IFS INSTITUTE FOR SOFTWARE

OST

Automatisierte Analyse von Clean-Code Regeln mit IDE-Plugins

Bachelorarbeit

Studiengang Informatik
OST – Ostschweizer Fachhochschule
Campus Rapperswil-Jona

Frühlingssemester 2022

Autoren: Rafael Fuhrer, Pascal Schneider
Betreuer: Prof. Dr.-Ing. Frieder Loch
Experte: Dr. Michael Sollfrank
Gegenleser: Thomas Corbat

Zusammenfassung

Clean Code — sauber geschriebener Quelltext — ist ein elementarer Bestandteil der Softwarequalität. Man versteht darunter, dass Code gut lesbar, sauber strukturiert und einfach verständlich sein soll. Clean Code Prinzipien sollten bei jedem Softwareprojekt eine hohe Priorität geniessen, da sie vor allem die Wartbarkeit und die Erweiterbarkeit begünstigen.

Während unserer Studienarbeit haben wir einen Prototyp eines Plugins für die Visual Studio Code IDE entwickelt, welches geschriebenen Code auf die Einhaltung von Clean Code Prinzipien prüft. Dieser war in der Lage Quelltext gegen ein begrenztes Set an Regeln zu prüfen und Verstösse in der IDE farbig hervorzuheben, damit EntwicklerInnen diese Clean Code Prinzipien von Beginn weg einhalten und aufwändige Nachbesserungen vermeiden. Im Rahmen unserer Bachelorarbeit haben wir den Prototyp wieder aufgegriffen, mit dem Ziel ihn weiterzuentwickeln und als Open Source Projekt zu veröffentlichen. Dabei wurde die Architektur vom clientseitigen Plugin zum funktionelleren Language Server Plugin verändert, sowie der Satz an Clean Code Regeln die überprüft werden erweitert. Sämtliche Komponenten sind darauf ausgelegt, dass sie einfach ausgebaut werden können.

Vorbereitend für diese Weiterentwicklung haben wir unser Wissen über den Clean Code Begriff durch Literaturrecherchen vertieft. Zusätzlich stellten wir für die Veröffentlichung als Open Source Projekt Recherche bezüglich Lizenzbedingungen, Beitragsrichtlinien, Verhaltenskodex und benötigter Dokumentation an.

Mit der Durchführung und Auswertung von Nutzertests, bei denen die Testpersonen über unterschiedliche Fähigkeitsniveaus in der Programmierung und Wissen über die Clean Code Prinzipien verfügen, konnten wir das Plugin um einige nutzerrelevante Funktionen erweitern. Dadurch stehen nun eine Black- und Whitelist zur Verfügung und über ein Konfigurationsfile können einzelne Regeln deaktiviert, sowie weitere Feineinstellungen vorgenommen werden.

Mit der Veröffentlichung des Plugins als eigenständiges Open Source Projekt haben wir die Grundlage für ein Tool geschaffen, welches mit Hilfe der weltweiten Gemeinschaft um weitere Programmiersprachen, sowie Clean Code Regeln wachsen kann und sich langfristig als hilfreiches Werkzeug gegen schlechten Quellcode erweisen könnte.

Management Summary

Kontext und Problem

Clean Code — sauber geschriebener Quelltext — ist ein elementarer Bestandteil der Softwarequalität. Man versteht darunter, dass Code gut lesbar, sauber strukturiert und einfach verständlich sein soll. Clean Code Prinzipien sollten bei jedem Softwareprojekt eine hohe Priorität geniessen, da sie vor allem die Wartbarkeit und die Erweiterbarkeit begünstigen. Die Einhaltung dieser Regeln geht leider auch erfahrenen Programmierenden häufig vergessen. Die Auswirkungen von unsauberem Code sind vielfältig und meistens mit hohen Kosten verbunden, da diese Probleme oft erst spät im Entwicklungsprozess adressiert und geflickt werden. Während unserer Studienarbeit haben wir einen Prototyp eines Plugins für die Visual Studio Code IDE entwickelt, welches geschriebenen Code in Echtzeit auf die Einhaltung von Clean Code Prinzipien prüft.

Methoden und Resultate

Das Problem möchten wir mit einem Plugin angehen, welches Verstösse gegen Clean Code Prinzipien in der IDE farbig hervorhebt. Ziel ist es, dass Prinzipien von Beginn weg eingehalten und aufwändige Nachbesserungen vermieden werden können.

Aufbauend auf der bisherigen Arbeit konsultierten wir weitere Fachliteratur zum Thema Clean Code und überprüften die herausgearbeiteten Regeln. Zusätzlich wurde das Plugin um weitere Clean Code Regeln, sowie Konfigurationsoptionen erweitert. Die Architektur wurde zu einem Language Server Plugin umgebaut und alle enthaltenen Komponenten wurden auf möglichst einfache Erweiterbarkeit ausgelegt. Das Projekt wurde schlussendliche als Open Source Projekt veröffentlicht und damit der weltweiten Gemeinschaft zugänglich gemacht.

Weitere Arbeiten

Das Plugin könnte von folgenden Erweiterungen und Vertiefungen in Forschungsfeldern profitieren:

- Unterstützung für weitere Programmiersprachen
- Erweiterung des Regelsets zur Überprüfung von mehr Clean Code Prinzipien
- Unterstützung weiterer Entwicklungsumgebungen
- Integration in andere Tools die Softwarequalität unterstützen bzw. auswerten
- Forschung und Erweiterung im Bereich der Mustererkennung, die von guten und schlechten Codebeispielen lernt und das Gelernte dann auf neuen Code anwenden kann

Acknowledgements

Als erstes möchten wir uns bei unserem Betreuer, Prof. Dr.-Ing. Frieder Loch für seine Hilfe und die Inputs zu allen Belangen dieser Arbeit, sowie seine Unterstützung bei der Durchführung von Nutzertests, bedanken.

Weiter möchten wir uns bei unseren Testpersonen, Kevin Greminger, Jonas Hauser und Christoph Scheiwiller für ihre Unterstützung bei unseren Nutzertests, sowie dem daraus resultierenden Feedback bedanken.

Zuletzt möchten wir uns auch bei Tuija Krebs und Lorenz Keppler für das Korrekturlesen dieser Arbeit bedanken.

Inhaltsverzeichnis

Kapitel 1

Einführung

In diesem Kapitel wird eine Einführung und Einordnung in das von uns behandelte Thema gegeben. Wir geben dafür eine Definition der Themen unserer Arbeit, um sie zwischen den behandelten Bereichen einzuordnen.

1.1 Thematische Einordnung

Im Rahmen unserer Arbeit wollen wir das in unserer Studienarbeit [41] begonnene Clean Code Plugin weiterentwickeln. Das Plugin soll in die VS Code IDE installiert werden und Code automatisch auf Clean Code Bestimmungen überprüfen, sowie Verstösse gegen diese Regeln anzeigen. Unsere Bachelorarbeit, kurz BA, tangiert damit die Themen Softwarequalität in Form von Clean Code, Open Source und Didaktik. In den folgenden Abschnitten geben wir eine kurze thematische Einführung und Einordnung zu diesen Themen. In der nachfolgenden Abbildung 1.1 haben wir das Zusammenspiel der einzelnen Themen grafisch veranschaulicht.

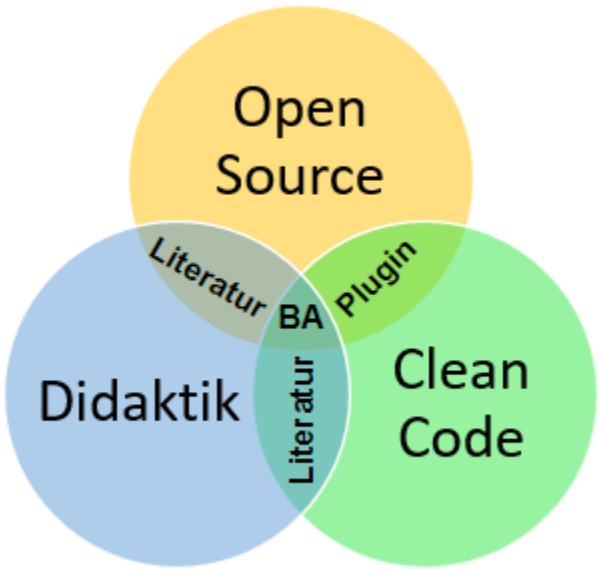


Abbildung 1.1: Übersicht über die Themengebiete der Arbeit

1.1.1 Softwarequalität

Clean Code ist ein elementarer Bestandteil der Softwarequalität [43]. Der Begriff Qualität kommt vom lateinischen *qualitas* und bedeutet übersetzt so viel wie Beschaffenheit, Merkmal, Eigenschaft oder Zustand und hat mehrere Bedeutungen [55]. 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 diesem Begriff stecken viele, sich teilweise überschneidende, Informatikthemen. So sind beispielsweise Performance und

Benutzerfreundlichkeit miteinander verknüpft, da eine langsame Anwendung nur höchst selten benutzerfreundlich sein wird. [41, S. 05]

1.1.2 Clean Code

Unter Clean Code versteht man, dass Code gut lesbar, sauber strukturiert, einfach verständlich und erweiterbar sein soll. Bei Clean Code sollen die folgenden Aspekte klar verständlich sein: der Ausführungsablauf, die Interaktionen von mehreren Objekten untereinander, Rolle und Verantwortlichkeiten jeder Klasse, die Funktionalität jeder Methode, sowie der Zweck jeder Variablen und Konstanten [58]. Clean Code Prinzipien sollten bei jedem Softwareprojekt eine hohe Priorität geniessen, da sie vor allem die Wartbarkeit und die Erweiterbarkeit begünstigen.

1.1.3 Open Source

Open Source bezieht sich auf etwas, das von allen Menschen verändert und weitergegeben werden kann, weil das Design öffentlich zugänglich ist. Der Begriff ist im Zusammenhang mit der Softwareentwicklung entstanden, wo er eine Herangehensweise an die Erstellung von Computerprogrammen beschreibt. Open-Source-Projekte, -Produkte oder -Initiativen beruhen auf den Grundsätzen des offenen Austauschs, der gemeinschaftlichen Beteiligung, der Transparenz, der Leistungsorientierung und der gemeinschaftsorientierten Entwicklung. Open-Source-Software, also Software, die unter einer Open Source Lizenz publiziert wurde, ist Software mit Quellcode, den jeder einsehen, verändern und verbessern kann. [39]

1.1.4 Didaktik

Das Wort *Didaktik* kommt aus dem Griechischen und kann von *didasko* abgeleitet werden, was übersetzt *ich lehre, belehre, unterrichte“* bedeutet [6, S. 45]. Didaktik bezeichnet die Wissenschaft des Unterrichts, die sich mit dem Lernen in allen Formen und Lehren aller Art unabhängig vom Lehrinhalt befasst [6, S. 45]. Im Rahmen unserer Arbeit tangieren wir das Thema Didaktik, da das entwickelte Plugin nicht nur als Kontrolle von Clean Code Prinzipien eingesetzt werden soll, sondern flankierend auch Mittel zum Verstehen und Erlernen dieser Prinzipien bereitstellen soll.

1.2 Grundlage

Im Rahmen unserer Studienarbeit [41] haben wir während des Herbstsemesters 2021 den ersten Prototyp eines Clean Code Plugins für die VS Code IDE erstellt. Der Prototyp ist in der Lage Variablen- und Funktionsnamen gegen Benennungsregeln zu prüfen. Ausserdem wird die Anzahl an Argumenten, die einer Funktion mitgegeben werden, gezählt und bei mehr als 3 Argumenten eine Warnung ausgegeben.

Die Erweiterung ist aufgeteilt in mehrere Untersysteme die voneinander entkoppelt ihre Teile der Regelprüfung durchführen. Das erste Modul ist für das Parsen des Quelltextes zuständig und bringt alle relevanten Informationen in eine Form, in der sie von den weiteren Modulen benutzt werden können. Das zweite Modul ist das Herzstück der Erweiterung und ist für die Regelprüfung zuständig. Die Regellogik ist für jede Regel einzeln implementiert und wird von übergeordneten Validierungsabläufen wo nötig auf die entsprechenden Stücke des Quelltextes angewandt die vom ersten Modul gesammelt wurden. Zum Schluss werden alle gefundenen Verstösse an das dritte Modul übergeben, welches dafür zuständig ist, diese Verstösse dem Programmierenden in einer Form mitzuteilen, welche genug Information beinhaltet, um sie zukünftig zu verringern.

Im Moment unterstützt das Plugin lediglich die Programmiersprache Java und läuft im selben Thread wie die Entwicklungsumgebung. Dies könnte beim Prüfen sehr grosser Code Files oder sehr rechenintensiven Regeln zu Problemen in Form vom Einfrieren der gesamten IDE führen. Da das Plugin in diesem Zustand aufgrund der vorhergehend beschriebenen Einschränkungen noch nicht veröffentlichungswürdig ist, soll es im Rahmen dieser Bachelorarbeit auf einen funktionelleren Stand gebracht werden.

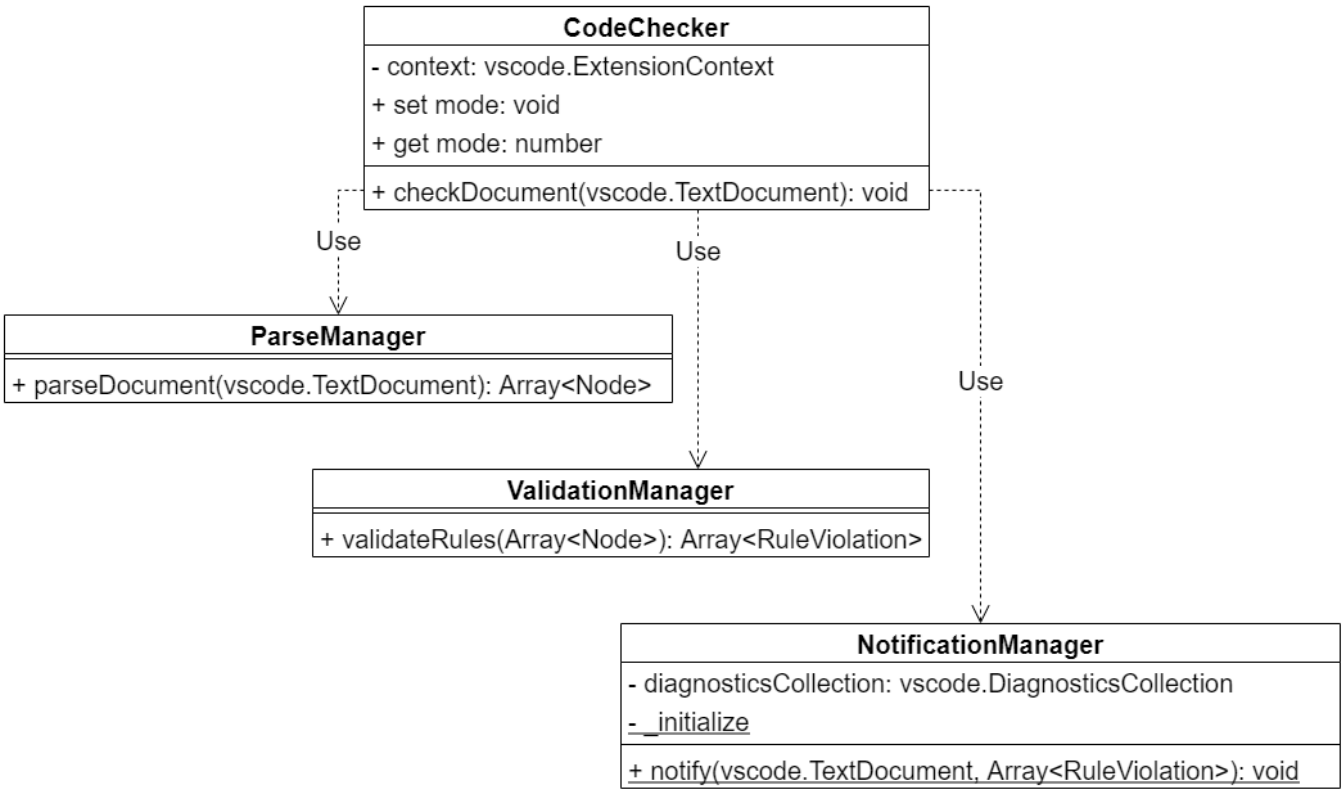


Abbildung 1.2: Architekturübersicht des Prototyps

1.3 Motivation

Da es bisher nur wenige Tools gibt, die Entwickelnde während dem Programmieren bei der Clean Code Thematik unterstützen, haben wir im Rahmen unserer Studienarbeit ein Tool für die Visual Studio Code Entwicklungsumgebung entwickelt, welches einen Teil dieser Lücke füllen soll. Unsere Erweiterung soll diesem Problem entgegenwirken, indem es sich in die Entwicklungsumgebung einklinkt und Verletzungen der Clean Code Regeln direkt anzeigt. Dadurch kann das Problem direkt am Ursprung behoben werden. Die Erweiterung hatte nach der Studienarbeit noch einen rudimentären Funktionsumfang und einen experimentellen Status, Sie soll im Rahmen dieser Bachelorarbeit aufgegriffen und weiterentwickelt werden. Der Fokus liegt hierbei darauf einen in der Praxis nutzbaren Stand zu erreichen, die Benutzerfreundlichkeit zu erhöhen und den Umfang des Regelwerks zu erweitern. Am Schluss der Arbeit soll die Erweiterung unter einer freien Lizenz als Open Source Projekt veröffentlicht und so der allgemeinen Entwicklergemeinschaft zugänglich gemacht werden.

Kapitel 2

Literaturrecherche

In diesem Kapitel beschreiben wir die im Rahmen der Bachelorarbeit durchgeführten Literaturrecherchen und die daraus gewonnen Erkenntnisse. Wir greifen dafür Vorwissen aus der Studienarbeit auf, analysieren weitere Werke zum Thema Clean Code und vergleichen diese.

2.1 Clean Code

Clean Code ist ein elementarer Bestandteil der Softwarequalität. Softwarequalität setzt sich jedoch aus mehreren Unterthemen zusammen. Diese sind gemäss ISO/IEC 25010 [28] in der Tabelle 2.1 definiert:

Clean Code begünstigt aber vor allem den Themenpunkt Wartbarkeit und hier insbesondere die Unterpunkte Modularität, Modifizierbarkeit und Testbarkeit. Bei der Anwendung von Clean Code Prinzipien entsteht mehr Modularität und damit verbessert sich automatisch die Modifizierbarkeit und Testbarkeit. Somit ist Clean Code ein elementarer Bestandteil der Softwarequalität. Wie die Recherchen unserer Studienarbeit zeigen [41, S.14 - S.17], gehören diese Prinzipien zu den am wenigsten beachteten und es existieren im Vergleich zu den anderen Themen in der Liste erst wenige Tools, die den Programmierenden bei der Anwendung der Best Practices unterstützen.

Tabelle 2.1: 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

Quelle: Studienarbeit [41, S. 06]

2.1.1 Werkzeuge der Softwareentwicklung

Einem Entwickelnden stehen heutzutage viele Werkzeuge zur Verfügung, die bei der Softwareentwicklung Unterstützung bieten. Um unsere Arbeit im Kontext der Softwareentwicklung als Gesamtes einordnen zu können, gehen wir hier noch grundlegend auf die weiteren Tools ein. Diese lassen sich in die Kategorien, siehe Tabelle 2.2 gruppieren [32]. [41, S. 06]

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

Tabelle 2.2: 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, Apache 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
Continuous 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

Quelle: Studienarbeit [41, S. 07]

2.2 Clean Code Literatur

Im Rahmen unserer Studienarbeit haben wir zur Identifikation möglicher Clean Code Regeln für unsere Erweiterung das Buch *Clean Code* [31] von Robert C. Martin analysiert. Dieses Buch hatten wir gewählt, weil es sich dabei um ein in der Branche weit bekanntes Werk handelt, auf welches wir auch vor diesem Projekt schon öfters gestossen sind. Bereits in der Studienarbeit haben wir auf weitere nennenswerte Werke dieses Themenbereichs verwiesen. Konkret haben wir dabei die folgenden 3 Bücher identifiziert: *Pragmatic Programmer* [48], *Code Complete 2* [33] und *The Art of Readable Code* [9]. Im Rahmen der Studienarbeit mussten wir uns jedoch auf ein Werk als Grundlage unserer Arbeit festlegen und konnten die anderen Werke nicht genauer analysieren. Dies haben wir für diese Bachelorarbeit nachgeholt,

denn auch Robert Martin gibt im ersten Kapitel von *Clean Code* den Hinweis, dass es mehrere "Denkschulen" bzw. "Auffassungen" der Clean Code Thematik gibt. Entsprechend fehlt eine generelle Definition für Clean Code bzw. Regeln für die Umsetzung. Aufgrund dessen berücksichtigen wir Fachliteratur, damit wir sicherstellen können, dass unser Clean Code Plugin nicht zu sehr von einem einzelnen Werk beeinflusst wird. Entsprechend fokussieren wir uns bei den nachfolgenden Analysen auf die Unterschiede im Vergleich zum Buch *Clean Code*.

Da es für den Begriff Clean Code keine allgemeine Definition gibt, interpretieren diesen alle Autoren unterschiedlich. Dies spiegelt sich auch so in ihren Werken wider. Wir konnten vor allem in 2 Ebenen eine Differenzierung feststellen. Während einige Autoren sich exklusiv auf das Schreiben von Code an sich fokussieren, stellen andere das Umfeld rund um den Programmierenden in den Mittelpunkt. Ausserdem unterscheiden sich die Werke im Detaillierungsgrad der Regeln und Beispiele. Diese Unterschiede haben wir in der Abbildung 2.1 grafisch veranschaulicht. Auf der Y-Achse zeigen wir auf, wie detailliert die Autoren die von Ihnen aufgestellten Prinzipien beschreiben, während die X-Achse beschreibt, ob sich die aufgestellten Prinzipien mehr auf konkrete Codingregeln oder mehr auf das Umfeld, also das Projekt als Ganzes fokussieren.

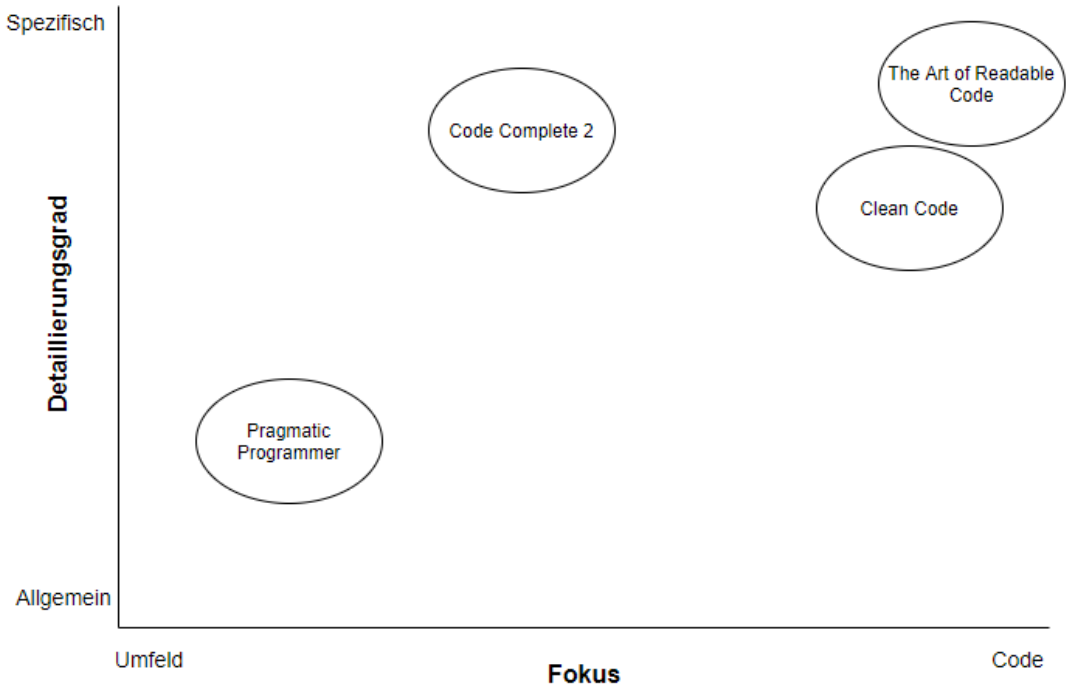


Abbildung 2.1: Einordnung der analysierten Clean Code Werke

In den Abschnitten Clean Code bis The Art of Readable Code gehen wir genauer auf die einzelnen Werke und ihre jeweilige Interpretation des Clean Code Begriffs ein.

2.2.1 Clean Code

In *Clean Code* beschreibt Robert C. Martin Regeln, wie nach seiner Denkschule sauberer Code geschrieben wird. Auf Organisatorisches oder das Umfeld der Programmierenden geht er nicht detailliert ein. Eine genauere Analyse dieses Buches bzw. der aufgestellten Regeln findet sich in Kapitel 2 unserer Studienarbeit [41, S. 08 - S. 17]. Der Autor interpretiert den Begriff Clean Code hier als konkrete Regeln unter deren Anwendung sauberer Code geschrieben werden soll. In der Abbildung 2.1 haben wir dieses Werk daher auf der Achse Detaillierungsgrad bei spezifisch und auf der Fokus Achse bei Code angeordnet.

2.2.2 Pragmatic Programmer

In *Pragmatic Programmer* [48] gehen Andy Hunt und Dave Thomas nur am Rande und nur indirekt auf das Thema programmieren an sich ein. Im Buch stehen Prinzipien der Handwerkskunst des Programmierens im Vordergrund, wie zum Beispiel das Management eines Codeprojekts. Dabei geht es aber um Gewohnheiten, welche man sich aneignen

kann, um am Ende ein besseres Ergebnis zu erzielen. In den ersten beiden Kapiteln gehen sie dabei auf das Mindset ein, welches ihrer Meinung nach vorhanden sein muss um guten, sauberen Code zu schreiben. So sehen sie beispielsweise die Bereitschaft zum lebenslangen Lernen, sowie gute Kommunikationsfähigkeiten als Voraussetzungen. In den weiteren Kapiteln gehen sie zuerst auf relevante Tools, zum Beispiel Versionierungstools ein, danach auf Ressourcenmanagement, um schliesslich grundlegende Prinzipien wie paralleles Programmieren zu behandeln. Abgeschlossen wird das Buch mit zwei Kapiteln rund um das Thema Projektmanagement. Hier werden Themen wie das Anforderungsmanagement detaillierter untersucht.

Vereinzelt wird über mehrere Kapitel verteilt auf die Thematik von Code Prinzipien wie DRY (Don't Repeat Yourself) eingegangen. Dabei handelt es sich aber überwiegend um generalisierte Prinzipien, welche so auch im Buch *Clean Code* zu finden sind. Eine abweichende Meinung zu konkreten Regeln im Buch *Clean Code*, konnten wir hier nicht identifizieren.

Bei *Pragmatic Programmer* handelt es sich um ein Buch, welches einen guten Überblick über die verschiedenen Aspekte der Softwareentwicklung gibt, ohne dabei konkret und detailliert auf die einzelnen Themen einzugehen. Wir haben dieses Werk daher in der Abbildung 2.1 auf der Achse Detaillierungsgrad in der unteren Hälfte, näher bei Allgemein angeordnet. Die Autoren interpretieren den Begriff Clean Code allgemeiner und weiter als die Autoren der anderen Werke. Sie fokussieren sich primär auf das Umfeld sowie das Mindset des Programmierenden. Die Prinzipien sind dabei allgemeiner gehalten als in den anderen Werken. Es finden sich auch kaum konkrete Beispiele, was aber zum Teil auch der Tatsache geschuldet ist, dass sich das Buch weniger auf konkrete Codingregeln fokussiert. In der Abbildung 2.1 haben wir das Werk auf der Fokus Achse näher an Umfeld als an Code angeordnet.

2.2.3 Code Complete 2

Beim umfassendsten der vier Bücher, *Code Complete 2* [33], von Steve McConnell handelt es sich um Fachliteratur, welche das Thema Clean Code weiter definiert. Nach seiner Auffassung umfasst das Thema auch organisatorische Aspekte. In den ersten Kapiteln wird eine Einführung in das Thema Software und Programmierung gegeben. Danach geht McConell auf das Thema Anforderungsmanagement und Softwarearchitektur ein. Ebenfalls behandelt werden Themen wie defensives und sicheres Programmieren, Datentypen und Controll Flow im Programm. Wir haben das Werk daher in der Abbildung 2.1 auf der Fokus Achse ungefähr mittig, aber leicht mehr am Umfeld als an Code orientiert angeordnet. Kapitel 20 widmet sich schliesslich explizit dem Thema Softwarequalität. Da das Buch im Jahr 2003 erschienen ist, sind einige Aussagen wie die Auflistung der Bestandteile der Softwarequalität, verglichen mit der aktuellen ISO Definition [28] nicht mehr aktuell. Abgeschlossen wird das Buch mit den Themen Testing, Debugging, Refactoring, Tuning und Projektmanagement.

Bei *Code Complete 2* und *Pragmatic Programmer* handelt es sich um ähnliche Werke. Die grössten Unterschiede liegen im Detaillierungsgrad, da *Code Complete 2* mehr ins Detail geht. Ausserdem finden sich im Buch mehr konkrete Code Beispiele als in *Pragmatic Programmer*. Wir haben das Buch in der Abbildung 2.1 auf der Achse des Detaillierungsgrads bei spezifisch angeordnet. Bei der Analyse der Code Beispiele ist uns vor allem das Alter des Buches aufgefallen. Ausserdem wurden viele Beispiele dieses Buches in C++ gegeben während Robert Martin in *Clean Code* seine Beispiele in Java oder Java ähnlichem Pseudocode aufzeigt. Dies erschwert einen direkten Vergleich, da die Programmiersprachen zum Teil über unterschiedliche Sprachfeatures verfügen, aus welchen sich individuelle Regeln ableiten lassen.

Wir sind daher zum Schluss gekommen, dass wir die meisten Clean Code Regeln von *Code Complete 2* nicht direkt mit *Clean Code* vergleichen können. Der Autor interpretiert den Begriff Clean Code allgemein und fokussiert sich neben konkreten Coding Regeln auch auf organisatorische Aspekte und das Umfeld des Programmierenden, sowie auf grundlegende Theorie rund um die Thematik der Softwareentwicklung.

