werten das Ziel daher als nicht erfüllt.

Tabelle A.4: Nicht Funktionale Anforderungen

Anforderungs-ID	Kategorie	Erfüllt?	Begründung
NF-1	KANN	Ja	Siehe T03 im Testbericht (Anhang)
NF-2	MUSS	Ja	Siehe T04 im Testbericht (Anhang)
NF-3	MUSS	Ja	Siehe Architekturdokumentation (Anhang)
NF-4	MUSS	Ja	Siehe SonarQube Analyse (Anhang)
NF-5	KANN	Nein	Siehe Bericht (Kapitel 5.1)
NF-6	KANN	Ja	Siehe T05 im Testbericht (Anhang)

A.7.2 Auswertung des Zeitaufwands

Hier blicken wir nochmal auf die für das Projekt aufgewendete Zeit zurück. In Summe wurden pro Projektmitarbeiter ∼240 Arbeitsstunden, insgesamt also ∼480 Arbeitsstunden aufgewendet.

Zu Beginn des Projekts haben wir geschätzt, wie viel Zeit wir pro Kategorie und Woche ca. aufwenden werden. Auf diese Schätzung möchten wir an diesem Punkt nochmal zurück kommen und die Schätzung mit der Realität vergleichen. In der Tabelle A.5 haben wir die aufgewendete Zeit über das gesamte Projekt nach Kategorie aufgeführt. Dabei ist zu beachten, dass sämtliche abgehaltenen Besprechungen, inklusive der wöchentlichen Besprechung mit dem Betreuer in der Kategorie Projektmanagement & Dokumentation enthalten sind.

Tabelle A.5: Analyse Zeitaufwand

Kategorie	Zeitaufwand geschätzt (in Std.)	Zeitaufwand real (in Std.)
Entwicklung	180	197
Projektmanagement & Dokumentation	100	184
Infrastruktur	20	18
Recherche	60	51
Testing	40	33
Reserve	84	0
Summe	484	483

Wie zu sehen ist war unsere Schätzung über den gesamten Zeitaufwand überraschend genau. Bei den einzelnen Kategorien zeigen sich aber viele Abweichungen. Wichtig ist zu beachten, dass wir die Reserve bewusst gross definiert hatten um flexibel auf Verzögerungen reagieren zu können. Entsprechend waren grössere Abweichungen nach oben bei gewissen Kategorien zu erwarten. In der Auswertung kann man sehen, dass wir knapp 200 Arbeitsstunden, bei geplanten 180 Stunden, in die Entwicklung des Plugins investiert haben. Bei der Kategorie Projektmanagement &

Dokumentation hatten wir 100 Stunden vorgesehen, haben aber 184 Stunden aufwenden müssen. Hier hatten wir vor allem unterschätzt, wie viel Zeit die Meetings und das schreiben des Berichts in Anspruch nehmen, daher die grosse Abweichung. Den Zeitaufwand für die Infrastruktur konnten wir sehr genau abschätzen. Dies dürfte an unseren Erfahrungen im Engineering Projekt gelegen haben. Ausserdem sind bei der Infrastruktur keine unerwarteten Probleme aufgetaucht. Bei der Recherche haben wir etwas weniger Zeit als wir geschätzt haben aufgewendet. Dies dürfte unter anderem daran gelegen haben, dass wir alle relevanten Informationen, schön strukturiert aus einer einzelnen Buchquelle recherchieren konnten. Das Testing hat ebenfalls etwas weniger Zeit in Anspruch genommen, als wir geschätzt haben. Grund dafür war unserer Meinung nach, dass nur sehr wenige Probleme aufgetreten und die meisten Tests von Anfang an funktioniert haben. Die Reserve war sehr gut abgeschätzt und wurde fast komplett aufgebraucht, um unsere Fehleinschätzung beim Projektmanagement und der Dokumentation zu kompensieren.

Anhang B

Testprotokoll

Das folgende Testprotokoll wurde beim durchgeführten Nutzertest erstellt.

B.1 Methodik

Die Tests wurden von unserem Kommilitonen Christoph Scheiwiller, der Teilzeit als Java Entwickler arbeitet, durchgeführt. Die Tests wurden auf seinem Desktop Computer mit einem Intel Core i7–4790K Prozessor, einer NVidia GeForce GTX 970 Grafikkarte und 16GB RAM durchgeführt. Das verwendete Betriebssystem war Windows 10 mit dem aktuellen Patchstand per 19.12.2021. Für die Tests wurden Code Dateien aus Christophs geschäftlichem Alltag verwendet, um möglichst praxisnahe Beispiele verwenden zu können. Diese Dateien können aber aus Compliance gründen nicht dem Code Repository hinzugefügt werden. Die Durchführung der Tests wurden wegen der Pandemiesituation via Teams begleitet und in diesem Testprotokoll protokolliert.

B.2 Durchführung der Tests

In den folgenden Tabellen haben wir das bei der Nutzertestung erstellte Testprotokoll abgebildet.

Tabelle B.1: Test 1: Ausführen des Plugins

Test Nr.	T01
Test Bezeichnung	Grundfunktionalität
Erwartetes Ergebnis	Visual Studio Code lässt sich mit installiertem & aktiviertem Plugin öffnen. Die Ausführung des *check code* Kommandos markiert Verstösse gegen die Clean Code Prinzipien.
Tatsächliches Ergebnis	Visual Studio Code lässt sich mit installiertem Plugin ausführen. Mit dem *check code* Kommando werden die Verstösse gegen die Clean Code Prinzipien erkannt und markiert.
Kommentar	Prüft FA-1 der im Projektplan spezifizierten funktionalen Anforderungen.
Test Ergebnis	Bestanden

Tabelle B.2: Test 2: Problem Log

Test Nr.	T02
Test Bezeichnung	Hervorhebung von Verstössen im Problem Log
Erwartetes Ergebnis	Alle Verstösse werden mit Angabe der Position im Problem Log angezeigt.
Tatsächliches Ergebnis	Alle Verstösse werden mit Angabe der Position im Problem Log angezeigt.
Kommentar	Prüft FA-2 der im Projektplan spezifizierten funktionalen Anforderungen.
Test Ergebnis	Bestanden

Tabelle B.3: Test 3: Prüfzeit Allgemein

Test Nr.	T03
Test Bezeichnung	Prüfzeit
Erwartetes Ergebnis	Bei der Ausführung der Regelprüfung soll die Zeit bis die Verstösse angezeigt werden, weniger als 5 Sekunden betragen.
Tatsächliches Ergebnis	Bei der Ausführung vergeht weniger als eine Sekunde bis die Ergebnisse angezeigt werden.
Kommentar	Prüft NF-1 der im Projektplan spezifizierten nicht funktionalen Anforderungen.
Test Ergebnis	Bestanden

Tabelle B.4: Test 4: Prüfzeit bei Grossen Dateien

Test Nr.	T04
Test Bezeichnung	Prüfzeit bei Grossen Dateien
Erwartetes Ergebnis	Die Prüfzeit bei Quelltext mit mehr als 1000 Zeilen beträgt weniger als 5 Sekunden.
Tatsächliches Ergebnis	Die Prüfzeit bei einem Quelltext mit mehr als 1000 Zeilen beträgt weniger als 1 Sekunde.
Kommentar	Prüft NF-2 der im Projektplan spezifizierten nicht funktionalen Anforderungen.
Test Ergebnis	Bestanden

Anhang C

Aufgabenbeschreibung

Die Aufgabenbeschreibung der Hochschule in ihrer originalen Form.

