F&E-Arbeit

Gamification zur nachhaltigen Steigerung der Code-Qualität in coderadar

Forschungs- und Entwicklungs-Arbeit

An der Fachhochschule Dortmund
im Fachbereich Informatik
Studiengang Praktische Informatik
erstellte F&E-Arbeit
zur Erlangung des akademischen Grades
Master of Science

von
Johannes Teklote
geb. am 02.01.1996
Matr.-Nr. 7091992

Betreuer:
Prof. Dr. Andreas Harrer
Vorname Nachname, Abschluss

Dortmund, Datum der Abgabe

Kurzfassung

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Laut [Bal09] entfallen 80% des Aufwands in der Softwareentwicklung auf deren Wartung, wobei 40% davon benötigt werden, um die zu wartende Software zu verstehen. Ein erheblicher Teil der Kosten für die Softwareentwicklung entfällt damit auf das Verstehen der Software. Wenn insgesamt die Entwicklungskosten reduziert werden sollen, stellt dieser Umstand einen wesentlichen Hebel dar. Denn der Wartungsaufwand im Übrigen ist vielfach kaum zu verringern, weil sich die Anforderungen an die Software im Laufe der Zeit ändern, weil auftretende Probleme behoben werden müssen oder weil Abhängigkeiten auf andere Software, Frameworks oder Komponenten angepasst werden müssen. Bereits durch das Erfüllen gewisser Qualitätsstandards lässt sich jedoch der für das Verstehen des Codes notwendige Aufwand reduzieren.

Qualitativ hochwertige Software zeichnet sich nach [ISO11] dadurch aus, dass sie stabil ist und gut gewartet, analysiert, erlernt und verändert werden kann. Software, die diese Anforderungen erfüllt, beinhaltet Code, der bestimmten Richtlinien entspricht. Zu diesen Richtlinien gehört beispielsweise, dass Code auf einer Ebene einheitlich eingerückt wird, dass öffnende geschweifte Klammern in der gleichen Zeile stehen wie der Code, der die Klammer erfordert, dass Variablennamen sprechend sind und nicht nur aus einem Buchstaben bestehen oder dass Parameter auf ihre Gültigkeit überprüft werden, bevor mit ihnen gearbeitet wird.

Die Einhaltung solcher Regeln scheint auf den ersten Blick nur von geringer Bedeutung zu sein, allerdings lenken beispielsweise Unstimmigkeiten in der Formatierung beim Lesen des Codes ab und sie erschweren es, sich auf den eigentlichen Code zu konzentrieren. In [Pra15] und [Spi11] werden diese Formatverstöße als Hintergrundrauschen beschrieben, das vom eigentlichen Code ablenkt.

Code, der nach den oben beschriebenen Regeln geschrieben wurde, ist laut [Pra15] leichter verständlich und dadurch auch leichter erlernbar und veränderbar. Um die Mitglieder des Entwicklungsteams dazu zu bringen, sich an die vereinbarten Regeln zu halten, gibt es zwei Möglichkeiten. Zum einen kann im Reviewprozess des Projekts ein automatischer Test eingebaut werden, der dafür sorgt, dass grundsätzlich funktionierender Code dann nicht angenommen wird, wenn er gegen festgelegte Regeln verstößt, wenn er also beispielsweise Stylingrichtlinien nicht einhält oder es potentiell unsichere Variablenzugriffe gibt. Zum anderen können die Teammitglieder durch Belohnungen dazu motiviert werden, von sich aus die festgelegten Regeln einzuhalten und dies auch selbst zu überprüfen. Es stellt sich nun die Frage, ob eine solche Motivation durch Gamification erreicht werden kann. Bei Gamification werden Methoden genutzt, die aus Spielen bekannt sind. Dazu gehören beispielsweise Levels, Badges oder Leader Boards, die das Team oder ein einzelnes Teammitglied erreichen kann.

Die oben beschriebene Ablehnung von eingereichtem Code auf Grund von Verstößen gegen Richtlinien führt dazu, dass die Teammitglieder die vereinbarten Programmierrichtlinien mit der Zeit von sich aus einhalten, ohne immer darauf hingewiesen werden zu müssen. Es ergibt sich also ein Lerneffekt, der bewirkt, dass die Teilnehmer in Zukunft qualitativ hochwertigen Code einreichen. Im Rahmen dieser Arbeit soll untersucht werden, ob beziehungsweise in wie weit auszuwählende Gamification-Elemente ebenfalls die Motivation erhöhen, qualitativ hochwertigeren Code einzureichen. Neben qualitativ hochwertigem Code könnte so durch einen gewissen „Spaßfaktor“ auch eine höhere Zufriedenheit der Teammitglieder erreicht werden.

1.2 Zielsetzung

Ziel dieser Arbeit ist die Erweiterung der Codeanalyseplattform coderadar [1] um Gamification-Elemente zur Steigerung der Code-Qualität. Dazu werden zunächst Analysemechanismen zur Bewertung des Codes ausgewählt. Anschließend werden Gamification-Elemente ausgewählt und in coderadar implementiert. Mit einer Balancierungsmatrix wird dann die Bewertung des Codes gewichtet und in eine absolute Punktzahl umgerechnet. Auf dieser Punktzahl basieren dann Levels, Leader Boards und ähnliches. Ein Feldversuch zur

[1] `https://github.com/adessoAG/coderadar`

Feststellung, in wie weit die implementierten Maßnahmen die Softwarequalität nachhaltig verbessern können, soll im Rahmen der F&E-Arbeit nicht stattfinden. Dies geschieht erst im Rahmen der Masterthesis.

1.3 Vorgehensweise

Zunächst werden in Kapitel 2 die technischen Grundlagen für diese Arbeit vorgestellt. Dazu gehört zunächst die Definition von Gamification im Rahmen dieser Arbeit in Abschnitt 2.1. Im Anschluss daran wird in Abschnitt 2.2 beschrieben, was qualitativ gute Software genau ausmacht. Dazu wird ausgeführt, welche Qualitätsmetriken es für Software gibt und wie ihre Werte zu verstehen sind.

In Kapitel 3 wird dann die Problemstellung analysiert. Dazu wird zunächst in Abschnitt 3.1 das grundlegende Problem beschrieben. Dieses Problem wird anschließend in Abschnitt 3.2 in den bestehenden Kontext eingeordnet. Auf Basis dieser Ergebnisse werden dann Anforderungen an eine Software herausgearbeitet, die im Rahmen dieser Arbeit implementiert werden und die beschriebenen Probleme lösen soll. Abschließend erfolgt dann die wissenschaftliche Einordnung der Arbeit in Abschnitt 3.4.

Anschließend erfolgt in Kapitel 4 der Entwurf eines Lösungsansatzes für das oben beschriebene Problem. Dazu wird zunächst in Abschnitt 4.1 vorgestellt, wie der Code analysiert werden kann. Ein besonderes Augenmerk wird dabei auf mögliche Herausforderungen während der Analyse gelegt. Anschließend wird in Abschnitt 4.2 auf Basis der bestehenden coderadar-Architektur eine sinnvolle Möglichkeit zur Ergänzung der Analyse und Gamification erarbeitet. Danach wird in Abschnitt 4.3 eine Matrix erarbeitet, in der erläutert wird, in welchem Umfang welche Metrik in die Berechnung eines Scores für die Güte der Software einfließt. Des Weiteren wird in Abschnitt 4.4 erarbeitet, wie welche Gamification-Elemente im User-Interface verwendet werden können, um den berechneten Score darzustellen und Motivation zur Verbesserung dieses Scores zu liefern.

Kapitel 5 beschäftigt sich dann mit der programmatischen Umsetzung des beschriebenen Lösungsansatzes.

Abschließend werden in Kapitel 6 die Ergebnisse dieser Arbeit zusammengefasst. Dazu wird zunächst in Abschnitt 6.1 die Umsetzung der gestellten Anforderungen verifiziert.

Anschließend wird in Abschnitt 6.2 ein Fazit aus der Implementierung gezogen. Zum Schluss wird ein Ausblick auf die weitere Arbeit mit der erarbeiteten Gamification in coderadar gegeben.

2 Technische Grundlagen

Dieses Kapitel geht auf die technischen Grundlagen der Arbeit ein. Dazu gehört zunächst die Erläuterung von Gamification und Gamification-Elementen in Abschnitt 2.1. Danach werden in Abschnitt 2.2 grundlegende Kriterien für qualitativ hochwertige Software aufgestellt.

2.1 Gamification

In diesem Abschnitt wird der Begriff Gamification erläutert. Dazu geschieht zunächst in Abschnitt 2.1.1 eine Definition des Begriffs sowie eine Einordnung der Gamification in die Formen des Spielens nach [DDKN11]. Anschließend wird in Abschnitt 2.1.2 vorgestellt, wie Motivation erreicht beziehungsweise gesteigert werden kann. In Abschnitt 2.1.3 werden dann einige Gamification-Elemente vorgestellt. Im Laufe der Arbeit wird dann ausgearbeitet welche Elemente verwendet werden und welchen Zweck sie erfüllen sollen.

2.1.1 Definition und Einordnung

Nach [SHMK14] ist Gamification die Nutzung der Motivationskraft von Spielen in anderen Bereichen, die nicht der Unterhaltung dienen. Um dies zu bewerkstelligen werden Elemente aus dem Gamedesign verwendet. Zu diesen Elementen gehören beispielsweise Punktestände, Level, Bestenlisten, Fortschrittsanzeigen oder Aufgaben beziehungsweise Herausforderungen (vgl. Abschnitt 2.1.3). Durch die Ausrichtung dieser Elemente an einem übergeordneten Ziel kann die Motivation der Spieler auf das Erfüllen dieses Ziels gerichtet werden (vgl. Abschnitt 2.1.2). Im Rahmen dieser Arbeit besteht dieses Ziel darin, möglichst qualitativ hochwertigen Code zu produzieren. Entsprechend würde ein

Spieler, der qualitativ hochwertigeren Code einreicht, mehr Punkte bekommen und somit auch schneller im Level aufsteigen.

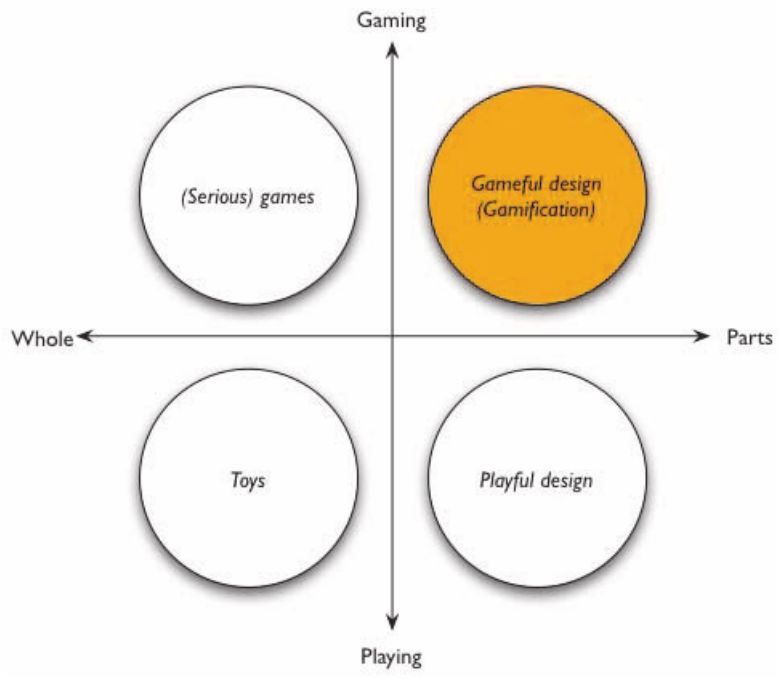

Abbildung 2.1: Die verschiedenen Arten des Spielens nach [DDKN11].