2.2.4 The Art of Readable Code

In ihrem Buch *The Art of Readable Code* [9] zeigen Dustin Boswell und Trevor Foucher simple, praktische Techniken, um besseren Code zu schreiben. Sie gehen dabei wie Martin nur wenig auf Organisatorisches oder das Umfeld ein, sondern fokussieren sich ganz auf das Schreiben von Code. Dabei behandeln sie die Themen Benennung, Controll Flow, Kommentare, Expressions, Testing und Architektur.

Dieses Buch lässt sich gut mit Robert Martins *Clean Code* Buch vergleichen, da es konkrete Regeln inklusive Code Beispiele beinhaltet. Da Themen ausserhalb der Programmierung nur kurz angeschnitten werden, haben wir das Werk in der Abbildung 2.1 auf der Fokus Achse ganz bei Code angeordnet. Da stets detaillierte Beispiele zu jeder Regel gegeben werden, haben wir das Werk auf der Achse des Detaillierungsgrads bei spezifisch angeordnet.

Im Abschnitt Regelanalyse haben wir Regeln identifiziert, welche gegen Regeln aus dem Buch *Clean Code* verstossen oder nicht direkt kompatibel sind. Die Autoren interpretieren den Begriff Clean Code hier ähnlich wie Robert Martin als konkrete Regeln unter deren Anwendung sauberer Code geschrieben wird.

2.3 Regelanalyse

Da wir beim Buch *The Art of Readable Code* eine ähnliche Interpretation des Clean Code Begriffs wie beim Buch von Robert Martin feststellen konnten und der Fokus genauso auf konkreten Regeln für das Schreiben von sauberem Code lag, haben wir uns entschieden genauer auf dieses Buch einzugehen. Die Denkschulen die *Code Complete 2* und *Pragmatic Programmer* vertreten erachten wir grundsätzlich als kompatibel mit diesen beiden Büchern. Da der Fokus bei diesen Büchern jedoch auf organisatorischen und persönlichen Aspekten liegt, bieten sie für unsere Arbeit nicht denselben Mehrwert. So ist es beispielsweise nicht möglich, mit einer IDE-Erweiterung zu prüfen, ob die Programmierende fit und ausgeruht ist, oder ob das Projekt mittels sinnvollem Projektmanagement betrieben wird.

Spezifische Wörter wählen

Diese Regel besagt, dass eine Variable mit möglichst spezifischen Worten benannt werden soll. Soweit sind sich die Autoren mit Martin einig. Allerdings verwenden sie eine Variable mit dem Identifikator `NumNodes` und damit eine Abkürzung. Ein ähnliches Beispiel lässt sich in *Code Complete 2* [33], Kapitel 11 finden. Robert Martin hat in Kapitel 2.7 in *Clean Code* [31] aber die Regel "Codierungen Vermeiden" aufgestellt, welche das Verwenden solcher Abkürzung untersagt.

```
// Variable für die Anzahl Nodes in einem Baum

// Vorschlag nach The Art of Readable Code
NumNodes

//Vorschlag nach Clean Code
NumberOfNodes
```

Für unser Plugin werden wir hier Robert Martin folgen, denn seine Regel ist für uns sinnvoller und in der Anwendung klarer. Ausserdem basiert unser Plugin auf Wortlisten, welche nur vollständige Wörter enthalten. Eine technische Umsetzung, welche Abkürzungen berücksichtigt, haben wir in dieser Form daher nicht vorgesehen. Für Einzelfälle kann aber die Whitelist verwendet werden, damit die Abkürzung keine Meldungen mehr erzeugt.

Kommentare

In Kapitel 1 von *The Art of Readable Code* gehen die Autoren darauf ein, dass Quellcode möglichst kurz, aber dennoch verständlich gehalten werden soll. Bezüglich der Verständlichkeit geben sie auch folgendes Beispiel bezüglich Kommentare zur Unterstützung der Verständlichkeit.

```
// Fast version of "hash = (65599 * hash) + c"
hash = (hash << 6) + (hash << 16) - hash + c;
```

Robert Martin vertritt in *Clean Code* in den Kapiteln 4.1 & 4.2 die Meinung, dass Kommentare, wenn immer möglich weggelassen werden sollen und der Code für sich selbst sprechen sollte. Ähnlichen Beispiele wie beim Beispiel oben hat Martin in seinem Buch explizit als schlechten Code klassifiziert. In den weiteren Kapiteln gehen die Autoren von *The Art of Readable Code* noch weiter auf das Thema Kommentare ein. Dabei sind sie sich in vielen Punkten mit Robert Martin einig. So sollen Kommentare zum Beispiel möglichst kurz und präzise sein. Die beiden Denkschulen unterscheiden sich hier also nicht grundsätzlich, sondern nur in einigen Nuancen.

Da wir in unserem Plugin die Sinnhaftigkeit von Kommentaren im Moment nicht überprüfen können, spielt diese Uneinigkeit für das Plugin momentan keine Rolle. Wir würden hier aber Robert Martins Denkschule folgen, da wir ebenfalls der Meinung sind, das Code der keine erklärenden Kommentare benötigt immer besser ist, als Code der dies tut.

2.4 Open Source Projekt

Für die Veröffentlichung unseres Plugins als Open Source Projekt müssen diverse Aspekte beachtet werden. Es muss festgelegt werden, unter welcher Lizenz der Source Code stehen soll, oder wie die Zusammenarbeit mit der Community geregelt werden soll.

2.4.1 Bestandteile eines Open Source Projekts

Als Erstes haben wir abgeklärt, aus welchen zwingenden Bestandteilen ein Open Source Projekt besteht. Die Firma GitHub, welche Online Repositories für Open Source Projekte bereitstellt, hat eine entsprechende Anleitung [16] publiziert. In dieser werden unter Sektion 3, die folgenden vier essenziellen Bestandteile für ein Open Source Projekt genannt.

- Eine Open Source Lizenz
- Ein README
- Beitragsrichtlinien
- Ein Verhaltenskodex

Wir haben uns dafür entschieden, dass wir alle diese Bestandteile in der englischen Sprache abfassen werden. Dadurch können wir eine grössere Community an potenziellen Entwicklern erreichen und die Erweiterung für alle englischsprachigen Personen zugänglich machen.

Lizenz

Da es heutzutage über 100 verschiedene Open Source Lizenzen gibt [13], mussten wir zuerst einschränken, welche Lizenzen wir genauer evaluieren wollen. Wir haben uns dafür entschieden uns auf die am weitesten verbreiteten Lizenzen zu fokussieren, da wir uns davon eine höhere Bereitschaft zur Nutzung oder Mitwirkung versprachen. Eine Liste mit den populärsten Lizenzen stellt die Open Source Initiative zur Verfügung [25]. Im Abschnitt Popular Licenses werden die folgenden neun Lizenzen als besonders populär oder mit starken Communities aufgelistet.

- Apache License 2.0
- BSD 3-Clause New or Revised license
- BSD 2-Clause Simplified or FreeBSD license
- GNU General Public License (GPL)
- GNU Library or "Lesser" General Public License (LGPL)
- MIT license
- Mozilla Public License 2.0
- Common Development and Distribution License
- Eclipse Public License version 2.0

Die Lizenzen lassen sich nach David Wheeler [49] in die drei Kategorien *permissive, weakly protective* und *strongly protective* aufteilen. Permissive Lizenzen erlauben es dabei, dass Software proprietär werden kann. Weakly Protective Lizenzen erlauben den Einsatz der Software innerhalb proprietären Softwareprojekten, die genutzte Software unter dieser Lizenz, sowie alle Änderungen müssen aber offen und zugänglich bleiben. Strongly Protective Lizenzen verlangen, dass Softwareprojekte, welche Softwarebestandteile unter einer solchen Lizenz nutzen ebenfalls unter dieser Lizenz veröffentlicht werden. Ein proprietärer Einsatz wird dadurch verunmöglicht.

Da wir beide das Projekt möglichst offen gestalten wollen, um damit z.B. auch eine kommerzielle Nutzung des Source Codes zu ermöglichen, haben wir uns entschieden eine Lizenz aus der permissiven Kategorie zu verwenden. Entsprechend haben wir hier nur die Lizenzen dieser Kategorie, namentlich MIT License, Apache License 2.0, BSD 3-Clause [18] und Eclipse Public License 2.0 [11], genauer analysiert. Alle diese Lizenzen haben gemeinsam, dass sie eine kommerzielle Nutzung sowie Verbreitung des Quelltextes erlauben. Ausserdem darf unter diesen Lizenzen Quelltext stets privat genutzt und frei modifiziert werden. Bei allen Lizenzen muss ausserdem stets die Lizenz bzw. das Lizenzfile mit ausgeliefert werden. Sämtliche Lizenzen schränken ausserdem die Garantie und Haftung ein [18]. Die expliziten, für uns relevanten, Unterschiede zwischen den Lizenzen haben wir in der Tabelle 2.3 aufgezeigt.

Tabelle 2.3: Vergleich permissiver Lizenzen

Lizenz	**Statusänderungen**	**Markenzeichenrechte**	**Patentschutz**	**Quellen**
Apache License 2.0	Müssen dokumentiert werden	Explizit verboten	Explizit gegeben	[18]
MIT License	Keine Bestimmung	Implizit verboten	Keine Bestimmung	[18]
BSD 3-Clause	Keine Bestimmung	Implizit verboten	Keine Bestimmung	[26]
Eclipse Public License 2.0	Keine Bestimmung	Implizit verboten	Keine Bestimmung	[11]

Während die MIT, die BSD 3-Clause sowie die Eclipse Public Lizenz sich untereinander kaum unterscheiden, sticht die Apache License 2.0 mit einigen besonderen Bestimmungen heraus. So müssen Änderungen an lizenziertem Material dokumentiert werden. Ausserdem steht in dieser Lizenz explizit, dass keine Markenrechte mit der Lizenz übertragen werden. Es ist aber davon auszugehen, dass diese Rechtseinschränkung bei allen anderen Lizenzen ebenfalls implizit gegeben ist [18]. Letztlich bietet die Apache License 2.0 noch explizit Anerkennung des Patentschutzes für Beitragende. Entsprechend können Patenthalter ihren patentierten Quelltext in das Projekt einbringen, während sie gleichzeitig ihren Patentschutz behalten können. Besonders diese letzte Bestimmung ist für uns interessant, da es im Bereich der Code Qualität viele kommerzielle Anbieter gibt. Mit den Lizenzbestimmungen der Apache License 2.0 könnten private Firmen patentierte Beiträge einbringen, ohne ihren Patentschutz zu verlieren. Um uns diese Möglichkeit offen zu halten und weil die Lizenz im Vergleich zu den anderen Lizenzen keine Nachteile bietet, haben wir uns daher entschieden, unser Projekt unter der Apache License 2.0 zu veröffentlichen.

2.4.2 Readme

Das Readme File ist für Endnutzer und interessierte Mitwirkende der erste Kontaktpunkt mit dem Projekt. Daher auch der Name Readme, der übersetzt *lies mich* bedeutet. Entsprechend ist es wichtig ein ansprechendes Readme bereitzustellen, welches möglichst alle potenziellen Fragen dieser Gruppen beantwortet. Die folgenden Punkte sollten demnach von jedem Readme abgedeckt sein [27].

- Eine generelle Beschreibung des Systems oder Projektes.
- Der Projektstatus ist vor allem dann wichtig, wenn sich das Projekt noch in der Entwicklung befindet. Erwähnen Sie in der Datei geplante Änderungen und die Entwicklungsrichtung oder geben Sie direkt an, wenn ein Projekt fertig entwickelt ist.
- Die Anforderungen an die Entwicklungsumgebung für die Integration.
- Eine Anleitung für die Installation und Bedienung.
- Eine Auflistung der verwendeten Technologien und gegebenenfalls Links zu weiteren Informationen zu den Technologien.
- Open-Source-Projekte, die Entwickler eigenständig verändern oder erweitern, sollten in der Readme.md einen Abschnitt zur „gewünschten Zusammenarbeit“ enthalten. Wie geht man mit Problemen um? Wie sollen Entwickler die Änderungen vorantreiben?
- Bekannte Bugs und eventuelle Fehlerbehebungen. (Soweit vorhanden)
- FAQ-Bereich mit allen bisher gestellten Fragen.
- Copyrights und Lizenzinformationen.

Das Readme sollte dabei generell immer auf den Endnutzer der Applikation abgestimmt sein. Da das Readme in unserem Fall die Plugin Präsentation im Visual Studio Code Store mit übernehmen wird, legen wir hier ein besonderes Augenmerk auf die Endnutzerfreundlichkeit unseres Readmes. Bei der Strukturierung ist dies ebenfalls eingeflossen. Die für den Endnutzer relevanten Informationen werden prominent im oberen Teil des Readmes platziert.

Das durch uns erstellte Readme kann im öffentlichen Repository [1] eingesehen werden.

2.4.3 Beitragsrichtlinien

In den Beitragsrichtlinien wird festgelegt, wie interessierte Mitwirkende zum Projekt beitragen können. Dies wird meistens mittels einer weiteren Datei namens *contributing*, ähnlich des Readmes kommuniziert. Die Contributing Datei gibt dabei dem Projekteigentümer die Möglichkeit zu kommunizieren, wie Mitwirkende zum Projekt beitragen können und erhöht gleichzeitig ihre Chancen, dass ihr Beitrag auf Anhieb akzeptiert wird. Die Contributing Datei kann so für beide Seiten Zeit einsparen, da Diskussionen über problematische Beiträge grösstenteils vermieden werden können [19].

Wir haben uns für unsere Beitragsrichtlinien an der Contributing Datei von Atom [14] orientiert, da es sich bei Atom wie bei VS Code um eine IDE handelt und es alle für uns relevanten Aspekte abdeckt.

Unser Contributing File kann ebenfalls im öffentlichen Repository [2] eingesehen werden.

2.4.4 Verhaltenskodex

Der Verhaltenskodex soll eine gesunde und konstruktive Zusammenarbeit durch die Festlegung von Umgangsregeln garantieren. Im Verhaltenskodex wird von den Projekteigentümern festgelegt, welche Erwartungen an Projektteilnehmende gestellt werden. Damit soll eine positive und soziale Atmosphäre rund um das Projekt geschaffen werden.

Neben der Klarstellung Ihrer Erwartungen soll ein Verhaltenskodex Folgendes beschreiben [17].

- Wo gilt der Verhaltenskodex?
- Für wen gilt der Verhaltenskodex?
- Was passiert, wenn jemand dagegen verstösst?
- Wie und wo man Verstösse gegen den Verhaltenskodex melden kann

Wir haben uns daher dazu entschieden, den von mehr als 40'000 Open Source Projekten genutzten und daher gut erprobten *Contributer Convenant* [8] für unser Projekt einzusetzen.

2.4.5 Git Plattform

Das Open Source Projekt soll auf einer Git Hosting Plattform veröffentlicht werden. Die drei bekanntesten Plattformen, die dafür in Frage kommen würden, wären Github [3] von Microsoft, GitLab [4] von der GitLab Foundation und Codeberg. Codeberg ist dabei ein relativ junges Projekt, das von Spenden finanziert wird und sich als komplett offene Alternative zu den anderen beiden Plattformen, die von Firmen finanziert werden, positioniert. Da wir bei der Entwicklung des ersten Prototyps in der Studienarbeit bereits GitLab als Git Plattform gewählt haben und bei dieser Arbeit so beibehalten haben, war die Wahl von GitLab für uns naheliegend. So können beispielsweise auch die bestehenden Pipeline Files weiterverwendet werden und die Projektstruktur muss nicht angepasst werden. Kleinere Plattformen haben wir nicht in Betracht gezogen, da bei diesen weitere Abklärungen zur Stabilität und Langlebigkeit nötig gewesen wären, sowie daraus resultierende Risiken, die wir nicht eingehen wollten.

2.5 Diskussion

Bei der Analyse der vier verschiedenen Clean Code Büchern haben wir vier verschiedene Denkschulen über das Thema Clean Code identifiziert. Jede dieser Denkschulen hat ihre Daseinsberechtigung. Die unterschiedlichen Interpretationen des Clean Code Begriffs lassen sich auf die unterschiedliche Gewichtung von Schwerpunkten und dem Fokus auf bestimmte Themen zurückführen. Dies führt dazu, dass die verschiedenen Denkschulen, abgesehen von einigen Nuancen nicht grundsätzlich inkompatibel zueinander sind. Da sich die Bücher *Pragmatic Programmer* und *Code Complete 2* weniger auf das Schreiben von Code, sondern primär auf das organisatorische und persönliche Umfeld des Programmierenden fokussieren, haben diese Werke wenig Relevanz für diese Arbeit oder unsere Erweiterung. Die restlichen darin gelernten Konzepte sind aber von hoher Bedeutung in anderen Teilbereichen der Softwarequalität. Eine wichtige

[1] https://gitlab.com/cleancode2/cleancode-plugin
[2] https://gitlab.com/cleancode2/cleancode-plugin/-/blob/develop/CONTRIBUTING.md
[3] https://github.com/
[4] https://gitlab.com/

Erkenntnis konnten wir jedoch aus dem Buch *Code Complete 2* gewinnen. Es hat uns aufgezeigt, dass es neben generell gültigen Regeln auch viele sprachspezifische Regeln gibt, welche berücksichtigt werden müssen. Dieser Umstand war uns davor nicht in diesem Umfang bewusst. Wir haben daher eine Erweiterung der Architektur für sprachspezifische Regeln auf die Roadmap des Plugins gesetzt.

Bei der Analyse von *The Art of Readable Code* im direkten Vergleich mit *Clean Code* konnten wir sehr viele Gemeinsamkeiten feststellen. Bei einigen wenigen Punkten konnten wir Uneinigkeiten erkennen und diese genauer analysieren. Wir sind allerdings in diesen Fällen zum Schluss gekommen, dass uns die Denkschule von Robert Martin mehr überzeugt. In den späteren Kapiteln setzen aber auch diese beiden Werke einen unterschiedlichen Fokus. Während Robert Martin sich in Clean Code mehr auf Themen wie Unit Testing und Parallelität fokussiert, beleuchten Dustin Boswell und Trevor Foucher in *The Art of Readable Code* die Thematiken der Loops & Logik, sowie dem Refactoring genauer. Entsprechend konnten wir hier keine unterschiedlichen Auffassungen oder Meinungen mehr identifizieren.

Die Anforderungen eines Open Source Projekts waren dank der guten Übersichtsseite die Github bereitstellt schnell abgeklärt. Auch die Umsetzung der einzelnen Bestandteile bereitete keine Herausforderungen, da alle Informationen strukturiert verfügbar waren. Lediglich die Wahl der Lizenz stellte uns vor eine Herausforderung, da ein direkter Vergleich mehrerer Lizenzen untereinander sich als schwierig herausstellte. Dies, weil die Lizenzen jeweils unterschiedliche Fokusse auf Aspekte des Lizenzrechts legen. Mit der Apache Version 2 Lizenz haben wir aber schliesslich eine Lizenz gefunden, die gut zu unserem Projekt passt.

Kapitel 3

Konzepte zum Vermitteln der Clean Code Regeln

In diesem Kapitel werden wir ein didaktisches Konzept für das Wiki erarbeiten. Im Wiki erläutern wir die einzelnen Clean Code Regeln genauer, unter anderem mit konkreten Beispielen. Bei Verstössen kann in den Problemmeldungen die Wiki Seite referenziert werden, falls die Programmierenden weiterführende Erklärungen zu den Problemen suchen. Da gerade bei Programmierenden auf dem Anfängerniveau das allgemeine Verständnis der Thematik noch wenig ausgeprägt ist, sollen diese Wiki Seiten didaktisch sinnvoll aufgebaut werden.

3.1 Literaturrecherche zu erklärenden Konzepten

Um das Konzept für das Wiki auszuarbeiten, haben wir als erstes eine Quelle gesucht, an welcher wir uns orientieren können. Dabei haben wir zuerst in theoretischen Quellen nach einer Möglichkeit gesucht, ein praktisches Konzept abzuleiten. Daher haben wir unseren Fokus auf Literatur gelegt, die in einem ähnlichen Themenbereich Didaktik zur Vermittlung einsetzt. So sind wir auf das Buch *Entwurfsmuster von Kopf bis Fuß* [42] gestossen. Wir haben dieses Buch als Inspiration gewählt, da es primär für die gleiche Zielgruppe geschrieben wurde und Programmierenden die Anwendung von gängigen Entwurfsmustern beibringen soll. Die Art und Weise, wie die Autoren die einzelnen Entwurfsmuster erläutern variiert leicht von Muster zu Muster. Grundsätzlich sind sie aber sehr ähnlich zueinander aufgebaut.

Als konkretes Beispiel haben wir die Erklärung des Factory Musters in Kapitel 4 [42, S. 109] ausgewählt. Dieses Beispiel haben wir gewählt, weil dieses Kapitel bei uns selbst den besten Lerneffekt aufgewiesen hat. Alle Kapitel sind aber generell sehr identisch aufgebaut. Die Erläuterung beginnt mit einem kurzen Einführungstext und einer Grafik einer Bäckerin, mit der eine Analogie zu einer Pizzabäckerei aufgebaut wird. Ausserdem wird eine kurze Zusammenfassung zum Problem gegeben. Auf den nachfolgenden Seiten (S. 110–113) wird das Problem, unter Zuhilfenahme von (negativen) Code Beispielen genauer erläutert. Dabei wird die am Anfang aufgebaute Analogie einer Pizzabäckerei genutzt, um für den Leser verständliche Code Beispiele zu erzeugen. Nachdem das Problem ausreichend erläutert wurde, wird in dem mit Hilfe der Analogie aufgebauten Code, eine Lösung ausgearbeitet (S. 114). Somit stehen Beispiele für guten Code, welcher das Entwurfsmuster umsetzt zur Verfügung. Abgeschlossen wird die Erklärung mit einer Sektion für häufig gestellte Fragen und deren Antworten (S. 115). Dabei werden in den Fragen proaktiv auch Vor- und Nachteile, die beim Einsatz dieses Entwurfsmusters entstehen aufgezeigt.

3.2 Konzept für Erkärseiten

Unsere eigenen Wiki Seiten sollen einen didaktischen Mehrwert bieten und behandeln ein ähnliches Thema wie das oben erwähnte Buch. Wir haben uns daher dafür entschieden unser Konzept an demjenigen des Buchs anzulehnen, allerdings mit einigen Änderungen. Unsere Hilfsseiten sollen ebenfalls die Grundlagen vermitteln und als Nachschlagewerk dienen, müssen aber auch kompakt und übersichtlich sein und die wichtigsten Fakten einfach ersichtlich darlegen.

Ausgehend von diesen Überlegungen haben wir das in der Abbildung 3.1 dargestellte Konzept erarbeitet.

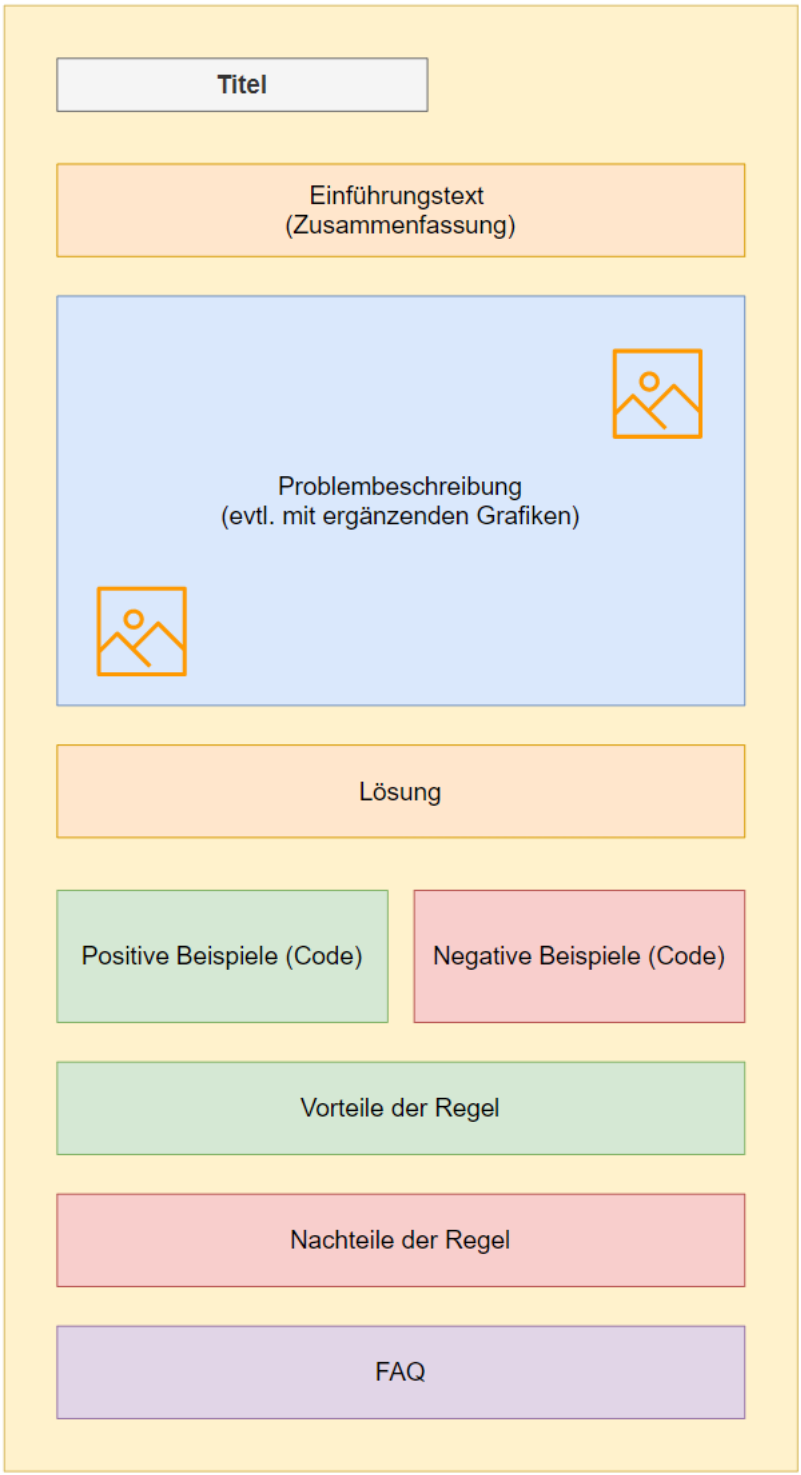


Abbildung 3.1: Konzept der Erklärseiten

Unsere Wiki Seiten beginnen mit einem Einführungstext, der im Stile einer Zusammenfassung formuliert werden soll. Danach folgt eine Problembeschreibung in Textform, die soweit möglich mit Grafiken und der Verwendung von Analogien ergänzt werden. Wir haben die Verwendung von Analogien bewusst als optional festgelegt, da es nicht zwingend bei jeder Regel möglich sein wird eine passende Analogie aufzustellen. Wir sind der Meinung, dass es in diesem Falle besser ist, keine Analogie zu verwenden, als die Verwendung einer verwirrenden Analogie zu erzwingen. Auf die Problembeschreibung folgt eine kurze Beschreibung der Lösung in Textform. Diese wird um positive und negative Codebeispiele ergänzt. Die Code Beispiele sollen sich dabei gegenübergestellt werden, damit für den Lesenden ein direkter Vergleich möglich ist. Darauf folgt eine Auflistung der Vor- und Nachteile, die bei der Verwendung dieser Regel entstehen. Dies hat zum Zweck das der Lesende besser abschätzen kann, ob der Einsatz der Regel für ihn bzw. sein Projekt Sinn macht oder nicht. Abgeschlossen wird die Wiki Seite mit einem FAQ, welches stetig ergänzt werden kann, um häufig auftretende Fragen direkt zu beantworten.

3.3 Umsetzung der Erklärseiten

Wir haben unsere Erklärseiten für die vom Plugin unterstützten Regeln gemäss dem Modell in Abbildung 3.1 umgesetzt. Die Seiten sind im öffentlichen Repository [1] einsehbar. Wir haben jeweils eine englische [2] und eine deutschsprachige [3] Version jeder Seite angefertigt. Dies geschah mit der Intention, Einsteigern unterschiedlicher Herkunft das Verständnis der Erklärung zu vereinfachen.

Bei der Umsetzung konnten wir uns meistens an das ausgearbeitete Modell, wie in Abbildung 3.1 dargestellt, halten. Wir haben allerdings festgestellt, dass es schwierig ist, eine Grafik zu finden die ergänzend zur Erklärung eingesetzt werden kann. Aufgrund von Lizenzbedingungen müsste eine solche Grafik zur freien Nutzung unter offener Lizenz zur Verfügung stehen oder selbst erstellt werden. Ausserdem haben wir kaum nützliche Analogien gefunden, die wir in den Erklärungen hätten verwenden können.

3.4 Diskussion

Dieser Teil der Recherchen für unsere Arbeit stellte die grösste Herausforderung dar. Wir taten uns anfänglich schwer gute Quellen zu finden, um daraus ein Modell für unsere Erklärseiten ableiten zu können. Dies lag vor allem daran, dass wir im Bereich von wissenschaftlichen Studien über die Frage wie man Programmieren unterrichtet gesucht haben. Daraus konnten wir aber keine Informationen identifizieren, die wir zum Ableiten eines praktischen Modells hätten verwenden können. Eine Lösung fanden wir durch einen Hinweis unseres Advisors dann an unerwarteter Stelle. Wir konnten ein Modell aus einem Buch, welches Entwurfsmuster von Programmiercode für Anfänger erklärt, ableiten. Daraus haben wir gelernt, dass man ein Modell nicht nur aus rein theoretischen Quellen ableiten, sondern auch praktische Anwendungsbeispiele zu Rate ziehen kann. In vielen Fällen führt der Ansatz mit praktischen Anwendungsbeispielen auch schneller zu einem brauchbaren Ergebnis. Optimalerweise werden beide Ansätze ergänzend zueinander eingesetzt.

Bei der Umsetzung der Erklärseiten hat uns das zuvor ausgearbeitete Modell sehr geholfen. Damit stand von Beginn an ein Bauplan unserer Wikiseite zur Verfügung. So konnten wir uns bei der Umsetzung auf den Inhalt konzentrieren und mussten uns nicht parallel um eine ansprechende Strukturierung kümmern. Dies hat den ganzen Prozess deutlich vereinfacht und dadurch auch merklich beschleunigt.

[1] https://gitlab.com/cleancode2/cleancode-plugin/-/tree/master/docs/wiki/
[2] https://gitlab.com/cleancode2/cleancode-plugin/-/tree/master/docs/wiki/english/
[3] https://gitlab.com/cleancode2/cleancode-plugin/-/tree/master/docs/wiki/german

Kapitel 4

Nutzertests

Vor der Veröffentlichung unserer Software soll diese von aussenstehenden Nutzern unserer Zielgruppe (ProgrammiererInnen) getestet werden. Damit stellen wir sicher, dass die Qualität und Nutzbarkeit unserer Software gewährleistet ist. In diesem Kapitel beschreiben wir unser Vorgehen bei den Nutzertests, deren Auswertung und die Verwertung der Ergebnisse.