Plugin zum Erlernen von Clean Code

Beteiligte Personen

Diese Arbeit wird verfasst von Pascal Schneider; pascal.schneider@ost.ch Rafael Fuhrer; rafael.fuhrer@ost.ch

Betreuer dieser Arbeit ist Prof. Dr.-Ing. Frieder Loch; frieder.loch@ost.ch

Problembeschrieb

Es existieren zahlreiche Empfehlung, wie guter und verständlicher Programmcode verfasst werden kann. Diese werden oft unter dem Begriff Clean Code zusammengefasst, der von Martin Fowler geprägt wurde. Diese Empfehlungen betreffen, zum Beispiel, aussagekräftige Variablennamen zu verwenden oder Funktionen anhand eines konsistenten Schemas zu benennen. Diese Regeln werden in der Praxis allerdings nicht immer befolgt.

Ein Ansatz dem entgegen zu treten ist es, Programmierende direkt beim Verletzen dieser Regeln auf diese hinzuweisen. Dies kann in Form eines Plugins für eine Entwicklungsumgebung (z.B. VSCode) oder für einen Linter (z.B. ESLint) geschehen. Linter sind in der Lage die Einhaltung bestimmter Richtlinien zu prüfen. Für viele Regeln, beispielsweise das Einhalten eines konsistenten Benennungsschemas, ist allerdings eine tiefergehende Analyse des Programmcodes notwendig. So müssen Bezeichner vom System automatisch in ihre relevanten semantischen Bestandteile zerlegt werden (z.B. isActive à is active).

Formulierung des Konkreten Auftrags

Diese Studienarbeit soll relevante Clean Code-Regeln sammeln und ein System entwerfen und implementieren, deren Einhaltung zu prüfen. Es sollen verbreitete objektorientierte Sprachen (z.B. JAVA, TypeScript) betrachtet werden.

Hierzu sollen folgende Teilschritte durchgeführt werden:

- **Literaturrecherche** zur Identifikation von Clean-Code Regeln. Es werden relevante wissenschaftliche Literatur und hochwertige Onlinequellen herangezogen. Die Regeln werden strukturiert gesammelt. Es wird abgeglichen, welche Regeln mit bestehenden Tools bereits geprüft werden können. Es wird festgelegt, welche Regeln in der Arbeit adressiert werden.
- **Konzeption** eines Systems, mit dem die ausgewählten Regeln geprüft werden können. Weiterhin wird konzipiert, wie die Kommunikation der Ergebnisse der Analyse an den Benutzenden erfolgen soll. Das System beschreibt, was für ein Fehler erkannt wurde und erklären wie dieser behoben werden kann. Es wird entschieden wie das System implementiert wird.
- Das konzipierte System wird **implementiert**. Die Implementierung wird in einem git-Repository durchgeführt und dokumentiert.

- **Evaluation** des Projektergebnisses in der praktischen Anwendung. Nach Möglichkeit sollen nicht am Projekt beteiligte Personen herangezogen werden und die Evaluation in einem möglichst realistischen Anwendungsszenario erfolgen. Es sollen Verbesserungsvorschläge anhand der Ergebnisse beschrieben werden.

Umfang und Form der erwarteten Resultate

Die Ergebnisse der Arbeit (Software und Dokumentation) sollen als Open Source-Projekt publiziert werden. So wird es ermöglicht, dass die Ergebnisse ohne Einschränkung von der OST und anderen Parteien weiterverwendet werden können. Bei der Veröffentlichung der Ergebnisse und den verwendeten Libraries werden Permissive-Licenses (z.B. MIT) bevorzugt.

Anfangs- und Abgabetermin

- Start der Bearbeitung: **20. September 2021**
- Abgabe der Arbeit: **24. Dezember 2021 (17:00 Uhr)**

Zulässige Hilfsmittel und weitere Betreuung

Alle verwendeten Hilfsmittel werden in der Arbeit aufgeführt. Die Betreuung erfolgt durch die genannte Betreuungsperson.

Anhang D

SonarQube Auswertung

Im folgenden Abschnitt soll die Code Qualität unserer Visual Studio Code Extension anhand einer Statischen Code Analyse durch SonarQube aufgezeigt werden.

D.1 Allgemeine Code Qualität

In der Nicht Fuktionalen Anforderung (NF-4) hatten wir definiert, dass unser Quelltext eine Gesamtnote von A erreichen muss. In Bild D.1 ist zu sehen, dass wir in allen Kategorien diese Note erreichen.

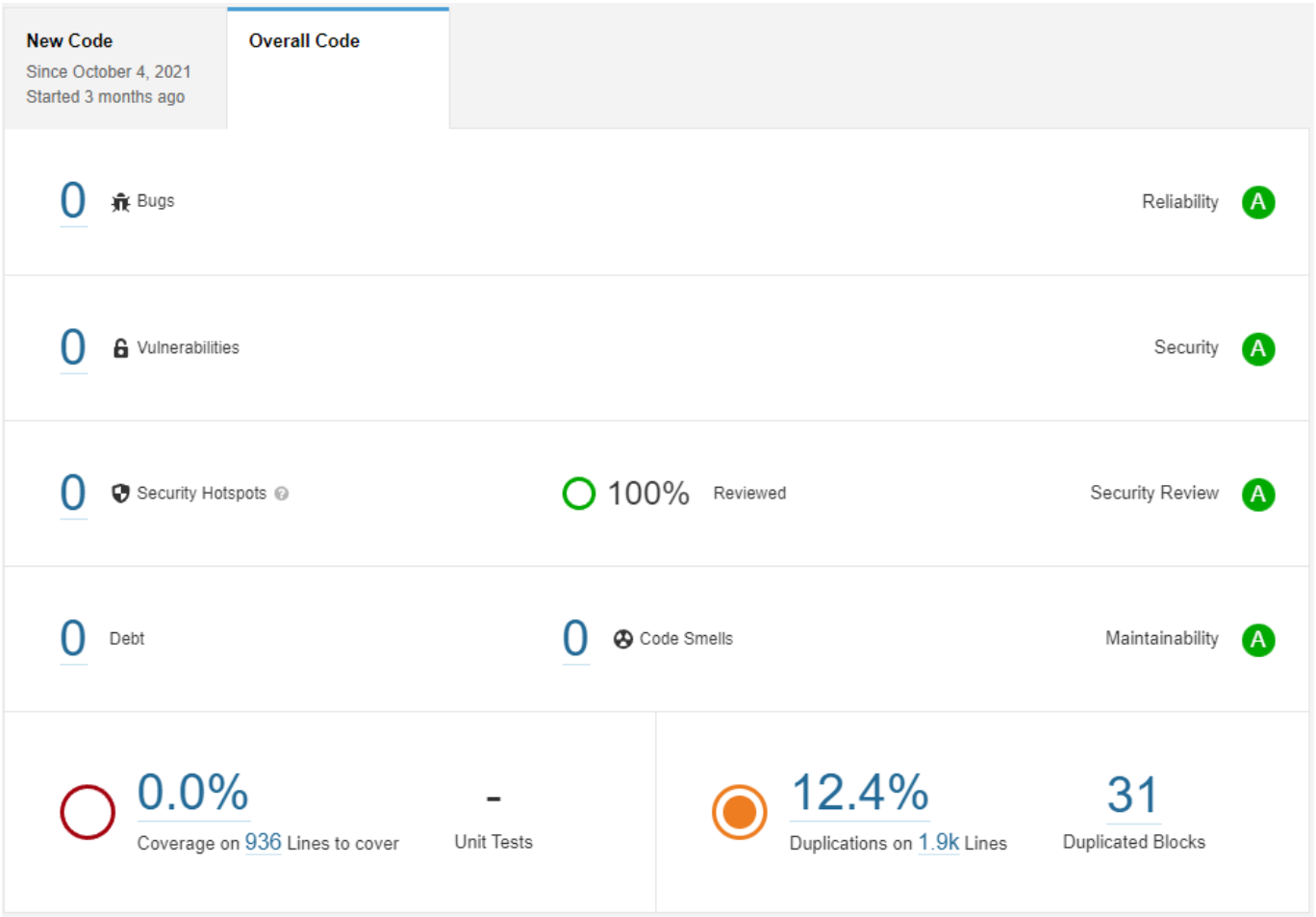


Abbildung D.1: Übersicht Statische Code Analyse in SonarQube

Wir haben sogar erreicht, dass wir in keiner Kategorie offene Punkte haben.