Der Begriff Gamification ist von dem Wort „Gaming “ abgeleitet. Gaming bezieht sich dabei auf ein seriöses Spielen. Dies steht dabei dem „Playing “ gegenüber (vgl. Abbildung 2.1). Playing beschreibt nach [Wal03] ein Spielen in einer offenen Welt, die Oft nur durch die Kreativität der Spieler beschränkt ist. Der Aufbau der Spielwelt, beispielsweise mit Spielzeugen wie Lego, ist dabei ein zentraler Aspekt. Gaming unterscheidet sich[DDKN11] von Playing dadurch, dass das Gaming einem Regelwerk unterliegt, welches die Teilnehmer beim Erreichen eines gesetzten Ziels einschränkt und somit eine Herausforderung darstellt. Am Beispiel Schach verdeutlicht, liegt das Ziel darin, den gegnerischen König zu schlagen, ohne dass der eigene König geschlagen wird. Einschränkend gilt hierbei, dass die Spieler immer abwechselnd ziehen und die Figuren in ihrer Beweglichkeit eingeschränkt sind. Zusätzlich gibt es teilweise noch eine zeitliche Einschränkung, sodass für alle Züge zusammen in Summe nur eine gewisse Zeit zur Verfügung steht.

Gamification ist die Übertragung von Elementen aus dem Gaming in Bereiche, die nicht

mit dem Spielen oder der Unterhaltung generell verwandt sind. Am Beispiel Joggen verdeutlicht bedeutet das, dass als Ziel eine vorher festgelegte Distanz pro Woche definiert wird. Die Möglichkeiten, dieses Ziel zu erreichen, werden dadurch eingeschränkt, dass zum Erreichen nur eine festgelegte Zeit zur Verfügung steht und der Spieler gegebenenfalls die zu erreichende Distanz nicht in einem Mal zurücklegen kann und von daher mehrere Läufe aufsummieren muss.

2.1.2 Spielermotivation

Motivation ist laut [SHMK14] ein psychologischer Prozess, der ein zielgerichtetes Handeln initiieren oder verstärken kann. Für die Effektivität von Gamification ist entscheidend, dass beim Spieler Motivation generiert wird, das gesetzte Ziel zu erreichen. Dazu kann das Fördern von Motivation aus verschiedenen Perspektiven betrachtet werden, entsprechend gibt es verschiedene Arten, auf die Motivation gefördert werden kann.

Bedürfnisse des Spielers Spieler haben laut [SHMK14] verschiedene Bedürfnisse, die sie erfüllen wollen. Diese sind dabei abhängig von der Ausprägung unterschiedlicher Charakterzüge eines Spielers. Generell gibt es das Bedürfnis nach Erfolg, Einfluss und Statussymbolen und nach Zusammengehörigkeit. Dadurch, dass einem Spieler ein Erfolg oder eine Gruppenzugehörigkeit in Aussicht gestellt wird, hat der Spieler nun die Möglichkeit diese Bedürfnisse zu erfüllen und er wird versuchen, die notwendigen Anforderungen zu erfüllen.

Positive und negative Eindrücke Aus der Betrachtung der verhaltensorientierten Perspektive geht laut [SHMK14] hervor, dass positive und negative Eindrücke oder Erfahrungen zukünftiges Verhalten beeinflussen. Entsprechend kann durch positives und negatives Feedback oder Belohnungen und Bestrafungen zukünftiges Verhalten gesteuert werden.

Erfüllung von Erwartungen Laut [SHMK14] ist Motivation aus der kognitiven Perspektive abhängig von den gestellten Erwartungen. Diese Erwartungen können allgemeingültig sein und beispielsweise anhand des Alters oder der Fähigkeiten oder individuell

durch den Spieler festgelegt werden. Motivation wird dabei durch die Erfüllung beziehungsweise das Bedürfnis nach Erfüllung dieser Erwartungen generiert.

Selbstverwirklichung Selbstverwirklichung ergibt sich nach [SHMK14] aus den Bedürfnissen nach Autonomie, Kompetenz und sozialer Verbundenheit. Die Erfüllung dieser Bedürfnisse fördert laut [SHMK14] die intrinsische Motivation des Spielers.

Nutzerinteressen Die Interessensperspektive berücksichtigt laut [SHMK14] individuelle Vorlieben und inhaltliche Aspekte. Motivation ergibt sich demnach aus der Beziehung des Spielers zu der Aufgabe oder dem Kontext. Idealerweise kann dies laut [SHMK14] zu einem vollständigen Eintauchen in die Aufgabe führen, dem sogenannten „Flow “. [ZC11] erklärt Flow als einen Zustand, in dem der Anspruch einer Aufgabe den Fähigkeiten des Spielers exakt entspricht. Übersteigen die Fähigkeiten den Anspruch der Aufgabe, langweilt sich der Spieler und er ist weniger motiviert. Übersteigt der Anspruch die Fähigkeiten hingegen, gerät der Spieler unter Leistungsdruck und er sorgt sich um die Erfüllung der Aufgabe, was laut [Ast00] bis hin zu Angstzuständen führen kann. Auch dadurch sinkt die Motivation des Spielers. Das Erreichen der Flow-Zone, in der die Fähigkeiten des Spielers den Ansprüchen der Aufgabe entsprechen, erfordert demnach eine genaue Abstimmung dieser beiden Aspekte.

Emotionales Empfinden Auch über die emotionale Ebene lässt sich laut [SHMK14] Motivation generieren. [Ast00] führt an, dass unterschiedlich formulierte Aufgabenstellungen Motivation fördern oder einschränken können. Demnach lässt sich Motivation durch die Verminderung negativer Emotionen wie Angst, Ärger oder Neid und die Verstärkung positiver Emotionen wie Sympathie oder Freude fördern.

2.1.3 Gamification-Elemente

Um in einem Spiel die Motivation des Spielers zu steigern, ist wichtig, dass er ein Feedback zu seinem aktuellen Fortschritt bekommt. Am Beispiel Schach geschieht dies die Stellung der Figuren und somit die Feldhoheit. Am Beispiel der Gamification von Joggen könnte dies ein Abbild der Gesamtdistanz und ein entsprechender Positionsmarker mit

der aktuell zurückgelegten Distanz sein. Wie oben beschrieben, werden für diese Darstellung Elemente aus dem Gamedesign verwendet. Im Folgenden werden einige Elemente beispielhaft vorgestellt und ihr Nutzen erläutert.

Punkte Punkte dienen laut [SHMK14] zur Darstellung der aktuellen Situation. Sie können auch zur Veranschaulichung von Belohnungen verwendet werden und geben ein direktes Feedback. Dieses Feedback kann durch das Erhalten von Punkten positiv beziehungsweise durch das Verlieren von Punkten negativ sein.

Badges Badges oder Auszeichnungen sind Statussymbole innerhalb der Gamification. Sie erfüllen das Bedürfnis des Spielers nach Erfolg und Anerkennung (vgl. [SHMK14]). Darüber hinaus fungieren sie laut [SHMK14] als eine Form der Gruppenidentifikation, da sie gemeinsame Erfahrungen kommunizieren. Für Spieler, die ein bestimmtes Badge noch nicht erreicht haben, können sie außerdem als Ziel fungieren, dessen Belohnung dann das Erhalten dieses Badge ist.

Fortschrittsbalken Fortschrittsbalken veranschaulichen einen Fortschritt hinsichtlich eines Ziels. Sie gelten dabei individuell für einen Spieler und zeigen die Erreichbarkeit des jeweiligen Ziels.

Leistungsdiagramme Leistungsdiagramme setzten einen individuellen Score in Relation zu anderen Scores. Diese anderen Scores können von anderen Spielern oder anderen Zeitpunkten stammen. Stammen sie von anderen Spielern, zeigt das Diagramm, wie der Spieler sich im Vergleich zu anderen Spielern im Bezug auf ein Ziel hin entwickelt. Betrachtet das Diagramm hingegen unterschiedliche Zeitpunkte des selben Spielers, liegt der Fokus auf der Verbesserung der Fähigkeiten des Spielers und fördert das Meistern dieser (vgl. [SHMK14]).

Quests Quests sind laut [SHMK14] kleine Aufgaben oder Herausforderungen mit einem klar definierten Ziel. Der Fortschritt auf dieses Ziel hin muss dabei transparent sein, sodass es für den Spieler ersichtlich ist, welche Konsequenzen sein Aktionen im Bezug

auf das Erreichen des Ziels haben. Das Erreichen dieses Ziels ist dabei immer mit einer Belohnung verknüpft.

Avatare und Profile Avatare beziehungsweise Profile und die Entwicklung dieser repräsentieren den Spieler. Spieler können eine Verbundenheit zu ihren Avataren entwickeln, sodass ein Fortschritt des Avatars einen persönlichen Fortschritt bedeutet (vgl. [SHMK14]).

Leaderboards Leaderboards oder Bestenlisten stellen den Erfolg der Spieler an der Spitze dar. Spieler, die nicht an der Spitze stehen, könnten allerdings demotiviert werden, da ihre schlechteren Leistungen veröffentlicht werden (vgl. [SHMK14]). Dies führt dazu, dass der Einsatz von Leaderboards kritisch zu sehen ist. Durch die Verwendung von Team-Leaderboards können diese negativen Effekte allerdings etwas reduziert werden, da sie den Fokus auf die Teamleistung legen. Zum einen wird dadurch kein einzelner Spieler „bloßgestellt “ und zum anderen hat das Team ein gemeinsames Ziel auf das es hinarbeiten kann, wodurch sich laut [SHMK14] die Verbundenheit im Team erhöht.

2.2 Kriterien für qualitativ hochwertigen Code

Ein zentraler Aspekt dieser Arbeit ist die Messung von Codequalität. Dazu muss zunächst definiert werden, was im Rahmen dieser Arbeit unter qualitativ hochwertigem Code verstanden wird. Laut [ISO11] sind das die im folgenden aufgelisteten Aspekte. Einige dieser Aspekte lassen sich jedoch nicht ohne Kenntnis über den Projektkontext oder ohne die Ausführung des Programms im laufenden Betrieb bewerten.

2.2.1 Funktionale Vollständigkeit

Die funktionale Vollständigkeit beispielsweise ist immer von den Anforderungen innerhalb des Projekts abhängig. Ohne Kenntnisse über diese Anforderungen lassen sich auch keine Aussagen über den Erfüllungsgrad dieser Anforderungen und somit über die funktionale Vollständigkeit treffen. Darüber hinaus ist die Überprüfung von Semantik innerhalb

der Software nur sehr schwer möglich, da dafür Intelligenz notwendig ist, die ein Computerprogramm nicht liefern kann. Des weiteren ist es nicht möglich, zu prüfen, ob ein Programm fehlerfrei ist, es lässt sich lediglich validieren, dass ein Programm für eine gegebene Liste an Eingaben die gewünschten Ausgaben erzeugt (vgl. [Dij72]). Entsprechend wird in dieser Arbeit die funktionale Vollständigkeit als Kriterium für die Codequalität nicht berücksichtigt.

2.2.2 Nutzbarkeit

Die Aspekte Nutzbarkeit, Erlernbarkeit und Barrierefreiheit beziehen sich auf die Benutzeroberfläche des Programms. Da eine in Java[1] geschriebene Benutzeroberfläche nur in einigen Anwendungen vorhanden ist, können hier nur schwer gerechte Bewertungen vorgenommen werden, da viele Projekte nicht in diesem Aspekt bewertet werden können. Die Bewertung von Projekten, die für die Benutzeroberfläche nicht Java verwenden oder keine Benutzeroberfläche haben, wäre verzerrt, da ein Ausgleich für die in diesem Kriterium vergebenen Punkte geschaffen werden müsste. Daher wird im Rahmen dieser Arbeit der Aspekt der Nutzbarkeit vernachlässigt.

2.2.3 Performanz

Auch ist es schwer, die Performanz und den Ressourcenverbrauch einer Funktion zu ermitteln, ohne den semantischen Hintergrund zu kennen oder die Funktion auszuführen. Einige Aspekte des Ressourcenverbrauchs, beispielsweise Netzwerklast oder Speicherverbrauch, lassen sich nur schwer voraussagen und müssen im laufenden Betrieb gemessen werden. Daher wird dieser Aspekt im Rahmen dieser Arbeit vernachlässigt.