4.1 Vorgehen

Beim Sammeln von Testdaten muss stets eine Abwägung zwischen Datenqualität und -quantität vorgenommen werden. Langzeittests mit mehreren Testpersonen würden uns die qualitativ hochwertigsten Daten liefern. Aufgrund des zeitlichen und organisatorischen Aufwands haben wir uns allerdings dazu entschieden, nur einen einzigen Langzeittest mit detaillierter Auswertung durchzuführen und diesen mit mehreren Kurzzeittests zur ergänzen. Dies sollte zu ähnlich umfassenden Rückschlüssen führen und einiges an Zeit einsparen, die wir in die Weiterentwicklung investieren können. Der Langzeittest soll dabei mit einem Programmiereinsteiger durchgeführt werden. So können wir auch den didaktischen Erfolg des Plugins messen und auswerten. Wir prognostizieren, dass Anfänger die meisten Vorteile aus der Nutzung des Plugins ziehen können. Diese Hypothese überprüfen wir während des durchgeführten Testverfahrens. Nach dem Langzeittest wird ein offenes Interview mit dem Tester geführt und protokolliert. Auf diesem Gespräch basiert die Auswertung des Langzeittests.

Wir führten die Kurzzeittests zweimal durch, weil wir möglichst früh Feedback einholen wollten, um im Rahmen dieser Arbeit darauf reagieren zu können. Die erste Testphase wurde in den Projektwochen zehn bis elf, bei einem unvollständigen Entwicklungsstand, mit der im Marketplace veröffentlichten Version 0.0.2 des Plugins durchgeführt. Die zweite Phase wurde in den Projektwochen 15 und 16, kurz vor Fertigstellung des Plugins, mit der Version 0.0.4 durchgeführt.

Die Langzeittestperson, hat zusätzlich in beiden Testphasen den Fragebogen der Kurzzeittests ausgefüllt.

4.1.1 Fragebogen

Für eine sinnvolle Auswertung der Daten, haben wir uns entschieden, einen einheitlichen Fragebogen für die Testpersonen im Kurzzeittest vorzubereiten. Für den Fragebogen haben wir uns von der von SurveyMonkey bereitgestellten Vorlage [45] inspirieren lassen und diese um weitere projektspezifische Fragen ergänzt. Unsere eigenen Fragen zielen darauf ab, zu erfassen, wie sich das Plugin auf den Programmierprozess auswirkt. Beispielsweise durch eine Verlangsamung oder Beschleunigung der Programmiergeschwindigkeit oder Störungen des Programmierenden durch Problemmeldungen.

Die Vorlage für den Fragebogen haben wir dem Anhang beigefügt.

4.1.2 Testgruppen

Wir haben für die Durchführung der Nutzertests drei Gruppen definiert (siehe Tabelle 4.1), in welche wir unsere Testpersonen einteilen konnten. Bei der Einteilung haben wir uns am Erfahrungsgrad und an den abgeschlossenen Ausbildungen im Informatikbereich orientiert. Diese Klassifizierung haben wir vorgenommen um bei der Auswertung

der Kurzzeittests die durchschnittliche Zufriedenheit jeder Gruppe, anhand des Grades an Expertise, erfassen und aufzeigen zu können.

Tabelle 4.1: Testgruppen

Gruppenname	**Ausbildungsgrad (in der Informatik)**	**Programmiererfahrung**
Einsteiger	Keine Abgeschlossene Informatik Ausbildung	< 2 Jahre
Fortgeschrittene	Abgeschlossene Berufslehre / Studium	2–12 Jahre
Experte	Doktortitel	> 12 Jahre

4.2 Hypothesen

Nachfolgend stellen wir Hypothesen bezüglich relevanter Themen des Plugins auf und stellen eine Ergebnisprognose für die Nutzertests auf. Nach Testabschluss werden wir bei der Auswertung die Hypothesen referenzieren und diese mit den tatsächlichen Ergebnissen vergleichen.

4.2.1 Installation der Software

Unser Plugin wird mittels eines einzelnen Klicks auf den *Install* Button aus dem Visual Studio Marketplace [1] installiert. Wir erwarten, dass unsere Testpersonen dies in beiden Testphasen als sehr einfach bewerten werden.

Zur Überprüfung dieser Hypothese, haben wir die Frage F01 *Wie einfach war die Installation unserer Software?* im Fragebogen platziert.

4.2.2 Benutzerfreundlichkeit

Zu Beginn der ersten Testphase wird noch wenig Dokumentation zur Funktionalität des Plugins für Endnutzer zur Verfügung stehen. Dieser Umstand kann zu Verwirrung führen und würde sich dadurch negativ auf die Benutzerfreundlichkeit auswirken. Wir erwarten daher in der ersten Testphase mittlere bis schlechte Bewertungen von unseren Nutzern zu diesem Punkt. In der zweiten Testphase wird deutlich mehr Dokumentation und generelle Hilfestellungen für den Benutzer vorhanden sein. Ausserdem werden wir zu diesem Zeitpunkt einen Grossteil der Rückmeldungen aus der ersten Testphase eingearbeitet haben. Wir erwarten daher in der zweiten Testphase gute Bewertungen zu diesem Punkt.

Zur Überprüfung dieser Hypothese, haben wir die Frage F02 *Wie benutzerfreundlich ist die Benutzeroberfläche unserer Software?* im Fragebogen platziert.

4.2.3 Laufzeitstabilität

Das Plugin verfügt über ein grundlegendes Exception Handling. Dies soll verhindern, dass das Plugin regelmässig crasht. Das Plugin wurde jedoch noch nie grossflächigen Tests unterzogen. Daher besteht die Möglichkeit, dass bei den Nutzertests Bugs gefunden werden, die zum Crash der Software führen. Wir erwarten daher, dass wir eher wenige Rückmeldungen über Crashes erhalten werden.

Zur Überprüfung dieser Hypothese, haben wir die Frage F03 *Wie oft bleibt unsere Software hängen oder stürzt sie ab?* im Fragebogen platziert.

4.2.4 Erfüllung des Einsatzzwecks

Unser Plugin verfügt momentan im Vergleich zu der Anzahl existierenden Clean Code Prinzipien immer noch über einen rudimentären Funktionsumfang. Entsprechend erwarten wir hier, dass unsere Testpersonen dies in beiden Test-

[1] https://marketplace.visualstudio.com/vscode

phasen mittelmässig bewerten werden, wobei sich die Wertung bei der zweiten Testphase, aufgrund des erhöhten Funktionsumfangs, leicht verbessern sollte.

Zur Überprüfung dieser Hypothese, haben wir die Frage F04 *Wie erfolgreich ist unsere Software, was die Erfüllung ihres Einsatzzwecks (Verbesserung der Codequalität) angeht?* im Fragebogen platziert.

4.2.5 Dokumentation

Während der ersten Testphase ist Dokumentation für den Benutzer nur unvollständig vorhanden. Wir erwarten daher mittelmässige bis schlechte Bewertungen unserer Testpersonen. Bei der zweiten Testphase sollte die Dokumentation deutlich erweitert und das Feedback aus der ersten Testphase eingearbeitet worden sein. Wir erwarten in der zweiten Testphase daher mehrheitlich mittelmässige und gute Bewertungen.

Zur Überprüfung dieser Hypothese, haben wir die Frage F05 *Wie nützlich ist die unserer Software beiliegende Dokumentation?* im Fragebogen platziert.

4.2.6 Verbesserungsvorschläge

Wir erwarten aufgrund des früheren Nutzerfeedbacks aus der Studienarbeit [41, S. 29], dass als Verbesserungsvorschläge die Ausweitung auf weitere zu prüfende Regeln und die Ausweitung der Unterstützung von weiteren Programmiersprachen vorgebracht werden. Ausserdem erwarten wir den Vorschlag einer Deaktivierungsmöglichkeit für einzelne Regeln, da nicht alle Regeln in jeder Situation bzw. für jedes Projekt zweckdienlich sind.

Zur Überprüfung dieser Hypothese, haben wir die Frage F11 *Wie können wir unsere Software Ihrer Meinung nach verbessern?* im Fragebogen platziert.

4.2.7 Weiterempfehlungen

Aufgrund der bisherigen Hypothesen schätzen wir die Wahrscheinlichkeit einer Weiterempfehlung des Plugins in seinem jetzigen Zustand als mittel bis gering ein. Wir gehen aber davon aus, dass sich die Bereitschaft zwischen der ersten und der zweiten Testphase, aufgrund der eingearbeiteten Verbesserungen, steigern wird.

Zur Überprüfung dieser Hypothese, haben wir die Frage F06 *Wie wahrscheinlich ist es, das Sie unsere Software anderen Personen weiterempfehlen würden?* im Fragebogen platziert.

4.2.8 Störende Meldungen

Wir prognostizieren, dass die meisten momentan implementierten Regeln von der Mehrheit der Programmierenden als sinnvoll erachtet werden. Trotzdem könnten Meldungen in Einzelfällen angezweifelt und daher als störend wahrgenommen werden. Wir gehen aber davon aus, dass solche Fälle selten vorkommen werden. Daher erwarten wir in beiden Testphasen positive Rückmeldungen.

Zur Überprüfung dieser Hypothese, haben wir die Frage F07 *Wie empfinden Sie die Meldungen unsere Software während der Programmierarbeit?* im Fragebogen platziert.

4.2.9 Veränderung der Softwarequalität

Da unser Plugin lediglich Probleme aufzeigt und den Code nicht automatisch verändert, sehen wir keinen Weg, wie unser Plugin die Softwarequalität aktiv verschlechtern könnte. Einzig eine vom Plugin verursachte Beeinträchtigungen des Programmierenden, die eine Verschlechterung der Konzentrationsfähigkeit nach sich ziehen und dadurch negative Auswirkungen auf die Codequalität haben, wären denkbar. Wir erwarten keine derartigen Störungen und rechnen daher mit leicht verbesserten oder gleichbleibenden Rückmeldungen in der Auswertung des zweiten Fragebogens. Die Antworten hängen auch von den Testpersonen bzw. ihrem individuellen empfinden zu diesem Punkt ab.

Zur Überprüfung dieser Hypothese, haben wir die Frage F08 *Wie sehr schätzen Sie, hat sich die Qualität des von Ihnen geschriebenen Codes während der Nutzung unserer Software verändert?* im Fragebogen platziert.

4.2.10 Effizienzveränderung

Da der Fragebogen überwiegend von Kurzzeittestenden ausgefüllt wird, gehen wir davon aus, dass die Testpersonen ihre Effizienz als unverändert oder leicht verschlechtert empfinden werden. Diese Annahme basiert auf der Tatsache,

dass der positive Effekt von Clean Code sich erst über eine längere Zeit ausspielt. Zum Beispiel wenn der Code eines Teammitgliedes überprüft wird, oder von einem Teammitglied erweitert werden soll. Da der Programmierende selbst seinen Code kurzfristig meistens sehr genau kennt und versteht, entstehen für ihn Effizienzvorteile ebenfalls erst über längere Zeit.

Zur Überprüfung dieser Hypothese, haben wir die Frage F09 *Wie hat sich Ihrem Gefühl nach die Geschwindigkeit mit der Sie Code schreiben können während der Nutzung des Plugins verändert?* im Fragebogen platziert.

4.2.11 Berechtige Meldungen

Da wir primär Regeln umgesetzt haben, welche wenig Interpretationsspielraum bei ihrer Evaluation zulassen, gehen wir von einer hohen Genauigkeit unserer Meldungen aus. Einige Regeln lassen aber auch mehr Interpretationsspielraum offen. Da jeder eine andere Auffassung von Clean Code vertritt, gehen wir davon aus, dass die Meldungen überwiegend als meistens korrekt klassifiziert werden.

Zur Überprüfung dieser Hypothese, haben wir die Frage F10 *Wie hilfreich schätzen Sie die Meldungen des Plugins ein?* im Fragebogen platziert.

4.2.12 Gruppenauswertung

Je grösser das Vorwissen um die Clean Code Prinzipien ist, desto grösser sollten gemäss unseren Erwartungen die Ansprüche an unser Plugin sein. Wir gehen daher davon aus, dass die durchschnittliche Bewertung in den Gruppen abnehmen wird, je grösser das Vorwissen ist. Entsprechend erwarten wir von der Einsteigergruppe die beste, von der Gruppe der Fortgeschrittenen die mittlere und von der Gruppe der Experten die niedrigste durchschnittliche Wertung. Generell rechnen wir mit minimalen Unterschieden zwischen den Testgruppen.

Wir nehmen an, dass sich die durchschnittliche Wertung zwischen der ersten und der zweiten Testphase bei allen drei Gruppen leicht verbessern wird, da dem Plugin nur Funktionen hinzugefügt werden und an der bestehenden Funktionalität, abgesehen vom Beheben von Fehlern, nichts abgeändert wird.

Zur Überprüfung dieser Hypothese, wird der Durchschnitt aller Fragen im Fragebogen für die jeweilige Gruppe berechnet und ausgewertet.

4.3 Durchführung

In der nachfolgenden Sektion beschreiben wir die Durchführung der Nutzertests. Mittels des Fragebogens, den unsere Testpersonen in zwei Testphasen ausfüllen werden, sollen unsere aufgestellten Hypothesen gemessen und entweder bestätigt oder widerlegt werden.

4.3.1 Testpersonen

Die in der Tabelle 4.2 aufgelisteten Testpersonen konnten wir für die Durchführung unserer Nutzertests rekrutieren. Wir haben dabei darauf geachtet, jede Gruppe mit mindestens einer Testperson zu repräsentieren.

Tabelle 4.2: Testpersonen

Name	**Testgruppe**	**Ausbildungsgrad (in der Informatik)**	**Programmiererfahrung**
C. Scheiwiller	Fortgeschrittene	Abgeschlossene Berufslehre, im Bachelorstudium	8 Jahre
F. Loch	Experte	Doktortitel, Professor für Informatik	15 Jahre
K. Greminger	Einsteiger	Keine, in der Berufsausbildung	$<$ 1 Jahr
J. Hauser	Fortgeschrittene	Abgeschlossene Berufslehre, im Bachelorstudium	8 Jahre

4.4 Auswertung

4.4.1 Auswertung der Kurzzeittests

In dieser Sektion stellen wir jeweils die Auswertung der ersten und der zweiten Testphase der Kurzzeittests nebeneinander und analysieren die Ergebnisse.

F01: Wie einfach war die Installation unserer Software?

Die Einfachheit der Installation des Plugins wird, wie in der Abbildung 4.1 zu sehen, in beiden Testphasen überragend positiv bewertet. Da das Plugin über den Visual Studio Marketplace [2] mittels eines Klicks installiert werden kann, entspricht hier das gemessene Ergebnis unserer Hypothese aus 4.2.1. Eine weitere Verbesserung könnte nur erreicht werden, wenn das Plugin zusammen mit der IDE vorinstalliert ausgeliefert würde.

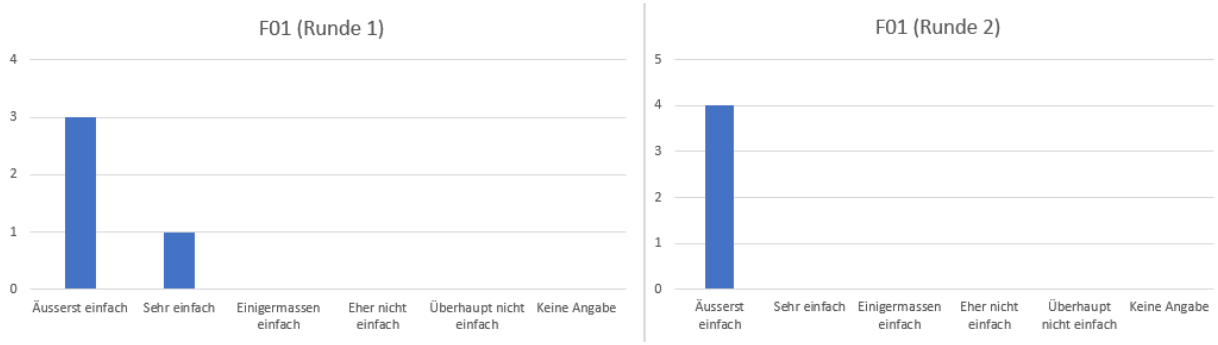


Abbildung 4.1: Frage 1, Testphasen 1 & 2

F02: Wie benutzerfreundlich ist die Benutzeroberfläche unserer Software?

Wie in der Abbildung 4.2 zu sehen ist, haben wir entgegen unseren Erwartungen, bereits in der ersten Testphase mehrheitlich mittlere Bewertungen für die Benutzerfreundlichkeit unseres Plugins bekommen. Die Begründung dazu war überwiegend, dass das Plugin die von der IDE bereitgestellten Möglichkeiten gut nutze, aber gleichzeitig nichts zu nennen ist, was aussergewöhnlich gut wäre, wodurch die Benutzerfreundlichkeit besser wahrgenommen wurde, als wir es initial angenommen hätten. Die einzige Person, welche eine schlechte Bewertung abgegeben hat, verstand mangels Dokumentation die Funktion des Plugins nicht und nahm an es funktioniere nicht, woraus die schlechte Bewertung resultierte.

In der zweiten Testphase haben wir mehrheitlich gute Bewertungen für die Benutzerfreundlichkeit bekommen, was laut Testpersonen an der Verfügbarkeit einer besseren Dokumentation lag, wodurch sich unsere Hypothese aus 4.2.2 in diesem Punkt bestätigt.

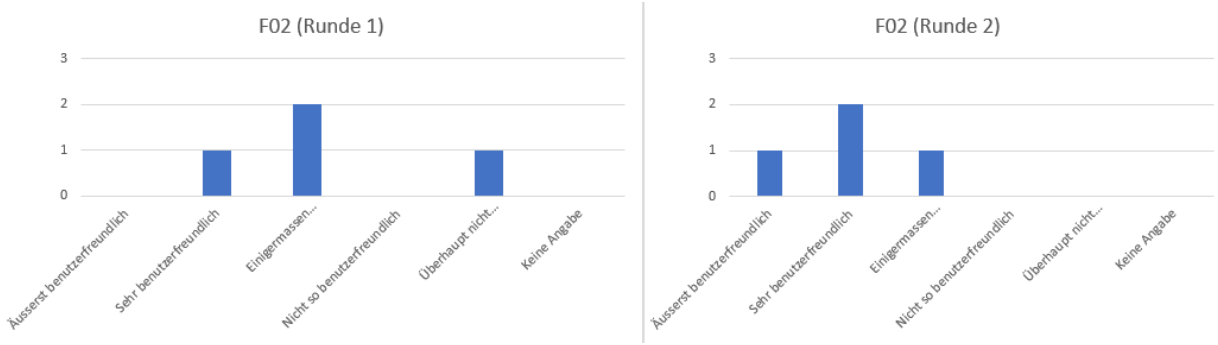


Abbildung 4.2: Frage 2, Testphasen 1 & 2

[2] https://marketplace.visualstudio.com/

F03: Wie oft bleibt unsere Software hängen oder stürzt sie ab?

Die Laufzeitstabilität des Plugins ist, wie in der Abbildung 4.3 ersichtlich, in beiden Testphasen sehr positiv bewertet worden. Eine ungünstige Formulierung der Frage sah keine Antwortoption für den Fall vor, dass das Plugin nie abstürzt, weshalb eine Testperson auf eine Angabe verzichtete, um deutlich zu machen, dass das Plugin nicht selten abstürzt, sondern nie. Wir haben daher bei allen Testpersonen die *selten* als Antwort zurückgegeben haben nachgefragt, worauf alle angaben, dass das Plugin im Testzeitraum nie abgestürzt ist. Die Auswertung fällt daher besser aus, als unsere in 4.2.3 aufgestellte Hypothese, da wir mit einigen wenigen Abstürzen gerechnet haben, woraus wir schließen, dass die eingebaute Fehlerbehandlung gut funktioniert.

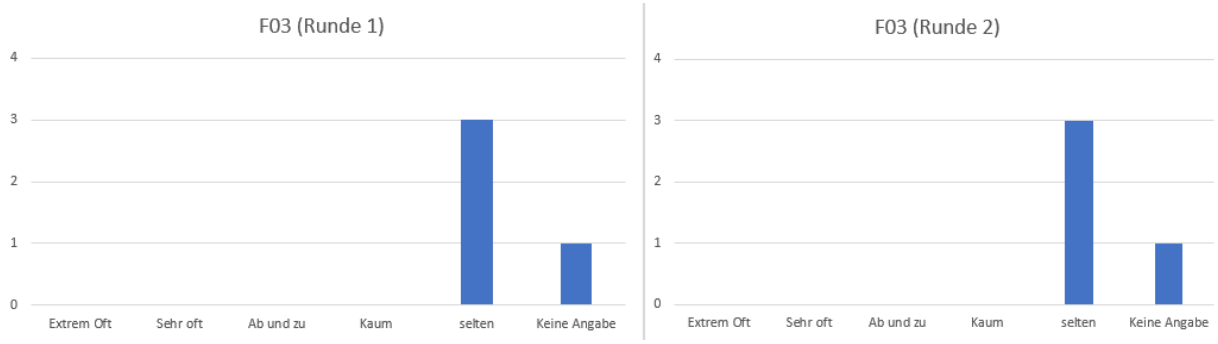


Abbildung 4.3: Frage 3, Testphasen 1 & 2

F04: Wie erfolgreich ist unsere Software, was die Erfüllung ihres Einsatzzwecks (Verbesserung der Codequalität) angeht?

Bei der Bewertung der Erfüllung des Einsatzzwecks haben wir, wie in der Abbildung 4.4 dargestellt, in der ersten Testphase, aufgrund geringen Umfangs und Funktionalität, mehrheitlich mittlere Bewertungen erhalten. In der zweiten Testphase haben wir durch den leicht gesteigerten Funktionsumfang mehrheitlich gute Bewertungen erhalten was unserer initialen Annahme der Verbesserung in der zweiten Phase entspricht. Unsere Hypothese aus 4.2.4 hat sich in diesem Punkt also bestätigt.

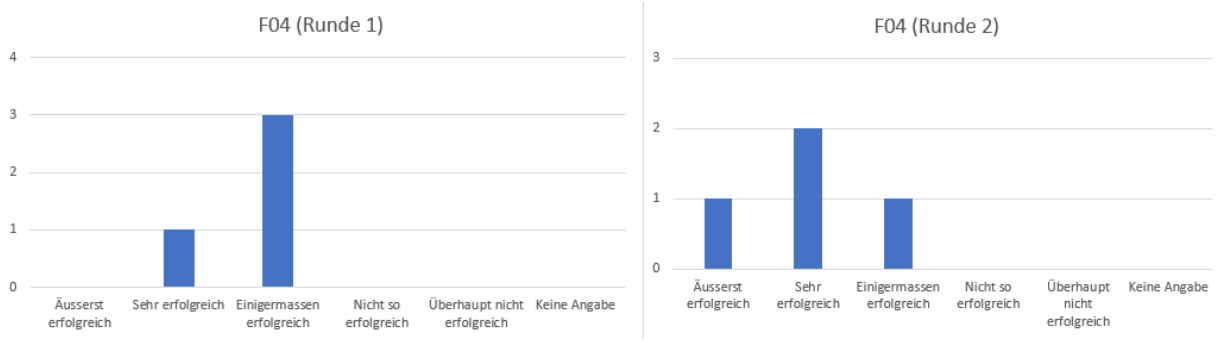


Abbildung 4.4: Frage 4, Testphasen 1 & 2

F05: Wie nützlich ist die unserer Software beiliegende Dokumentation?

Bezüglich der dem Plugin beiliegenden Dokumentation, lässt sich wie in der Abbildung 4.5 ersichtlich, eine deutliche Verbesserung der Bewertungen in der zweiten Testphase feststellen. Eine Testperson hat die Dokumentation in der ersten Testphase nicht berücksichtigen können und daher *Keine Angabe* gemacht. In der ersten Testphase wurde die Dokumentation mittelmässig bis schlecht bewertet und entspricht daher unseren Erwartungen. Das hinzufügen weiterer Dokumentation hat hier sichtlich geholfen, wie die Auswertung der zweiten Testphase zeigt. Die Auswertung entspricht daher unserer in 4.2.5 aufgestellten Hypothese.

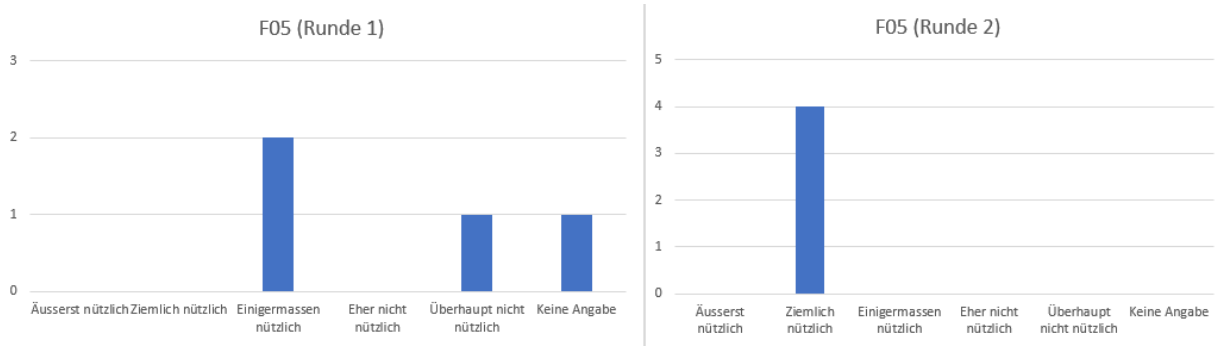


Abbildung 4.5: Frage 5, Testphasen 1 & 2

F06: Wie wahrscheinlich ist es, das Sie unsere Software anderen Personen weiterempfehlen würden?

Die Bereitschaft das Plugin an andere Personen weiterzuempfehlen war anfänglich, wie in der Abbildung 4.6 zu sehen eher gering. Als Begründung gaben unsere Testpersonen an, dass der Funktionsumfang noch zu beschränkt sei und noch zu viele kleine Fehler vorhanden seien. In der zweiten Testphase hat die Bereitschaft zur Weiterempfehlung stark zugenommen, was mit dem gesteigerten Funktionsumfang, dem Beheben der gemeldeten Fehler und der Einarbeitung der Rückmeldungen aus der ersten Testphase begründet wurde. Die Ergebnisse von beiden Testphasen entsprechen somit grundsätzlich unseren Erwartungen. Die Bereitschaft zur Weiterempfehlung hat sich zwischen den Testphasen aber stärker gesteigert als erwartet, daher folgern wir, dass das Plugin von unserer Testgruppe positiver wahrgenommen wird, als wir dies beim Aufstellen unserer Hypothese in 4.2.7 initial angenommen haben.

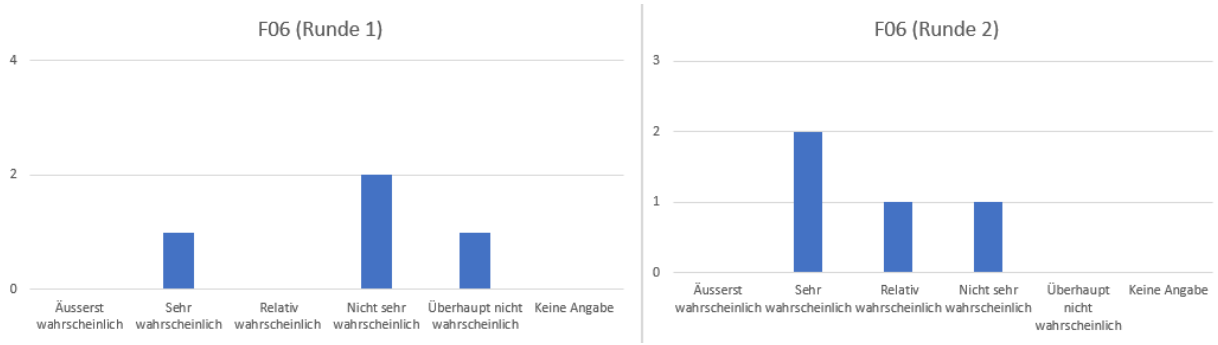


Abbildung 4.6: Frage 6, Testphasen 1 & 2

F07: Wie empfinden Sie die Meldungen unsere Software während der Programmierarbeit?

Wie in der Abbildung 4.7 dargestellt, haben wir in beiden Testphasen von allen Testnutzern die Antwort *wenig störend* erhalten, was genau unserer in 4.2.8 aufgestellten Hypothese entspricht. Die Meldungen wurden als *wenig störend* wahrgenommen, da es laut unseren Testpersonen doch hin und wieder zu fehlerhaften Meldungen gekommen ist, welche aber stets durch fehlende Wörter im Dictionary verursacht wurden.

Abbildung 4.7: Frage 7, Testphasen 1 & 2

F08: Wie sehr schätzen Sie, hat sich die Qualität des von Ihnen geschriebenen Codes während der Nutzung unserer Software verändert?

Während der ersten Testphase, haben wir, wie in der Abbildung 4.8 zu sehen, überwiegend die Rückmeldung *Teilweise verbessert* erhalten. Eine Testperson hat keine Angabe gemacht, denn sie hatte fälschlicherweise angenommen das Plugin funktioniere nicht korrekt und sah sich daher nicht im Stande, diese Frage zu beantworten. In der zweiten Testphase haben sich die Resultate kaum verändert, wir haben weiterhin überwiegend die Rückmeldung *Teilweise verbessert* erhalten. Begründet wurde dies in beiden Testphasen jeweils damit, dass einige der Regeln, allen voran die Benennungsregeln bereits zur Verbesserung beigetragen hätten, was unsere in 4.2.9 aufgestellte Hypothese bestätigt.