D.2 Technical Debt

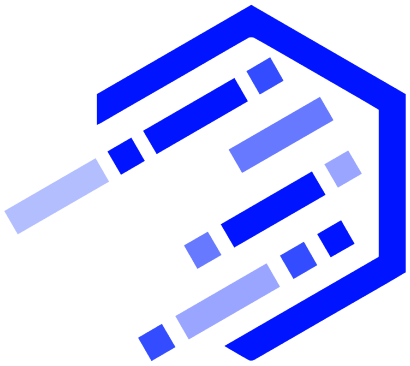
Daraus, dass wir keine offenen Probleme im Quelltext haben, ergibt sich dann auch, dass wir keine "Technical Debt" auf zukünftige Entwickler abwälzen. In Bild D.2 kann man dies ablesen.

Abbildung D.2: Übersicht Statische Code Analyse in SonarQube

Ebenfalls in Bild D.2 wird eine Coverage von 0% ausgewiesen. Diese stammt daher, dass wir unsere Unit Tests zwar automatisiert als Teil der Pipeline ausführen, aber kein Tool zur Auswertung der Code Coverage dieser Tests integrieren konnten.

Anhang E

Architekturdokumentation



CLEANCODE

Clean Code Plugin Architekturdokumentation

Rafael Fuhrer, Pascal Schneider

2021-11-30

Über arc42

arc42, das Template zur Dokumentation von Software- und Systemarchitekturen.

Erstellt von Dr. Gernot Starke, Dr. Peter Hruschka und Mitwirkenden.

Template Revision: 7.0 DE (asciidoc-based), January 2017

Einführung und Ziele

Diese Sektion beschreibt die wesentlichen Anforderungen und treibenden Kräfte, die bei der Umsetzung der Softwarearchitektur und Entwicklung des Systems berücksichtigt werden müssen.

Aufgabenstellung

Dieses Clean Code Plugin ist im Zusammenhang mit einer Studienarbeit die an der Fachhochschule OST durchgeführt wurde entstanden. Nachfolgend der komplette Aufgabenbeschrieb in seiner originalen Fassung.

Problembeschrieb

Es existieren zahlreiche Empfehlung, wie guter und verständlicher Programmcode verfasst werden kann. Diese werden oft unter dem Begriff Clean Code zusammengefasst, der von Martin Fowler geprägt wurde. Diese Empfehlungen betreffen, zum Beispiel, aussagekräftige Variablennamen zu verwenden oder Funktionen anhand eines konsistenten Schemas zu benennen. Diese Regeln werden in der Praxis allerdings nicht immer befolgt. Ein Ansatz dem entgegen zu treten ist es, Programmierende direkt beim Verletzen dieser Regeln auf diese hinzuweisen. Dies kann in Form eines Plugins für eine Entwicklungsumgebung (z.B. VSCode) oder für einen Linter (z.B. ESLint) geschehen. Linter sind in der Lage die Einhaltung bestimmter Richtlinien zu prüfen. Für viele Regeln, beispielsweise das Einhalten eines konsistenten Benennungsschemas, ist allerdings eine tiefergehende Analyse des Programmcodes notwendig. So müssen Bezeichner vom System automatisch in ihre relevanten semantischen Bestandteile zerlegt werden (z.B. `"isActive"` zu `"is"` und `"active"`)

Anforderungen

Wir haben uns aufgrund der Einfachheit der Anwendungsfälle gegen den Einsatz von Use Cases entschieden, da diese einen hohen Mehraufwand mit sich bringen. Wir setzen dafür auf einen schlanken Mix aus User Stories und MUSS/KANN Anforderungen.

ProgrammiererIn

Als ProgrammiererIn möchte ich das Plugin in meiner IDE aktivieren können und alle problematischen Codestellen automatisch markiert bekommen.

Table 1: Funktionale Anforderungen

Anforderungs-ID	**Kategorie**	**Priorität**
FA-1	MUSS	Das Plugin soll beim Speichern die geöffnete Datei scannen und alle Regelverstösse grafisch hervorheben.
FA-2	KANN	Alle Regelverstösse sollen im Problem Log, inklusive ihrer Position in der Datei, in der IDE ausgegeben werden.
FA-3	KANN	Als ProgrammiererIn kann ich das Plugin bequem über den Marketplace meiner IDE installieren (keine manuelle Installation nötig).

Qualitätsziele

Wir haben die Qualitätsziele in Form von Nicht-Funktionalen Anforderungen erfasst, aufgeteilt in MUSS und KANN Anforderungen, welche in Tabelle 2 aufgelistet sind.

Stakeholder

Die Stakeholder werden in der folgenden Tabelle mit Rollen- oder Personennamen, sowie deren Erwartungshaltung bezüglich der Architektur und der Dokumentation festgehalten.

Rolle	Kontakt	Erwartungshaltung
Betreuer	*Prof. Frieder Loch*	*Wissensgewinn, Gute Umsetzung auf welcher weitere Projekte aufsetzen können*
Projektmitarbeiter	*Rafael Fuhrer* *Pascal Schneider*	*Gutes Resultat, Erlernen neuer Technologien, Saubere Umsetzung und Dokumentation*

Table 2: Nicht Funktionale Anforderungen

Anforderungs-ID	Kategorie	Priorität
NF-1	KANN	Die Prüfung der Regeln durch das Plugin soll nicht mehr als 5s Zeit pro Durchlauf in Anspruch nehmen.
NF-2	MUSS	Das Plugin funktioniert auch bei grossen Dateien (>1000 Zeilen) wie vorgesehen.
NF-3	MUSS	Das Plugin kann einfach um weitere Clean Code Regeln erweitert werden.
NF-4	MUSS	Die Code Qualität gemäss SonarQube Scans ist mit der Gesamtnote A gegeben.
NF-5	KANN	Das Projekt erreicht eine Unit-Test Abdeckung von mindestens 80%.
NF-6	KANN	EntwicklerInnen die mit dem Projekt nicht vertraut sind, sollen bei einem einfachen Problem die betroffene Codestelle innerhalb von 30 Minuten finden können.

Randbedingungen

Wir haben uns, aufgrund der bewusst offen formulierten Aufgabenstellung, die Randbedingungen in Tabelle 4 gegeben.

Kontextabgrenzung

Die Kontextbeziehungen haben wir im Bild 1 abgebildet.

Eine Erläuterung zur Beziehung zwischen den verschiedenen Kontextelementen, sowie deren Eigenschaften finden sich hier.