2.2.4 Kompatibilität

Auf die Kompatibilität zu anderer Software können in hier nur schwer Rückschlüsse gezogen werden. Dafür würden Kenntnisse über diese andere Software benötigt. Anschließend müssten die Aufrufe und Implementierungen von Schnittstellen untersucht werden. Java bietet allerdings verschiedene Möglichkeiten, um Schnittstellen zu anderer Software

[1] `https://java.com/de/`

aufzurufen. Dies kann über Representational State Transfer (REST)- beziehungsweise Simple Object Access Protocol (SOAP)-Anfragen, für die es verschiedene Implementierungen gibt, als auch über das Importieren dieser anderen Software als Abhängigkeit geschehen. Die Entwicklung der Muster zur Erkennung dieser Aufrufe und Implementierungen geht dabei über den zeitlichen Rahmen dieser Arbeit hinaus. Auch zur Koexistenz neben anderen Systemen lassen sich nur schwer Aussagen tätigen, da das Programm hierzu im laufenden Betrieb untersucht werden muss.

2.2.5 Zuverlässigkeit

Die Zuverlässigkeit von Software lässt sich durch Codeanalysen gut bewerten. Zum einen kann unter dem Aspekt des Reifegrads des Programms die Testabdeckung untersucht werden. Durch eine hohe Testabdeckung kann erreicht werden, dass verifiziert wird, dass die Methoden das machen, was sie sollen. Dabei muss aber auch darauf geachtet werden, dass die Tests dazu geeignet sind, den Code zu testen. Es ist auch möglich, eine hohe Testabdeckung zu erreichen, ohne dass die Funktionsweise tatsächlich geprüft wurde. Die Testabdeckung besitzt dementsprechend nur eingeschränkte Aussagekraft zur Zuverlässigkeit der Software.

Ein weiterer Aspekt der Zuverlässigkeit ist die Fehlertoleranz. Damit eine Anwendung Tolerant gegenüber Fehlerfällen ist, müssen diese Fehlerfälle erkannt und durch eine alternative Abarbeitungsvariante behandelt werden können. Fehlerfälle wie beispielsweise invalide Nutzereingaben können innerhalb der statischen Codeanalyse durch entsprechend gewählte Testfälle überprüft werden. Dies setzt allerdings voraus, dass solche Tests existieren. Es müsste also gezeigt werden, dass das Programm fehlerfrei ist, was nach [Dij72] nicht möglich ist. Auch die Toleranz gegenüber Fehlern außerhalb des Programms, beispielsweise Netzwerkfehler, Stromausfälle oder Speicherfehler, sind ein Aspekt der Fehlertoleranz. Innerhalb einiger Frameworks gibt es hierfür teilweise Maßnahmen, Spring[2] bietet hier beispielsweise den Cirquit Breaker[3]. Es könnte in den Fällen, in denen solche Frameworks verwendet wurden, eine Validierung stattfinden, dieser Sonderfall ist aber zu speziell als dass er in dieser Arbeit berücksichtigt werden soll. Entsprechend wird der Aspekt der Fehlertoleranz im Rahmen dieser Arbeit für die Zuverlässigkeit nicht berücksichtigt.

[2] `https://spring.io/`
[3] `https://spring.io/projects/spring-cloud-circuitbreaker`

Die Wiederherstellbarkeit ist ein weiterer Bereich innerhalb der Zuverlässigkeit. Sie wird allerdings hauptsächlich vom System auf dem das Programm ausgeführt wird beeinflusst. Maßnahmen an dieser Stelle sind beispielsweise eine geeignete Backup-Strategie oder die Replikation einzelner der Anwendung oder einzelner Teile dieser. Entsprechend wird die Wiederherstellbarkeit für die Messung der Codequalität hier auch nicht berücksichtigt.

2.2.6 Sicherheit

Die Sicherheit eines Programms ist in [ISO11] definiert als Grad zu dem das Programm Daten vor unbefugten Zugriffen schützt. Das Programm nun dahingehend zu analysieren, dass alle Funktionen mit den entsprechenden Zugriffsbeschränkungen versehen worden sind, ist wieder eine semantische Überprüfung und somit nur schwer umzusetzen. Dies gilt auch für die Validierung, ob vor jedem Zugriff eine Authentifizierung korrekt erfolgt ist.

Die Nachverfolgbarkeit von Aktionen innerhalb des Programms ist ein weiterer Aspekt der Sicherheit. Hierzu gehört zum einen, das belegt werden kann, dass eine Aktion oder ein Ereignis stattgefunden hat und zum anderen wer diese Aktion ausgeführt hat. Im Programm können hierzu beispielsweise Log-Ausgaben verwendet werden. Um mit Hilfe von Mustererkennung im Code zu validieren, dass entsprechende Ausgaben getätigt werden, müssten zunächst Muster zur Erkennung von Befehlen zum Ausführen von Log-Ausgaben entwickelt werden, was den zeitlichen Rahmen dieser Arbeit überschreitet. Im laufenden Betrieb können diese Log-Ausgaben zwar gelesen werden, allerdings ist deine Simulation des laufenden Betriebs innerhalb der Codeanalyse nicht möglich. Darüber hinaus müsste erkennbar sein, wo die Grenzen zwischen den einzelnen Aktionen sind, sodass bewertet werden kann, ob eine Log-Ausgabe für eine Aktion getätigt wird. Aufgrund des semantischen Aufbaus der einzelnen Aktionskontexte und der Vielzahl an Implementierungsmöglichkeiten zum Ausgeben von Log-Informationen ist die Analyse hier sehr komplex und wird im Rahmen dieser Arbeit nicht berücksichtigt.

Ein Möglichkeit, die Sicherheit eines Programms innerhalb einer Codeanalyse zu bewerten, ist die Analyse auf Schwachstellen. Hierzu kann eine Open Web Application Security Project (OWASP)-Top-Ten-Analyse verwendet werden. Die OWASP-Top-Ten ist eine Liste mit den zehn kritischsten Sicherheitsbedenken für Web-Applikationen (vgl.

[Fou20]). Dadurch können potentielle Sicherheitslücken innerhalb des Programms aufgedeckt werden. Neben der OWASP-Top-Ten gibt es weitere Indizes für Sicherheitslücken oder Problemstellen in Software, die ebenfalls verwendet werden können.

2.2.7 Wartbarkeit

Die Wartbarkeit beschreibt laut [ISO11] die Effektivität und Effizienz mit der Modifikationen oder Korrekturen umsetzen lassen. Dazu gehört unter anderem die Auftrennung des Programms in Module und die entsprechende Wiederverwendbarkeit dieser Module. Da die Auftrennung des Programms in einzelne Module entlang der semantischen Grenzen geschieht, ist dieser Aspekt nur schwer zu analysieren. Durch das Zählen von Passagen mit doppeltem Code kann allerdings eine Aussage über die Wiederverwendbarkeit von Funktionen und Modulen getroffen werden.

Ein weiterer Bereich ist die Analysierbarkeit und Modifizierbarkeit. Diese Aspekte beschreiben den Grad zu dem Änderungen einen Einfluss auf die Korrektheit des Programms haben beziehungsweise voraussichtlich haben werden. Um entsprechende Aussagen tätigen zu können, ist es wichtig, die einzelnen Programmteile mit ihren Funktionen und eventuellen Seiteneffekten zu verstehen. Hierbei hilft wie oben beschrieben die Einhaltung von Stilrichtlinien und Konventionen, welche innerhalb einer statischen Codeanalyse sehr gut gemessen werden kann.

Ein weiterer Aspekt in der Analysierbarkeit ist das Wissen um den Code. Je komplexer der Code ist, desto schwerer wird es, ihn zu verstehen. Die Komplexität lässt sich dabei innerhalb einer statischen Codeanalyse mit beispielsweise der McCabe-Metrik messen (vgl. [McC76]).

Auch die Verteilung von Wissen um den Code innerhalb des Entwicklungsteams gehört zur Analysierbarkeit. Je weniger Teammitglieder sich in einem Teil des Codes auskennen, desto geringer ist das Verständnis des Codes. Wenn es soweit kommt, dass sich nur noch ein Teammitglied in einem Teil des Codes auskennt, besteht ein Wissensmonopol. Sollte es dazu kommen, dass dieses Teammitglied nun beispielsweise auf Grund von Krankheit oder Urlaub nicht erreichbar ist, müssen sich die anderen Teammitglieder erneut einarbeiten, was sich negativ auf die Analysierbarkeit auswirkt.

2.2.8 Portierbarkeit

Die Portierbarkeit beschreibt, wie gut sich ein Programm von einem System auf ein anderes transferieren lässt. Die Systeme können sich dabei beispielsweise in der verbauten Hardware, der Laufzeitumgebung oder der Prozessorarchitektur unterscheiden. Da es sich bei dem Code, den coderadar analysieren kann, um Java-Code handelt, sind die Programme automatisch weitestgehend unabhängig vom Betriebssystem, da es für unterschiedliche Betriebssysteme unterschiedliche Ausführungsumgebungen gibt. Für Unterschiede in Betriebssystemen wie unterschiedliche Separatoren in Dateipfaden bietet Java ebenfalls Lösungen. Die verbaute Hardware spielt an der Stelle für das Programm lediglich leistungstechnisch eine Rolle, da die Laufzeitumgebung von Java die Hardwarestruktur entsprechend abstrahiert.

Ein weiter Aspekt in der Portierbarkeit ist die Installierbarkeit. Sie beschreibt den Grad der Effektivität und Effizienz, mit dem das Programm installiert beziehungsweise deinstalliert werden kann. Dies ist immer vom Programm selber und seinen Abhängigkeiten abhängig. Es gibt verschiedene Möglichkeiten, ein Java-Programm auszuliefern. Zentral ist dabei aber immer ein Java-Archiv, welches über die Konsole ausgeführt werden kann. Entsprechend muss eine Java-Laufzeitumgebung installiert sein, damit das Programm ausgeführt werden kann. Dieses Archiv kann beispielsweise innerhalb eines Docker[4] Images verpackt werden, was zusätzlich noch eine Installation von Docker erfordert. Alternativ kann das Java-Archiv auf einen Web-Applikationsserver deployt werden, was wiederum die Installation eines solchen Applikationsservers erfordert. Die Installierbarkeit ergibt sich somit nicht nur aus dem Programm selbst, sondern auch aus den jeweiligen Abhängigkeiten und der Auslieferungsform. Entsprechend ist es nur schwer möglich, Aussagen über die Installierbarkeit zu treffen.

Die Austauschbarkeit des Programms beschreibt den Grad zu dem das Programm ein anderes Programm, welches dem selben Zweck dient, in der selben Umgebung ersetzen kann. Um hierzu eine Aussage treffen zu können, wird Kenntnis über das zu ersetzende Programm benötigt. Daher wird dieser Aspekt im Rahmen dieser Arbeit vernachlässigt.

[4] `https://www.docker.com/`

2.2.9 Auswahl der Qualitätskriterien

Wie oben beschrieben, eignen sich nicht alle Aspekte aus [ISO11] zur Bewertung der Codequalität innerhalb der Codeanalyse. In diesem Abschnitt werden die Aspekte vorgestellt, die im Rahmen dieser Arbeit als Kriterien für qualitativ hochwertige Software verwendet werden sollen. Die im folgenden aufgelisteten Metriken stammen aus [Che21] und [Son21b].

Zuverlässigkeit Die Zuverlässigkeit des Programms wird im Rahmen dieser Arbeit unter anderem anhand der Testabdeckung gemessen. Der oben beschriebene Fall, dass einige Tests die Funktionsweise des zu testenden Codes nicht korrekt prüfen, wird hier nicht berücksichtigt. Es wird davon ausgegangen, dass die geschriebenen Tests zur Überprüfung der Funktionsweise geeignet sind. Die gefundenen Bugs und Code-Smells und der entsprechend geschätzte Aufwand zur Behebung dieser fließen ebenfalls in die Bewertung der Zuverlässigkeit mit ein.