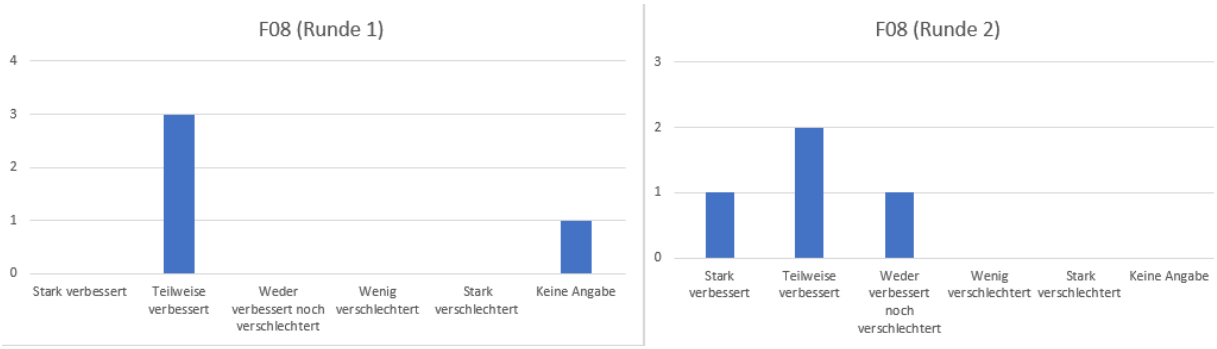


Abbildung 4.8: Frage 8, Testphasen 1 & 2

F09: Wie hat sich Ihrem Gefühl nach die Geschwindigkeit mit der Sie Code schreiben können während der Nutzung des Plugins verändert?

Wie in der Abbildung 4.9 dargestellt, haben sich die Mehrheit der Antworten in beiden Testphasen im Bereich *Weder beschleunigt noch verlangsamt* befunden. Auf Nachfrage haben die meisten unserer Testpersonen angegeben, dass das Plugin keinen Einfluss auf ihre Effizienz beim Schreiben von Code gehabt hat. Eine Testperson hat jeweils *Eher verlangsamt* als Antwort angegeben, da sie als Einsteiger mehr Meldungen vom Plugin erhielt als die anderen Testpersonen, entsprechend gab es im Vergleich auch mehr zu verbessern und mehr nachzulesen. Diesen Umstand führen wir auf das unterschiedliche Niveau an Vorwissen zurück, daher bestätigt sich das Ergebnis dieser Auswertung unsere Hypothese aus 4.2.10.

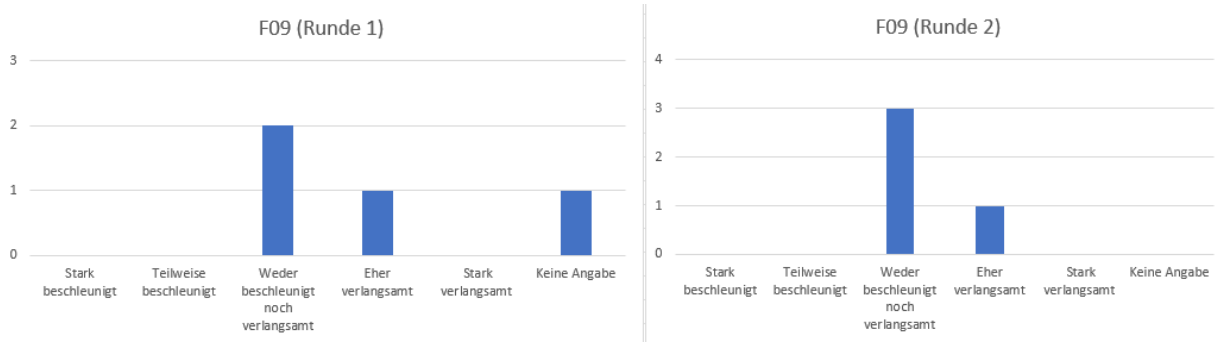


Abbildung 4.9: Frage 9, Testphasen 1 & 2

F10: Wie hilfreich schätzen Sie die Meldungen des Plugins ein?

In beiden Testphasen wurde von allen Testpersonen, wie in der Abbildung 4.10 zu sehen, die Antwort *Meistens Hilfreich* ausgewählt. Auf Nachfrage gaben unsere Testpersonen an, dass Meldungen die sie als inkorrekt eingestuft hätten, meistens auf Fehler wie z.B. fehlende Wörter im Dictionary zurückzuführen waren, was unsere in 4.2.11 aufgestellte Hypothese bestätigt.

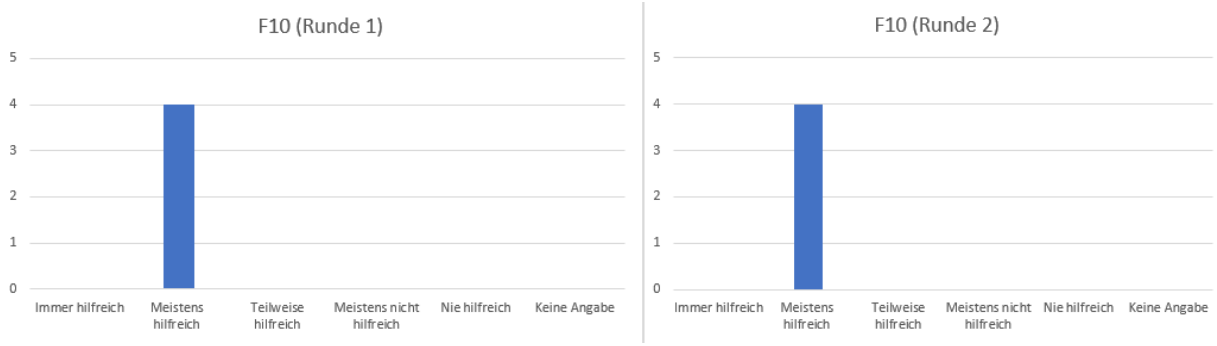


Abbildung 4.10: Frage 10, Testphasen 1 & 2

F11: Wie können wir unsere Software Ihrer Meinung nach verbessern?

Bei der Befragung unserer Testpersonen, haben wir um Verbesserungsvorschläge gebeten. Die Auswertung dieser Vorschläge, sowie deren Umsetzung, haben wir in der Tabelle 4.3 festgehalten.

Im Zusammenhang mit der Durchführung der Tests wurden uns auch einige Bugs und unerwünschtes Verhalten gemeldet, welche wir in der Tabelle 4.4 aufgelistet haben.

Tabelle 4.3: Verbesserungsvorschläge

Vorschlag	**Anzahl Nennungen**	**Umsetzung**	**Bemerkungen**
Dokumentation ausweiten	2	Dokumentation erweitert & vertieft	Wiki aus Readme verlinkt, Roadmap hinzugefügt
Funktionsumfang erweitert	3	Hinzufügen der LawOfDemeter Regel, Konfigurationsoptionen, sowie einer White- und Blacklist	Der weitere geplante Funktionsumfang kann der Roadmap im Repository entnommen werden
Einzelne Regeln deaktivierbar machen	1	Hinzufügen dieser Option im Konfigurationsfile	—

Tabelle 4.4: Gemeldete Probleme

Problembeschreibung	**Lösung**	**Bemerkung**
Wörter auf der Whitelist lösen trotzdem falsche Worttypmeldungen aus (Nomen / Verb).	Ein Wort auf der Whitelist gilt neu als Wildecard. Also als Nomen und als Verb. So werden diese Meldungen verhindert.	—
Wörter auf der Blacklist lösen trotzdem noch andere Problemmeldungen aus.	Wenn ein Wort auf der Blacklist ist, werden keine anderen Wortspezifischen Fehlermeldungen mehr ausgegeben.	Damit ist die Blacklistmeldung die Einzige, die angezeigt wird.
Gebräuchliche Wörter wie *firstname* (zusammengeschrieben) sind nicht im Dictionary vorhanden.	Die Wörter wurden dem Dictionary manuell hinzugefügt.	—

4.4.2 Gruppenauswertung

Bei der Gruppenauswertung zur durchschnittlichen Bewertung wurden alle Fragen gleich gewichtet. Aus den 5 möglichen Antworten gab die bestmögliche Antwort jeweils fünf Punkte und die schlechtes möglichste jeweils einen Punkt, wobei das Ergebnis jeweils auf einen Zehntel gerundet wurde. Im Falle der Frage 03 (Wie oft ist das Plugin abgestürzt) wurde keine Angabe als fünf Punkte gewertet, da es gemäss Nachfrage nie zu einem Absturz kam. Bei allen anderen Fragen wurde die Antwort nicht berücksichtigt, wenn in der Bewertung keine Angabe stattfand.

Um eine statistisch signifikante Antwort auf unsere beiden Hypothesen aus 4.2.12 abgeben zu können, war die durch uns befragte Testgruppe mit lediglich 4 Personen deutlich zu klein. Die beiden Hypothesen können daher hier nicht belegt oder widerlegt werden. Wir möchten die Ergebnisse trotzdem aufführen, da sich Rückschlüsse daraus ziehen lassen, wenn auch nicht im Bezug auf die beiden ursprünglich aufgestellten Hypothesen.

Wie in der Abbildung 4.11 zu sehen, hat sich die durchschnittliche Bewertung in der zweiten Testphase im Vergleich zur ersten Testphase um ca. 0.5 beziehungsweise 10% der möglichen Punkte erhöht. Dies bestätigt unsere Beobachtung in den Auswertungen der anderen Fragen, dass sich das Plugin zwischen beiden Testphasen messbar verbessert hat. Da alle Testgruppen uns in beiden Testphasen nahe beieinander liegende Bewertungen gegeben haben, gehen wir davon aus, dass das Plugin über alle Erfahrungsniveaus hinweg ähnlich wahrgenommen wird.

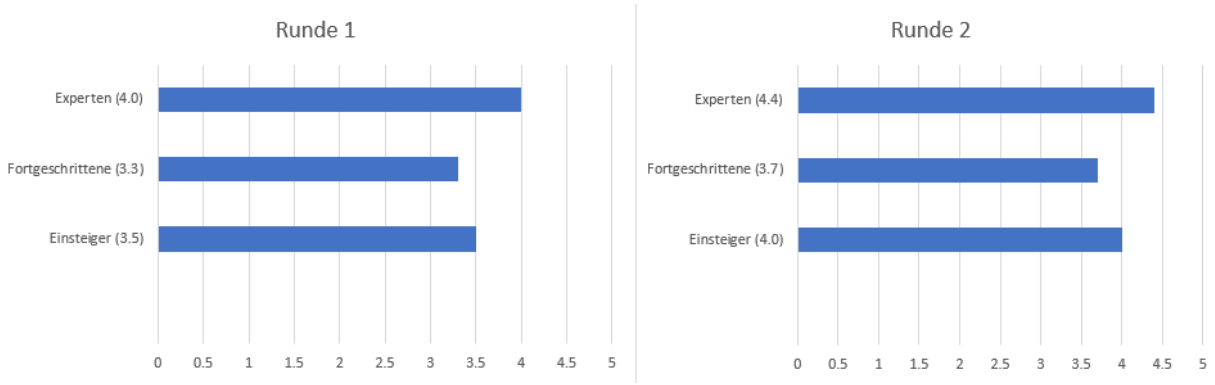


Abbildung 4.11: Gruppenauswertung, Testphasen 1 & 2

4.5 Auswertung des Langzeittests

Der Langzeittest wurde nach dem Ende des Testzeitraums von ca. 4 Wochen mittels eines Interviews ausgewertet. Im Anhang findet sich ein Protokoll des Interviews, auf welches wir in der nachfolgenden Sektion referenzieren.

4.5.1 Erkenntnisse

Die wichtigste Erkenntnis welche wir der Auswertung (*Frage 3*) entnehmen konnten, war, dass ein messbarer Lerneffekt bei der Nutzung unseres Plugins stattgefunden hat. Hat unsere Testperson zu Beginn des Testzeitraums noch viele Meldungen über problematische Benennungen von Variablen und Funktionen bekommen, haben sich diese im Laufe des Testzeitraums in ihrer Zahl immer weiter reduziert. Gemäss der Selbsteinschätzung der Testperson hat hier also definitiv ein Lerneffekt stattgefunden, wodurch positive didaktische Effekte im Hinblick auf Clean Code Prinzipien bei der Nutzung des Plugins festgehalten werden können. Dies bezieht sich jedoch nur auf die Regeln zur Benennung von Variablen und Funktionen, da die Testperson trotz des längeren Testzeitraums für eine fundierte Aussage nicht oft genug mit den anderen Regeln in Kontakt kam (*Nachfrage 3A*).

Bezüglich der Didaktik konnten wir in der Auswertung (*Frage 4 und Nachfrage 4A*) auch feststellen, dass die Erklärseiten, welche bei den Problemmeldungen verlinkt sind, ihren Zweck erfüllen. Demnach sind sie für einen Anfänger verständlich aufgebaut, klar formuliert und enthalten keine verwirrenden Textpassagen. Weiter verbessert werden könnten die Seiten, nach Meinung der Testperson, indem UML Klassendiagramme dem Beispielcode beigefügt und wo möglich weitere grafische Veranschaulichungen der Prinzipien hinzugefügt würden.

Laut Testperson (*Fragen 1, 2, 6, 7 und 8*) ist unser Nutzererlebnis, von einigen kleineren Bugs abgesehen, auch bei längerer Nutzungszeit durchwegs positiv. Dies lässt darauf schliessen, dass das verbesserte Exception handling korrekt funktioniert. Unsere Testperson würde das Plugin in der derzeitigen Form ausserdem allen Programmierenden, unabhängig ihres Kenntnisstandes weiterempfehlen, was ebenfalls auf eine grundlegende Zufriedenheit mit dem Tool schliessen lässt.

Laut Auswertung (*Frage 5 und Nachfrage 5A*) hat sich die Dokumentation über den Testzeitraum merklich verbessert. Zu Beginn noch kaum vorhanden und unvollständig, wird diese nun als solide eingestuft. Weitere konkrete Verbesserungsvorschläge für die Dokumentation wurden von der Testperson benannt:

- Law of Demeter check Limitationen (genauere Beschreibung des aktuellen Standes im Readme)
- Direkte Verlinkung des Readme aus dem Wiki heraus zur direkten Ersichtlichkeit der Regeln
- Erweiterung um eine Roadmap (Ausblick auf die Zukunft des Plugins)

Zuletzt konnten wir in der Auswertung die folgenden generellen Verbesserungsvorschläge aufnehmen:

- Erweiterung des Plugins um weiteren Funktionsumfang (weitere Regeln die geprüft werden)
- Hinzufügen einer Möglichkeit einzelne Problemmeldungen zu unterdrücken
- Ausweitung des Sprachsupports auf weitere Programmiersprachen

4.6 Diskussion

Die Durchführung der Nutzertests hat einiges mehr Zeit in Anspruch genommen, als erwartet. Unsere saubere, strukturierte und vor allem einheitliche Durchführung des Testprozesses mit jeder Testperson war wichtig, um die Ergebnisse miteinander vergleichen zu können, gleichzeitig aber auch zeitaufwändig. Eine wichtige Erkenntnis war der Fehler bei Befragung und Auswertung auf Word und Excel zu setzen, statt auf Tools wie Limesurvey [3] zurückzugreifen, welche diesen Prozess unterstützt und vereinfacht hätten.

Des Weiteren haben wir bei der Frage über die Laufzeitstabilität die beiden theoretisch validen Optionen, dass das Plugin bei einer Testperson entweder nie abstürzt oder gar nicht läuft nicht ausreichend in Betracht gezogen. Dieser Umstand irritierte einige unserer Testnutzer, da sie nicht wussten, ob sie nun angeben sollten, dass das Plugin selten abstürzt oder einfach keine Angabe machen sollten was uns die Auswertung erschwerte. Folgend sollte man sich beim Formulieren von Multiple Choice Fragen immer die beiden möglichen Extreme überlegen und in den Antwortmöglichkeiten abbilden.

Ein weiteres Problem war die kleine Anzahl Testpersonen, was es uns erschwerte, statistisch signifikante Aussagen zu einer befragten Thematik machen zu können. In unserer ersten Testphase hat eine Person fälschlicherweise angenommen, das Plugin funktioniere nicht wie gewünscht und hat dies entsprechend vermerkt. Solche Missinterpretationen einer Testperson führen zu grossen Auswirkungen in der Auswertung, da bei vier Testpersonen eine schlechte Bewertung bereits stark ins Gewicht fällt.

Die Ergebnisse entsprachen meist unseren Erwartungen und im Falle einer falschen Hypothese fielen sie jeweils besser aus als angenommen. Das aus den Befragungen gewonnene Feedback war enorm wertvoll da es uns dabei half wichtige Features und vorherrschenden Probleme für den Endnutzer zu identifizieren und zu priorisieren. Viele dieser Informationen wären uns ohne die Durchführung von Nutzertests nie zur Verfügung gestanden und hätten zu einem schlechteren Gesamtergebnis geführt. Mit der beschriebenen Durchführung konnten wir sicherstellen, dass die wichtigsten Features zum Ende des Projektes implementiert und die drängendsten Probleme behoben wurden.

[3] https://www.limesurvey.org/de/

Kapitel 5

Technische Umsetzung

In diesem Kapitel gehen wir auf Details der technischen Umsetzung, Softwarearchitektur und die zugrundeliegenden Entscheidungsfindungen ein. Dem Anhang liegt eine detaillierte Architekturdokumentation bei.

5.1 Vorgehen

Bei der Umsetzung sind wir gemäss Projektplan vorgegangen, welcher zu Beginn des Projekts definiert wurde. In einem ersten Schritt haben wir den bestehenden Prototyp des Plugins genau analysiert und die grundlegende Architektur der Blacklist und Whitelist, des Language Servers, sowie der Webhilfe grob festgelegt. Darauf folgte die Einrichtung verschiedener Git Repositories, in welchen unterschiedliche Aspekte unserer Arbeit organisiert wurden, sowie die Erhebung funktionaler und nichtfunktionaler Anforderungen und die Realisierung des Language Servers. Die eigentlich zu diesem Zeitpunkt geplante Umsetzung der Blacklist & Whitelist wurde aufgrund von unerwartetem Arbeitsaufwand um ca. 3 Wochen verschoben. Danach wurden bereits identifizierte Clean Code Regeln, sowie eine Webhilfe die diese Regeln erklärt umgesetzt und später im Projekt Nutzertests in zwei Phasen durchgeführt, wobei die Probanden unterschiedliche Programmiererfahrung hatten. Abschliessend wurde deren Feedback soweit möglich in das Plugin eingearbeitet und das Plugin als Open Source Projekt der weltweiten Gemeinschaft zur Verfügung gestellt.

5.2 Umsetzung

In dieser Sektion werden Umsetzung des Projektes und Umgang mit damit einhergehenden Engineering Problemen aufgezeigt.

5.2.1 Eingesetzte Werkzeuge

Für die Programmierarbeiten haben wir die Visual Studio Code IDE eingesetzt. Als Versionsverwaltungstool kam Git zum Einsatz. Als Build Tool kam npm zum Einsatz, da Microsoft dieses für Visual Studio Code Extensions voraussetzt bzw. Projekte direkt damit initialisiert werden. Die Repositories wurden für den längsten Teil der Projektzeit auf einer privaten GitLab Instanz gehostet. Zum Ende des Projekts wurde der Code und die Dokumentation dann in ein öffentliches Repository auf der offiziellen GitLab Instanz verlegt. Für die Continuous Integration und das Artefakte Repository wurden die von GitLab bereitgestellten Tools (GitLab CI/CD, GitLab Variables) der öffentlichen Gitlab Instanz verwendet. Für die statische Code Analyse wurden EsLint [1] und SonarQube [2] eingesetzt. Die Dokumentation, sowie die Erklärseiten wurden in Markdown geschrieben. Für das automatische Testen des Quellcodes kamen das Testing Framework Mocha [3], die Assertion Library Chai [4] und Istanbul [5] zur Berechnung der Testabdeckung zum Einsatz.

[1] https://eslint.org/
[2] https://www.sonarqube.org/
[3] https://mochajs.org/
[4] https://www.chaijs.com/
[5] https://istanbul.js.org/

5.2.2 Infrastruktur

Im Rahmen des Projekts haben wir drei Repositories auf einer selbst gehosteten GitLab Instanz aufgesetzt, deren jeweiliger Einsatzzweck in der Tabelle 5.1 dargestellt ist. Zeitweise existierte ein viertes Repository namens *cleancode-wiki* in welchem während der Testphase die Wikiseiten abgelegt waren. Dieses musste erstellt werden, weil die Wikiseiten für die Tests öffentlich zugänglich sein mussten und alle anderen Repositories auf privat eingestellt waren. Das öffentliche Repository existierte zu diesem Zeitpunkt noch nicht. Der Inhalt wurde später, genau wie die Architekturdokumentation, in das öffentliche Repository überführt.

Tabelle 5.1: Projektrepositories

Namen	**Zweck**	**Bemerkungen**
cleancode-plugin	Verwaltung des Quelltextes	Veröffentlichtes Repository
report	LaTeX Files dieses Berichts	Privates Repository
architecture-documentation	LaTeX Files der Architekturdokumentation	Wurde ins cleancode-plugin Repository integriert

Pipeline

Beim Code Repository nutzen wir die von GitLab bereitgestellte GitLab CI für unsere Pipeline, welche wie in Abbildung 5.1 grafisch veranschaulicht, aus den vier Schritten (*lint, test, sonarcloud-check und publish*) besteht. Im *linting* Schritt wird der von uns Verwendete Linter EsLint ausgeführt. Im *test* Schritt werden die Unittests ausgeführt. So stellen wir sicher, dass nur Code der Codebasis hinzugefügt wird, der keine durch unsere Unittests abgedeckte Funktionalität zerstört. Im *sonarcloud-check* Schritt wird eine statische Codeanalyse mit SonarQube ausgeführt, welche automatisch problematische Codestellen findet. Der Code wird damit also auf potenzielle Sicherheitslücken, Code Smells, Bugs und weitere Code Metriken geprüft. Wir nutzen dabei die Sonarcloud Instanz, die für Open Source Projekte kostenlos verwendet werden kann. Entsprechend ist die Auswertung der Codequalität öffentlich einsehbar [6]. Diese drei Schritte zusammen erlauben es uns, zu jedem Zeitpunkt eine angemessene Codequalität sicherstellen zu können. Die ersten drei Schritte werden bei jedem Merge Request sowie bei jedem Commit in die Branches Master und Develop automatisch ausgeführt. Der vierte Schritt wird nur auf getagten Commits ausgeführt, er dient dazu, dass Plugin automatisch im Visual Studio Marketplace zu veröffentlichen.

Weitere Details zur Infrastruktur und Pipeline können der Architekturdokumentation im Anhang entnommen werden.

5.2.3 Architektur

Eine Übersicht der Grobarchitektur zeigt die Abbildung 5.2.

Ausgangslage

Die rein clientseitige Erweiterung aus der Studienarbeit ist aufgeteilt in drei Untersysteme zum Parsen des Quelltextes, Überprüfen der Clean Code Regeln und Benachrichtigen der Programmierenden. Die Hauptlogik der Erweiterung verwaltet diese drei Systeme und stösst sie an, wenn sie benötigt werden. Je nach Einstellung geschieht dies bei Änderungen an einer Quell-Datei, beim Speichern einer Quell-Datei oder wenn die Prüfung manuell angestossen wird.

Das System zum Parsen der Quelltexte instanziert den korrekten Parser aufgrund der verwendeten Programmiersprache (aktuell nur Java) und extrahiert alle zur Regelprüfung nötigen Informationen aus dem Programmcode. Diese bereitet es dann in Form von Knoten auf, welche von der Prüflogik weiterverwendet werden können.

Die Prüflogik führt je nach Art von Knoten die benötigten Regelprüfungen durch und sammelt alle Verstösse. Diese werden mit Informationen über den Regelverstoss angereichert und am Ende der Prüfung gesammelt zurückgegeben.

[6] https://sonarcloud.io/summary/overall?id=cleancode2_cleancode-plugin

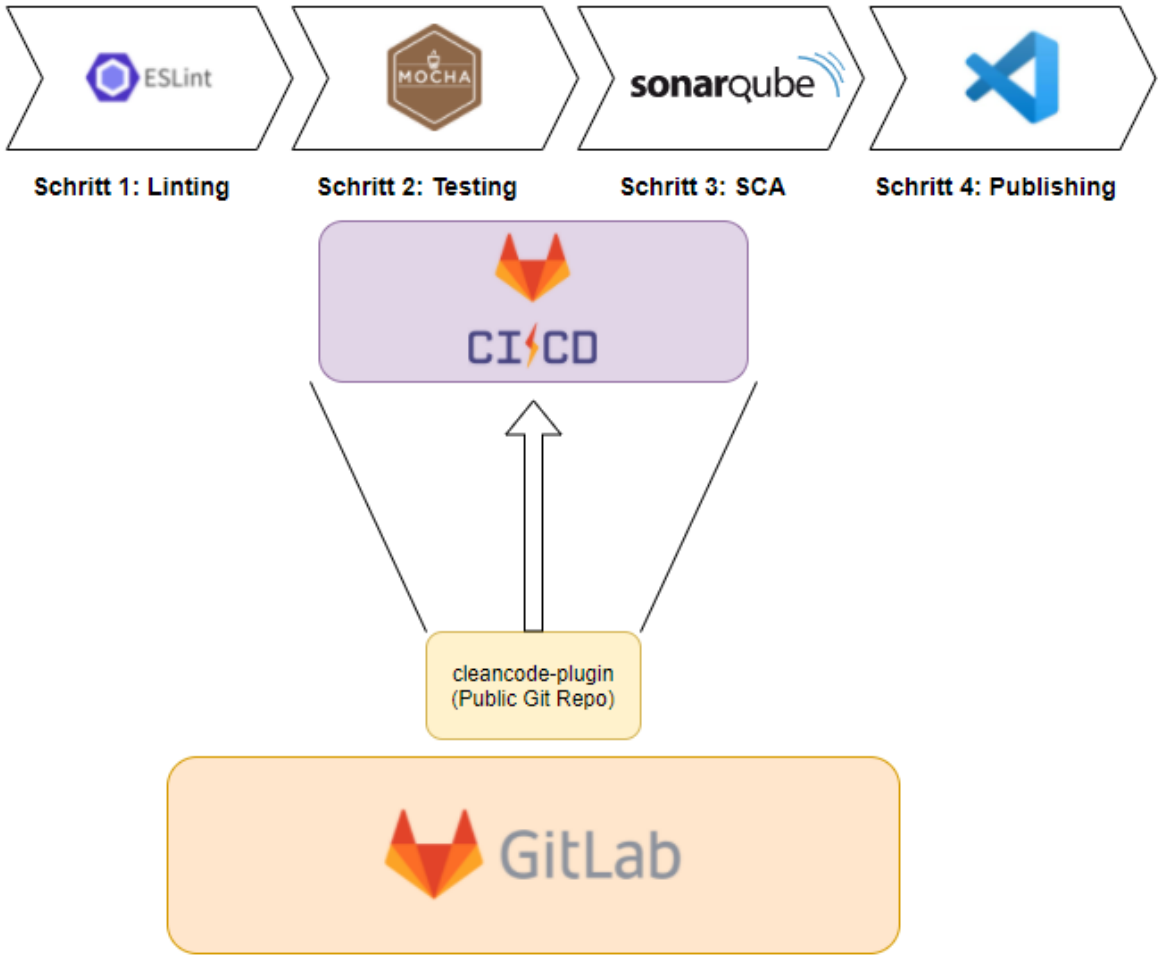


Abbildung 5.1: Pipelineschritte

Das Benachrichtigungssystem wandelt die gesammelten Regelverstösse in Diagnosemitteilungen um welche von der Entwicklungsumgebung angezeigt werden können und liefert diese zurück and die IDE.

Erweiterung

Wir haben uns dafür entschieden die Erweiterung in eine Languageserver Erweiterung [7] auszubauen. Die Hauptgründe hierfür waren die Auslagerung der Prüflogik in einen eigenen Thread, um zu verhindern, dass langlaufende Berechnungen die Entwicklungsumgebung einfrieren lassen, sowie die Logik der Erweiterung von Visual Studio Code abzukoppeln, sodass diese auch, wie in der Abbildung 5.3 aufgezeigt, für andere Entwicklungsumgebungen als Sprachserver zur Verfügung stehen könnte. Die Erweiterung ist nun aufgeteilt in zwei Teile, einen Client und einen Server. Die Client Erweiterung wird wie eine normale Visual Studio Code Erweiterung in der Entwicklungsumgebung ausgeführt, während der Server auf einem eigenen Thread gestartet wird. Beide kommunizieren über Node IPC miteinander mittels dem language server protocol.

Der Client ist sehr leichtgewichtig und besteht hauptsächlich aus dem Verbindungsaufbau zum Server und einigen Funktionen zur Konfiguration. Der Server beinhaltet die komplette Parse- und Prüflogik und hat daher eine sehr ähnliche Architektur wie das Plugin aus der Studienarbeit, allerdings wird die Logik nicht mehr direkt von Visual Studio Code angestossen, sondern die Entwicklungsumgebung schickt bei Änderungen eine aktuelle Version einer Quell-Datei an den Server, welcher dann daraufhin die Prüfung ausführt. Ebenfalls überwacht der Client ob sich an den Einstellungen etwas ändert und meldet dies dem Server. Dieser führt daraufhin die Regelprüfung auf allen geöffneten Dateien erneut durch, da gewisse Einstellungen das Resultat der Prüfung verändern können.

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

5.2.4 Erweiterungseinstellungen

Für die Benutzerfreundlichkeit der Erweiterung haben wir einige Einstellungen zur Verfügung gestellt mit denen die Regelprüfungen und Rückmeldungen feinjustiert werden können. Grundsätzlich können diese Optionen über die Einstellungen von Visual Studio Code genutzt werden, woraufhin sie im globalen settings.json für die Maschine abgelegt werden. Um die Einstellungen pro Projekt individuell vornehmen und mit einem Team teilen zu können, besteht die Möglichkeit eine erweiterungsspezifische Einstellungs-Datei im Root-Ordner des Projekts zu erstellen. Befindet sich eine solche Datei im geöffneten Projekt, werden die globalen Einstellungen ignoriert.

Black- und Whitelist

Wörter, welche nicht im Wörterbuch vorhanden sind, aber trotzdem korrekt oder zumindest nicht als falsch ausgewiesen werden sollen, können der Whitelist hinzugefügt werden. Bei der Prüfung auf Wortkorrektheit und angemessene Bezeichnung (Klassennamen, Methodennamen) werden diese Worte immer als passend angesehen. Umgekehrt können Wörter, die zwar eigentlich korrekt sind, aber nicht verwendet werden sollen, der Blacklist hinzugefügt werden, wodurch diese beim Prüfen von Bezeichnern danach als fehlerhaft erkannt und dementsprechend ausgewiesen werden. Bei Wörtern die sowohl auf der White- wie auch der Blacklist sind, wird die Blacklist als übergeordnet gewertet und das Wort als fehlerhaft ausgewiesen.