Table 4: Randbedingungen

Randbedingungs-ID	**Bezeichnung**	**Beschreibung**
RB-1	Versionierung	Einsatz von semantic Versioning bei der Plugin Versionierung
RB-2	Linting	Einsatz eines Linters wie z.B. ESLint
RB-3	Dokumentation	Die gesamte Dokumentation des Plugins erfolgt nach einer gängigen Architekturdokumentationsvorlage

Table 5: Kontextbeziehungen

Kontextelement	**Format**	**Kommunikation via**
Wordlist	JSON-Datei	Direkter Dateizugriff via Clean Code Plugin
Clean Code Plugin	Typescript	Zugriff auf den Quelltext via interner Visual Studio Code API
Code Editor	Typescript	Stellt Interface für ProgrammiererIn zur Verfügung

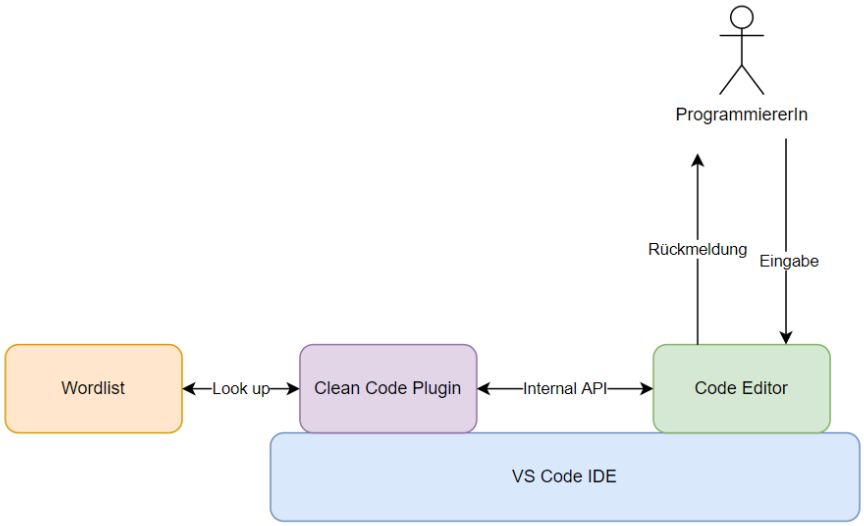


Figure 1: Kontextdiagramm

Lösungsstrategien

Programmiersprache

Da wir uns für die Umsetzung der Aufgabenstellung in Form eines Plugins in der Visual Studio Code IDE entschieden haben, kamen für die Umsetzung nur Javascript und Typescript in Frage. In der Folge haben wir uns unter anderem aufgrund der gebotenen Typesicherheit für Typescript entschieden.

Clientseitiges Plugin vs. Language Server

Da in der ersten Version des Plugins die Regelprüfungen im Vordergrund stehen und wir uns in der Prüfung auf Java beschränken, haben wir uns für die Implementation in einem rein clientseitigen Plugin und gegen einen Language Server entschieden. Der grösste Vorteil eines Language Servers würde darin liegen, dass er die N * M Problematik (Unterstützung von N Spracherweiterungen für M Editoren) stark vereinfacht, indem die Spracherweiterung nur einmal implementiert und aus verschiedenen Editoren abgefragt werden kann.[1]

Wortliste

Wir brauchen für die geplante Umsetzung von Regeln, welche die Benennung von Variablen und Funktionen prüfen, ein englisches Wörterbuch (Dictionary). Dieses kann entweder extern gehostet und vom Plugin via API abgefragt werden, oder direkt mit dem Plugin (als File) ausgeliefert werden. Wir haben uns, aufgrund von Privacy und Compliance Gründen, für die direkte Auslieferung im Fileformat entschieden. Wenn wir das Wörterbuch extern gehostet hätten, wäre es möglich gewesen die anfallenden Daten zu sammeln und auszuwerten. Die dabei weiter anfallenden und theoretisch auswertbaren Metadaten, wie z.B. der Zeitpunkt wann jemand programmiert und wann nicht, fallen unter den Datenschutz. Im Zusammenhang mit kommerziellen Programmierarbeiten, wo der geschriebene Quelltext meistens als vertraulich einzustufen ist und nicht extern geteilt werden darf, hätte dieser Lösungsansatz ebenfalls gegen Vorgaben verstossen, was den Einsatz des Plugins in diesen Umgebungen verunmöglicht hätte. Das finale Wörterbuch wurde von uns selbst aus öffentlich zugänglichen Informationen die unter CC-0 Lizenz stehen zusammengestellt und beinhaltet fast hunderttausend der meistgenutzten englischen Wörter inklusive ihrer Definition(en) und Worttyp Informationen.

Benachrichtigung der ProgrammiererIn

Die Rückmeldung an die ProgrammiererIn besteht aus einer Hinweismeldung, die den Fehler beschreibt und Informationen an welcher Stelle im Quelltext sich das Problem befindet. Diese Meldung wird sowohl im “Problem”-Fenster von Visual Studio Code, als auch durch farbige Unterstreichung im Quelltext

[1] https://code.visualstudio.com/api/language-extensions/language-server-extension-guide

Fenster angezeigt. Dazu wird die von der Visual Studio Code Extension API zur Verfügung gestellte DiagnosticCollection verwendet.

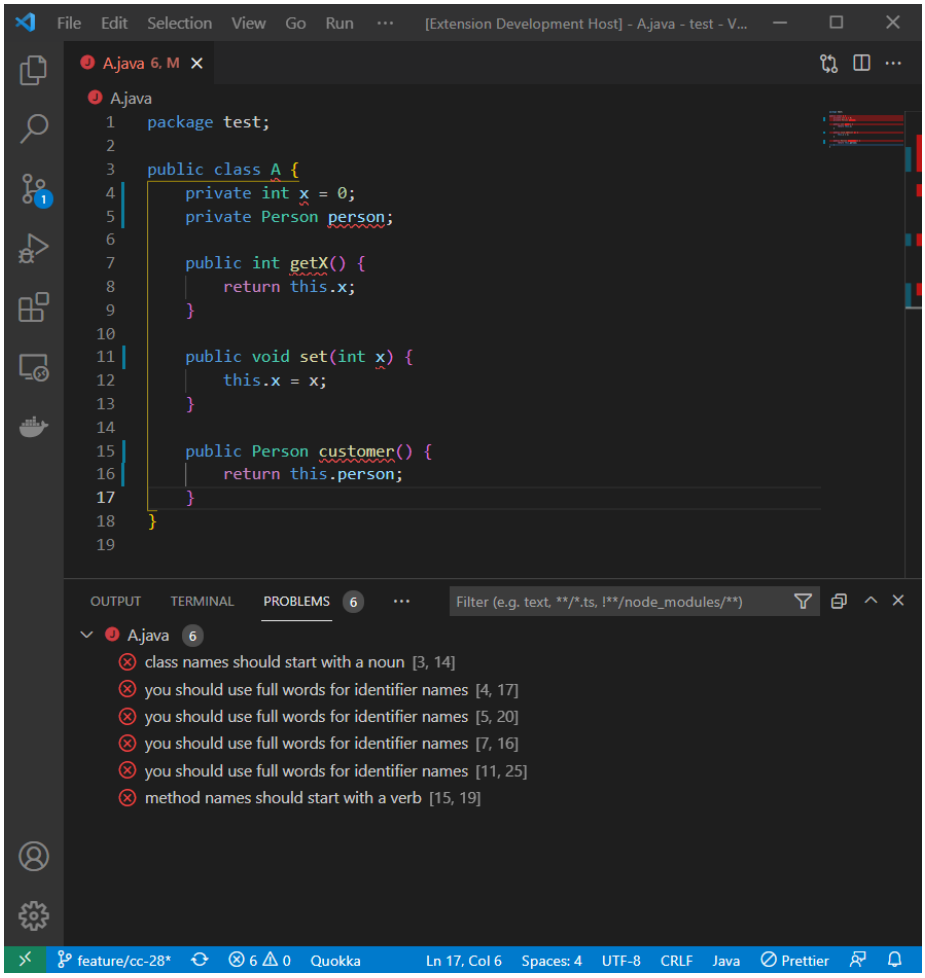


Figure 2: Beispiel Benachrichtigung

Auslösung der Code Prüfung

Die Prüfung des Quelltextes erfolgt je nach Benutzereinstellung als Folge eines von drei Auslösern. Die erste Möglichkeit ist, dass die Prüfung nur durch den manuellen Aufruf des Befehls “check code” erfolgt. Die zweite Möglichkeit ist, dass die Prüfung nach jedem Speichervorgang automatisch ausgelöst wird. Hierzu wird eine Funktion an das `onDidSaveTextDocument` Event der Visual Studio Code Extension API angehängt. Die dritte mögliche Einstellung bewirkt, dass die Prüfung automatisch nach jeder Änderung des Quelltextes ausgeführt wird.

Das Event welches hier genutzt wird ist `onDidChangeTextDocument`.