Metriken, die für die Messung der Zuverlässigkeit verwendet werden, sind folgenden:

- *test_success_density*
  Die *test_success_density* gibt den Anteil erfolgreicher Tests in einer Software an. Je mehr Tests fehlschlagen, desto unzuverlässiger ist der Code.
- *uncovered_lines*
  uncovered_lines gibt an, wie viele Programmzeilen nicht im Rahmen von Tests ausgeführt werden. Sollte diese Zeilen einen Fehler beinhalten, kann dieser nicht durch Tests gefunden werden.
- *line_coverage*
  Die *line_coverage* beschreibt den Anteil an Zeilen, die durch Tests abgedeckt sind.
- *coverage*
  coverage beschreibt eine Mischung aus Zeilenüberdeckung und Bedingungsüberdeckung und gibt so noch besser Aufschluss darüber, welcher Code durch Tests abgedeckt ist.
- $effort\_to\_reach\_maintainability\_rating\_a$

- *code_smells*
  code_smells sind potentielle Problemstellen im Code. An diesen Stellen ist der Code verwirrend oder schwer zu verstehen und zu warten. Sie müssen nicht zwangsweise zu Fehlern führen, sind aber Stellen, auf die der Entwickler besonderes Augenmerk richten sollte, da sie zu unvorhergesehenem verhalten führen können.

- *bugs*
  bugs sind Fehler im Programmcode, die einer direkten Behebung bedürfen. Diese Fehler führen dazu, dass das Programm abstürzt oder falsche Aktionen ausführt.

- *reliability_rating*
  Das *reliability_rating* bezieht sich auf die Anzahl und die Schwere der gefundenen Probleme. Die Schwere geht dabei von *BLOCKER*, ein Problem welches sofort behoben werden muss, da es zum Absturz des Systems führen kann, bis *MINOR*, was Qualitätsmängel beschreibt, die die Entwicklerproduktivität beeinflussen. Zu ersterem gehören unter anderem Speicherlecks, zu letzterem gehören beispielsweise zu lange Zielen, die das Lesen des Codes erschweren.

- *violations*
  In den *violations* werden alle Arten von Problemstellen unabhängig von ihrer Schwere zusammengefasst.

- *lines_to_cover*
  lines_to_cover beschreibt die Anzahl an Code-Zeilen, die durch Tests abgedeckt werden müssen.

- *sqale_rating*
  Das *sqale_rating* beschreibt die Zeit die anteilig zur gesamten Entwicklungszeit, die notwendig ist, um technische Schulden im Projekt auszugleichen.

Sicherheitsschwachstellen Im Rahmen der Analyse der Sicherheit des Programms wird eine Analyse auf mögliche Schwachstellen durchgeführt. Zu diesen Schwachstellen gehören unter anderem Aspekte aus den OWASP-Top-Ten, kommunikationsspezifische Schwachstellen wie offene Cross-Origin-Einstellungen, Sicherheitslücken aus der Common Weakness Enumeration (CWE) oder sprachspezifische Schwachstellen wie Code-Injektion bei dynamischer Code-Ausführung.

Cross-Origin-Einstellungen sorgen dafür, dass Anfragen an einen Server nur von bestimmten Adressen aus erlaubt sind. Dadurch kann beispielsweise verhindert werden, dass ein dritter Dienst auf Daten aus dem Speicher des Browsers zugreift und so Anfragen im Namen des Nutzers stellt.

Die CWE ist eine Auflistung bekannter Sicherheitslücken in Systemen, Frameworks und Sprachen. Für viele dieser Sicherheitslücken existieren bereits Lösungen, für andere müssen eigene Lösungen implementiert werden.

Bei der dynamischen Ausführung von Code kann zur Zeit der Kompilierung nicht gewährleistet werden, welche Art von Objekt verwendet wird, dies steht erst zur Ausführungszeit fest. Ein Angreifer könnte an dieser Stelle nun ein manipuliertes Objekt injizieren und so die Funktion der Software beeinflussen.

Sollten entsprechende Schwachstellen vorhanden sein, senkt dies die Codequalität an dieser Stelle deutlich.

Zur Bewertung werden hier die gefundenen *vulnerabilities* verwendet. Eine *vulnerability* bezeichnet dabei eine Stelle im Code, die eine Möglichkeit für einen Angriff bietet.

Wiederverwendbarkeit Zur Bewertung der Wiederverwendbarkeit von Code innerhalb des Programms wird die Anzahl von doppeltem Code bewertet. Doppelter Code deutet darauf hin, dass Funktionen innerhalb des Programms hätten zusammengefasst werden können, wodurch potentiell weniger Fehlerquellen bestehen und die Komplexität reduziert werden kann. Auch die Dokumentation von Methoden, die potentiell wiederverwendet werden können, trägt einen Teil dazu bei, dass besagte Wiederverwendung stattfindet.

Die Wiederverwendbarkeit wird im Rahmen dieser Arbeit mit folgenden Metriken gemessen:

- *duplicated_lines_density*
  Die *duplicated_lines_density* beschreibt den Anteil von sich wiederholenden Code-Zeilen im Vergleich zum gesamten Code. Je öfter, gleicher oder ähnlicher Code erneut geschrieben wurde, desto schlechter sind Teile des Programms wiederverwendbar.

- *public_undocumented_api*
- *MissingJavadocMethod* Damit Code wiederverwendet wird, muss schnell und einfach erkennbar sein, welche Funktion ein bestimmter Teil des Codes erfüllt. Dies kann durch eine passende Dokumentation des Codes, der potentiell wiederverwendet werden kann, erleichtert werden. *MissingJavadocMethod* misst, wie oft diese Dokumentation nicht vorhanden ist.

Analysierbarkeit Die Analysierbarkeit ergibt sich im Rahmen dieser Arbeit aus dem Grad der Einhaltung von Stilrichtlinien und Konventionen. Es wird davon ausgegangen, dass nonkonformer Code schwerer zu lesen, zu verstehen und entsprechend zu analysieren ist. Die Komplexität des Codes und die Dokumentation spielen hier ebenfalls eine Rolle.

Um die Analysierbarkeit bewerten zu können, wird unter anderem die *cognitive_complexity* herangezogen. Die *cognitive_complexity* beschreibt die Komplexität einer Methode. Die Berechnung ist an die McCabe-Metrik angelehnt, der Wert ergibt sich aus der Anzahl gewisser Schlüsselwörter innerhalb einer Methode. Darüber hinaus werden viele der von Checkstyle bereitgestellten Metriken verwendet, die sich mit der Einhaltung von Stylingrichtlinien befassen. Dazu gehören beispielsweise *MissingJavadocMethod*, *MethodName* oder *ClassName*. Letztere überprüfen, dass Namen für Bezeichner, beispielsweise Methodennamen oder Klassennamen, einem gewissen Schema entsprechen. Es kann zwar nicht validiert werden, dass der Name des Bezeichners der Funktion des selbigen entspricht, allerdings erleichtert dieses Schema generell die Lesbarkeit.

Komplexität Die Komplexität eines Codeblocks oder einer Funktion kann mit der McCabe-Metrik gemessen werden. Die Metrik berechnet für eine Methode einen Komplexitätswert, welcher wie folgt zu interpretieren ist:

- $<$10: niedrige Komplexität
- 11 - 20: mittlere Komplexität
- 21 - 50: hohe Komplexität
- $>$50: sehr hohe bis undurchschaubare Komplexität

Ein hoher Wert hat entsprechend einen negativen Effekt auf die Analysierbarkeit. Neben der McCabe-Metrik werden auch noch andere Komplexitätsmetriken verwendet.

Um die Komplexität zu messen, gibt es mehrere Möglichkeiten:

- *ognitive_complexity*
- *BooleanExpressionComplexity*
  Die *BooleanExpressionComplexity* bezieht sich auf die Komplexität boolescher Ausdrücke. Durch Verschachtlungen von Negationen, Und- und Oder-Ausdrücken können diese teils sehr komplex werden. Da diese Ausdrücke häufig in *if*- oder *switch*-Abfragen für die Entscheidung verwendet werden, welcher Zweig eines Programms durchlaufen werden soll, ist es wichtig, dass sie gut zu verstehen sind.
- *ClassFanOutComplexity*
  Die *ClassFanOutComplexity* beschreibt die Anzahl an Abhängigkeiten einer Klasse oder eines Interface. Jede Abhängig kann dazu führen, dass mehr Code ausgeführt wird, der potentiell zum Verständnis der aktuellen Funktion notwendig ist. Je mehr Abhängigkeiten eine Klasse oder ein Interface hat, desto höher ist also auch die Komplexität.
- *CyclomaticComplexity*
  Die *CyclomaticComplexity* misst die McCabe-Komplexität.
- *NPathComplexity*
  Die *NPathComplexity* ist eine Erweiterung der *CyclomaticComplexity*, bei der stärker auf die Kombination boolescher Ausdrücke eingegangen wurde.
- *JavaNCSS*
  JavaNCSS beschreibt die Anzahl an Anweisungen innerhalb einer Klasse oder Methode. Je mehr Anweisungen vorhanden sind, desto höher ist die Komplexität.

Wissensmonopole Wie oben beschrieben, sorgen Wissensmonopole dafür, dass Codeverständnis gegebenenfalls neu erarbeitet werden muss. Grade im Zusammenhang mit komplexem Code ist dies ein Problem. Durch die Verwendung von Kommentaren und weiterer Dokumentation kann Wissensmonopolen ein Stück weit entgegengewirkt werden.

Wissensmonopole in komplexem Code haben dementsprechend einen negativen Einfluss auf die Analysierbarkeit.

3 Projektkontext

Dieses Kapitel beschäftigt sich mit der Aufarbeitung der Problemstellung. Dazu wird zunächst in Abschnitt 3.1 das Problem genauer ausformuliert. Anschließend wird in Abschnitt 3.2 das Problem in den Kontext von coderadar eingeordnet. Auf dieser Basis werden dann in Abschnitt 3.3 die Anforderungen an die zu entwickelnden Erweiterungen in coderadar gestellt. Abschließend erfolgt die wissenschaftliche Einordnung der Arbeit in bereits bestehende Werke.

3.1 Problembeschreibung

Zwei Eigenschaften von Entwicklern sind nach [WCSP96], dass sie faul und hochmütig sind. Zum einen halten sie ihren eigenen Code oft für qualitativ sehr gut und zum anderen wollen sie aufgrund ihrer Faulheit bereits geschriebenen Code nicht noch einmal bearbeiten müssen.

Dazu kommt, dass Projekte oft unter Zeitdruck oder mit nur sehr geringem Budget entwickelt werden. Nach einer Studie von [KLME02] wurden die Zeitaufwände für etwa ein drittel der untersuchten Projekte zu gering eingeschätzt, lediglich knapp die Hälfte der untersuchten Projekte hatten einen geringeren zeitlichen Aufwand als zu Beginn geschätzt. Entsprechend fällt auch oft die Budgeteinschätzung aus, da die Zeitaufwände der Entwickler und entsprechende Lizenzkosten einen großen Anteil an den Entwicklungskosten haben.

Dies führt dazu, dass oft wenig Zeit dafür bleibt, auf qualitativ hochwertigen Code zu achten. Stattdessen wird oft die schnellere, dafür meist aber auch unsauberere Variante bevorzugt. Dazu gehört, dass keine Dokumentation, beispielsweise über JavaDoc, erstellt wird, dass mehrere fachlich nicht zusammengehörende Funktionalitäten in einer Klasse

implementiert werden oder das einheitliche Stylingrichtlinien außer acht gelassen werden.

Wie bereits in Abschnitt 1.1 erwähnt, steigt dadurch der Aufwand, der zum Verstehen der Software notwendig ist. Somit steigen auch die Kosten, für die entsprechende Projektphase.

Mit der Hilfe von Analysewerkzeugen wie beispielsweise coderadar (vgl. 3.2) ist es möglich, entsprechende unsaubere Varianten zu finden. Die Herausforderung an dieser Stelle ist, wie Eingangs beschrieben, Entwickler dazu zu bringen, direkt qualitativ hochwertigen Code zu schreiben. Dies dauert zwar erfahrungsgemäß etwas länger, spart dann aber den Aufwand, den Code ein weiteres Mal bearbeiten zu müssen, sodass im Endeffekt Zeit gespart wird.