Rückmeldungslevel einzelner Regeln

Für die verschiedenen Regelverstösse (Blacklist, Anzahl Funktionsparameter, Law of Demeter, Benennung von Bezeichnern, Wortart) kann individuell konfiguriert werden ob und wie diese Verstösse geprüft und ausgewiesen werden sollen. Die Prüfung kann entweder komplett ausgeschaltet werden, oder die Rückmeldung kann als Warnung oder Fehler eingestellt werden.

5.2.5 Herausforderungen

Folgende Herausforderungen traten während der Umsetzung des Plugins auf.

Weblinks

Da die Darstellungsmöglichkeiten in den Problemmeldungen der Visual Studio Code IDE auf reinen Text eingeschränkt ist, wurden die detaillierten Erklärungen auf webbasierte Erklärseiten ausgelagert. Die Diagnostics API, welche verwendet wird, um die Problemmeldungen an die IDE zu übermitteln verfügt bereits über ein Attribut, worüber sich ein Weblink mitgeben lässt. Leider wurde dieses Attribut bisher nicht im Language Server Package implementiert. Laut den Entwicklern ist dies zwar seit mehr als zwei Jahren geplant, wurde aber bis heute nicht realisiert [8]. Da eine Abkehr von einem Language Server Plugin inakzeptabel war, mussten die Links wie in der Abbildung 5.4 zu sehen in die Textbeschreibung der Problemmeldung integriert werden. Dies hat zum Nachteil, dass der Entwickelnde die URL manuell kopieren und im Browser einfügen muss, was sich schlecht auf die Benutzererfahrung auswirkt.

Die Textimplementation wurde über ein *.env* File realisiert, worin der gemeinsame Hauptteil der URL, sowie die individuellen URL Teile zu den Erklärseiten gespeichert sind. Dadurch ist eine einfache Anpassung der URL bzw. der URL Bestandteile an einem zentralen Ort möglich.

Kommentarregel

Im Rahmen der Arbeit war eine Regel geplant, die bei Kommentaren eine Warnung ausgibt, da die meisten Kommentare nicht notwendig sind, bzw. sich durch besseren Code ersetzen lassen. Für die Umsetzung dieser Regel sind wir auf den von uns in diesem Projekt verwendete Java Parser angewiesen. Dieser muss im erstellten konkreten Syntax Baum Kommentarnodes erstellen, damit wir diese mit dem von uns implementierten Visitor finden können. Diese Nodes werden leider von unserem Parser nicht angelegt. Das Fehlen dieser Nodes verunmöglicht leider die Implementation dieser Regel. Mögliche Lösungsansätze wären das Schreiben eines eigenen Parsers oder das Wechseln auf einen anderen Parser, was beides im Rahmen dieser Arbeit nicht möglich war.

Laden des JSON Wörterbuchs in den Unit Tests

Die Art wie die automatisierten Tests ausgeführt werden änderte sich im Vergleich zur Studienarbeit. Nutzten wir für unsere Unit-Tests damals noch die Strukturen von Visual Studio Code die eigentlich für Integrationstests gedacht

[8] https://github.com/microsoft/vscode/issues/11847

waren, wollten wir diese bei der Bachelorarbeit isoliert ausführen, ohne dass im Hintergrund eine Instanz der Entwicklungsumgebung hochgefahren wird. Dies führte dazu, dass alle Tests, die vom Wörterbuch abhängen nicht mehr funktionierten.

Nach einer Analyse des Problems stellten wir fest, dass der Inhalt der JSON-Datei bei Testläufen als `undefined` erkannt wurde. Wir probierten verschiedene Einstellungsmöglichkeiten beim Build und bei unserer Testbibliothek, welche leider alle das Problem nicht beheben konnten.

Wir lösten das Problem, indem wir die Verantwortung für das Laden des Wörterbuchs aus den Klassen, welche dieses verwenden, in deren Aufrufer verschoben und so beim Testen nicht auf unsere Wörterbuch-Datei angewiesen waren, sondern eine Testattrappe verwenden konnten.

Konfigurations-Datei aus Projektordner dem Language Server verfügbar machen

Die benutzerfreundlichste und sauberste Lösung einer Black-/Whiteliste stellte für uns eine Konfigurations-Datei im Projektordner dar. Diese Lösung erlaubt zum einen eine schnelle und simple Änderung der Einstellungen und macht es zum anderen trivial diese Einstellungen mit anderen Programmierenden zu teilen. Bei der Instanziierung der Erweiterung können file-watchers angegeben werden welche Notifikationen aussenden, falls die überwachten Dateien geändert werden. Der Server kann im Hintergrund auf diese Meldungen hören und die veränderte Konfiguration benutzen. Diese Dateien werden allerdings ausschliesslich bei Änderungen und nicht auch beim Start des Plugins übermittelt. Daher definierten wir eine API auf der Seite des Clients, über welche der Server eine Datei aus dem Projektordner anfordern kann. Der Client sucht diese dann und sendet ihren Inhalt zurück. Mit dieser Lösung spielt es für den Server keine Rolle, ob er aufgrund von Änderungen an der Konfiguration oder beim Systemstart nach der Datei sucht, der Ablauf ist in jedem Fall derselbe.

Pipeline

Bei der Realisierung der Pipeline, wie in der Sektion 5.2.2 beschrieben, trat beim Schritt *sonarcloud-check* ein Problem auf. Die SonarQube Auswertung beinhaltet auch eine Analyse der Testabdeckung. Der dort eingesetzte Sonarscanner führt die Unittests jedoch nicht selbständig aus, sondern ist auf ein separates, bereits erstelltes, Testfile angewiesen. Im Falle von Typescript wäre das zum Beispiel ein *lcov.info* File. Dieses File kann in dem von uns verwendeten Tool zur Berechnung zur Testabdeckung erstellt werden. Unittests und die SonarQube Analyse in der Pipeline wurden in zwei separaten Schritten ausgeführt, wodurch die Datei zwischen beiden Schritten verschwindet. Die Lösung war die von der GitLab CI bereitgestellte Konfiguration *artifacts:untracked*, welche erlaubt Dateien, die nicht von Git getrackt werden, als Artefakte hinzuzufügen und sie somit in nachgehenden Pipelineschritten zugänglich zu machen. Wie in der Abbildung 5.5 zu sehen, wurde die Option im *test* Schritt aktiviert, dementsprechend ist die Datei im nächsten Schritt verfügbar und der Sonarscanner kann die Test Coverage auslesen und in der Analyse berücksichtigen.

5.2.6 Testing

Dieser Teil fokussiert sich auf das Testen der technischen Funktionalität des Plugins. Details zu den Herausforderungen im Rahmen der durchgeführten Nutzertests können dem Kapitel 4 entnommen werden.

Um die technische Funktionalität unserer Erweiterung zu gewährleisten haben wird die bestehende Unit Test Suite aus der Studienarbeit ausgebaut, um auch alle neue Logik abzudecken. Dafür kommen als Test Framework Mocha [9], als Assertion Library Chai [10] und zur Berechnung der Testabdeckung Istanbul [11] zum Einsatz. Mit über 90% Testabdeckung, ersichtlich in Bild 5.6, übetreffen wir hier unser Ziel von 80% klar.

[9] https://mochajs.org/
[10] https://www.chaijs.com/
[11] https://istanbul.js.org/

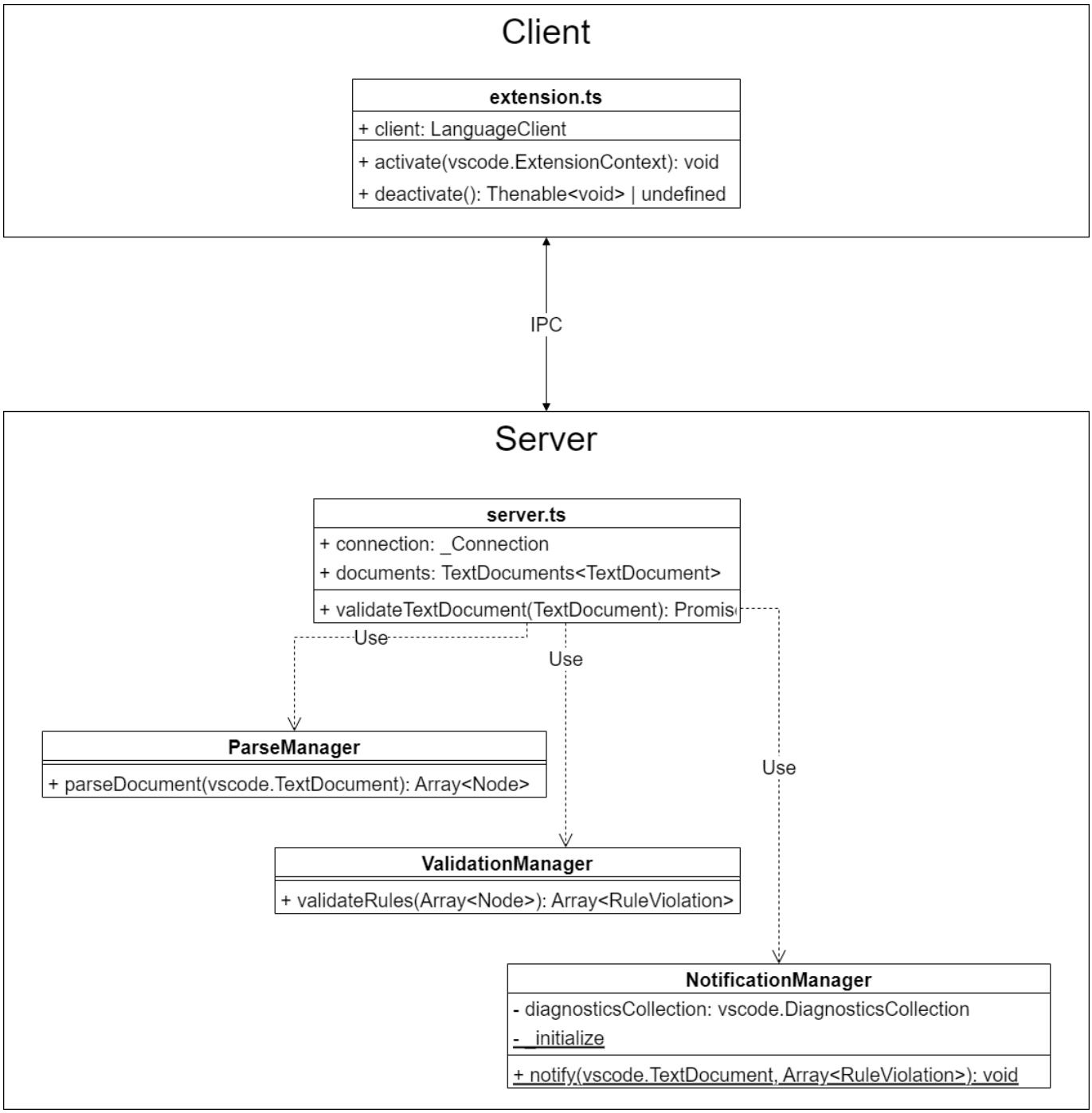


Abbildung 5.2: Grobarchitektur des fertigen Language Server Plugins

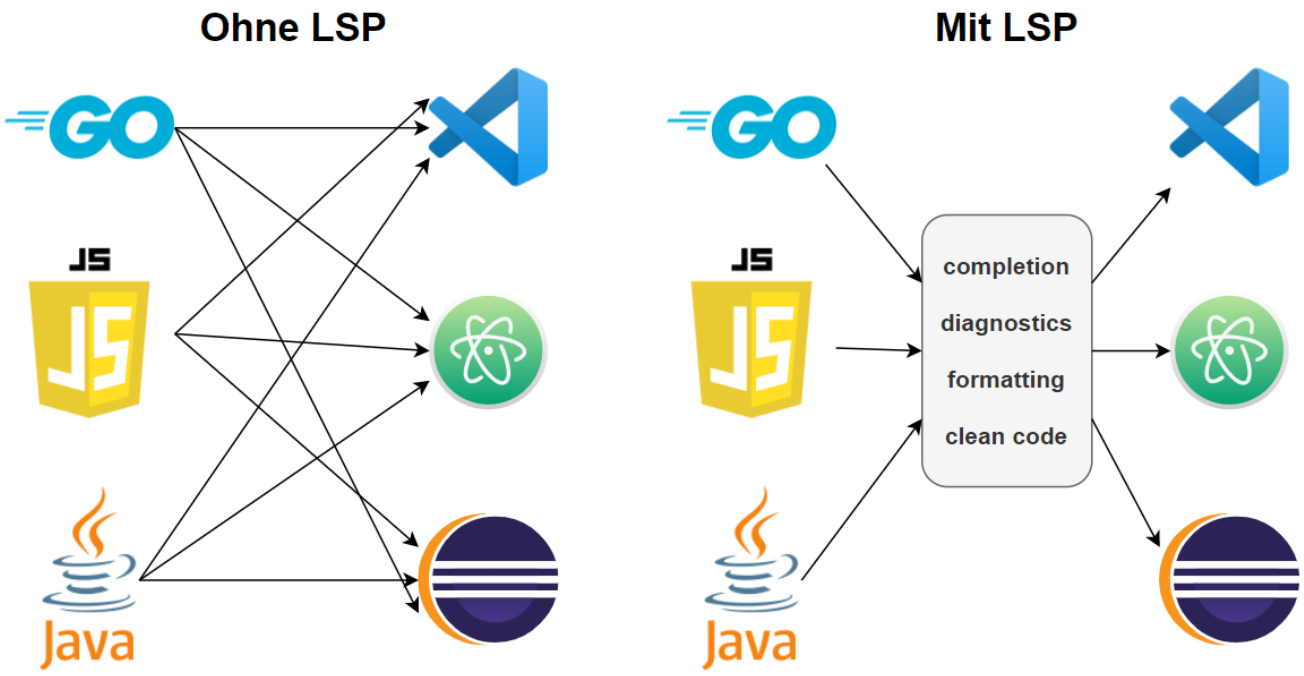


Abbildung 5.3: Das LSP im Zusammenspiel mit Programmiersprachen und IDEs [34]

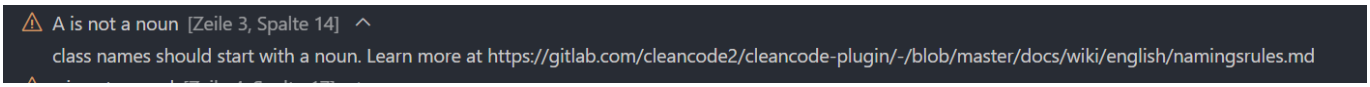
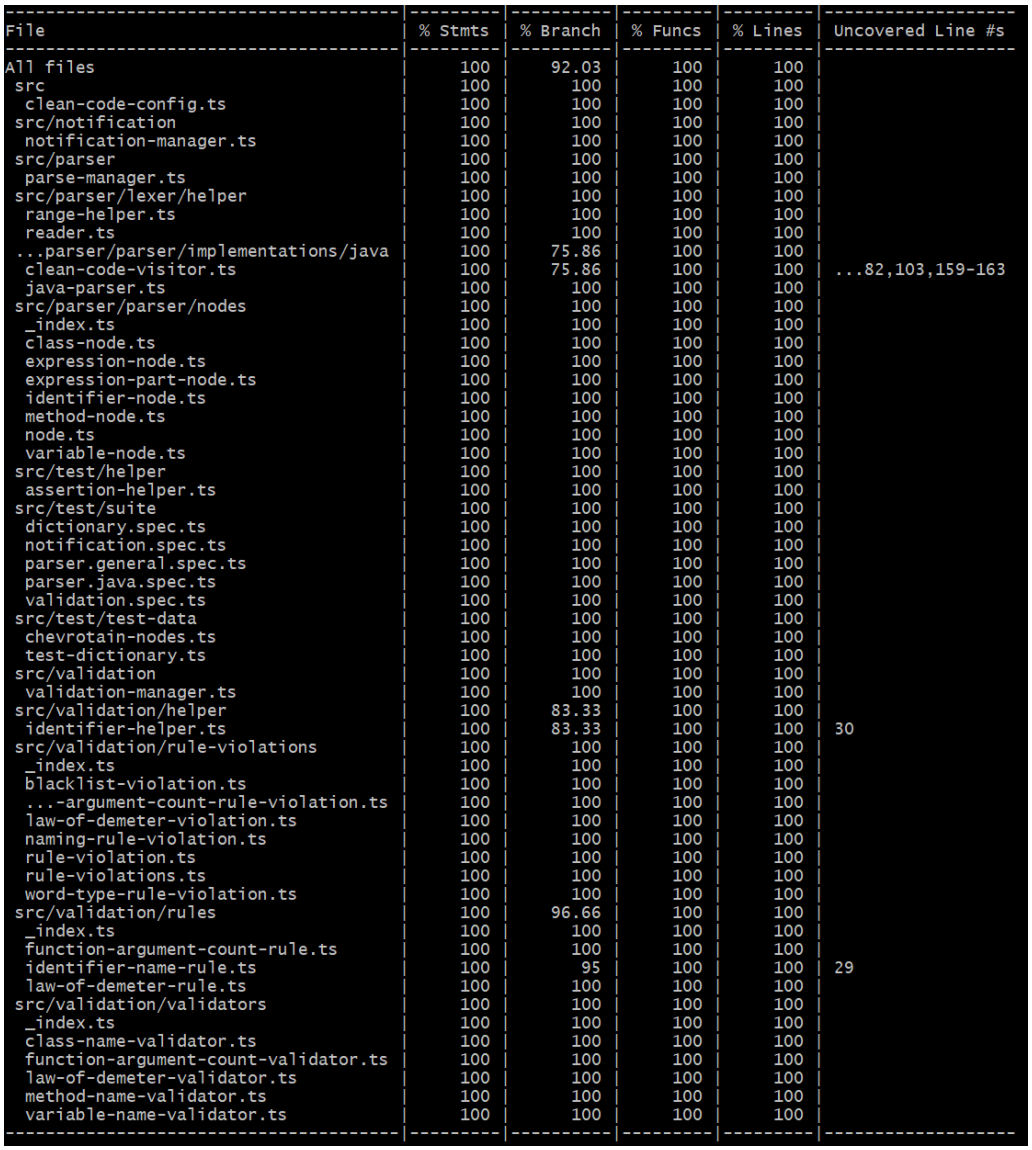


Abbildung 5.4: Implementation Weblink in Problemmeldungen

```
17  image: sonarsource/sonar-scanner-cli:latest
18
19  test:
20    stage: test
21    only: ['merge_requests', 'develop', 'master']
22    artifacts:
23      untracked: true
24    script:
25      - cd client && npm install && cd ../server && npm install && cd .. && npm install
26      - npm test
27
28  sonarcloud-check:
29    stage: sonarcloud-check
30    variables:
31      SONAR_USER_HOME: "${CI_PROJECT_DIR}/.sonar"  # Defines the location of the analysis task cache
32      GIT_DEPTH: "0"  # Tells git to fetch all the branches of the project, required by the analysis task
33    cache:
34      key: "${CI_JOB_NAME}"
35      paths:
36        - .sonar/cache
37    script:
38      - sonar-scanner
39    allow_failure: true
40    only:
41      - merge_requests
42      - master
43      - develop
44
```

Abbildung 5.5: Pipeline Schritte test & sonarcloud-check



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	92.03	100	100	
src	100	100	100	100	
clean-code-config.ts	100	100	100	100	
src/notification	100	100	100	100	
notification-manager.ts	100	100	100	100	
src/parser	100	100	100	100	
parse-manager.ts	100	100	100	100	
src/parser/lexer/helper	100	100	100	100	
range-helper.ts	100	100	100	100	
reader.ts	100	100	100	100	
...parser/parser/implementations/java	100	75.86	100	100	
clean-code-visitor.ts	100	75.86	100	100	...82,103,159-163
java-parser.ts	100	100	100	100	
src/parser/parser/nodes	100	100	100	100	
_index.ts	100	100	100	100	
class-node.ts	100	100	100	100	
expression-node.ts	100	100	100	100	
expression-part-node.ts	100	100	100	100	
identifier-node.ts	100	100	100	100	
method-node.ts	100	100	100	100	
node.ts	100	100	100	100	
variable-node.ts	100	100	100	100	
src/test/helper	100	100	100	100	
assertion-helper.ts	100	100	100	100	
src/test/suite	100	100	100	100	
dictionary.spec.ts	100	100	100	100	
notification.spec.ts	100	100	100	100	
parser.general.spec.ts	100	100	100	100	
parser.java.spec.ts	100	100	100	100	
validation.spec.ts	100	100	100	100	
src/test/test-data	100	100	100	100	
chevrotain-nodes.ts	100	100	100	100	
test-dictionary.ts	100	100	100	100	
src/validation	100	100	100	100	
validation-manager.ts	100	100	100	100	
src/validation/helper	100	83.33	100	100	
identifier-helper.ts	100	83.33	100	100	30
src/validation/rule-violations	100	100	100	100	
_index.ts	100	100	100	100	
blacklist-violation.ts	100	100	100	100	
...-argument-count-rule-violation.ts	100	100	100	100	
law-of-demeter-violation.ts	100	100	100	100	
naming-rule-violation.ts	100	100	100	100	
rule-violation.ts	100	100	100	100	
rule-violations.ts	100	100	100	100	
word-type-rule-violation.ts	100	100	100	100	
src/validation/rules	100	96.66	100	100	
_index.ts	100	100	100	100	
function-argument-count-rule.ts	100	100	100	100	
identifier-name-rule.ts	100	95	100	100	29
law-of-demeter-rule.ts	100	100	100	100	
src/validation/validators	100	100	100	100	
_index.ts	100	100	100	100	
class-name-validator.ts	100	100	100	100	
function-argument-count-validator.ts	100	100	100	100	
law-of-demeter-validator.ts	100	100	100	100	
method-name-validator.ts	100	100	100	100	
variable-name-validator.ts	100	100	100	100	

Abbildung 5.6: Übersicht Testabdeckung

5.3 Ergebnis

Aus der Arbeit resultierte ein Visual Studio Code Plugin, welches Java Quelltext gegen ein Set von Clean Code Regeln prüft. Dieses wurde anschließend als Open Source Projekt veröffentlicht. Verstösse gegen diese Regeln werden in der IDE farbig markiert und im Problemlog inklusive einer kurzen Problembeschreibung angezeigt. In den Problemmeldungen ist ausserdem ein Link zu einer Erklärseite enthalten, welche das Problem und potenzielle Lösungen erläutert. In Bezug auf die Architektur wurde das Plugin offen aufgebaut, damit weitere Clean Code Regeln oder Support für weitere Programmiersprachen einfach hinzugefügt werden können. Neue Versionen des Plugins werden automatisch im Visual Studio Marketplace veröffentlicht und können dort von jedem interessierten Entwickelnden kostenlos mittels einer Ein-Klick Installation heruntergeladen werden. Das Plugin kann sowohl über die IDE internen Einstellungen wie auch über das Hinzufügen einer Konfigurationsdatei im Root Verzeichnis konfiguriert werden. Darunter können auch eine projektspezifische White- und Blacklist konfiguriert werden, welche bei der Regelprüfung von Bezeichnern berücksichtigt wird.

Eine allgemeine Übersicht über alle implementierten Features gibt die Tabelle 5.2.

Tabelle 5.2: Plugin Features

Name	**Typ**	**Beschreibung**	**Implementiert**
Benennungs Regeln	Clean Code Regeln	Überprüft Benennungsregeln von Variablen, Funktionen und Klassen	Prototyp
Funktionsargumente Regel	Clean Code Regeln	Überprüft die maximale Anzahl von Funktionsparametern	Prototyp
Law of Demeter Regel	Clean Code Regeln	Überprüft Verstösse gegen das Law Of Demeter	Neu
Konfigurationsdatei	Konfiguration	Erlaubt die Konfiguration (Warninglevel, Deaktivierung) einzelner Regeln	Neu
Black- und Whitelist	Konfiguration	Erlaubt es Wörter aus dem Dictionary zu entfernen oder eigene neue Wörter hinzuzufügen	Neu
Kontextmenü Eintrag	Konfiguration	Erlaubt das Hinzufügen von Wörtern in die Black- bzw. Whitelist mittels Rechtsklick in der IDE	Neu
Weblinks	Anderes	Verlinkt in Problemmeldungen zu Erklärseiten mit detaillierten Beschreibungen sowie Problemlösungsvorschlägen	Neu

Nachfolgend zeigen wir die implementierte Funktionalität grafisch auf.

Das fertige Plugin, welches eine Auswertung einer Testdatei vorgenommen hat, ist in der Abbildung 5.7 ersichtlich.

Ein Beispiel einer Konfigurationsdatei mit erweiterter Black- und Whitelist ist in der Abbildung 5.8 zu sehen.

Die Interaktion mit der Black- und Whitelist ist in der Abbildung 5.9 veranschaulicht.

Das Plugin kann wie in der Abbildung 5.10 zu sehen ist, einfach über den Visual Studio Marketplace installiert werden.

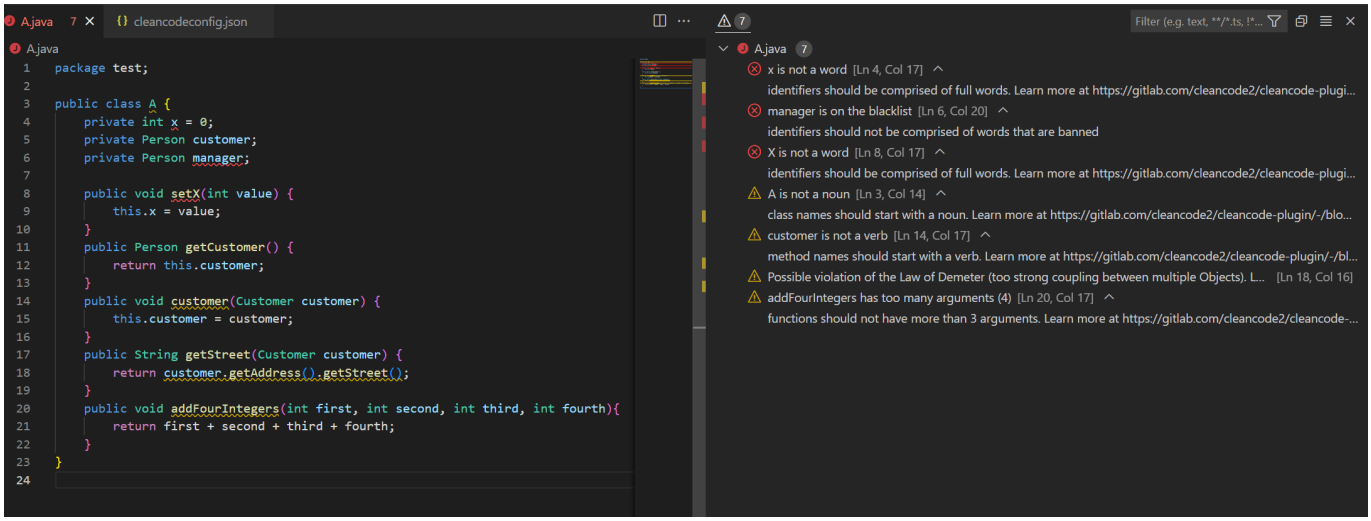


Abbildung 5.7: Beispiele von gefundenen Regelverstössen

Abbildung 5.8: Konfigurations-Datei

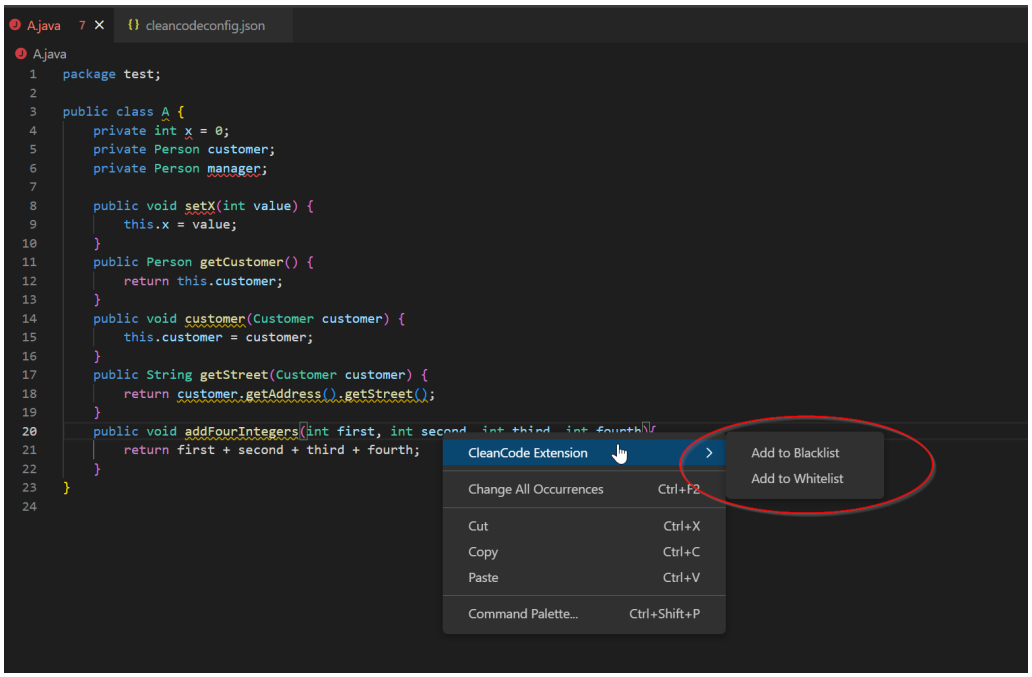


Abbildung 5.9: Kontextmenüeinträge der Erweiterung

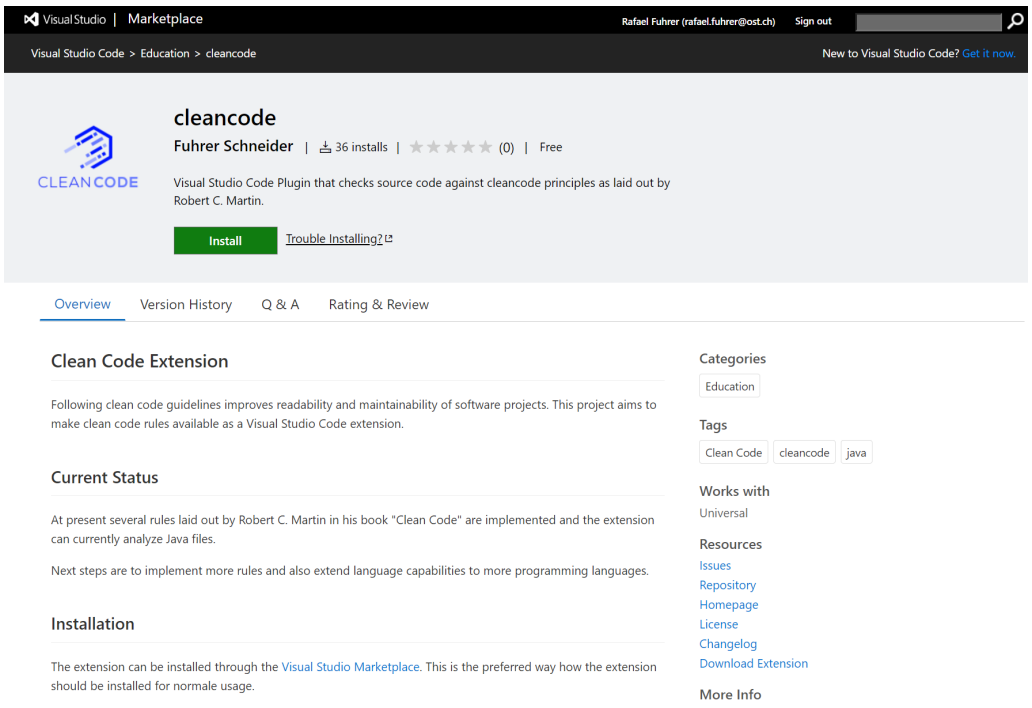


Abbildung 5.10: Veröffentlichtes Plugin im Visual Studio Marketplace

5.4 Diskussion

Bei der Umsetzung dieses Projekts mussten diverse Herausforderungen bewältigt werden. Einige wurden erfolgreich gelöst, andere, wie beispielsweise die Implementation von Weblinks in den Problemmeldungen, konnten nicht zu unserer Zufriedenheit abgeschlossen werden. Dies lag teilweise auch ausserhalb unseres Einflussbereichs, da bei den Weblinks zum Beispiel die Implementation der Diagnosemitteilungen des Language Servers noch nicht alle Attribute unterstützt die Visual Studio Code eigentlich akzeptieren würde. Es gab keine Probleme, die uns über längere Zeit aufhielten, so hat auch das Infrastruktursetup während der gesamten Zeit fehlerfrei funktioniert. Besonders schnell gelungen ist uns, durch Vorwissen aus der Studienarbeit, der Aufbau der Pipeline. In dieser Arbeit wurden die bestehenden Schritte lediglich überarbeitet und der publish Schritt hinzugefügt.