Bausteinsicht

Das Plugin ist in mehrere Untersysteme gegliedert, welche durch eine Hauptklasse nacheinander angestossen werden. Dies mit dem Hintergrund, dass einzelne Unterschritte unabhängig vom Rest der Applikation überarbeitet oder bei Bedarf später ausgetauscht werden können.

Gesamtsystem

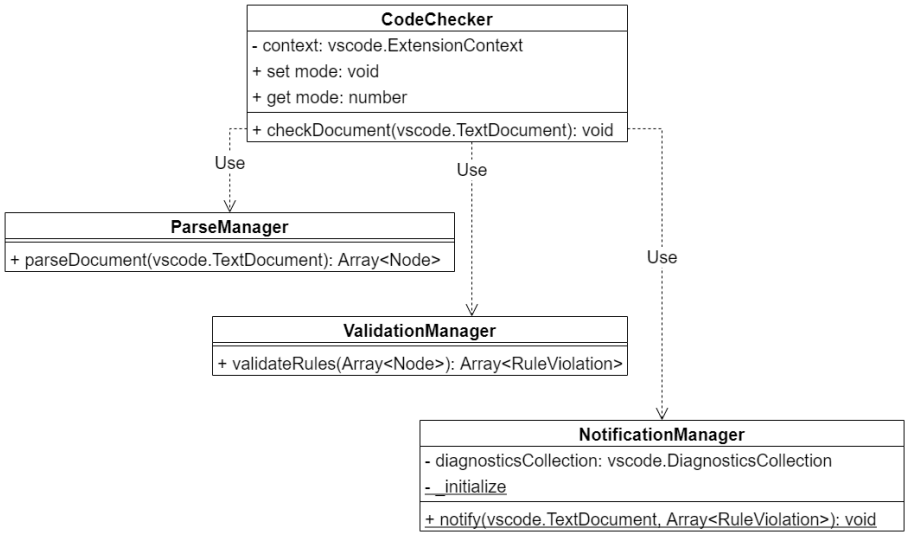


Figure 3: Bausteinsicht Gesamtsystem

Beschreibung Die Klasse `CodeChecker` ist die Einstiegsklasse, welche von Visual Studio Code angesprochen wird, wenn die Prüflogik des Plugins ausgeführt werden soll. Des weiteren setzt und liest die Klasse die Benutzereinstellungen des Plugins.

Enthaltene Bausteine Intern stösst der `CodeChecker` die drei Untersysteme zum Parsen des Quelltextes, Validieren der Regeln und Benachrichtigen der ProgrammiererIn an.

Wichtige Schnittstellen Die Methode `checkDocument` nimmt ein Dokument entgegen und prüft ob dieses die implementierten Clean Code Regeln einhält. Das Property `mode` liest und schreibt die Benutzereinstellung, die steuert zu welchem Zeitpunkt die Prüfung ausgeführt werden soll.

Wichtige Konzepte Sowohl für Objekte welche sprachabhängige Implementa-

tionen benötigen (z.B. Parser, Lexer), als auch für Objekte die in unterschiedlicher Ausführung implementiert werden (z.B. Regeln, Validatoren), werden Interfaces definiert. Zum einen damit die aufrufenden Klassen keine Implementationsdetails kennen müssen und zum anderen damit zur Laufzeit situativ unterschiedliche Implementationen genutzt werden können.

Klassen, welche eine Grundfunktionalität zur Verfügung stellen aber für einzelne Einsatzzwecke durch Vererbung erweitert werden sollen, markieren wir als abstrakt. Ein Beispiel hierzu ist die `Node`-Klasse, welche die Funtionalität zur Standortbestimmung innerhalb des Quelltextes implementiert – alle spezifische Logik der unterschiedliche Arten von Nodes allerding in den abgeleiteten Klassen liegt.

Um private Properties ausserhalb einer Klasse verfügbar zu machen benutzen wir keine Methoden sondern die get/set Funktionalität von Typescript. Beispiel:

```
class Token {
  private _range: vscode.Range;

  get range(): vscode.Range {
    return this._range;
  }

  set range(range: vscode.Range) {
    this._range = range;
  }
}

...
var token = new Token();
token.range = new Range(1, 1);
...
```

ParseManager

Zweck Der `ParseManager` nimmt einen Quelltext entgegen und gibt eine Liste aller `Nodes` zurück, welche zur Regelprüfung relevant sind.

Schnittstellen Die Methode `parseDocument` ist die öffentliche Schnittstelle dieses Untersystems.

ValidationManager

Zweck Der `ValidationManager` nimmt die Liste der `Nodes` entgegen und prüft diese gegen die definierten Regeln. Er sammelt die verletzten Regeln und gibt nach der Prüfung eine Liste von Regelverletzungen zurück.

Schnittstellen Die Methode `validateRules` ist die öffentliche Schnittstelle dieses Untersystems.

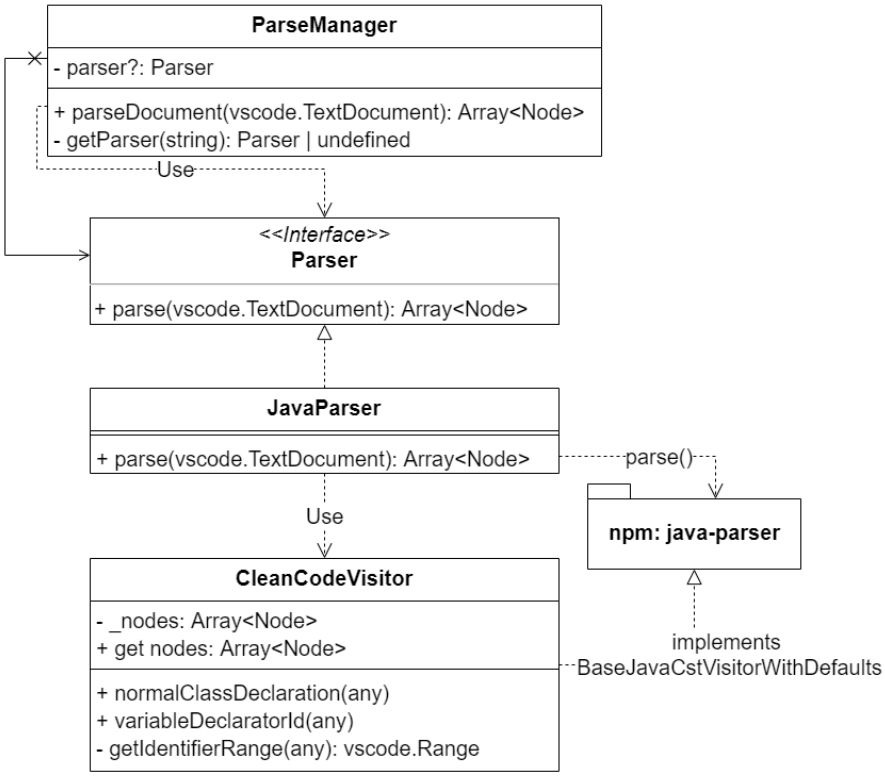
NotificationManager

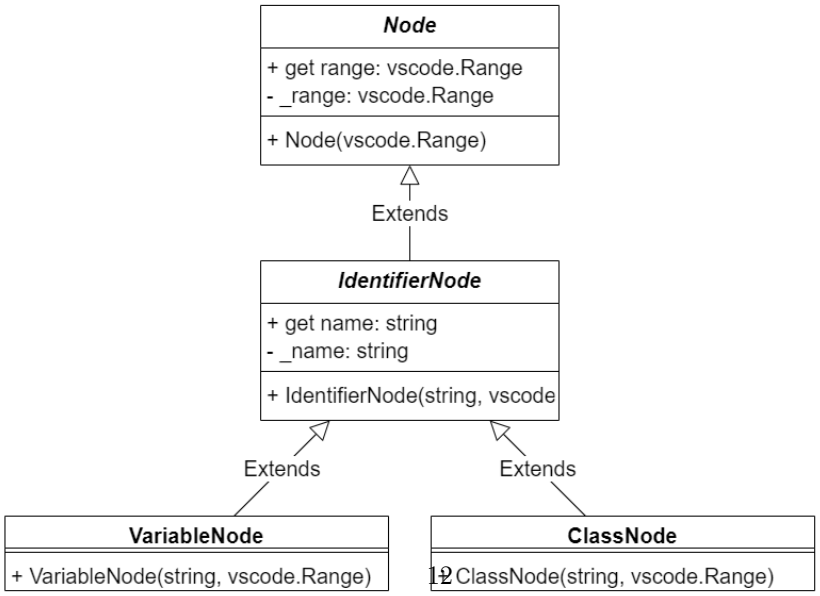
Zweck Der `NotificationManager` ist zuständig für die Benachrichtigung der ProgrammiererIn. Er nimmt die Liste von Regelverletzungen entgegen und zeigt diese der ProgrammiererIn visuell an.