3.2 Kontext coderadar

coderadar ist ein Analysewerkzeug für Java-Code. Es dient dazu, automatisiert Codeanalysen auf dem Versionskontrollsystem durchzuführen. Die Ergebnisse dieser Codeanalysen werden in Form von Metriken festgehalten, die dem Entwicklungsteam bereit gestellt werden. Darüber hinaus ermöglicht coderadar eine Ansicht der Metriken je Commit, sodass frühere Projektversionen betrachtet werden können.

coderadar verwendet intern verschiedene Plugins zur Codeanalyse. Bisher sind ein Checkstyle[1]- und ein Lines of Code (LOC)-Plugin implementiert. Das Checkstyle-Plugin ist über die Angabe einer Checkstyle-Konfigurationsdatei konfigurierbar. Ein Vorteil von Checkstyle an dieser Stelle ist, dass Checkstyle im Gegensatz zu anderen Analysewerkzeugen wie beispielsweise SonarQube[2] dateibasiert arbeitet. Dadurch ist es möglich, den Dateiinhalt aus einem Commit zu laden und zu analysieren, ohne das gesamte Projekt berücksichtigen zu müssen.

Die Architektur von coderadar erlaubt es, zusätzlich zu den bereits bestehenden Analyse-Plugins, eigene Plugins zu schreiben. Diese können per Anwahl einer Checkbox in der Benutzeroberfläche für ein Projekt zu- oder abgewählt werden. Damit im Rahmen dieser

[1] `https://github.com/checkstyle/checkstyletext`
[2] `https://github.com/SonarSource/sonarqube`

Arbeit ein Vergleich zwischen den einzelnen Projekten möglich ist, sollen im Rahmen der Fallstudie später alle verfügbaren Plugins verwendet werden.

Neben den Möglichkeiten zu Codeanalyse bietet coderadar eine Nutzerverwaltung. Ein Nutzer kann ein Projekt anlegen und analysieren lassen. Dazu werden eine GitHub[3]-URL, ein Projektname und, falls gewünscht, ein Start- und ein Enddatum für die Analyse angegeben. Im nächsten Schritt können dann die zu verwendenden Analyse-Plugins ausgewählt werden. Zusätzlich wird ein Muster für zu analysierende Dateien angegeben. So können bestimmte Dateien von der Analyse ausgeschlossen werden, beispielsweise ist es nicht sinnvoll, die Testabdeckung innerhalb der Implementierung der Tests zu messen. coderadar untersucht dann alle Commits im genannten Zeitraum auf Dateien, die dem angegebenen Muster entsprechen. Wird eine entsprechende Datei gefunden, wird sie mit den konfigurierten Analyse-Plugins untersucht, und entstehenden Metriken werden zu dem Commit hinzugefügt. Ein Nutzer kann ein erstelltes Projekt auch für andere Nutzer zur Kollaboration freigeben.

Nachdem coderadar ein Projekt geclont und analysiert hat, gibt es verschiedene Visualisierungen für die Ergebnisse. Zum einen können die Rohdaten der Metriken je Commit angezeigt werden.

Darüber hinaus können die Ergebnisse in einer „City-View“ angezeigt zu lassen (vgl. Abbildung 7.1). Dabei wird für jede Änderung in einem Commit ein „Haus“ erstellt. Das Aussehen eines solchen Hauses hängt dabei von den vier Faktoren Länge, Breite, Höhe und Farbgebung ab. Jeder dieser Faktoren lässt sich dabei mit dem Wert einer Metrik belegen. Die City-View kann sowohl einen Commit als auch zwei Commits vergleichend anzeigen. Bei der vergleichenden Ansicht besteht die Möglichkeit, entweder beide City-Views nebeneinander oder beide Views ineinander anzuzeigen.

Neben der City-View gibt es auch noch eine „Dependency-Map“ (vgl. Abbildung 7.2). Die Dependency-Map zeigt die Ordnerstruktur des Projektes sowie die Abhängigkeiten der einzelnen Module untereinander an. Die einzelnen Unterordner lassen sich dabei aufklappen, sodass auch innerhalb eines Unterordners die Abhängigkeiten visualisiert werden. Auch hier ist sowohl eine Einzelansicht eines Commits als auch eine vergleichende zweier Commits möglich. Änderungen zwischen den zwei Commits werden entsprechend farblich markiert.

[3] `https://github.com`

coderadar listet die einzelnen Commits entweder in einer Liste oder in der Baumansicht (vgl. Abbildung 7.3) auf. Die Liste ist nach Erstellungsdatum des Commits sortiert. In der Baumansicht werden die Commits anhand der Branches angezeigt, denen sie zugeordnet sind. Zusätzlich bietet coderadar die Funktion, für einen ausgewählten Commit die Ordnerstruktur und den Dateiinhalt anzuzeigen.

Ein Nutzer kann auch ein Team anlegen. Innerhalb eines Teams gibt es Mitglieder und Administratoren. Ein Administrator kann im Namen des Teams ein Projekt anlegen und analysieren. So ist das Projekt nicht direkt von dem Nutzer abhängig. Löscht der Nutzer also seinen Account oder verlässt das Team, kann das Team mit dem Projekt immer noch weiterarbeiten. Ein Nutzer kann dabei auch mehreren Teams angehören.

3.3 Anforderungen

In diesem Abschnitt werden die Anforderungen an die im Rahmen dieser Arbeit durchzuführenden Ergänzungen erarbeitet. An dieser Stelle werden dabei lediglich die funktionalen Anforderungen berücksichtigt, da zum einen die nichtfunktionalen Anforderungen zu einem Teil durch die bestehende Basissoftware abgedeckt sind und zum anderen die nichtfunktionalen Anforderungen für die weitere Arbeit nicht von Bedeutung sind. Die Identifikation der einzelnen Anforderungen erfolgt über einen Buchstaben und eine Nummer. Der erste Buchstabe beschreibt dabei, welcher Teil der Anwendung diese Anforderung erfüllen soll. „A“ steht dabei für den Analyseprozess, „B“ für Backend, „F“ für Frontend. Die Nummer dient zur Identifikation einer Anforderung im jeweiligen Teilbereich und wird hochgezählt. Die Anforderungen an den Analyseprozess und die anderen Funktionen des Backends werden hier getrennt betrachtet, da die Ergänzung des Analyseprozesses um die in Abschnitt 2.2.9 gestellten Kriterien Abwägungen zwischen verschiedenen Lösungsansätzen enthalten und somit mit einigem Aufwand verbunden sein werden.

A.1 Der Analyseprozess für die einzelnen Commits muss dahingehend ergänzt werden, dass der ganze Projektstand an diesem Commit analysiert werden kann und nicht nur die geänderten Dateien. Dies dient dazu, dass auch Werkzeuge wie SonarQube verwendet werden können, die weitere Metriken bereitstellen.

A.2 Der Analyseprozess muss mit verschiedenen Java-Versionen und Build-Management-Werkzeugen umgehen können. Speziell sind die Java-Versionen von Java 8 bis Java 11 zu berücksichtigen. Des weiteren sollen die Build-Management-Werkzeuge Gradle und Maven berücksichtigt werden.

A.3 Das System muss eine Möglichkeit zur Untersuchung des Reifegrades des Projekts zum Zeitpunkt des zu analysierenden Commits bieten. Dazu wird die Testabdeckung innerhalb des Projekts bewertet. Dabei sollen sowohl die Anweisungs- als auch die Zweigüberdeckung berücksichtigt werden.

A.4 Das System muss eine Möglichkeit zur Untersuchung der Sicherheit des Projekts zum Zeitpunkt des zu analysierenden Commits bieten. Hier sollen OWASP-Top-Ten, CWE und weitere Ursachen für *vulnerabilities* in SonarQube verwendet werden.

A.5 Das System muss eine Möglichkeit zur Untersuchung der Wiederverwendbarkeit von Teilen des Projekts zum Zeitpunkt des zu analysierenden Commits bieten. Hierzu wird das Projekt auf doppelten Code hin untersucht.

A.6 Das System muss eine Möglichkeit zur Untersuchung der Analysierbarkeit des Projekts zum Zeitpunkt des zu analysierenden Commits bieten. Hierzu soll die Einhaltung von Stilrichtlinien und Konventionen betrachtet werden.

A.7 Das System muss eine Möglichkeit zur Untersuchung der Komplexität von Teilen des Projekts zum Zeitpunkt des zu analysierenden Commits bieten. Die Bewertung findet dabei unter anderem anhand der Komplexitätsmetrik nach [McC76] statt.

A.8 Das System muss eine Möglichkeit zur Untersuchung auf Wissensmonopole in Teilen des Projekts zum Zeitpunkt des zu analysierenden Commits bieten. Dazu wird die Anzahl an Autoren einer Datei zusammen mit der Komplexität nach [McC76] verwendet.

B.1 Das System muss für jeden Nutzer eine Gesamtpunktzahl, ein Level, die Punktzahl im aktuellen Level, eine Anzahl an Coins, die Liste erhaltener Badges, die Liste absolvierter Quests und die dem Nutzer zugeordneten Contributors speichern. Ein Contributor ist dabei ein identifizierter Git-Account in einem Projekt. Einem Nutzer können mehrere Contributors zugeordnet sein, da ein Nutzer zum einen in mehreren Projekten aktiv sein kann und zum anderen mit mehreren Git-Accounts in einem Projekt arbeiten kann.

B.2 Das System muss für jeden Commit aus den Werten der einzelnen Metriken sowie der Metrikmatrix einen Score berechnen können.

B.3 Das System muss für jeden Contributor eine Gesamtpunktzahl berechnen. Diese Gesamtpunktzahl ergibt sich aus der Summe der Scores über alle Commits des Contributors.

B.4 Das System muss mehrere Level verwalten. Ein Level hat dabei einen nummerischen Bezeichner, eine Gesamtpunktzahl, die notwendig ist, um dieses Level zu erreichen und die Anzahl an Punkten, die in diesem Level notwendig ist, um das nächste Level zu erreichen. Mit steigendem Level soll dabei die Anzahl an benötigten Punkten bis zum nächsten Level größer werden.

B.5 Das System muss Aufgaben, sogenannte Quests, verwalten. Eine Aufgabe hat einen eindeutigen Identifikator, einen Titel, einen Beschreibungstext, eine Anzahl an Punkten und eine Anzahl an Coins, die ein Nutzer für das Erfüllen bekommt.

B.6 Das System muss nach der Analyse eines Commits den errechneten Score automatisch zum jeweiligen Nutzer und Contributor dazurechnen. Bei dem Contributor wird lediglich die Gesamtpunktzahl erhöht. Bei dem Nutzer muss zunächst die Gesamtpunktzahl erhöht werden. Reicht die Gesamtpunktzahl aus, um das nächste Level zu erreichen, muss das Level entsprechend hochgezählt werden. Zusätzlich muss die Punktzahl im aktuellen Level angepasst werden. Wird ein neues Level erreicht, ist die Punktzahl im

aktuellen Level die Anzahl erhaltener Punkte, abzüglich der Punkte, die für den Levelaufstieg nötig waren. Wird kein neues Level erreicht, werden die erreichten Punkte auf die Punkte im aktuellen Level addiert.

B.7 Das System muss für jeden Nutzer eine Liste an Personalisierungen speichern. Zu speichern sind ein Farbschema für die Benutzeroberfläche, ein Profilbild, eine Liste anzuzeigender Badges und das aktuelle Ziel des Nutzers.

F.1 Das System muss dem Nutzer sein aktuelles Level, seine Punktzahl im aktuellen Level und die Punktzahl, die im aktuellen Level benötigt wird, in der generellen Übersicht anzeigen.

F.2 Das System soll eine Übersicht über verschiedene Bestenlisten anzeigen.

F.3 Das System muss für jedes Projekt eine Bestenliste in der Übersicht der Bestenlisten anzeigen. Diese Bestenliste wird über die Gesamtpunktzahl der einzelnen Contributors erstellt. Ist ein Contributor einem Nutzer zugeordnet, muss auch dessen Nutzername angezeigt werden.