Von besonderer Bedeutung war in dieser Arbeit die Durchführung von Nutzertests mit mehreren Personen. Dadurch wurde wertvolles Feedback generiert, welches es uns Erlaubte in der Umsetzung die Schwerpunkte exakt zu legen. Auch erhielten wir viele Hinweise, die wir selbst als Projektinterne so nicht wahrgenommen haben. So haben uns unsere Testnutzer zum Beispiel darauf hingewiesen, dass Wörter auf der Whitelist trotzdem noch eine *Falscher Worttyp* Meldung bei Bezeichnern auslösen können. Dies konnten wir dadurch beheben, dass Wörter von der Whitelist nun für alle Benennungsregeln als Wildcards gelten.

Wir sind mit unserem Ergebnis zufrieden. Es ist uns gelungen, das Plugin als Open Source Projekt zu veröffentlichen. Es kann nun im Visual Studio Marketplace heruntergeladen werden und steht damit der weltweiten Community zur Verfügung. Einziger negativer Punkt ist der noch immer geringe Umfang an geprüften Clean Code Regeln, was auch die Auswertung der Nutzertests widerspiegelt. Im Rahmen der Arbeit konnten wir, trotz dem Wunsch nach mehr neuen Regeln im Funktionsumfang, diesen nur leicht ausbauen.

Kapitel 6

Zusammenfassung

In diesem Kapitel fassen wir die Bachelorarbeit zusammen und evaluieren die gewonnenen Erkenntnisse. Ausserdem geben wir einen Ausblick, in welchen Bereichen das Plugin in der Zukunft weiterentwickelt werden könnte.

6.1 Gewonnene Erkenntnisse

Bei der Durchführung dieser Arbeit sind wir mit vielen Themen in Berührung gekommen und haben bei der Bewältigung diverser Herausforderungen viel gelernt. Nachfolgend möchten wir eine kurze Zusammenfassung über die wichtigsten Punkte geben.

6.1.1 Clean Code Denkschulen

Bei der Analyse weiterer Clean Code Literatur und deren direkter Vergleich zum Buch *Clean Code* haben wir festgestellt, dass der Begriff Clean Code in der Literatur sehr verschieden interpretiert wird. Während einige Autoren unter Clean Code das gesamte Umfeld der Softwareentwicklung, wie z.B. Projektmanagement miteinbeziehen, fokussieren sich anderer rein auf den geschriebenen Quelltext. Diese unterschiedlichen Auffassungen erschwerten uns aber auch einen direkten Vergleich von konkreten Clean Code Regeln in Bezug auf das Schreiben von Quelltext.

6.1.2 Open Source Projekt

Für die Veröffentlichung unseres Plugins als Open Source Projekt, mussten die elementaren Bestandteile eines solchen evaluiert werden. Die wichtigste Frage war dabei, welche Inhalte ein Readme, der Code of Conduct und die Contributing Anweisungen haben sollten. Ausserdem musste eine passende Lizenz ausgewählt werden. Open Source Lizenzen lassen sich in drei Grobkategorien einteilen. Dabei kann nur eine dieser Kategorien, die Kategorie der permissiven Lizenzen, völlig frei in proprietären Produkten verwendet werden. Für unser Projekt haben wir uns schliesslich für die Apache Version 2 Lizenz entschieden, da einige Spezialbestimmungen wie z.B. eine Bestimmung rund um das Patentrecht, sie zur am besten passenden Lizenz machten.

6.1.3 Nutzertests

Die Durchführung von Nutzertest, bzw. die daraus gewonnenen Erkenntnisse können für ein Projekt sehr bereichernd sein. Wichtig ist dabei, dass diese systematisch vorbereitet und ausgewertet werden. Die gewonnenen Erkenntnisse waren sehr wertvoll, um versteckte Bugs zu finden und um die Laufzeitstabilität zu messen. Ausserdem haben wir dadurch viel Input zu potenziellen Verbesserungen bekommen, welche wir soweit möglich auch umgesetzt haben. Eine persönlich betreute Durchführung und manuelle Auswertung können aber viel Zeit in Anspruch nehmen. Der Einsatz von Tools zur Unterstützung bei der Erhebung und Auswertung ist daher vor allem für grössere Testgruppen unerlässlich.

6.1.4 Architektur

Im Rahmen der Arbeit wurde das Plugin von einer rein Clientseitigen in eine Language Server Implementation abgeändert. Für die erfolgreiche Implementation sind ein fundiertes Vorwissen, wie LSP Erweiterungen aufgebaut sind,

und ein grundlegendes Verständnis über die Kommunikation zwischen Client und Server nötig. Dieser Umbau war die einzige tiefgreifende Änderung an der Architektur im Vergleich zur Studienarbeit [41, S. 26].

6.1.5 Testing

Das Testing Framework Mocha und die Assertion Library Chai waren bereits bei den automatisierten Tests der Studienarbeit [41, S. 29] im Einsatz. Damals wurde keine Testabdeckung berechnet, diese wird nun mit Istanbul berechnet. Durch die Änderungen am Aufbau der automatisierten Tests konnten wir dieses Tool bei der Bachelorarbeit nun Nutzen.

6.2 Weiterführende Themen

Mit der Veröffentlichung des Projekts als Open Source Projekt und der Fertigstellung dieses Berichts ist unsere Bachelorarbeit abgeschlossen. Mit unserer Arbeit haben wir eine Grundlage geschaffen auf der weiter aufgebaut und das Plugin um diverse weitere Funktionalität erweitert werden kann. Abschliessend möchten wir hier daher Ideen sammeln was für Weiterentwicklungen konkret denkbar wären. Wir unterscheiden dabei zwischen technischen Erweiterungen, die mit bestehenden Mitteln implementiert werden könnten, und Feldern der Forschung, die im Zusammenhang mit dem Plugin erforscht werden könnten.

Um das Plugin um technische Funktionalität erweitern zu können, braucht man grundlegendes Wissen rund um die Technologien Typescript, Mocha, Chai und Istanbul. Zudem ein vertieftes Verständnis über den Aufbau von Programmiersprachen, sowie ein vertieftes Wissen rund um Clean Code Prinzipien.

6.2.1 Technische Erweiterungsmöglichkeiten

Die folgenden technischen Erweiterungsmöglichkeiten haben wir in unserer Studienarbeit identifiziert [41, S. 34 - S. 35]:

- Ausweitung des Sprachsupports über Java hinaus (z.B. Support für Python, Typescript, Go etc ...)
- Die Möglichkeit eine Problemmeldung zu akzeptieren bzw. zu unterdrücken
- Die Möglichkeit einfach selbst weitere Regeln zu definieren, die dann vom Plugin geprüft werden
- Die Möglichkeit Clean Code Metriken zu sammeln und mit SCA Tools wie z.B. SonarQube zu analysieren & auszuwerten

Im Rahmen dieser Arbeit konnten wir ebenfalls einige mögliche Erweiterungen identifizieren:

- Ausweitung des IDE Supports auf weitere IDEs (z.B. Atom, IntelliJ, Eclipse)
- Die Möglichkeit sprachspezifische Wörterbücher zu unterstützen (Mehrsprachiger Support)
- Die Möglichkeit spezifische Regeln für einzelne Programmiersprachen zu unterstützen

6.2.2 Forschungsfelder

In unserer Studienarbeit haben wir bereits einige Forschungsfelder identifiziert [41, S. 35]. Dabei fokussierten wir uns vor allem auf Felder, welche dabei helfen können, Kontextverständnis von Code zu realisieren:

- Trainieren eines AI 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 via Aufrufe, 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] Stephan Augsten chrissikraus. Was ist ein parser? `https://www.dev-insider.de/was-ist-ein-parser-a-756662/`, Oktober 2018. zuletzt abgerufen am 15.06.2022.

[3] TechTarget Contributor. Language server protocol. `https://www.techtarget.com/whatis/definition/blacklist/`, Juli 2016. zuletzt abgerufen am 15.06.2022.

[4] Cambridge Dictionary. whitelist. `https://dictionary.cambridge.org/de/worterbuch/englisch/whitelist`, o. D. zuletzt abgerufen am 15.06.2022.

[5] Mozilla MDN Web Docs. Arbeit mit json. `https://developer.mozilla.org/de/docs/Learn/JavaScript/Objects/JSON`, Dezember 2020. zuletzt abgerufen am 15.06.2022.

[6] Josef Dolch. *Grundbegriffe der pädagogischen Fachsprache*. Ehrenwirth Verlag, 1965.

[7] en.wikipedia.org. node. `https://en.wikipedia.org/wiki/Node_(linguistics)`, Mai 2022. zuletzt abgerufen am 17.06.2022.

[8] Organization for Ethical Source. A code of conduct for open source communities. `https://www.contributor-covenant.org/`, o. D. zuletzt abgerufen am 15.06.2022.

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

[10] Eclipse Foundation. Eclipse. `"https://www.eclipse.org/`, o. D. zuletzt abgerufen am 15.06.2022.

[11] Eclipse Foundation. Eclipse public license (epl) frequently asked questions. `https://www.eclipse.org/legal/epl-2.0/faq.php`, o. D. zuletzt abgerufen am 15.06.2022.

[12] Martin Fowler. Refactoring. `https://www.refactoring.com/`, o. D. zuletzt abgerufen am 15.06.2022.

[13] Instituts für Rechtsfragen der Freien und Open Source Software. Lizenzcenter. `https://ifross.github.io/ifrOSS/Lizenzcenter`, o. D. zuletzt abgerufen am 15.06.2022.

[14] Github. Contributing to atom. `https://github.com/atom/atom/blob/master/CONTRIBUTING.md`, Februar 2022. zuletzt abgerufen am 15.06.2022.

[15] Github. Create a repo. `https://docs.github.com/en/get-started/quickstart/create-a-repo#commit-your-first-change`, o. D. zuletzt abgerufen am 15.06.2022.

[16] Github. Ein open source projekt anfangen. `https://opensource.guide/de/starting-a-project/`, o. D. zuletzt abgerufen am 15.06.2022.

[17] Github. Ihr verhaltenskodex. `https://opensource.guide/de/code-of-conduct/`, o. D. zuletzt abgerufen am 15.06.2022.

[18] Github. Licenses. `https://choosealicense.com/licenses/`, o. D. zuletzt abgerufen am 15.06.2022.

[19] Github. Setting guidelines for repository contributors. `https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/setting-guidelines-for-repository-contributors`, o. D. zuletzt abgerufen am 15.06.2022.

[20] Github. Setting guidelines for repository contributors. https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/setting-guidelines-for-repository-contributors, o. D. zuletzt abgerufen am 15.06.2022.

[21] Red Had. Was versteht man unter ci/cd? https://www.redhat.com/de/topics/devops/what-is-ci-cd, April 2018. zuletzt abgerufen am 15.06.2022.

[22] Red Had. Was ist eine ide? https://www.redhat.com/de/topics/middleware/what-is-ide, Januar 2019. zuletzt abgerufen am 15.06.2022.

[23] Red Had. Was ist eine java runtime-umgebung (jre)? https://www.redhat.com/de/topics/cloud-native-apps/what-is-a-Java-runtime-environment, April 2020. zuletzt abgerufen am 15.06.2022.

[24] Computer Hope. Exception handling. https://www.computerhope.com/jargon/e/exception-handling.htm, Februar 2017. zuletzt abgerufen am 15.06.2022.

[25] Open Source Initiative. Licenses & standards. https://opensource.org/licenses, o. D. zuletzt abgerufen am 15.06.2022.

[26] Open Source Initivative. The 3-clause bsd license. https://opensource.org/licenses/BSD-3-Clause, o. D. zuletzt abgerufen am 15.06.2022.

[27] IONOS. Readme: Alles wichtige im Überblick – inklusive vorlage. https://www.ionos.de/digitalguide/websites/web-entwicklung/readme-datei/, Juni 2020. zuletzt abgerufen am 15.06.2022.

[28] iso25000.com. Iso/iec 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010, o. D. zuletzt abgerufen am 15.06.2022.

[29] lauren orsini. Github for beginners: Don't get scared, get started. https://readwrite.com/understanding-github-a-journey-for-beginners-part-1/, September 2013. zuletzt abgerufen am 15.06.2022.

[30] Universität Marburg. Konkrete vs abstrakte syntax. https://www.mathematik.uni-marburg.de/~loogen/Lehre/ws05/Compilerbau/Syntax.pdf, o. D. zuletzt abgerufen am 15.06.2022.

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

[32] 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. zuletzt abgerufen am 15.06.2022.

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

[34] Microsoft. Language server extension guide. https://code.visualstudio.com/api/language-extensions/language-server-extension-guide, o. D. zuletzt abgerufen am 15.06.2022.

[35] Microsoft. Language server protocol. https://microsoft.github.io/language-server-protocol/, o. D. zuletzt abgerufen am 15.06.2022.

[36] Microsoft. Typescript is javascript with syntax for types. https://www.typescriptlang.org/, o. D. zuletzt abgerufen am 15.06.2022.

[37] Microsoft. Visual studio code. "https://code.visualstudio.com/, o. D. zuletzt abgerufen am 15.06.2022.

[38] Microsoft. Visual studio ide. https://visualstudio.microsoft.com/de/, o. D. zuletzt abgerufen am 15.06.2022.

[39] opensource.com. What is open source? https://opensource.com/resources/what-open-source, o. D. zuletzt abgerufen am 15.06.2022.

[40] Oracle. Java se desktop technologies. https://www.oracle.com/java/technologies/javase/desktop.html, o. D. zuletzt abgerufen am 15.06.2022.

[41] Pascal René Schneider Rafael Fuhrer. Entwicklung eines ide-plugins zur prufung von clean code-regeln. https://eprints.ost.ch/id/eprint/1005/1/HS%202021%202022-SA-EP-Schneider-Fuhrer-Clean%20Code%20Plugin.pdf, Dezember 2021. zuletzt abgerufen am 15.06.2022.

[42] Eric Freeman & Elisabeth Robson. *Entwurfsmuster von Kopf bis Fuß*. O'Reilly, 2021.

[43] 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. zuletzt abgerufen am 15.06.2022.

[44] Sonarsource SA. Quality gates. https://docs.sonarqube.org/latest/user-guide/quality-gates/, o. D. zuletzt abgerufen am 15.06.2022.

[45] SurveyMonkey. Vorlage umfrage zur software-bewertung. https://www.surveymonkey.de/mp/software-evaluation-survey-template/, o. D. zuletzt abgerufen am 15.06.2022.

[46] Talend. Api (application programming interface) — definition und vorteile. https://www.talend.com/de/resources/was-ist-eine-api/, o. D. zuletzt abgerufen am 15.06.2022.

[47] techterms.com. Thread. https://techterms.com/definition/thread, Dezember 2006. zuletzt abgerufen am 15.06.2022.

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

[49] David A. Wheeler. The free-libre / open source software (floss) license slide. https://dwheeler.com/essays/floss-license-slide.html, Januar 2017. zuletzt abgerufen am 15.06.2022.

[50] Wikipedia. Visual studio code. https://de.wikipedia.org/wiki/Visual_Studio_Code, November 2021. zuletzt abgerufen am 15.06.2022.

[51] Wikipedia. Git. https://de.wikipedia.org/wiki/Git, Mai 2022. zuletzt abgerufen am 15.06.2022.

[52] Wikipedia. Künstliche intelligenz. https://de.wikipedia.org/wiki/K%C3%BCnstliche_Intelligenz, Juni 2022. zuletzt abgerufen am 15.06.2022.

[53] Wikipedia. Lint (programmierwerkzeug). https://de.wikipedia.org/wiki/Lint_(Programmierwerkzeug), März 2022. zuletzt abgerufen am 15.06.2022.

[54] Wikipedia. Modultest. https://de.wikipedia.org/wiki/Modultest, Juni 2022. zuletzt abgerufen am 15.06.2022.

[55] Wikipedia. Qualität. https://de.wikipedia.org/wiki/Qualit%C3%A4t, April 2022. zuletzt abgerufen am 15.06.2022.

[56] Wikipedia. Statische code-analyse. https://de.wikipedia.org/wiki/Statische_Code-Analyse, April 2022. zuletzt abgerufen am 15.06.2022.

[57] Lawrence Williams. Inter process communication (ipc) in os. https://www.guru99.com/inter-process-communication-ipc.html, April 2022. zuletzt abgerufen am 15.06.2022.

[58] Gary Woodfine. What is clean code ? https://garywoodfine.com/what-is-clean-code/, Oktober 2018. zuletzt abgerufen am 15.06.2022.

Glossar

Einige Einträge dieses Glossars haben wir aus unserer Studienarbeit [41, S. 40 - S. 41] übernommen.

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. [52]*
Application Programming Interface (API)	*Eine API ist eine Programmierschnittstelle, die von einem Softwaresystem für andere Systeme zur Verfügung gestellt wird. [46]*
Blacklist	*In der IT ist eine schwarze Liste eine Sammlung von Entitäten, die für die Kommunikation mit einem Computer, einer Website oder einem Netzwerk oder für die Anmeldung bei einem solchen blockiert sind. [3]*
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. [21]*
Clean Code	*Sauberer, einfach und intuitiv verständlicher Quelltext mit klarer Struktur und angewandten Konzepten. [31] (S34)*
Code of Conduct	*In Bezug auf Open Source Projekte ist ein Verhaltenskodex (Code of Conduct) ein Dokument, das die Erwartungen an das Verhalten der ProjektteilnehmerInnen festlegt. [17]*
Konkreter Syntaxbaum	*Ein Syntaxbaum der für die Sprache definierten kontextfreien Grammatik. [30, S. 01]*
Contributing Guidelines	*Für den Projektverantwortlichen eines Open Source Projekts sind die Beitragsrichtlinien (Contributing Guidelines) ein Mittel, um zu kommunizieren, wie die Leute zum Projekt beitragen sollten. [20]*
Exception Handling	*Die Behandlung von Exceptions ist die Reaktion auf Ausnahmen bei der Ausführung eines Computerprogramms. Eine Exception tritt auf, wenn ein unerwartetes Ereignis eintritt, das eine besondere Verarbeitung erfordert. [24]*
Git	*Git ist eine freie Software zur verteilten Versionsverwaltung von Dateien. [51]*

Begriff	**Definition**
Integrated Development Environment (IDE)	*Integrierte Entwicklungsumgebung, wird von einem Programmierenden für das Entwickeln von Software verwendet. [22]*
Inter Process Communication (IPC)	*Interprozesskommunikation (IPC) wird für den Datenaustausch zwischen mehreren Threads in einem oder mehreren Prozessen oder Programmen verwendet. [57]*
Language Server Protokoll	*Das Language Server Protocol (LSP) definiert das Protokoll, das zwischen einem Editor oder einer IDE und einem Sprachserver verwendet wird, der Sprachfunktionen wie Autovervollständigung, Gehe zur Definition, Suche nach allen Referenzen usw. bietet. [35]*
Linter	*Ein Programm zur Durchführung von Statischer Code Analyse (Siehe SCA). [53]*
Java	*Java ist eine 1995 entwickelte objektorientierte Programmiersprache. [23]*
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. [5]*
Node	*In der Sprachwissenschaft ist eine Node (Knoten) ein Punkt in einem Syntaxbaum. [7]*
Pipeline	*In der Softwareentwicklung ist eine Pipeline ein System automatisierter Prozesse, welche Aktualisierungen beim Code schnell von der Versionskontrolle in die Produktion übertragen. [21]*
Parser	*Ein Programm welches eine syntaktische Analyse auf der Rückgabe eines Lexers durchführt. Bringt die Informationen in eine hierarchische Struktur. [2]*
QualityGate	*Die Quality Gates von SonarQube setzen eine Qualitätspolitik, indem sie sicherstellen, dass Code bereit für den Einsatz ist. [44]*
Readme	*README-Dateien sind ein hervorragender Ort, um ein Projekt detaillierter zu beschreiben oder eine Dokumentation hinzuzufügen. Der Inhalt Ihrer README-Datei wird automatisch auf der Hauptseite des Repositorys angezeigt. [15]*
Refactoring	*Umarbeitung von Quelltext für verbesserte Verständlichkeit und Wartbarkeit, ohne die Funktion zu ändern. [12]*
Repository	*Ein Verzeichnis oder Speicherplatz, in dem Projekte untergebracht werden können.*
Static Code Analysis (SCA)	*Static Code Analyse, ist eine Methode zur Fehlersuche, bei der der Quelltext untersucht wird, bevor ein Programm ausgeführt wird. [56]*

Begriff	**Definition**
Thread	*Die Threads eines Computerprogramms ermöglichen es dem Programm, aufeinanderfolgende Aktionen oder viele Aktionen gleichzeitig in einem oder mehreren Threads auszuführen. [47]*
Typescript	*TypeScript ist eine stark typisierte Programmiersprache, die auf JavaScript aufbaut und bessere Werkzeuge für die Anwendung bietet. [36]*
Unit Test	*Ein Test der ein einzelnes abgegrenztes Modul (Unit) der Software testet. [54]*
Visual Studio Code (VS Code)	*Eine in Typescript geschriebene IDE von Microsoft. [50]*
Whitelist	*Eine Whitelist ist eine Liste von Personen oder Dingen, die von einer bestimmten Autorität oder Gruppe als akzeptabel angesehen werden und denen man vertrauen sollte. [4]*
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 Erweiterung und Veröffentlichung unseres Clean Code Plugins zeitlich und inhaltlich geplant ist.

A.2 Gültigkeitsbereich

Dieses Dokument ist gültig für die Bachelorarbeit "Automatisierte Analyse von Clean-Code Regeln mit IDE-Plugins" im Frühlingssemester 2022 an der Fachhochschule OST in 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 aussehen soll. Viele EntwicklerInnen beachten einige oder auch alle diese Empfehlungen beim Programmieren nicht. Um EntwicklerInnen beim Schreiben von Quelltext direkt 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.

Dieses Plugin soll zum Ende der Bachelorarbeit für EntwicklerInnen nutzbar sein und als Open Source Projekt veröffentlicht werden.

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. Wir bauen für die Umsetzung auf dem Prototyp auf, welchen wir im Rahmen unserer Studienarbeit erstellt haben. Dieser soll um einen Language Server und weitere Regeln erweitert werden.

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 16 Semesterwochen aufgeteilt. Dies bedeutet ein durchschnittliches Arbeitspensum von ungefähr 22.5 Stunden pro Person pro Woche oder 360 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 21.02.-27.02.	2 28.02.-06.03.	3 07.03-13-03.	4 14.03.-20.03.	5 21.03.-27.03.	6 28.03.-03.04.	7 04.04.-10.04.	8 11.04.-17.04.	9 18.04.-24.04.	10 25.04.-01.05.	11 02.05.-08.05.	12 09.05.-15.05.	13 16.05.-22.05.	14 23.05.-29.05.	15 30.05.-05.06.	16 06.06.-12.06.
Projektmanagement & Dokumentation	10	10	10	10	10	10	10	10	10	10	10	10	10	10	20	30
Recherche	20	15	10	10	10	5	3	3	3	3	3	3	0	0	0	0
Implementation	0	0	5	10	15	20	20	20	20	20	20	20	15	10	10	0
Testing	0	0	0	0	0	0	0	5	5	5	5	5	5	5	5	0
Infrastruktur	2	2	2	2	2	2	5	0	0	0	0	0	5	10	10	5
Reserve	5	10	10	10	10	10	10	10	10	10	10	10	10	10	5	10
Total	**37**	**37**	**37**	**42**	**47**	**47**	**48**	**48**	**48**	**48**	**48**	**48**	**45**	**45**	**50**	**45**
Milestones		M1			M2			M3			M4			M5		M6

A.5.3 Phasen

Inception

In der ersten Phase werden der Projektrahmen definiert, sowie die notwendigen Recherchen durchgeführt.

Elaboration

Während der Elaboration wird detailliert beschrieben, wie die Architektur des bestehenden Plugins abgeändert werden soll.

Construction

Während dieser Phase entwickeln wir die neuen Features des Clean Code Plugins und testen diese fortlaufend.

Transition

In der letzten Phase wird das Plugin und die Dokumentation finalisiert. Das Plugin wird in der ersten Version im Store und überdies inklusive Dokumentation und Community Richtlinien als Open Source Projekt veröffentlicht.

A.5.4 Meilensteine

M1 Analyse

- Bestehendes Plugin analysieren
- Architektur Language Server grob
- Architektur Blacklist/Whitelist grob
- Git Repositories eingerichtet
- Architektur Webhilfe grob

M2 Langugage Server

- Funktionale Anforderungen dokumentiert
- Nichtfunktionale Anforderungen dokumentiert
- Language Server umsetzen
- Architekturdokumentation Language Server
- Blacklist/Whitelist umsetzen
- Architekturdokumentation Blacklist/Whitelist

M3 Weitere Regeln

- Weitere identifizierte Regeln umsetzen
- Webhilfe umsetzen
- Architekturdokumentation Webhilfe

M4 Usability

- Nutzertests durchführen
- Automatisierte Tests implementieren
- Architekturdokumentation abschliessen
- Technische Schulden eliminieren

M5 Publikation

- Nutzerrückmeldungen implementiert
- Code Repository in veröffentlichbaren Zustand bringen
- Architekturdokumentation in veröffentlichbaren Zustand bringen
- Projekt als Open Source Projekt veröffentlichen

M6 Abgabe

- Bericht in abgabefähigen Zustand bringen
- Letzte Fehler beheben
- Qualitätsziele erreichen (SonarQube, Linting)

A.5.5 Besprechungen

Als Fixpunkt gibt es eine wöchentliche Besprechung mit dem Betreuer. Diese ist jeweils auf Montag um 17 Uhr terminiert. Des Weiteren ist jeden Sonntagabend 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 der simplen Anforderungen bei der Pluginentwicklung dagegen 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 vom Store in meine IDE installieren 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	MUSS	Alle Regelverstösse sollen im Problem Log, inklusive ihrer Position in der Datei, in der IDE ausgegeben werden.
FA-3	MUSS	Als ProgrammiererIn kann ich das Plugin bequem über den Marketplace meiner IDE installieren (keine manuelle Installation nötig).
FA-4	MUSS	In der Problemmeldung der IDE ist ein Weblink vorhanden, der auf eine Seite mit genaueren Erläuterungen und Problemlösungsvorschlägen verweist.

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 wird in einem eigenen Thread ausgeführt und blockiert somit die IDE nicht.
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 sowohl einfach um weitere Clean Code Regeln wie auch um den Support für weitere Programmiersprachen 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.
NF-7	MUSS	Das Plugin ist als Open Source Projekt unter freier Lizenz mit entsprechenden Richtlinien zur Verwaltung und Weiterentwicklung veröffentlicht worden.

A.7 Auswertung

In diesem Abschnitt möchten wir die Arbeit Auswerten und Reflektieren. Hier fokussieren wir uns auf die Auswertung des Projektmanagement. 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 der Funktionalen Anforderungen

Anforderungs-ID	Kategorie	Erfüllt?	Begründung
FA-1	MUSS	Ja	Siehe Sektion 5.3
FA-2	MUSS	Ja	Siehe Sektion 5.3
FA-3	MUSS	Ja	Siehe Sektion 5.3
FA-4	MUSS	Ja	Siehe Sektion 5.2.5

Wie aus der Auswertung hervorgeht, haben wir alle unsere MUSS-Kriterien erfüllt.

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

Tabelle A.4: Auswertung der Nicht Funktionalen Anforderungen

Anforderungs-ID	**Kategorie**	**Erfüllt?**	**Begründung**
NF-1	KANN	Ja	Siehe Sektion 5.2.3
NF-2	MUSS	Ja	Dies wurde im Rahmen der Nutzertests getestet
NF-3	MUSS	Ja	Siehe Architekturdokumentation im Anhang
NF-4	MUSS	Ja	Siehe SonarQube Auswertung im Anhang
NF-5	KANN	Ja	Siehe Sektion 5.2.6
NF-6	KANN	Ja	Siehe SonarQube Auswertung & Architekturdokumentation
NF-7	MUSS	Ja	Siehe Kapitel 2.4

Auswertung des Zeitaufwands

In dieser Sektion werten wir die aufgewendete Zeit aus. Pro Projektmitarbeiter sollten *360 Stunden*, zusammen also *720 Stunden* aufgewendet werden. Die geplante, sowie die real aufgewendete Zeit haben wir in der Tabelle A.5 dargestellt.