Schnittstellen Die Methode `notify` ist die öffentliche Schnittstelle dieses Untersystems.

Detailsicht der einzelnen Teilsysteme

Parsing





Beschreibung Die Logik zum parsen des Quelltextes wird von der Klasse `ParseManager` verwaltet. Dieser wählt den für die aktuelle Programmiersprache korrekten `Parser`, welcher den Text entgegen nimmt und die benötigten `Nodes` ausliest.

Das Interface `Parser` definiert welche Schnittstelle ein sprachspezifischer Parser zur Verfügung stellen muss. Die tatsächliche Implementation des Parsers kann variieren. Steht für die Programmiersprache eine gute Parser-Implementation zur Verfügung, kann diese eingebunden werden. Die Rückgabe muss dann nur noch auf die vom Plugin verwendeten `Nodes` gemappt werden. Besteht noch kein passender Parser, kann dieser unter Zuhilfenahme des bestehenden Lexer-Interface und den zugehörigen Hilfsklassen selbst implementiert werden.

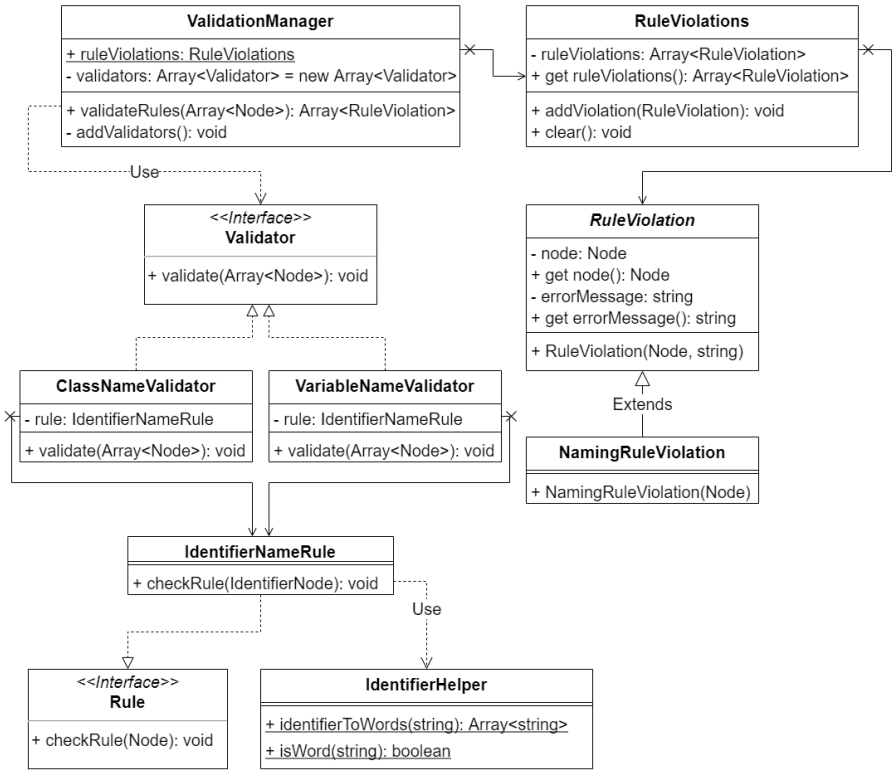
Enthaltene Bausteine Der `JavaParser` ist die Implementation des Parser-Interface für die Programmiersprache Java. Er benutzt das npm Paket java-parser, welches einen konkreten Syntaxbaum des gesamten Quelltextes zurück gibt. Dieser Baum wird mit dem `CleanCodeVisitor` traversiert und alle benötigten `Nodes` werden gesammelt und danach an den `ParseManager` zurückgegeben.

Spezifische `Nodes`, welche alle benötigten Informationen für die implementierten Regelprüfungen enthalten.

Wichtige Schnittstellen Die Methode `parseDocument` nimmt ein Dokument entgegen und gibt eine Liste aller regelrelevanten `Nodes` zurück.

Wichtige Konzepte Beim traversieren von Syntaxbäumen, welche durch die verwendeten parsing-Bibliotheken zurückgegeben werden, wird das Visitor Pattern eingesetzt. Dieses Pattern ermöglicht es alle Blätter eines Baumes zu besuchen und auf gewissen dieser Blätter Operationen auszuführen, ohne dass die eigentliche Baumklassen angepasst werden muss.

Validation



Beschreibung Die Logik zur Regelprüfung wird von der Klasse `ValidationManager` verwaltet. Zur Prüfung von spezifischen Regeln wird das Interface `Rule` implementiert, welches die `checkRule`-Methode zur Prüfung einer Regel definiert. Diese Regeln werden dem `ValidationManager` über Implementationen des `Validator` Interface zur Verfügung gestellt.

Der Zwischenschritt zwischen `Rule` und `ValidationManager` erhöht die Flexibilität, da dadurch die selbe Prüflogik für unterschiedliche `Nodes` eingesetzt werden kann, falls diese `Nodes` dieselben Regeln befolgen sollen.

Alle Regelverletzungen werden dem `ValidationManager` Property `RuleViolations` hinzugefügt und nach Prüfung aller Regeln von diesem zurückgegeben.

Enthaltene Bausteine Die Validator Implementation `ClassNameValidator` prüft ob ein Klassenname den Regeln bezüglich Namensgebung entspricht. Die Implementation `VariableNameValidator` führt dieselbe Prüfung auf Variablennamen aus.

Die `IdentifierNameRule` prüft ob die verwendeten `IdentifierNodes` den Regeln zur Namensgebung entsprechen.

Der `IdentifierHelper` implementiert Logik zum Auftrennen eines Bezeichners in einzelne Wörter und zur Prüfung ob diese einzelnen Wörter korrekte englische Wörter darstellen.