F.4 Das System muss eine Bestenliste über alle Projekte in der Übersicht der Bestenlisten anzeigen. Dazu wird für jedes Projekt ein Score errechnet. Dieser ergibt sich aus der Summe aller Gesamtpunktzahlen aller Contributors des Projekts, geteilt durch die Anzahl an Contributors. In dieser Bestenliste muss jeweils der Projektname und die Punktzahl angezeigt werden.

F.5 Das System muss eine Bestenliste über alle Nutzer in der Übersicht der Bestenlisten anzeigen. Dazu wird jeweils die Gesamtpunktzahl des Nutzers, geteilt durch seine Anzahl an Projekten verwendet. In der Bestenliste muss je Nutzer der Name, das Level und die Anzahl an Projekten dargestellt werden.

F.6 Das System muss in einer gesonderten Ansicht die Quests anzeigen. Dabei sollen in einem Abschnitt die noch nicht abgeschlossenen und in einem anderen Abschnitt die bereits abgeschlossenen Quests des Nutzers aufgelistet werden.

F.7 Das System muss verschiedene Einstellungsmöglichkeiten für Personalisierungen bieten. Diese werden mit steigendem Level des Nutzers freigeschaltet. Der Nutzer muss hier ein Farbschema, ein Profilbild, eine Liste an Badges und sein aktuelles Ziel einstellen können.

3.4 Wissenschaftliche Einordnung

Die Idee, Gamification-Elemente im Zusammenhang mit Code-Qualität zu nutzen, ist nicht neu. Ein Ansatz ist SonarQuest[4]. SonarQuest nutzt Analyseergebnisse von SonarQube[5], um auf Basis eines vordefinierten Regelwerks eine Menge an Kennzahlen und Wartungsaufgaben zu erstellen. Jeder Nutzer wird durch einen Avatar repräsentiert. Wie in herkömmlichen Rollenspielen kann ein Nutzer seinen Avatar mit Artefakten ausstatten und der Avatar kann eigene Fähigkeiten ausbilden. SonarQuest wird dabei immer von einem Gamemaster begleitet, der zu den definierten Wartungsaufgaben jeweils eine Geschichte erstellt. Daraus resultiert dann jeweils eine Quests, die von einem Spieler angenommen werden kann. Für das Erfüllen einer Quest und damit einer Wartungsaufgabe bekommt ein Nutzer Gold und Erfahrungspunkte als Belohnung. Neben der normalen Belohnung für eine Quest kann der Gamemaster auch Sonderaufgaben und Boni verteilen, um Aspekte wie Teamwork oder Termineinhaltung besonders zu belohnen.[Son18]

SonarQuest ist dabei sehr stark einem klassischen Rollenspiel nachempfunden und bringt somit einigen Mehraufwand mit sich. Zum einen müssen regelmäßig neue Quests geschrieben werden. Dabei sollten Quests so geschrieben sein, dass ihre Geschichte den Spieler dazu motiviert, die Aufgabe zu erfüllen. Laut [Son18] liegt diese Aufgabe jeweils beim Teamleiter. Dies kann den Teamleiter aber abhängig von der Anzahl an Wartungsaufgaben, Tiefe und inhaltlicher Schlüssigkeit der Hintergrundgeschichten der Aufgaben und dem literarischen Talent den Teamleiter sowohl zeitlich als auch in seinen Fähigkeiten

[4] `https://www.viadee.de/loesungen/java/sonarquest`
[5] `https://www.sonarqube.org/`

überfordern. Zum anderen basiert die Messwerte und damit die vergebenen Belohnungen von SonarQuest ausschließlich auf den Ergebnissen der Analyse von SonarQube. SonarQube bietet zwar eine Gesamtheitliche Analyse des Projekts, wodurch beispielsweise doppelter Code auch Klassenübergreifend auffällt, allerdings wird für die Analyse mit SonarQube immer das gesamte Projekt benötigt. Es ist dadurch deutlich komplexer, Unterschiede zwischen zwei Commits herauszustellen oder das Projekt an einem früheren Zeitpunkt in der Entwicklungshistorie zu untersuchen. SonarQube soll zwar auch für die Analyse der Commits verwendet werden, allerdings ist ein erheblicher Mehraufwand notwendig, um die Historie der Commits zu berücksichtigen. Außerdem sollen neben SonarQube noch weitere Quellen für Messwerte verwendet werden, was in SonarQuest nicht möglich ist. Darüber hinaus soll der Aufwand für das regelmäßige Schreiben der Quests wegfallen. Stattdessen sollen im Vorfeld einige Quests definiert werden, die die Entwickler dann erfüllen können. Eine Geschichte zu den einzelnen Quests soll es nicht geben.

Ein weiterer Ansatz wird in [PNV12] vorgestellt. Prause et al. haben dazu zwei Gruppen von Studenten untersucht, die jeweils ein ähnliches Softwareprojekt nach agilem Vorgehen in Pair Programming entwickeln sollten. Beide Gruppen wurden dabei jeweils von Lehrern begleitet. Der Versuch war dabei in zwei Phasen unterteilt. In der ersten Phase wurden lediglich Qualitätsmetriken gesammelt und ausgewertet, die Teilnehmer wurden darüber jedoch erst ab der zweiten Phase informiert. In dieser zweiten Phase wurden die Teilnehmer in einer täglichen Mail sowie in Meetings über den aktuellen Zwischenstand und ihren Punktestand informiert. Zusätzlich dazu wurden Informationen verschickt, die bei der Verbesserung der Code-Qualität helfen sollten. Nach Abschluss des Versuchs sollte zudem der Entwickler mit der besten Platzierung einen Amazon-Gutschein erhalten. Die Bewertung einer Datei fand dabei auf einer Skala von +10 bis -10 statt.

Als Basis für die Bewertung wurde allerdings lediglich die Vollständigkeit des JavaDoc herangezogen. Dies ist allerdings nur ein Aspekt für qualitativ guten Code nach [ISO11], auf Aspekte wie Analysierbarkeit, Sicherheit oder Zuverlässigkeit wurde an dieser Stelle nicht eingegangen. Die Punktzahl eines Nutzers ergab sich dann daraus, ob er mehr zu positiv oder negativ bewerteten Dateien beigetragen hat. Zudem weist [PNV12] selbst die Kritik auf, dass die Bewertungsformel für die Punkte der Teilnehmer undurchsichtig war und somit die Teilnehmer nicht beziehungsweise nur schwer nachvollziehen konnten, wie sie ihre Bewertung verbessern konnten. Das Pair Programming wird als weiterer Kritikpunkt aufgeführt, da zum einen nur ein Teilnehmer die Anerkennung für die Arbeit

beider Partner bekommt und zum anderen für den eigenen Account Beiträge von anderen Entwicklern eingereicht werden können. Im Rahmen dieser Arbeit sollen mehrere Aspekte aus [ISO11] verwendet werden. Außerdem soll die Bewertungsformel einsehbar und verständlich sein, sodass das Ergebnis nachvollziehbar ist. Die Verwendung von Pair Programming soll im Rahmen dieser Arbeit nicht berücksichtigt werden, stattdessen wird davon ausgegangen, dass der Code, der von einem Contributor eingereicht wurde, auch von diesem geschrieben wurde.

Ein weiterer Versuch wurde von [Pra15] unternommen. Ziel dieses Versuchs war es, Gamification einzusetzen, um Stylingrichtlinien durchzusetzen. Im Rahmen dieses Versuchs sollten zwei Entwicklerteams jeweils ähnliche Software implementieren. Als Basis für die Bewertung der Entwickler diente in diesem Versuch eine Punktzahl, die sich aus den mit Checkstyle gemessenen Verstößen gegen die vorgeschriebenen Stylingrichtlinien ergaben. Die Entwickler wurden dabei sowohl einzeln als auch als Team bewertet. Der Versuch war in zwei Phasen aufgeteilt, in der ersten Phase bekam Team A Rückmeldung über die Leistung sowohl des Teams als auch der einzelnen Personen, in der zweiten Phase bekam Team B dieses Feedback. Ein Ergebnis des Versuchs ist, dass beide Teams in der Zeit, in der sie Feedback bekamen, deutlich weniger Verstöße gegen die Stylingrichtlinien zu verzeichnen hatten. Auffällig ist auch, dass die Anzahl an Verstößen bei Team A nach Ende der Feedback-Phase wieder zugenommen hat. Auch war für die Bewertung des einzelnen Entwicklers sowie des jeweiligen Entwicklerteams die Anzahl an Verstößen lediglich zu Projektende von Bedeutung. Dies führte dazu, dass Team A kurz vor Ende seiner Feedback-Phase viel Arbeit in die Reduktion der Anzahl an Verstößen gesteckt hat.

Im Rahmen dieser Arbeit soll die Code-Qualität über den gesamten Zeitraum beobachtet und bewertet werden. Der Fokus soll nicht auf einer möglichst hohen Codequalität sondern auf dem Lernerfolg der Entwickler liegen. Daher wird die Codequalität kontinuierlich betrachtet und wenn möglich sollen individuelle Hilfestellungen zur Verbesserung der Codequalität gegeben werden.

4 Entwurf des Lösungsansatzes

In diesem Kapitel wird ein Lösungsansatz zur Integration von Gamification in coderadar zur nachhaltigen Steigerung der Codequalität entworfen. Dazu wird zunächst in Abschnitt 4.1 vorgestellt, wie die Codeanalyse in coderadar bisher durchgeführt wird und welche Herausforderungen bei der Erweiterung und der Implementierung weiterer Plugins bestehen. Danach wird in Abschnitt 4.2 die Architektur von coderadar mit der Integration des Lösungsansatzes beschrieben. In Abschnitt 4.3 wird dann auf Basis der Metriken aus 2.2.9 ein Matrix zur Gewichtung der gesammelten Metriken erstellt. Anschließend wird in Abschnitt 4.4 der Einsatz von Gamification-Elementen ausgearbeitet.

4.1 Codeanalyse

In diesem Abschnitt wird genauer auf die Codeanalyse eingegangen. Dazu wird zunächst in Abschnitt 4.1.1 der Analyseprozess innerhalb von coderadar beschrieben. Anschließend wird in Abschnitt 4.1.2 auf besondere Herausforderungen bei der Erweiterung des bisherigen Analyseprozesses eingegangen.

4.1.1 Analyseprozess

Die Codeanalyse in coderadar erfolgt jeweils Commit- und Dateiweise. Für jeden Commit werden die Dateien geladen, die geändert wurden. Diese Dateien werden dann in die konfigurierten Analyse-Plugins gegeben. Die Plugins implementieren dabei jeweils eine Analysemethode, die als Parameter den Pfad zur Datei und den Dateiinhalt bekommt. Die Analysemethode gibt die Metriken für die Datei mit den jeweils berechneten Werten und den Fundorten für die Makel im Quellcode, die zu Abwertungen geführt haben, zurück. Um die Bewertungen für einen Commit zu ermitteln, werden für die Metrikwerte

für alle geänderten Dateien in diesem Commit zusammengerechnet. Dadurch, dass diese Analyse für jeden Commit durchgeführt wird, entsteht auch eine Historie über die Commits.

4.1.2 Herausforderungen

Um einige der in Abschnitt 2.2 geforderten Metriken messen zu können, muss die bisher durchgeführte Analyse erweitert werden. Ein zur Messung dieser Metriken geeignetes Werkzeug ist SonarQube[1]. SonarQube bietet mit dem SonarScanner[2] Command Line Interface (CLI) ein Werkzeug für die Kommandozeile, um Projekte zu analysieren. Das Ergebnis dieser Analyse wird anschließend an einen SonarQube-Server geschickt und kann dort entweder über die Benutzeroberfläche oder eine REST-Schnittstelle eingesehen werden. An dieser Stelle wird SonarQube verwendet, da es zum einen weit verbreitet ist. Laut [Son21a] wurden allein in der „as a Service"-Version SonarCloud[3], die ausschließlich für Open-Source-Software genutzt werden kann, bereits über 180.000 Projekte analysiert. Dazu kommen noch von Firmen und Privatpersonen bereitgestellte Instanzen, die für Closed-Source-Software benötigt werden. Zum anderen wird SonarQube als Standard bei der adesso SE verwendet wird. An dieser Stelle orientiert sich diese Arbeit am Vorgehen der adesso SE, da im Rahmen der Masterarbeit, die auf dieser Arbeit aufbauen wird, bei der adesso SE ein Feldversuch zur Evaluierung der in dieser Arbeit vorgenommenen Implementierung durchgeführt werden soll.