Tabelle A.5: Analyse Zeitaufwand

Kategorie	**Zeitaufwand geschätzt (in Std.)**	**Zeitaufwand real (in Std.)**
Entwicklung	205	184
Projektmanagement & Dokumentation	190	315
Infrastruktur	47	29
Recherche	88	80
Testing	40	63
Reserve / Anderes	150	49*
Summe	720	720

*Schätzung des Restaufwands nach der Abgabe der schriftlichen Arbeit

Während die geschätzte Gesamtzeit, sehr genau getroffen wurde, gab es, wie in der Tabelle A.5 zu sehen, vor allem in der Kategorie *Projektmanagement & Dokumentation* grössere Abweichungen. Wir haben hier bei der Planung den administrativen Aufwand, darunter vor allem die Zeit die Meetings in Anspruch nehmen würden, unterschätzt. In den Kategorien Entwicklung und Infrastruktur jeweils etwas weniger Zeit als geplant aufgewendet. Im Falle der

Infrastruktur sind wir schneller vorangekommen als geplant. Die geplante Zeit der Entwicklung wurde wegen nicht überwindbaren Herausforderungen ausserhalb unseres Einflussbereichs, welche dazu führten, dass wir einige geplante Features nicht implementieren konnten, ebenfalls nicht vollständig aufgewendet. Im Vergleich dazu haben wir bei der Recherche ziemlich genau die geplante Zeit aufgewendet. Eine weitere grosse Abweichung gab es beim Testing. Auch hier haben wir den Aufwand bei der Planung unterschätzt und deutlich mehr Zeit aufwenden müssen. Die von uns eingeplante Reserve hat aber ausgereicht, um diese Mehraufwände vollständig abdecken zu können. Die restlichen Stunden in der Reserve Kategorie haben wir für andere Aufgaben wie der Bachelorpräsentation und die Vorbereitung auf die Bachelorprüfung, die nach der Abgabe der schriftlichen Arbeit anfallen veranschlagt. Entsprechend handelt es sich dabei um eine Schätzung des ungefähren Aufwands.

Anhang B

SonarQube Auswertung

Wir haben die Code Qualität unserer Visual Studio Code Extension anhand einer Statischen Code Analyse, die wir mit SonarQube durchgeführt haben, analysiert. In den nachfolgenden beiden Abschnitten gehen wir detailliert auf die Ergebnisse ein. Die Auswertung ist öffentlich einsehbar [1].

B.1 Allgemeine Code Qualität

Im Projektplan haben wir in der Nicht Funktionalen Anforderung (NF-4) definiert, dass unser Quelltext von SonarQube eine Gesamtbewertung von A erreichen muss. In der Abbildung B.1 ist zu sehen, dass wir in allen Kategorien diese Bewertung erreichen.

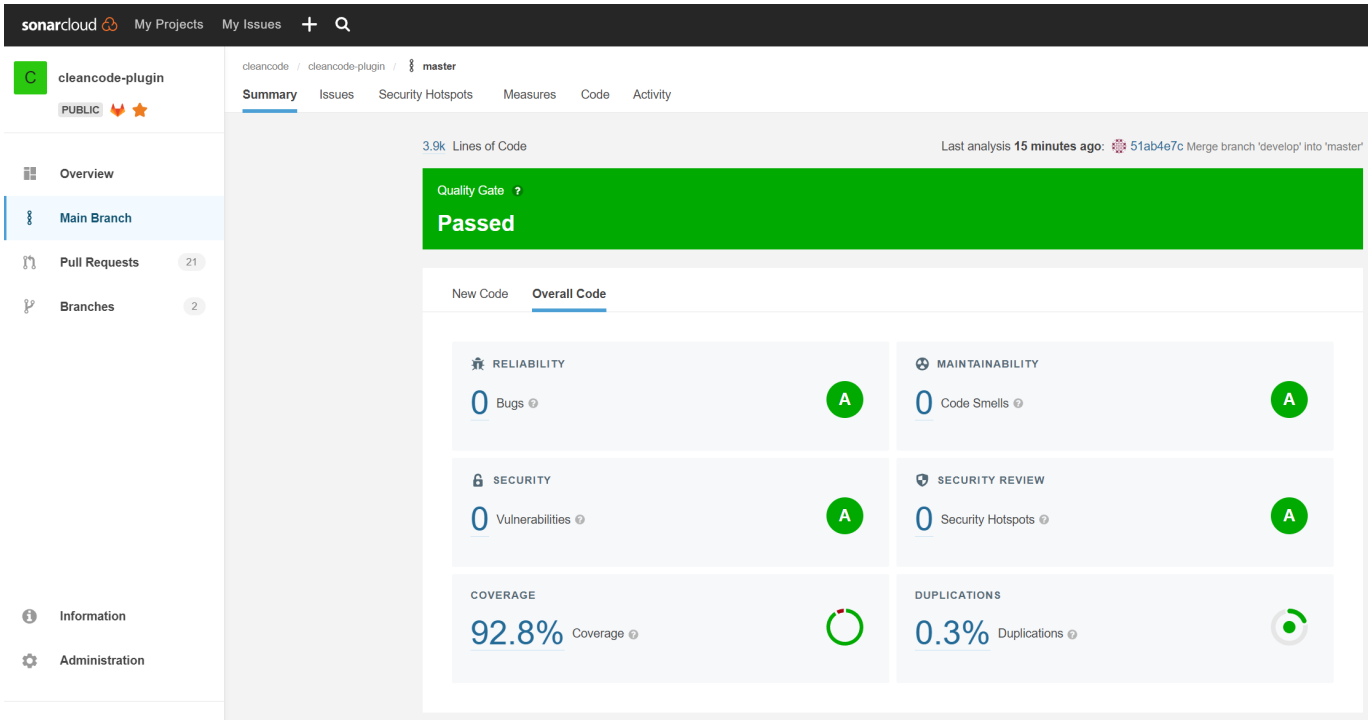


Abbildung B.1: SonarQube Gesamtauswertung

Die Abbildung zeigt ausserdem, dass es in keiner Kategorie unbehandelte Probleme gibt, was bedeutet das nach dieser Analyse keine technical debt existiert, welche wir auf zukünftige Entwickelnde abwälzen würden. Weitere Details zur *technical debt* können der detaillierten Auswertung [2] entnommen werden.

[1] https://sonarcloud.io/project/overview?id=cleancode2_cleancode-plugin
[2] https://sonarcloud.io/component_measures?metric=sqale_index&view=list&id=cleancode2_cleancode-plugin

Der Abbildung kann des Weiteren entnommen werden, dass wir eine *Test Coverage* von 92.8% erreichen. Dies liegt sogar über der optionalen Nicht Funktionalen Anforderung (NF-5), nach welcher eine *Coverage* von mindestens 80% erreicht werden soll. Weitere Details zur *Code Coverage* können der detaillierten Auswertung [3] entnommen werden.

B.2 Auswertung für neuen Code

Um sicherzustellen, dass das von uns vorgegebene Qualitätsniveau beibehalten wird, haben wir ein QualityGate konfiguriert, welches jeder Merge Request erfüllen muss um zusammengefügt werden zu können. Bei der Eröffnung eines Merge Requests wird automatisch eine Analyse angestossen. Das Ergebnis wird dann automatisch als Kommentar dem Merge Request beigefügt. Ein Beispiel einer solchen Auswertung zeigt die Abbildung B.2.

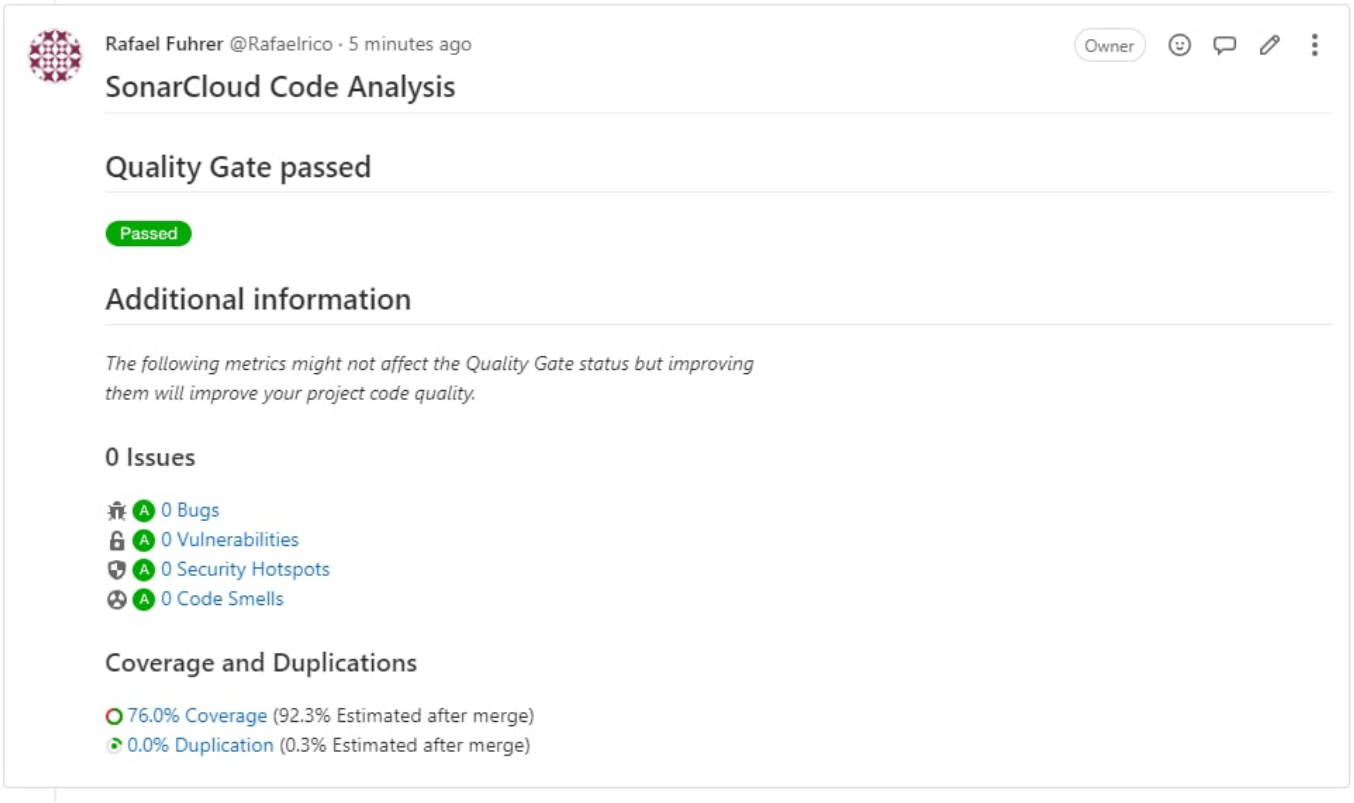


Abbildung B.2: Beispiel einer SonarQube Merge Request Auswertung

[3] https://sonarcloud.io/component_measures?metric=Coverage&view=list&id=cleancode2_cleancode-plugin

Anhang C

Fragebogen

F01: Wie einfach war die Installation unserer Software?

[ ] Äusserst einfach [ ]Sehr einfach [ ]Einigermassen einfach [ ] eher nicht einfach [ ] Überhaupt nicht einfach

Begründung:

F02: Wie benutzerfreundlich ist die Benutzeroberfläche unserer Software?

[ ] Äusserst benutzerfreundlich [ ] Sehr benutzerfreundlich [ ] Einigermassen benutzerfreundlich [ ] Nicht so benutzerfreundlich [ ] Überhaupt nicht benutzerfreundlich

Begründung:

F03: Wie oft bleibt unsere Software hängen oder stürzt sie ab?

[ ] Extrem Oft [ ] Sehr oft [ ] Ab und zu [ ] Kaum [ ] selten

Begründung:

F04: Wie erfolgreich ist unsere Software, was die Erfüllung ihres Einsatzzwecks (Verbesserung der Codequalität) angeht?

[ ] Äusserst erfolgreich [ ] Sehr erfolgreich [ ] Einigermassen erfolgreich [ ] Nicht so erfolgreich [ ] Überhaupt nicht erfolgreich

Begründung:

F05: Wie nützlich ist die unserer Software beiliegende Dokumentation?

[ ] Äusserst nützlich [ ] Ziemlich nützlich [ ] Einigermassen nützlich [ ] Eher nicht nützlich [ ] Überhaupt nicht nützlich

Begründung:

F06: Wie wahrscheinlich ist es, das Sie unsere Software anderen Personen weiterempfehlen würden?

[ ] Äusserst wahrscheinlich [ ] Sehr wahrscheinlich [ ] Relativ wahrscheinlich [ ] Nicht sehr wahrscheinlich [ ] Überhaupt nicht wahrscheinlich

Begründung:

F07: Wie empfinden Sie die Meldungen unsere Software während der Programmierarbeit?

[ ] Sehr störend [ ] Eher störend [ ] Ab und zu störend [ ] Wenig störend [ ] Überhaupt nicht störend

Begründung:

F08: Wie sehr schätzen Sie, hat sich die Qualität des von Ihnen geschriebenen Codes während der Nutzung unserer Software verändert?

[ ] Stark verbessert [ ] Teilweise verbessert [ ] Weder verbessert noch verschlechtert [ ] Wenig verschlechtert [ ] Stark verschlechtert

Begründung:

F09: Wie hat sich Ihrem Gefühl nach die Geschwindigkeit mit der Sie Code schreiben können während der Nutzung des Plugins verändert?

[ ] Stark beschleunigt [ ] Teilweise beschleunigt [ ] Weder beschleunigt noch verschlechtert [ ] Wenig verschlechtert [ ] Stark verschlechtert

Begründung:

F10: Wie hilfreich schätzen Sie die Meldungen des Plugins ein?

[ ] Immer hilfreich [ ] Meistens hilfreich [ ] Teilweise hilfreich [ ] Meistens nicht hilfreich [ ] Nie hilfreich

Begründung:

F11: Wie können wir unsere Software Ihrer Meinung nach verbessern?

F12: Falls unsere Software während des Tests abgestürzt ist, beschreiben Sie hier bitte wann es zu einem solchen gekommen ist.

Anhang D

Testprotokoll Langzeittest

Das folgende Gedächtnisprotokoll wurde nach der Befragung mit zuvor angefertigten Notizen erstellt.

Frage 1

Frage: *Du hast das Plugin jetzt während ca. 4 Wochen begleitend zu deiner Ausbildung zum Softwareentwickler getestet. Wie ist es dir dabei ergangen?*

Antwort: Im Grossen und Ganzen ist es ihm sehr gut ergangen. Das Plugin hat nicht wirklich gestört, hat ihm aber gleichzeitig einige Fehler aufgezeigt, die er regelmässig macht.

Nachfrage 1A

Frage: *Was für regelmässige Fehler meinst du genau?*

Antwort: In erster Linie Fehler bei der Benennung von Variablen. In der Berufsschule sind seine Lehrpersonen nie wirklich auf dieses Thema eingegangen. Zwar scheinen die meisten sich bei zur Verfügung gestellten Code an die gängigsten Regeln zu halten, aber im Unterricht wurde dieses Thema nie explizit thematisiert.

Frage 2

Frage: *Bist du mit dem Plugin von Anfang an gut zurechtgekommen, oder hat es auch Momente gegeben wo du Probleme hattest bzw. selber nicht weitergekommen bist?*

Antwort: Ja, es gab von Anfang an nicht wirklich Probleme. In der ersten Version des Plugins die er getestet hat, waren die Verlinkungen zu den Erklärseiten in den Problemmeldungen noch nicht vorhanden. Da musste nachgefragt werden, wo diese zu finden sind. Das Readme in der ersten Version war auch noch nicht vollständig und zum Teil noch fehlerhaft. Das hatte aber nicht wirklich einen Einfluss auf das Nutzererlebnis.

Frage 3

Frage: *Wir haben uns mit dem Plugin das Ziel gesetzt, dass wir auch das Wissen um die Clean Code Prinzipien indirekt vermitteln. Hast du deiner Einschätzung nach etwas über Clean Code gelernt und wenn ja was?*

Antwort: Ja, die Regeln zum Benennen von Variablen und Funktionen hat er mittlerweile recht gut verinnerlicht. Durch die stetigen Meldungen bzw. die farbig unterstrichenen Codeteile *nerven* beim Verbessern genug, dass im Kopf etwas hängen bleibt. Bei der Nutzung des Plugins ist dies mit der Zeit auch aufgefallen. Je weiter im Testzeitraum, desto weniger oft gab es Meldungen. Daher ist davon auszugehen, dass der jetzig geschriebene Code mittlerweile tatsächlich weniger dieser Fehler beinhaltet.

Nachfrage 3A

Frage: *Was ist mit den anderen beiden Regeln über das Law Of Demeter und die Anzahl Funktionsargumente?*

Antwort: Problemmeldungen dieser beiden Regeln sind deutlich weniger aufgetreten. Die Regeln waren vor dem Test ebenfalls nicht bekannt. Das kann aber gut daran gelegen haben, dass im Testzeitraum eher simpler Code geschrieben wurde (These der Testperson). Daher fand hier leider auch kein nachhaltiger Lerneffekt statt.

Frage 4

Frage: *Zu jeder Regel gab es Erklärseiten, die das der Regel zugrundeliegende Problem, Lösungsansätze, sowie Vor- und Nachteile beschreibt. Wie würdest du diese Seiten bewerten?*

Antwort: Die Seiten bzw. deren Inhalt waren alle klar verständlich. Der Aufbau wirkte auch stets schlüssig, verwirrende Aspekte wären keine aufgefallen.

Nachfrage 4A

Frage: *Hast du Verbesserungsvorschläge für die Seiten bzw. deren Aufbau?*

Antwort: Am Aufbau muss nichts geändert werden. Bei den Code Beispielen wäre es hilfreich, die Klassen und ihre Beziehungen mittels eines UML Diagrams darzustellen. So könnte der Beispielcode noch einfacher verstanden werden. Inhaltlich wünschenswert wären grafische Veranschaulichungen der Prinzipien soweit dies möglich ist.

Frage 5

Frage: *Wie würdest du die dem Plugin beliegende Dokumentation bewerten?*

Antwort: Anfänglich war leider kaum Dokumentation vorhanden. Gegen Ende des Testzeitraums kam aber erfreulicherweise immer mehr dazu. Der aktuelle Stand der Dokumentation wird also solide eingeschätzt. An gewissen Orten gibt es aber noch Potential zur Nachbesserung.

Nachfrage 5A

Frage: *Kannst du diese Orte in der Dokumentation bitte genauer benennen?*

Antwort: Im Readme steht unter Knows Issues lediglich, dass der Law Of Demeter check überarbeitet werden muss. Hier wäre es schön, wenn noch etwas genauer beschrieben werden würde was der aktuelle Stand und die aktuellen Implikationen sind. Ausserdem fehlen im Readme Verlinkungen ins Wiki. Die Regelerklärseiten sind so nur ersichtlich, wenn man das Plugin lokal installiert hat und auf einen Fehler trifft. Das ist schade, weil es vllt. auch Interessierte gibt, die sich zuerst die Regeln genauer anschauen möchten. Letztlich fehlt auch eine Roadmap, die aufzeigt *wohin die Reise gehen soll*, also was in Zukunft an Funktionalität im Plugin erwartet werden darf.

Frage 6

Frage: *Würdest du das Plugin anderen Programmierenden weiterempfehlen? Wenn ja wem und warum?*

Antwort: Grundsätzlich würde eine generelle Weiterempfehlung abgegeben werden, da Fehler bei Benennungen allen, auch erfahren Programmierenden passieren. Aber vor allem Programmierende auf Anfängerniveau können bei der Verwendung des Plugins profitieren und sofort *besseren* Code schreiben.

Frage 7

Frage: *Sind dir während des Testzeitraums Bugs aufgefallen?*

Antwort: Ja, sehr früh war ersichtlich, dass einige Systemaufrufe wie z.B. *System.out.println* als Law of Demeter Violation angezeigt werden. Ausserdem haben bei ihm die Plugin Settings, die man in der IDE vornehmen kann, nicht funktioniert.

Frage 8

Frage: *Kam es bei der Nutzung des Plugins zu anderen, gravierenden Problemen wie z.B. Abstürzen oder ähnlichem?*

Antwort: Nein, nichts was mit dem Plugin in Verbindung zu bringen wäre.

Frage 9

Frage: *Wie könnten wir das Plugin deiner Meinung nach weiter verbessern?*

Antwort:

- Zusätzlicher Funktionsumfang (weitere Regeln)
- Eine Möglichkeit einzelne Problemmeldungen, die man selbst als falsch einschätzt, zu unterdrücken (Suppression Kommentare)
- Ausweitung des Sprachsupports auf weitere Programmiersprachen

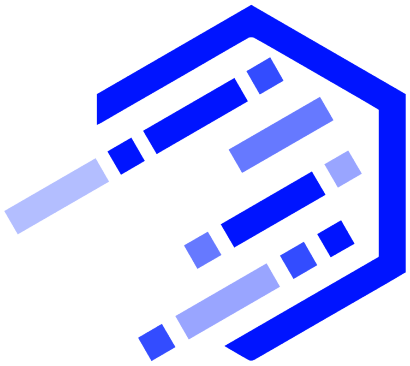
Protokollführer: Rafael Fuhrer

Befragte Person: Kevin Greminger

Ort, Datum: Dübendorf, 07.06.2022

Anhang E

Architekturdokumentation



CLEANCODE

Clean Code Extension architecture documentation

Rafael Fuhrer, Pascal Schneider

2022–03–28

About arc42

arc42, the Template for documentation of software and system architecture.

Created and maintained by Dr. Peter Hruschka, Dr. Gernot Starke and contributors.

Template Revision: 8.0 EN (based on asciidoc), February 2022

Introduction and Goals

This section describes the essential requirements for and driving forces behind the development of the architecture and implementation of the system.

Task

This clean code extension was initially developed in the course of a student research project [1] and extended as a bachelor thesis at the Eastern Switzerland University of Applied Sciences. The following section provides the complete task description translated from German.

Problem description

There exist numerous recommendations how one can write good and comprehensible program code. These recommendations are often grouped under the term “clean code”, which was coined by Martin Fowler. The recommendations concern naming variables or functions with meaningful names following a consistent scheme, for example. In pratice these rules are not always followed. One approach to rectify this would be to notify developers every time they violate a rule. This could be achieved through an IDE plugin (i.e. VSCode) or a linter extension (i.e. ESLint). Linters can check adherence to some rules. For many rules however a deeper analysis of the source code is required, for example adherence to a consistent naming scheme. For this example the system has to disassemble the identifiers into their relevant semantic parts (i.e. `"isActive"` to `"is"` and `"active"`).

Requirements

Due to the simplicity of the use cases, we decided against drawing use case diagrams. Instead we describe the requirements through a mix of user stories and MUST/CAN requirements. They are listed in table 1.

Programmer

As a developer I want to be able to activate the extension in my IDE for it to highlight all problematic parts in my source code.

[1] https://eprints.ost.ch/id/eprint/1005/1/HS%202021%202022-SA-EP-Schneider-Fuhrer-Clean%20Code%20Plugin.pdf

Table 1: Functional Requirements

ID	Priority	Description
FA-1	MUST	The extension checks the document for rule violations on save and marks them.
FA-2	MUST	All rule violations are displayed in the IDE problem log, including their position in the document.
FA-3	MUST	The extension is easy to add to the IDE through its marketplace (no manual installation required).
FA-4	MUST	The problem message contains a weblink for further information and explanations about the violated rule.

Quality Goals

The quality goals are statet in the form of non functional requirements, split into MUST and CAN requirements. They are listed in table 2.

Table 2: Non Functional Requirements

ID	Priority	Description
NF-1	CAN	Rule validation runs in its own thread and does not block the IDE.
NF-2	MUST	The extension works as intended even with large files (>1000 lines of code).
NF-3	MUST	The extension is easily extensible with additional rules and support for other programming languages.
NF-4	MUST	The source code of the extension has a rating of A in SonarQube.
NF-5	CAN	Unit test coverage exceeds 80%.
NF-6	CAN	Developer that don't yet know the source code, can find relevant parts within 30 minutes when fixing problems.
NF-7	MUST	The extension is published as an open source project including guidelines for contribution.

Stakeholders

The stakeholders of this project are listed in table 3 including their role, name and expectations concerning architecture and documentation.

Table 3: Stakeholders

Role	**Name**	**Expectation**
Advisor	Prof. Frieder Loch	Knowledge gain; Good result which can be extended in the future
Developer	Rafael Fuhrer Pascal Schneider	Good result; Learning new technologies; Clean implementation and documentation

System Scope and Context

We have pictured the relations between context elements in image 1. An explanation of all the context elements and their relations is listed in table 4.

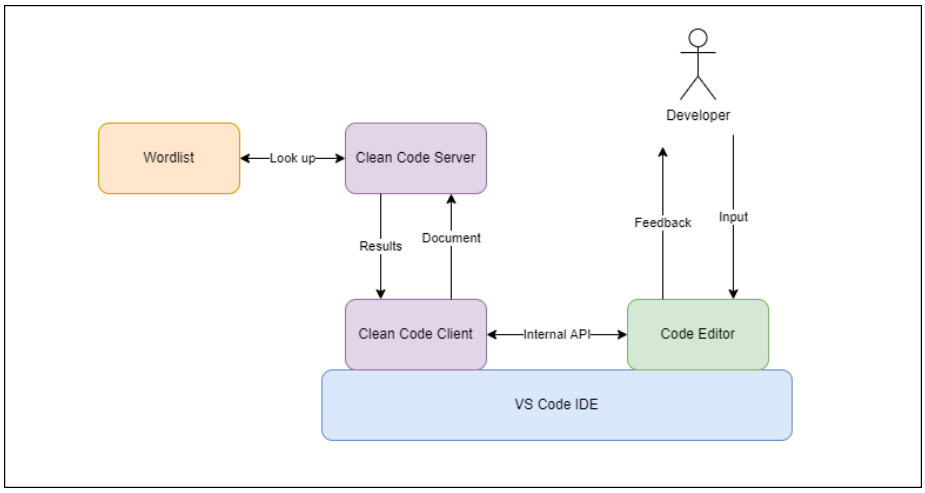


Figure 1: Context diagram

Table 4: Context relations

Element	**Format**	**Communication via**
Wordlist	JSON-File	Direct file access from Clean Code Server
Clean Code Server	TypeScript	Exchanges data with Clean Code Client through Node IPC
Clean Code Plugin	TypeScript	Accesses the source code via internal Visual Studio Code API
Code Editor	TypeScript	Provides interface to the programmer

Solution Strategy

Programming language

This extension was extended from a prototype clientside extension that we built during a student research project. As a Visual Studio Code extension it had to be either JavaScript or TypeScript and we chose TypeScript for its extended capabilities and type safety. To be able to reuse logic implemented before, we decided to keep both the language client and the language server in TypeScript.

Client side extension vs. language server

Due to the potentially large amount of computation required to check all the rules for a large file we decided to implement the extension in the form of a language server extension. Another deciding factor was the reusable nature of the server part of the extension. Once support for more languages than Java is implemented, the server could be used in plugins for different IDEs thus eliminating the problem of having to implement the same validation logic for N languages in M IDE plugins.[2]

Dictionary

For certain rules, in particular those regarding naming of identifiers, there is a need for an english dictionary. This dictionary could either be hosted externally and be queried by the extension via an API or be bundled into the extension as a file. Due to privacy and compliance reasons we decided to bundle the dictionary into the extension. If we had hosted the dictionary externally there would have been the possibility of collecting and evaluating the incomming calls. This also concerns metadata, i.e. at what time does someone write code, which would raise data protection concerns. In relation with commercial source code, which is usually confidential, the extension could have potentially violated guidelines and therefore not been usable in those environments. The final dictionary was collected by us from publicly available information that is available under CC-0 license. It is comprised of almost one hundred thousand of the most used english words including their definition. If a valid word is not contained it can be added manually to a whitelist and conversely if a word from the dictionary should not be used it can be added to a blacklist.

Notifying the programmer

The notification to the developer is comprised of a description of the problem, the exact position in the document as well as a link to an external wiki that explains the rule in more detail. This notification is displayed in the “Problems” window of Visual Studio Code as well as highlighted in the editor window through use of colored underlining. This is achieved with the help of the DiagnosticCollection

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

that is provided by the Visual Studio Code Extension API. It is possible to configure individual violatons to either be displayed as warnings, errors or disabled.

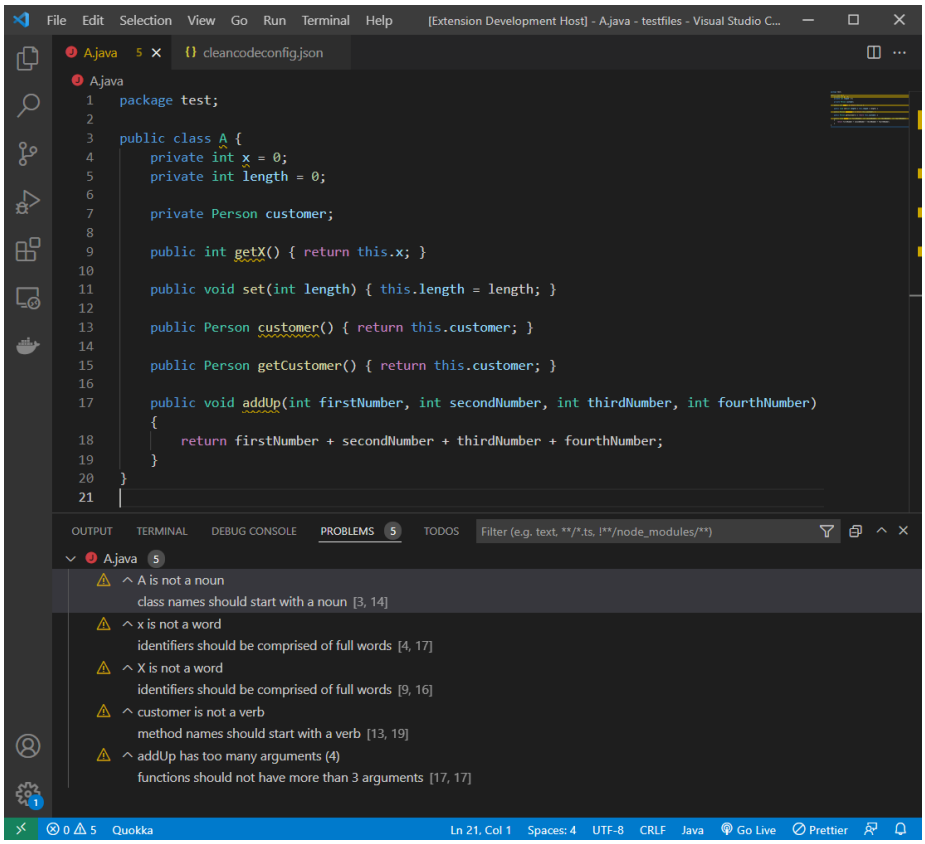


Figure 2: Notification Example

User settings

To enable users of the extension to have better control we added settings for a white- and blacklist to add missing words or block words from being used. Additionally there are settings to globally disable every violation individually ("off") or specify if it should be reported as a warning ("warning") or an error ("error").