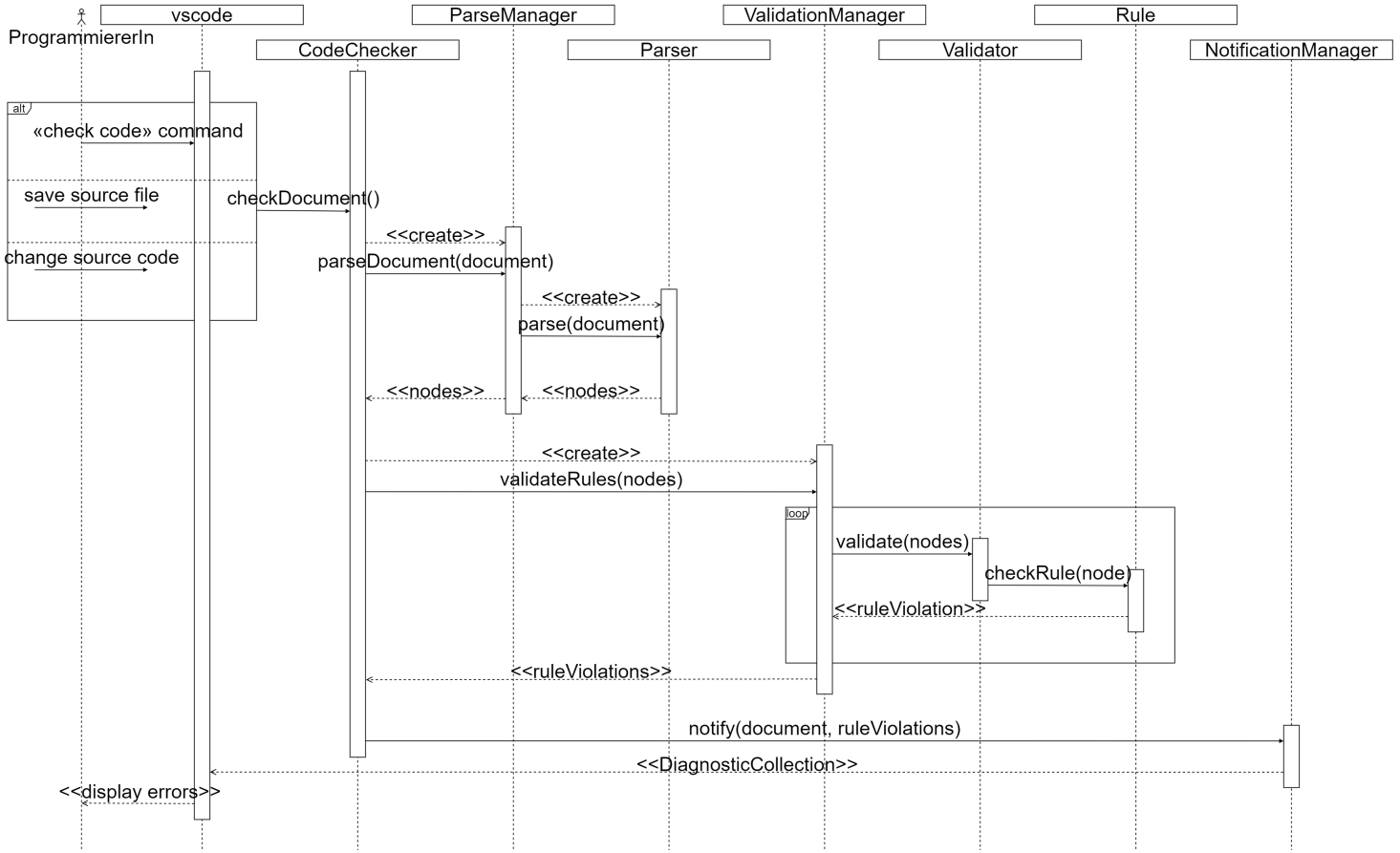
Die Klasse `RuleViolation` enthält alle Informationen zu Position innerhalb des Quelltextes und Art des Verstosses gegen die Regeln. Diese Informationen werden später zur Notifizierung der ProgrammiererIn benötigt.

Wichtige Schnittstellen Die Methode `validateRules` nimmt eine Liste von `Nodes` entgegen und prüft ob diese gegen eine der definierten Regeln verstossen.

Das Property `ruleViolations` sammelt alle gefundenen Regelverstösse.

Laufzeitsicht

Nachfolgend wird der Programmablauf von der Initialisierung über die Evaluation der Clean Code Regeln bis hin zur Rückgabe der Regelverstösse in Form eines Sequenzdiagramms dargestellt.


ProgrammiererIn
vscode
CodeChecker
ParseManager
Parser
ValidationManager
Validator
Rule
NotificationManager
alt
«check code» command
save source file
change source code
checkDocument()
<<create>>
parseDocument(document)
<<create>>
parse(document)
<<nodes>>
<<nodes>>
<<create>>
validateRules(nodes)
loop
validate(nodes)
checkRule(node)
<<ruleViolation>>
<<ruleViolations>>
notify(document, ruleViolations)
<<DiagnosticCollection>>
<<display errors>>

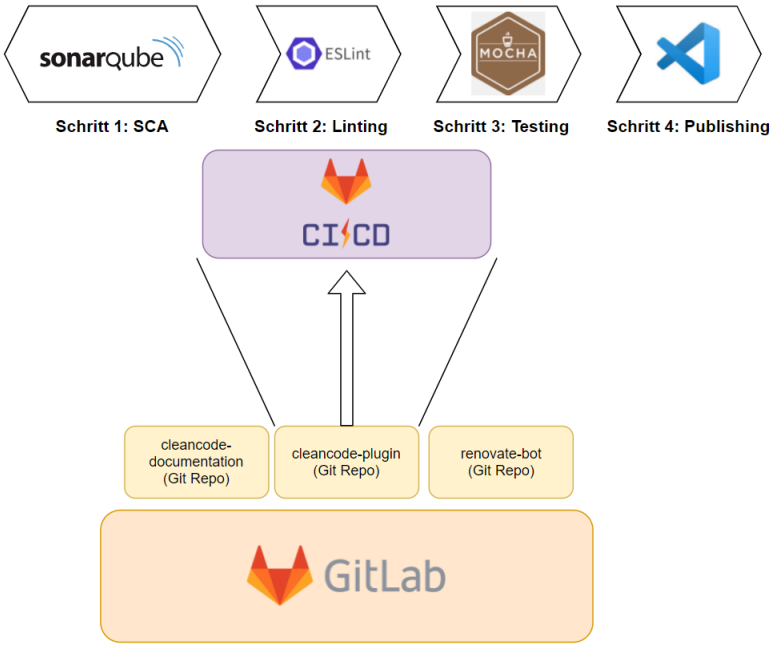
1. Zu Beginn des Programmablaufs wird ein `CodeChecker` instanziert. An diesen `CodeChecker`, genauer an dessen Methode `checkDocument`, wird der aktuell geöffnete Quelltext übergeben.
2. Die `checkDocument`-Methode instanziert ihrerseits einen `ParseManager`, der sich um das parsing kümmert und übergibt das `document` dessen Methode `parseDocument`.
3. Die `parseDocument`-Methode instanziert einen `Parser` und ruft dessen Methode `parse` auf, welcher sie ebenfalls das `document` übergibt.
4. Die `parse`-Methode parst nun das Dokument und gibt ein Array von `Nodes` an den `CodeChecker` zurück.
5. Der `CodeChecker` instanziert nun einen `ValidationManager`, der sich um die Validierung der Regeln kümmert und ruft dessen Methode `validateRules` auf. Als Parameter wird das zuvor erhaltene Array von `nodes` übergeben.
6. In der Methode `validateRules` wird nun in einer Schlaufe jeder registrierte `Validator` ausgeführt und ein Array von `RuleViolations` über den `ValidationManager` and den `CodeChecker` zurückgegeben.
7. Der `CodeChecker` ruft nun die `notify`-Methode des `NotificationManager` auf, welcher die Regelverletzungen in die `DiagnosticCollection` des Editors pusht.
8. Der Editor stellt die Mitteilungen in der `DiagnosticCollection` visuell dar.

Verteilungssicht

Da das Plugin lokal auf dem Rechner installiert wird, also nicht verteilt ausgeführt wird, verzichten wir an dieser Stelle auf ein eigenes Diagramm. An dieser Stelle wird auf das Kontextdiagramm verwiesen, aus welchem alle diesbezüglichen Informationen hervorgehen.

Infrastruktur

Die Infrastruktur ist im folgenden Diagramm dargestellt.



Das Projekt nutzt 3 Git-Repositories, von welchen jedes einem bestimmten Zweck dient.