An dieser Stelle bestehen aber auch einige Probleme. Zum einen hält SonarQube, abgesehen von einer manuell vorzunehmenden Versionierung, nur die aktuellen Analyseergebnisse vor. Um Analyseergebnisse von älteren Commits zu erhalten, muss für den jeweiligen Commit entweder eine Version angelegt werden oder das Projekt muss auf diesen Commit gesetzt werden. Um an dieser Stelle auch parallele Analysen zu ermöglichen, werden jeweils neue Projekte für die einzelnen Commits angelegt, die über den Commit-Hash eindeutig identifiziert werden.

Eine weitere Herausforderung besteht darin, aus der einzelnen Datei, die in coderadar für die Analyse zur Verfügung steht, den kompletten Projektstand zu bekommen. Hierfür

[1] `https://www.sonarqube.org/`
[2] `https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/`
[3] `https://sonarcloud.io/`

muss das komplette Projekt an dem zu analysierenden Commit ausgecheckt werden. An dieser Stelle muss eine Möglichkeit zur Vor- und Nachbereitung der Analyse der einzelnen Commits gegeben werden. Innerhalb der Vorbereitung kann dann das jeweilige Projekt ausgecheckt und analysiert werden, in der Nachbereitung werden reservierte Ressourcen dann wieder freigegeben.

Um nun die SonarQube-Analyse durchzuführen, werden allerdings die kompilierten Dateien benötigt. Um das Projekt kompilieren zu können, sind zunächst einige Informationen über die Projektstruktur notwendig. Projekte verwenden teilweise Build-Management-Werkzeuge wie Gradle[4] oder Maven[5]. Durch diese Werkzeuge können Abhängigkeiten, die über die Java Runtime Environment (JRE) hinaus gehen, automatisch aufgelöst werden. Können diese Abhängigkeiten nicht aufgelöst werden, kann auch das Projekt nicht kompiliert werden. Entsprechend können auch Tests fehlschlagen und die Metriken falsche oder keine Werte zurückgeben. Um herauszufinden, ob und wenn ja, welches Build-Management-Werkzeug verwendet wird, kann nach bestimmten Dateien im Projektverzeichnis gesucht werden. Gradle zeichnet sich an dieser Stelle durch eine Datei *build.gradle* aus, bei Maven existiert eine *pom.xml*. Ist eine dieser Dateien im Projektverzeichnis vorhanden, können die entsprechenden Befehle zum Kompilieren des jeweiligen Build-Management-Werkzeugs verwendet werden. Sind keine dieser Dateien vorhanden, wird der klassische Java-Befehl zum Kompilieren von Dateien verwendet.

An dieser Stelle treten ebenfalls einige Herausforderungen auf. Mit der Zeit werden Versionen von Abhängigkeiten, des Build-Management-Werkzeugs oder von Java geändert. Damit das Projekt weiterhin zuverlässig kompiliert werden kann, muss teilweise auf diese Versionsänderungen reagiert werden können. Haben sich Abhängigkeiten so stark verändert, dass die im Build-Management-Werkzeug angegebene Version nicht mehr verfügbar ist, kann das Projekt auch nicht mehr kompiliert werden. An der Stelle können dann keine Metriken über SonarQube bezogen werden. Generell kann auf neuere Versionen in den Abhängigkeiten nicht reagiert werden, da in den Abhängigkeiten teilweise benötigte Funktionen geändert werden oder wegfallen, sodass die ursprüngliche Funktionalität der Software nicht mehr gegeben ist. Darüber hinaus würden solche Änderungen Änderungen in der *build.gradle* oder der *pom.xml* erfordern und somit die Software verändern, was im Rahmen dieses Projekts nicht passieren soll.

[4] `https://gradle.org/`
[5] `https://maven.apache.org/`

Der Anpassung der Version des Build-Management-Werkzeugs kann im Beispiel von Gradle damit begegnet werden, dass ein Gradle Wrapper[6] verwendet wird. Dieser zieht die Versionsnummer für Gradle fest, lädt diese herunter und konfiguriert sie. Dadurch muss nicht jede benötigte Gradle-Version manuell heruntergeladen werden. Teilweise kann es hier auch passieren, dass die Java-Version, die von Gradle verwendet wird, eine andere ist, als die, die im Projekt verwendet wird. Dies führt dazu, dass beispielsweise die Java-Version 7 ausgelesen wird, Gradle aber die Version 8 benötigt. Da laut [Blo14] Java 8 Rückwärtskompatibel zu Java 6 und 7 ist, wird an dieser Stelle immer Java 8 verwendet wenn eine dieser drei Versionen ausgelesen wurde.

Eine Änderung in der Java-Version kann in einem Gradle-Projekt darüber erkannt werden, dass sich bestimmte Eigenschaften in der *build.gradle* ändern. Hier muss entsprechend eine Version der jeweiligen JRE installiert sein, damit das Projekt mit der korrekten Version kompiliert werden kann. Um die Java-Version entsprechend zu setzen, müssen die Umgebungsvariablen $JAVA\_HOME$ und $PATH$ angepasst werden.

Ein weiteres Problem besteht darin, dass einige Commits fehlerhaft beziehungsweise nicht kompilierbar sind. Beispielsweise fehlen bei solchen Commits Abhängigkeiten oder Imports oder das Projekt kann aufgrund von Codefehlern nicht kompiliert werden. Ein weiterer Grund können fehlschlagende Tests sein. Gradle und Maven führen während des Kompilierens des Projekts die im Projekt konfigurierten Tests durch. Dies ist notwendig, damit Aussagen über die Testabdeckung getroffen werden können. Die Build-Management-Werkzeuge sind an der Stelle so eingestellt, dass das Fehlschlagen eines Tests auch den Kompilierprozess fehlschlagen lässt. Entsprechend können diese Commits auch nicht per SonarQube analysiert werden.

Im Anschluss an das Kompilieren des Projekts wird die SonarQube-Analyse gestartet. Das SonarScanner CLI stellt die Analyseergebnisse nicht in Form einer Datei zur Verfügung, sondern lädt diese auf einen SonarQube-Server hoch. Dieser Server muss neben der Datenbank für coderadar und coderadar selbst deployt werden. Die Analyseergebnisse können dann, wie oben bereits erwähnt, über eine REST-Schnittstelle abgefragt werden. Damit das Anfragen der Analyseergebnisse möglich ist, muss im SonarQube-Server zunächst ein Projekt für diesen Commit angelegt werden. Dies ist auch über die REST-Schnittstelle möglich und muss zu Beginn einer jeden Analyse gemacht werden.

[6] `https://docs.gradle.org/current/userguide/gradle_wrapper.html`

Im Anschluss an die Analyse kann das jeweilig konfigurierte Projekt dann wieder entfernt werden, um Ressourcen zu schonen.

Damit nicht für jede Datei, die im zu untersuchenden Commit geändert wurde, eine Anfrage gestellt werden muss, wird zu Beginn eine große Anfrage an den SonarQube-Server gestellt. Die Ergebnisse dieser Anfrage werden dann nach dem jeweiligen Dateipfad aufgeschlüsselt gespeichert. In der Analyse der jeweiligen Datei wird dann auf diese zwischengespeicherten Daten zugegriffen. So muss nicht für jede zu analysierende Datei eine Anfrage via REST gestellt werden, und die Analyse einer Datei ist performanter.

4.2 Architektur

Um die weiteren geforderten Metriken erfassen zu können, werden weitere Maßnahmen zu Codeanalyse benötigt. Dies geschieht in coderadar durch die Ergänzung weiterer Analyse-Plugins. Dazu wird ein neues Untermodul im Modul *coderadar-plugins* angelegt. Zentraler Aspekt dieses Moduls ist die Klasse *SonarScannerSourceCodeFileAnalyzerPlugin*, welches das Interface *SourceCodeFileAnalyzerPlugin* implementiert. In coderadar wird via Java Reflection über alle Klassen iteriert, die das *SourceCodeFileAnalyzerPlugin*-Interface implementieren. Für jede Klasse wird dann die *analyze()*-Methode aufgerufen. Um die Metriken von SonarQube zu bekommen, wird hier entsprechend ein neues Modul geschrieben, welches eine entsprechende Analyse durchführt.

Darüber hinaus muss jede Analyse vor- und nachbereitet werden. Zur Vorbereitung gehört das Anlegen des SonarQube-Projekts, der Checkout des zu analysierenden Commits sowie die Analyse an sich. Zur Nachbereitung gehört das Entfernen des SonarQube-Projekts.

Die Vor- und Nachbereitung ist jeweils ein Prozess, der von der Dateianalyse getrennt zu betrachten ist, da er nicht je Datei, sondern je Commit durchgeführt werden soll. Aus diesem Grund wird die Vor- und Nachbereitung in eigens dafür zu erstellende Methoden ausgelagert. Um aus Sicht des *coderadar-core*-Moduls eine Unabhängigkeit zu den einzelnen Plugins herzustellen, wird an dieser Stelle in der *coderadar-plugin-api* ein neues Interface *WrappedAnalyzingProcess* eingeführt. Dieses Interface schreibt jeweils eine Methode zur Vor- und Nachbereitung der Analyse vor, *prepareAnalysis()* und *postprocessAnalysis()*. Im Analyseprozess wird dann zunächst über alle Implementierungen des

WrappedAnalyzingProcess iteriert und für jede Instanz wird die Methode *prepareAnalysis()* aufgerufen. Anschließend findet die Analyse des Commits, gefolgt von dem Aufruf der Methode *postprocessAnalysis()* für alle Implementierungen des *WrappedAnalyzingProcess*-Interface.

Des weiteren müssen die von den einzelnen Analyse-Plugins zurückgegebenen Metrikwerte mit Hilfe einer noch zu erarbeitenden Metrikmatrix in eine absolute Punktzahl umgerechnet werden. Diese Punktzahl muss dann dem Autor des Commits gutgeschrieben und sein Level angepasst werden.

Außerdem müssen gemäß den Anforderungen aus 3.3 Anpassungen in der Benutzerschnittstelle im Modul *coderadar-ui* vorgenommen werden. Um die benötigten Daten zu speichern und bereitzustellen, müssen entsprechende Anpassungen in der REST-Schnittstelle und dem Datenbankadapter, *coderadar-rest* beziehungsweise *coderadar-graph*, vorgenommen werden. Für die im Rahmen dieser Arbeit implementierten Erweiterungen müssen darüber hinaus Tests geschrieben werden. Diese teilen sich auf in Unit-Tests, die im jeweiligen Modul, und Integrationstests, welche im Modul *coderadar-test* implementiert werden.

Entsprechend der in diesem Absatz beschriebenen Änderungen wird coderadar wie im Komponentendiagramm in Abbildung 4.1 beschrieben aufgebaut sein. Orange eingefärbte Module werden dabei geändert, grün gefärbte Module werden hinzugefügt.