Although these settings can be set directly in Visual Studio Code users can create a `cleancodeconfig.json` file in their project folder to manage their extension settings. This approach makes it possible to fine tune the settings for each project and also share it with other developers easily. If a settings file is present, settings made in Visual Studio Code are ignored. This settings file should be

structured as the following:

```
{
  "maxNumberOfProblems": 100,
  "blacklist": ["foo", "bar"],
  "whitelist": ["api"],
  "blacklistViolation": "off",
  "functionArgumentCountViolation": "warning",
  "lawOfDemeterViolation": "error",
  "namingViolation": "off",
  "wordTypeViolation": "warning"
}
```

Building Block View

The extension is split into two parts, a client and a server. While the client is relatively light weight, the server itself is split into several subsystems, which perform their parts of the code validation independantly. These subsystems are tied together by the server extension that then invokes whichever system is required for the next step. This design enables us to replace single subsystems if required. The client and server communicate through the Language Server Protocol over the Node inter-process communication module.

Whitebox Overall System

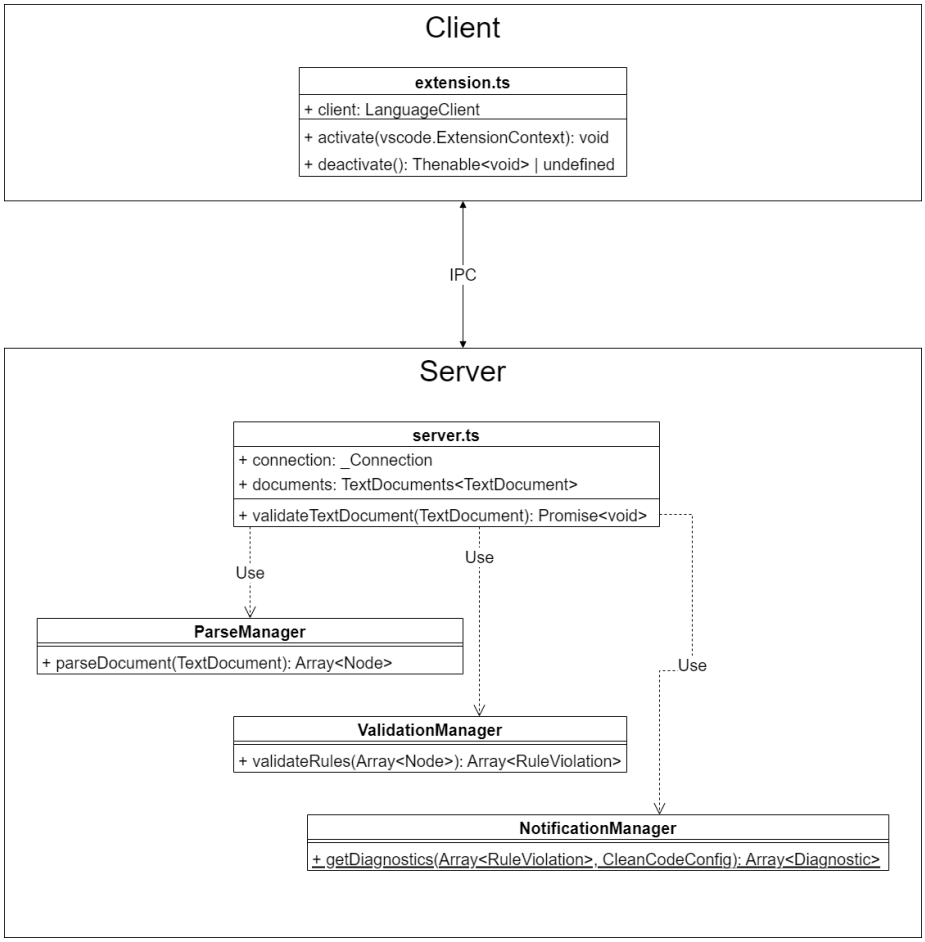


Figure 3: Overall System

Description The extension is actually comprised of two extensions that communicate through the LSP [3] over Node ipc. This allows offloading of computation-heavy operations to another process instead of blocking Visual Studio Code with long running functions.

Client The client is a lightweigt extension that runs in the same process as Visual Studio Code and just informs the server what files are opened, closed and edited. It furthermore communicates initial states and changes in user settings relevant to the extension.

Server The server is an implementation of a language server in Node. It runs in an independant process and could in theory be used as the language server for other plugins in other IDEs. The server runs the code checking logic and returns violations to the client.

Contained Building Blocks The server has three main subsystems, the `ParseManager` to transform source code into `Nodes`, the `ValidationManager` to check if any of those `Nodes` contain violations against our rules and lastly the `NotificationManager` which converts the results into useful feedback for the developer.

Important Interfaces All communication between the client and the server happens over the node ipc channel that is opened when the extension is initialized.

Important Concepts Interfaces are used for objects that need language specific implementations (i.e. parser, lexer) as well as for objects that are implemented in different variations (i.e. rules, validators). One reason for this is to hide implementation details from calling classes. Another reason is the ability to decide at runtime which implementation should be used for a given situation.

Abstract classes are used for classes that provide a set of basic funtionalities but are designed to be inherited and extended with specific functionality. An example for this is the class `Node` which provides the functionality to find its location in the source code but needs to be extended with the specific logic depending on the type of node.

The get/set functionality of TypeScript is used instead of traditional accessor methods to make private fields available outside of a class.

```
class Token {
  private_range: vscode.Range;

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

[3] https://microsoft.github.io/language-server-protocol/

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

...

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

Client

The client extension does not contain any of the logic required for the rule validation. It is intentionally kept as simple as possible so that it is less work to implement a corresponding client plugin in a different IDE while reusing the server extension as is. The singular TypeScript file is `extension.ts` which implements the functions `activate` and `deactivate` which are used by Visual Studio Code to activate and deactivate the extension. In addition there are some functions to handle interaction with the context menu entries the extension adds to editor views of supported languages and to look for a configuration file and make it available to the server.

Server

The server extension implements all of the logic for validating rules. It is comprised of dufferent subsystems to parse source code, validate rules and notify the developer. These subsystems don't rely on each other so that they can be reworked or replaced without the other subsystems needing any change.

Parsing

An overview of the parsing subsystem can be found in figure 4.

Description The logic for parsing source code is managed by the `ParseManager` class. It chooses the correct `Parser` for a given language which then extracts the required `Nodes` from the document.

The `Parser` interface defines what functions a language specific parser has to implement. The actual implementation of a parser can vary. If there are robust libraries available for a given language, one of those can be used to do the heavy lifting. The parser then only has to map the return of the used library to corresponding `Nodes` so that it returns in a format that the following subsystems understand. If no usable library exists a parser can be implemented with the help of the existing `Lexer` interface and the associated helper classes.

Contained Building Blocks The `JavaParser` class is an implementation of the `Parser` interface for the programming language Java. It uses the `java-parser` npm package which returns a concrete syntax tree for a given java document. The tree then is traversed using the `CleanCodeVisitor` during which the relevant `Nodes` are collected and returned to the `ParseManager` at the end.

Specific `Nodes` that contain all relevant information required for the implemented rule validations.

Important Interfaces The `parseDocument` method receives a document and returns a list of all relevant `Nodes`.

Important Concepts The visitor pattern is used to traverse the syntax tree that is built by the used parsing library. The pattern enables visiting all leaves of a tree and executing operations on chosen leaves while not changing anything on the underlying tree class.

Validation

An overview of the validation subsystem can be found in figure 5 and figure 6.

Description The class `ValidationManager` manages the validation of rules. For specific rules the interface `Rule` with its method `checkRule` is implemented. The implemented rules are provided to the `ValidationManager` through implementations of the `Validator` interface.

The additional layer between `Rule` and `ValidationManager` improves flexibility, because it enables the reuse of rule logic for several `Nodes`, if they have to abide by the same rules.

Rule infractions are added to the `ValidationManager` property `RuleViolations` which is returned after all the checks have run.

Contained Building Blocks `Validator` implementations check a list of specific `Nodes` against set of rules defined for those nodes.

`Rule` implementations check a single `Node` against a rule.

The class `IdentifierHelper` implements logic to break an identifier into its individual words and to check if those words are contained in the dictionary.

The `RuleViolation` class contains all the information about the position of a rule violation in the source code and about its type. This information is later used to notify the developer.

Important Interfaces The `validateRules` method takes a list of `Nodes` and checks whether they violate the implemented rules.

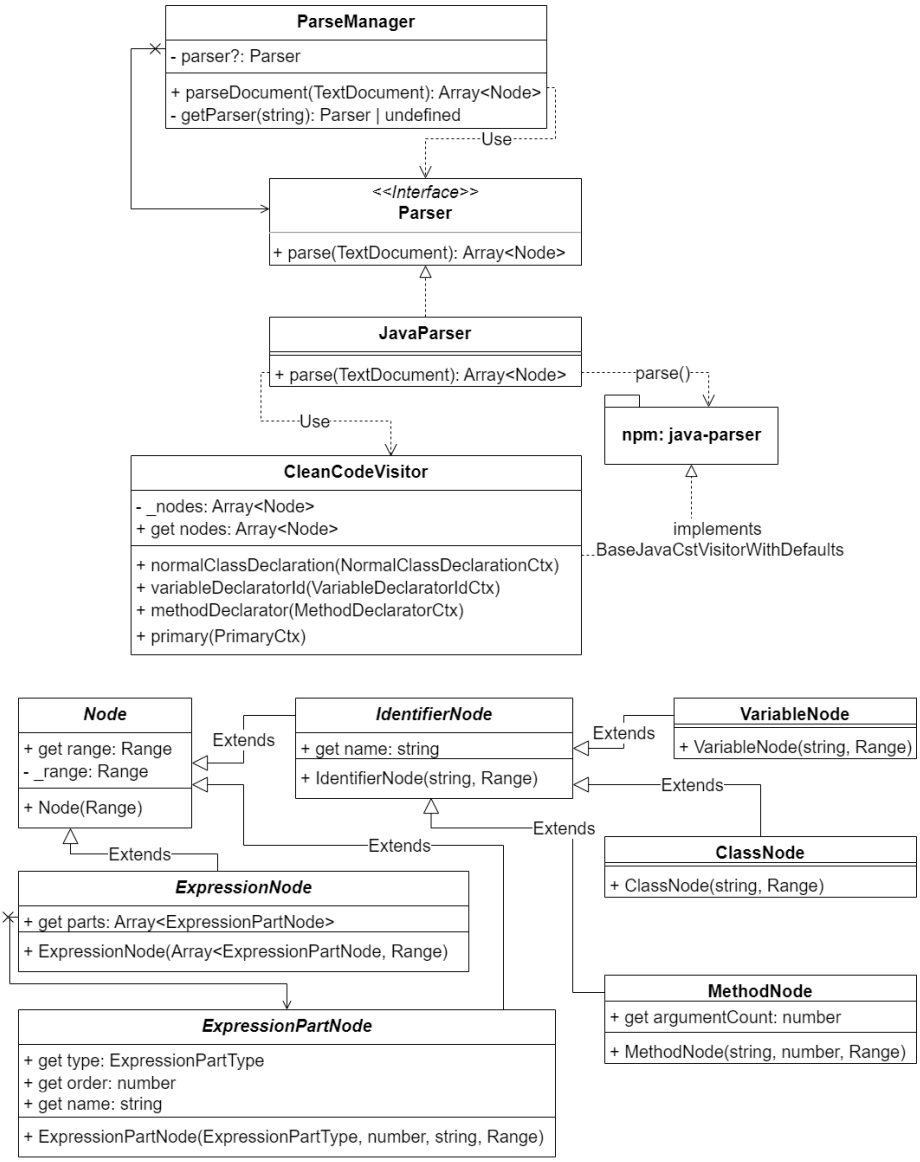


Figure 4: Overview Parsing Subsystem

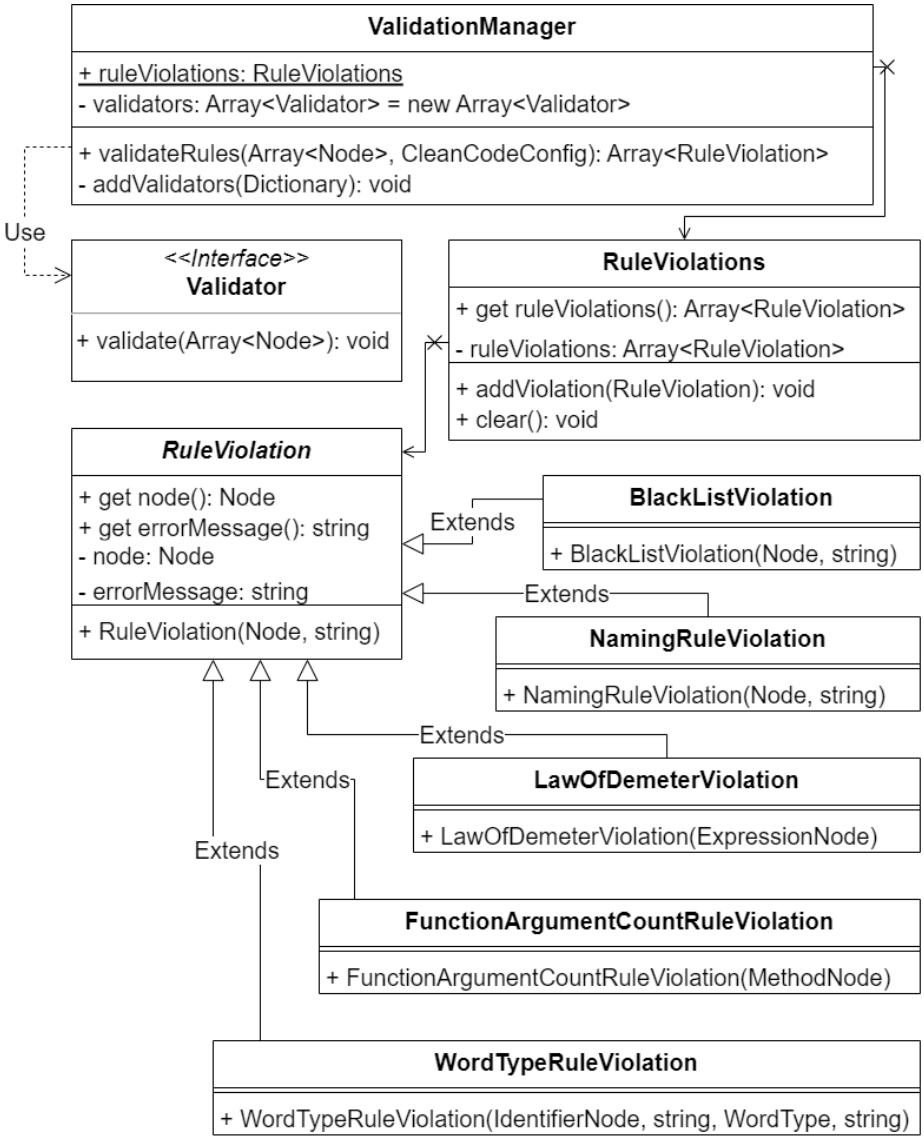


Figure 5: Overview Validation Subsystem

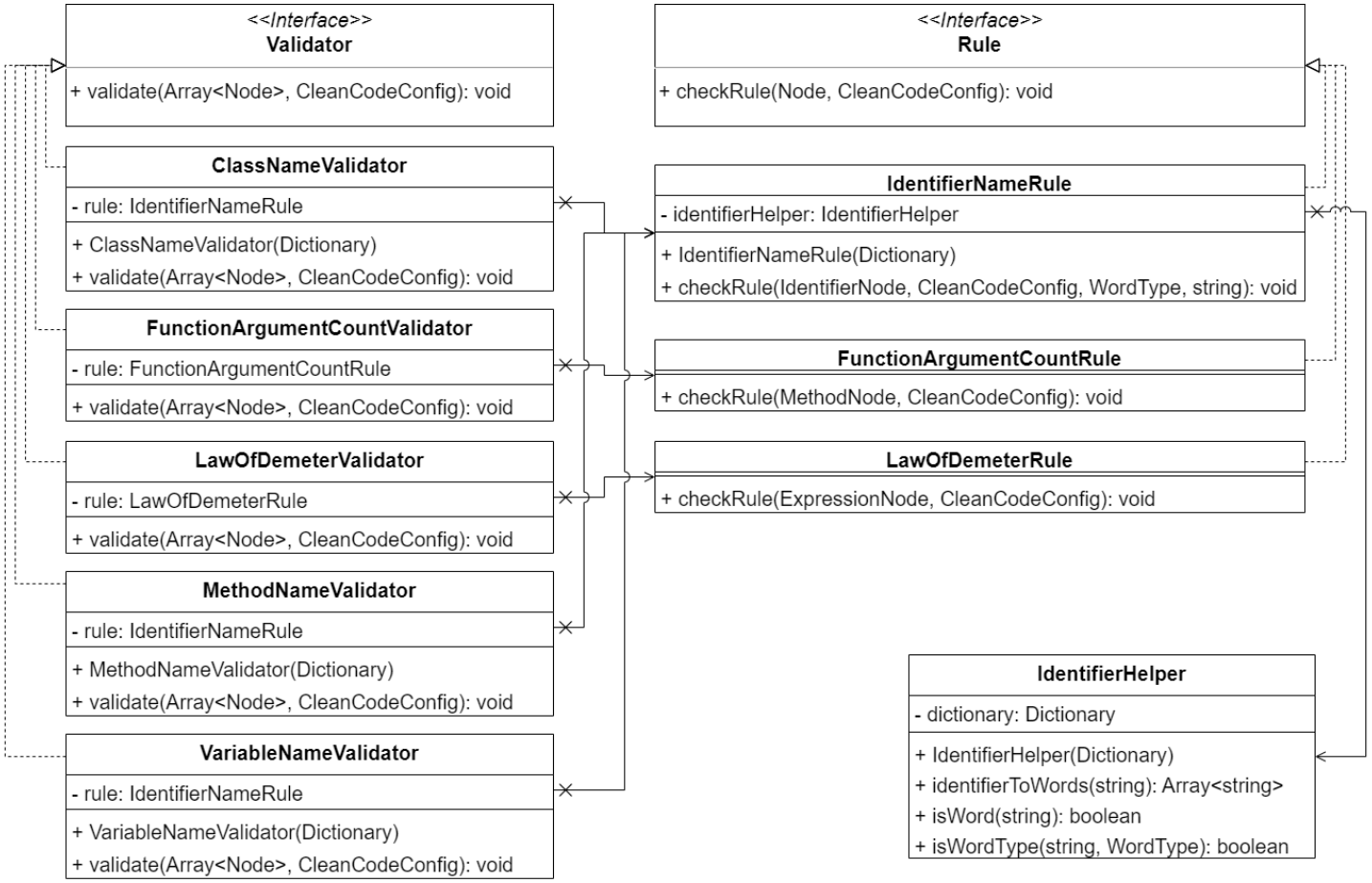


Figure 6: Validators and Rules

Notification

Description The class `NotificationManager` translates the rule violations that were found into Visual Studio Code `Diagnostic` objects and returns them.

Depending on the settings it returns them as either a warning, an error or not at all.

Important Interfaces The `createDiagnostics` method takes a list of `RuleViolation`s and the current `CleanCodeConfig` and returns a list of `Diagnostic`s.

Runtime View

If the extension configuration, an opened document or a file watched by the extension is changed, the information is routed to the server where either that one document or all opened documents are checked again.

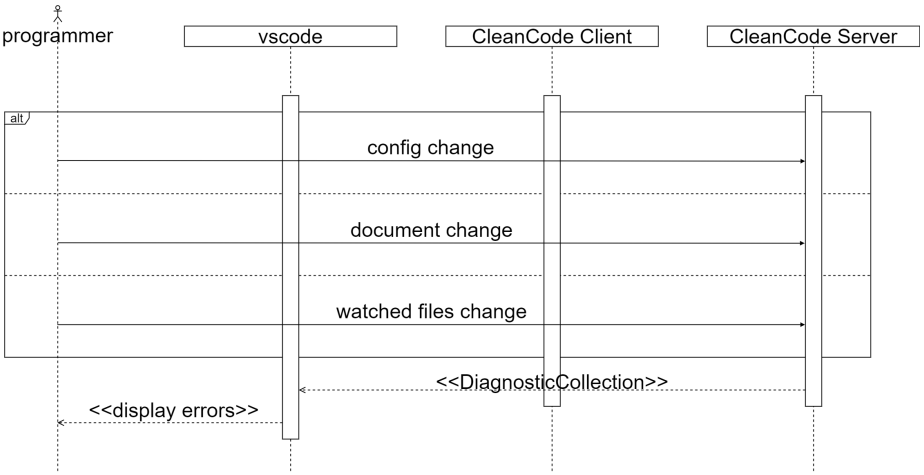


Figure 7: Overview Runtime View

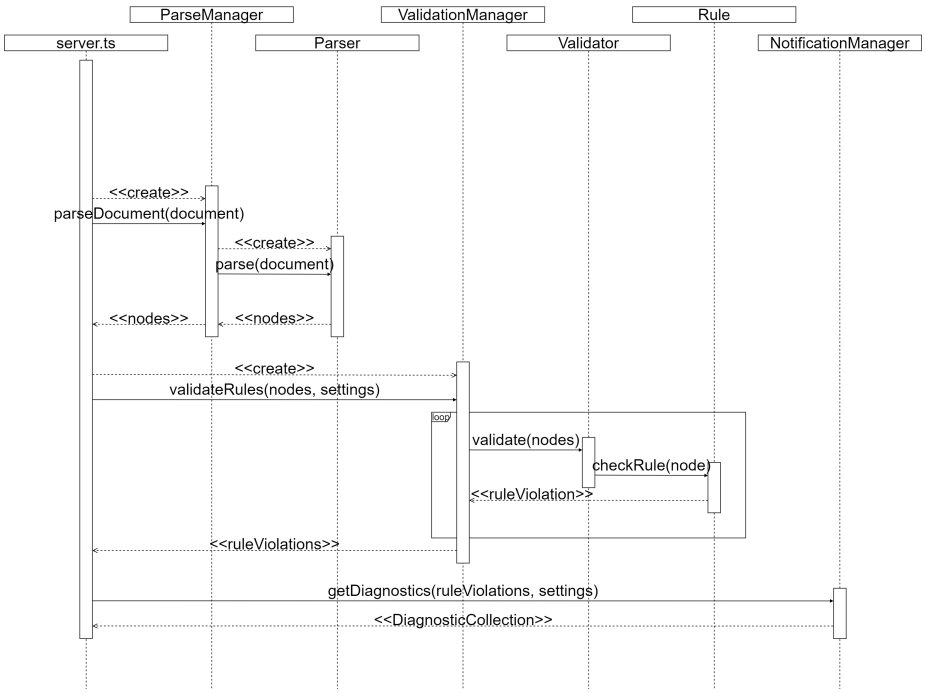


Figure 8: Server Runtime View

1. The `server` first creates an instance of `ParseManager` and calls its `parseDocument` method with the opened document as a parameter.
2. The `ParseManager` then creates an instance of the required language specific `Parser` implementation and calls its `parse` method with the document as a parameter.
3. The `parse` method parses the document into an array of `Nodes` which it returns to the `ParseManager` that in turn returns them to the `server`.
4. Now the `server` creates an instance of `ValidationManager` and calls its `validateRules` method with the array of `Nodes` generated by the `Parser`.
5. In the `validateRules` method every registered `Validator` is executed sequentially on the `Nodes` and all `RuleViolations` are added to an array which is returned to the `server` at the end.
6. Finally the `server` creates an instance of `NotificationManager` and calls its `getDiagnostics` method with the array of `RuleViolations` as a parameter. The `getDiagnostics` method transforms these `RuleViolations` into `Diagnostic` objects and returns an array of them to the `server`.
7. The `server` returns the collected `Diagnostic` collection to the client.

Deployment View

The extension is installed locally by developers, therefore we do without a diagram of the deployment. The relevant information can be seen in the context diagram in figure 1.

Infrastructure

The infrastructure is shown in figure 9.

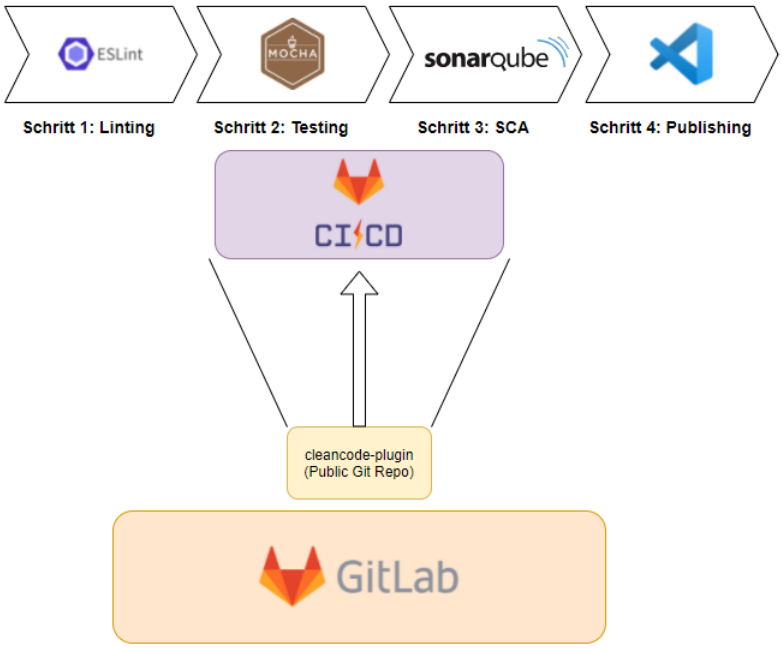


Figure 9: Infrastructure Overview

The project uses two git repositories each of which serves a specific purpose and is listed in table 5.

The cleancode-plugin repository contains a GitLab continuous integration pipeline declaration. The pipeline is comprised of four steps that are listed in table 6.

Table 5: Git Repositories

Repository Name	**Technologies**	**Function / Purpose**
cleancode-plugin	TypeScript & JavaScript, LaTeX, Markdown	Contains the source code of the extension including the build pipeline and all the documentation
report	LaTeX	Contains the files and documents of the bachelor thesis

Table 6: Pipeline Steps

Step Name	**Technology / Tool**	**Purpose**
lint	ESLint	Language specific static code analysis (linting).
test	Mocha, Chai, Istanbul	Execution of the defined unit tests to ensure functional correctness.
sonarcloud-check	SonarQube	Static code analysis (SCA) of added source code to ensure code quality and security.
publish	Azure & VSCode Marketplace	Publishing of the extension into the Visual Studio Code Marketplace through Azure DevOps.

Architecture Decisions

The first question that needed to be answered was what platform the tool should be developed for.

IDE or linter plugin?

Since the task didn't specify what form the the result should be in our options were either an IDE plugin or a linter extension. We decided on an IDE plugin for the following reasons.

- Language independant implementation respectively not being restricted to the languages a linter we could choose supports
- Notification in the form we want not in the form the linter dictates
- Direct access to the source code
- Able to interact with IDE hooks

A plugin that is written for a specific IDE allows us to make use of these advantages whereas with a linter extension we would be restricted to the functionality the linter itself has. Furthermore the user interactions would have been limited from the beginning.

The disadvantag this choice brings is that an IDE plugin is limited to the IDE. A linter extension on the other hand would have been available for every IDE that the linter supports. Nevertheless it is our opinion that the advantages outweigh the disadvantages. Furthermore this disadvantage is minimized by the implementation as a language server extension because only the client has to be rewritten for additional IDEs.

IDE choice

The next hurdle was figuring out what IDE we should target for our plugin. After some research we narrowed it down to the five most widely used IDEs as listed in table 7.

All the IDEs in table 7 meet the requirement that you can develop plugins for them and publish these plugins in an associated marketplace. We could have developed an open source plugin for any of them. It made sense for us though to choose an IDE that itself is released under an open source license. This choice eliminated Visual Studio as a proprietary product. We excluded Jetbrains' IntelliJ as well because even though it is open source their other IDEs are not. The remaining environments are published under an open source license. To narrow it further down we removed Eclipse because it is mainly for development in Java and our plugin should be able to support multiple programming languages. The decisive factor in our choice between Atom and Visual Studio Code was that the latter has a lot more contributors and stars on GitHub. This suggests that it is more widely used and therefore our plugin would be available to a bigger audience.

Table 7: Overview IDE Choices

Name	**Maintainer**	**Open Source**	**Supported Languages**
Visual Studio Code	Microsoft	Yes	Several (hundreds via extensions)
Visual Studio	Microsoft	No	C#, C, C++, VB (some others via extensions)
IntelliJ	Jetbrains	Yes	Java (some others via extensions)
Eclipse	Eclipse Foundation	Yes	Java (some others via extensions)
Atom	GitHub	Yes	Several (hundreds via extensions)

Due to these factors we decided on the implementation of a Visual Studio Code Extension.

Glossary

Term	Definition
Application Programming Interface (API)	An API is a programming interface that a software system provides.
Arc42	Arc42 is a template for documenting software architecture by Dr. Peter Hruschka and Dr. Gernot Starke.
Continuous Integration & Continuous Development (CI/CD)	These are concepts for automating all phases of software development from programming to deployment.
Clean Code	Clean, simple and intuitively comprehensible source code with clear structure and concepts.
Git	Git is free software for distributed version control of documents, i.e. source code.
Integrated Development Environment (IDE)	Software environment for programming source code that is used by developers.
Concrete Syntax Tree	Ordered tree that maps the syntactic structure of source code based on a context free grammar.
Language Server Protocol	The Language Server Protocol (LSP) defines the protocol used between an editor or IDE and a language server that provides language features such as autocomplete, go to definition, search for all references, and so on.
Context Free Grammar (CFG)	List of rules that describe all possible permitted texts in a language.
Linter	A program that runs a static code analysis on source code (see SCA).
Pipeline	In software development a pipeline is a system of automated processes that allow quick deployment of changes in the source code into production.
Static Code Analysis (SCA)	Static code analysis is a method for finding errors in source code before it is compiled into an executable.
Unit Test	A method that tests a single unit of code.
Visual Studio Code (VSCode)	An open source IDE by Microsoft written in TypeScript.