Table 6: Git Repositories

Repository Namen	**Technologie**	**Funktion / Zweck**
cleancode-documentation	LaTeX	Repo für die Projekt- und Architekturdokumentation
cleancode-plugin	Typescript & Javascript	Repo für den Plugin Code inkl. Pipeline Deklaration
renovate-bot	Javascript	Repo für den Renovate Bot, der für das automatische updaten der Plugin Libraries zuständig ist

Das cleancode-plugin Repository enthält die Deklaration für eine GitLab CI Pipeline. Die Pipeline besteht theoretisch aus 4 Schritten, während des Projekts haben wir aber nur die ersten 3 implementiert, da wir bis zum Ende des Projekts keinen veröffentlichungsfähigen Zustand des Plugins erreicht haben.

Table 7: Pipeline Schritte

Schritt Nr.	Technologie / Tool	Zweck
# 1	SonarQube	Statistische Code Analyse (SCA) des neu hinzugefügten Quelltextes zur Sicherstellung der Codequalität und der Sicherheit
# 2	ESLint	Sprachspezifische Static Code Analyse (Linting)
# 3	Mocha	Ausführung der definierten Unit Tests um die korrekte Funktionalität des Programms sicherzustellen
# 4	Azure & Visual Studio Code Marketplace	Veröffentlichung der Erweiterung im Visual Studio Code Marketplace mittels Azure Cloud Functions

Entwurfsentscheidungen

Zuerst musste festgelegt werden für welche Plattform bzw. für welches Tool wir unser Plugin entwickeln wollen.

IDE oder Linter Plugin?

Die Umsetzung könnte entweder durch ein IDE oder ein Linter Plugin realisiert werden. Wir wählten ein IDE-Plugin, da uns dieses Vorteile bietet, die mit einem Linter Plugin nicht realisierbar wären.

- Sprachunabhängige Implementierung möglich bzw. beliebige Sprachen können unterstützt werden
- (Graphische) Nutzerinteraktion & Hervorhebungen direkt im Quelltext
- Direkte Interaktionen mit dem Quelltext
- Einklinken in IDE-spezifische Funktionen (z.B. Run on Save)

Ein Plugin, welches direkt für eine IDE geschrieben wird ermöglicht es, all diese Vorteile für unser Plugin zu nutzen. Bei einem Linter Plugin wären wir erstens auf die Funktionalität des Linters selber beschränkt gewesen. Ausserdem hätten wir Einschränkungen bei der Nutzerinteraktion hinnehmen müssen, da eine Anzeige von Problemen nur über das Problem Log der IDE möglich gewesen wäre.

Diese Wahl bringt aber auch einen Nachteil mit sich. Unser Plugin ist an die IDE, für welche wir es entwickelt haben gebunden. Ein Linter Plugin hätte in jeder IDE eingesetzt werden können, welche den ausgewählten Linter unterstützen. Wir sind uns dieser Einschränkung bewusst und sind der Meinung, dass die Vorteile hier die Nachteile überwiegen.

Wahl der IDE

Als nächstes mussten wir nun festlegen, für welche IDE wir das Plugin entwickeln wollen. Nach einer kurzen Recherche haben wir die sechs der am häufigsten verwendeten IDEs herausgesucht. Nun standen die folgenden mögliche IDEs zur Auswahl:

Table 8: IDE Übersichtstabelle

IDE Name	**Maintainer**	**Open Source**	**Unterstütze Sprachen**
Visual Studio Code	Microsoft	Ja	Diverse (hunderte via Plugins)
Visual Studio IDE	Microsoft	Nein	C#, C, C++, VB (weitere via Plugins)
IntelliJ IDEA	Jetbrains	Ja	Java (weitere via Plugins)
Eclipse	Eclipse Foundation	Ja	Java (weitere via Plugins)
Atom	Github	Ja	Diverse (hunderte via Plugins)

Alle IDEs in dieser Liste erfüllen die Anforderung, dass man ein Plugin entwickeln und im dazugehörigen Marketplace veröffentlichen kann. Für alle diese Plattformen würde sich ein Open Source Plugin entwickeln lassen, was eine Vorgabe für dieses Projekt ist. Es war für uns aber naheliegend eine IDE zu wählen, welche ebenfalls unter einer Open Source Lizenz steht. Damit fällt Visual Studio IDE als proprietäres Produkt von Microsoft weg. IntelliJ von Jetbrains fällt ebenfalls weg, da IntelliJ und PyCharm zwar Open Source sind, die anderen IDEs für andere Programmiersprachen aber nicht. Alle anderen IDEs aus dieser Liste wurden vollständig unter einer Open Source Lizenz veröffentlicht. Da wir mit unserem Plugin langfristig mehrere Sprachen abdecken möchten, wollten wir

eine IDE auswählen, welche ebenfalls möglichst viele Programmiersprachen unterstützt, sich also nicht primär auf eine Programmiersprache fokussiert. Daher fällt Eclipse ebenfalls weg, da es sich primär auf die Programmiersprache Java fokussiert und das älteste Produkt in der Liste ist. Übrig bleiben Atom und Visual Studio Code. Da Visual Studio Code mehr als doppelt so viele Sterne und mehr als drei mal so viele Contributors auf Gihtub hat als Atom, ist es die grössere und weiter verbreitete IDE von den beiden.

Wir haben uns daher für die Umsetzung eines Plugins für die Visual Studio Code IDE entschieden.

Risiken und technische Schulden

- Wenn zu einer späteren Zeit Regeln implementiert werden sollen, deren Überprüfung einen grossen Rechenaufwand mit sich bringt, wird die Implementation als normales Plugin an seine Grenzen stossen. Dies liegt daran, dass die Ausführung des Plugins auf dem gleichen Thread geschieht, auf dem auch des Editorfenster ausgeführt wird. Um die Berechnung in einen eigenen Thread auszulagern muss das Plugin zu einer Language Server implementierung umgeschrieben werden.

Glossar

Begriff	Definition
Application Programming Interface (API)	*Eine API ist eine Programmierschnittstelle die von einem Softwaresystem für andere Systeme zur Anbindung zur Verfügung gestellt wird*
Arc42	*Arc42 ist eine Vorlage für die Dokumentation von Softwarearchitekturen von Dr. Peter Hruschka und Dr. Gernot Starke*
Continuous Integration & Continuous Deployment (CI/CD)	*Es handelt sich hierbei um Konzepte mit denen alle Phasen der Anwendungsentwicklung vom programmieren bis hin zum Deployment automatisiert werden*
Clean Code	*Sauberer einfach und intuitiv verständlicher Quelltext mit klarer Struktur und angewandten Konzepten*

Begriff	Definition
Git	*Git ist eine freie Software zur verteilten Versionsverwaltung von Dateien wie z.B. Quelltext*
Integrated Development Environment (IDE)	*Integrierte Entwicklungsumgebung, wird von einem Programmierenden für das Entwickeln von Software verwendet*
Konkreter Syntaxbaum (CST)	*Geordneter Baum, der die syntaktische Struktur eines Textes anhand einer kontextfreien Grammatik abbildet*
Kontextfreie Grammatik (CFG)	*Eine Liste von Regeln, welche alle möglichen erlaubten Texte in einer Sprache definieren*
Linter	*Ein Programm zur Durchführung von Statischer Code Analyse (Siehe SCA)*
Pipeline	*In der Softwareentwicklung ist eine Pipeline ein System automatisierter Prozesse, welche Aktualisierungen beim Quelltext schnell von der Versionskontrolle in die Produktion übertragen.*
Static Code Analysis (SCA)	*Static Code Analyse, ist eine Methode zur Fehlersuche, bei der der Quelltext untersucht wird, bevor ein Programm ausgeführt wird.*
Unit Test	*Ein Test der ein einzelnes abgegrenztes Modul (Unit) der Software testet*
Visual Studio Code (VS Code)	*Eine in Typescript geschriebene IDE von Microsoft*
Wordlist	*Eine Wörterliste die in unserem Fall eine Liste mit englischen Wörter inklusive deren Definition und Worttyp enthält*