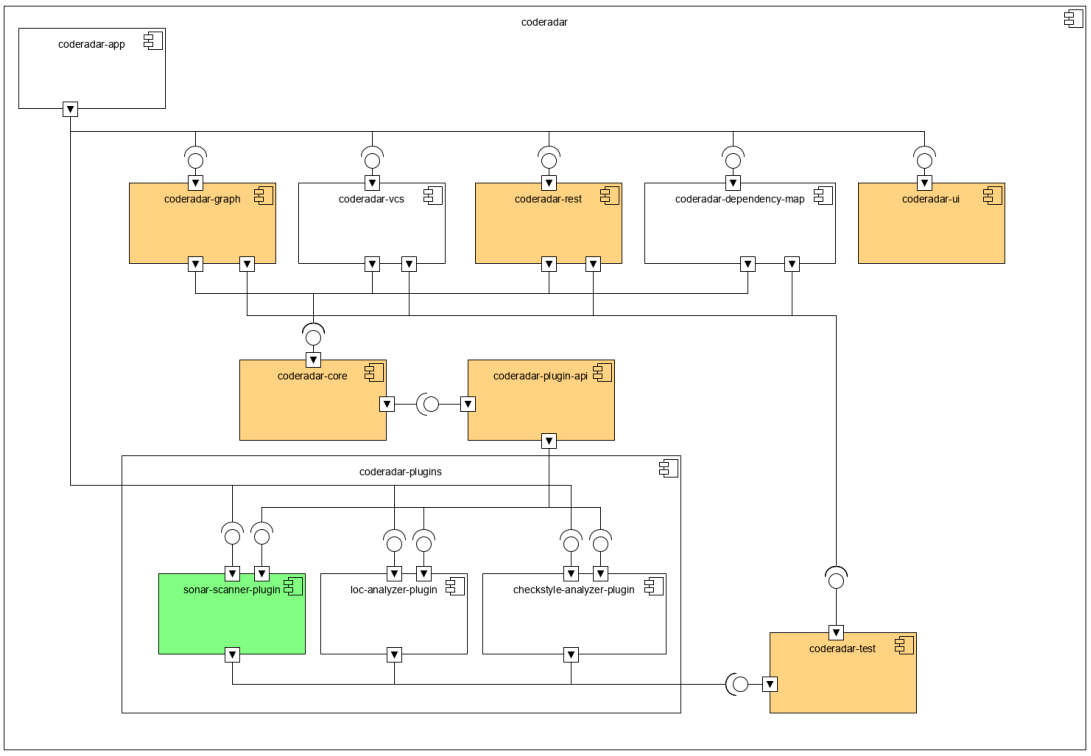

Abbildung 4.1: Die Struktur der Komponenten in coderadar nach den Erweiterungen aus dieser Arbeit.

4.3 Metrikmatrix

Um aus der Menge an Metriken, die coderadar mit der in dieser Arbeit entwickelten Erweiterung für jeden Commit sammelt, eine Summe an Punkten zu ermitteln, müssen die Ergebnisse der Metriken zusammengefasst werden. Da beispielsweise Vulnerabilities einen deutlich stärkeren Einfluss auf die Qualität des Codes haben, müssen die einzelnen Ergebnisse der Metriken gewichtet werden. Zur gewichteten Zusammenfassung der Metriken wird an dieser Stelle eine Metrikmatrix verwendet. Sie multipliziert die Gewichtung der jeweiligen Metrik mit ihrem Wert und erhält in Summe eine Punktzahl für einen Commit.

Eine weitere Aufgabe der Metrikmatrix ist die Ausbalancierung der Punkte, die ein Autor für einen Commit erhalten kann. Es darf nicht möglich sein, durch das wiederholte Ausführen bestimmter, trivialer Aktionen übermäßig viele Punkte zu erhalten, da sonst die Vergleichbarkeit zu anderen Autoren und Projektteams nicht mehr gegeben ist. Diese Balancierung wird ebenfalls über die Gewichtung vorgenommen.

Im Folgenden ist diese Metrikmatrix in Tabellenform dargestellt: 4.3

Metrik	Gewichtung
coderadar:size:eloc:java	x
coderadar:size:sloc:java	x
coderadar:size:cloc:java	x
coderadar:size:loc:java	x
CustomImportOrderCheck	x
TodoCommentCheck	x
UncommentedMainCheck	x
NeedBracesCheck	x
DeclarationOrderCheck	x
ExplicitInitializationCheck	x
FinalLocalVariableCheck	x
HiddenFieldCheck	x
AbstractClassNameCheck	x
AvoidStarImportCheck	x
AvoidStaticImportCheck	x
ClassDataAbstractionCouplingCheck	x
ClassFanOutComplexityCheck	x
CyclomaticComplexityCheck	x
FinalClassCheck	x
HideUtilityClassConstructorCheck	x
IllegalCatchCheck	x
ImportOrderCheck	x
JavadocStyleCheck	x
LineLengthCheck	x
MagicNumberCheck	x
MethodCountCheck	x

MultipleStringLiteralsCheck	x
NewlineAtEndOfFileCheck	x
OuterTypeFilenameCheck	x
OverloadMethodsDeclarationOrderCheck	x
PackageDeclarationCheck	x
ParameterAssignmentCheck	x
RedundantModifierCheck	x
RegexpMultilineCheck	x
StaticVariableNameCheck	x
SummaryJavadocCheck	x
TrailingCommentCheck	x
UpperEllCheck	x
VariableDeclarationUsageDistanceCheck	x
VisibilityModifierCheck	x
WhitespaceAroundCheck	x
WriteTagCheck	x
sonarscanner:test_success_density	x
sonarscanner:uncovered_lines	x
sonarscanner:cognitive_complexity	x
sonarscanner:coverage	x
sonarscanner:line_coverage	x
sonarscanner:sqale_rating	x
sonarscanner:vulnerabilities	x
sonarscanner:public_undocumented_api	x
sonarscanner:effort_to_reach_maintainability_rating_a	x
sonarscanner:duplicated_lines_density	x
sonarscanner:code_smells	x
sonarscanner:bugs	x
sonarscanner:reliability_rating	x
sonarscanner:violations	x
sonarscanner:lines_to_cover	x

Tabelle 4.1: Metrikmatrix mit der Gewichtung der einzelnen Metriken

Anhand der in Tabelle 4.3 dargestellten Gewichtsverteilung über die Metriken wird nach der Analyse die Anzahl an Punkten berechnet, die der Autor des Commits bekommt. Auf Basis der erhaltenen Punkte wird dann das aktuelle Level des Autors bei Bedarf angepasst.

4.4 Gamification-Elemente

Um den Punkte- und Levelstand der einzelnen Autoren aber auch der jeweiligen Projektteams darzustellen, werden verschiedene Gamification-Elemente verwendet. Im folgenden werden auf Basis der bisherigen Benutzerschnittstelle Prototypen für die Erweiterungen erstellt. Bei der Konzeptionierung der Prototypen wird besonders auf die beabsichtigte Wirkung der jeweiligen Gamification-Elemente eingegangen.

4.4.1 Punkte

Ein Autor erhält wie bereits erwähnt für das Einreichen eines Commits Punkte. Dadurch ist es möglich, positive Handlungen beziehungsweise qualitativ hochwertigen Code direkt mit vielen Punkten zu belohnen und qualitativ schlechten Code mit wenig oder keinen Punkten zu bestrafen. Die Anzahl an Punkten ermöglicht überdies einen einfachen Vergleich zwischen verschiedenen Autoren oder zwischen verschiedenen Zeitpunkten der Betrachtung des Profils eines Autors. Die Punkte dienen somit auch als Basis für einige der anderen Gamification-Elemente, auf die im Laufe dieses Abschnitts eingegangen wird. Der Nutzer bekommt seinen aktuellen Punktestand in Form eines Fortschrittsbalken zum nächsten Level angezeigt.

4.4.2 Level

Level dienen als zusätzliche Motivation für den Nutzer, möglichst viele Punkte zu erhalten. So können mit höherem Level bestimmte optische Zusatzfunktionen freigeschaltet werden. Der Fortschritt im aktuellen Level wird in den Ansichten, in denen der verfügbare Platz dies zulässt, beispielsweise nicht in der City-View oder der Dependency-Map, an prominenter Stelle mittels eines Fortschrittsbalkens dargestellt. An dieser Stelle sind folgende Funktionen mit dem entsprechend benötigten Level geplant.

Zusatzfunktion	Level
Das persönliche Profilbild kann angepasst werden	x
Der Fortschrittsbalken für das Level kann alternativ auch den Fortschritt auf ein anderes Ziel hin darstellen	x
Das Farbschema der Benutzeroberfläche kann angepasst werden	x
Die Auswahl an Badges, die anderen Nutzern angezeigt werden, kann personalisiert werden	x

5 Umsetzung

5.1 Programmierung

6 Abschluss

6.1 Verifizierung der Anforderungen

6.2 Fazit

Anhang

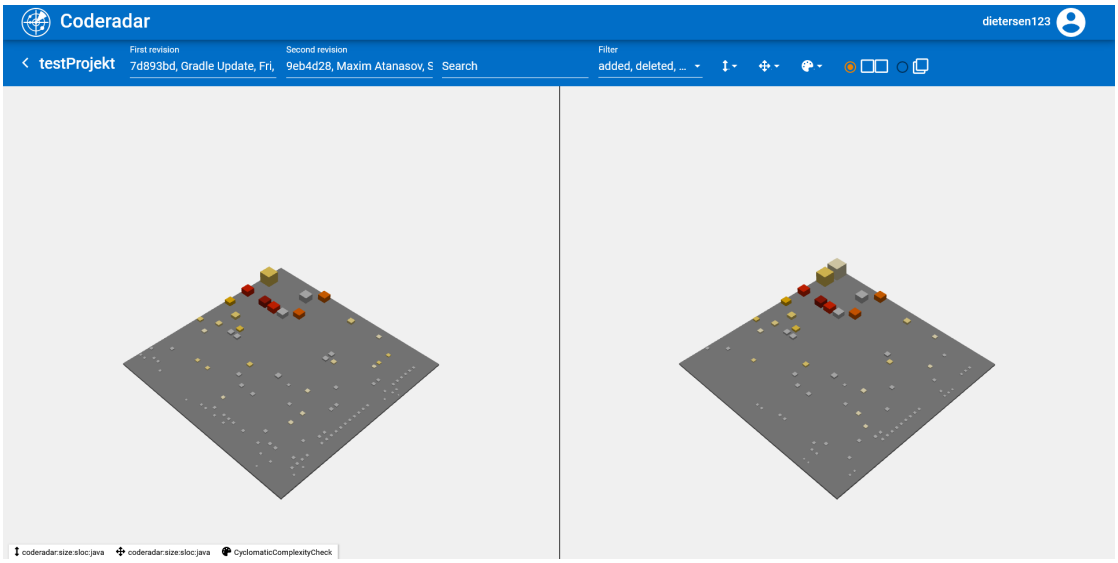

Abbildung 7.1: Ein Beispiel für die City-View.

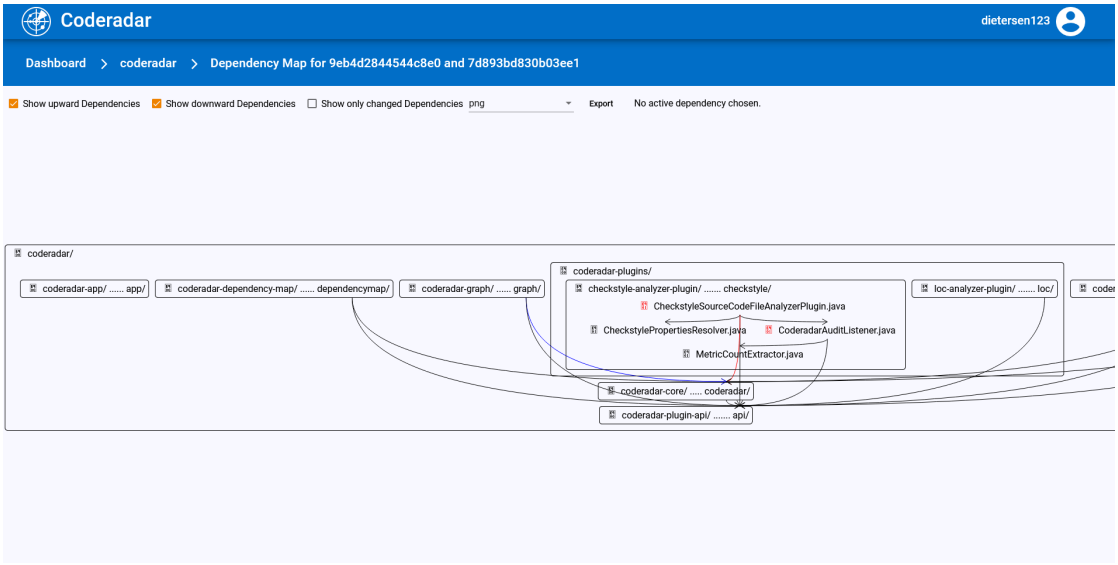

Abbildung 7.2: Ein Beispiel für die Dependency-Map.

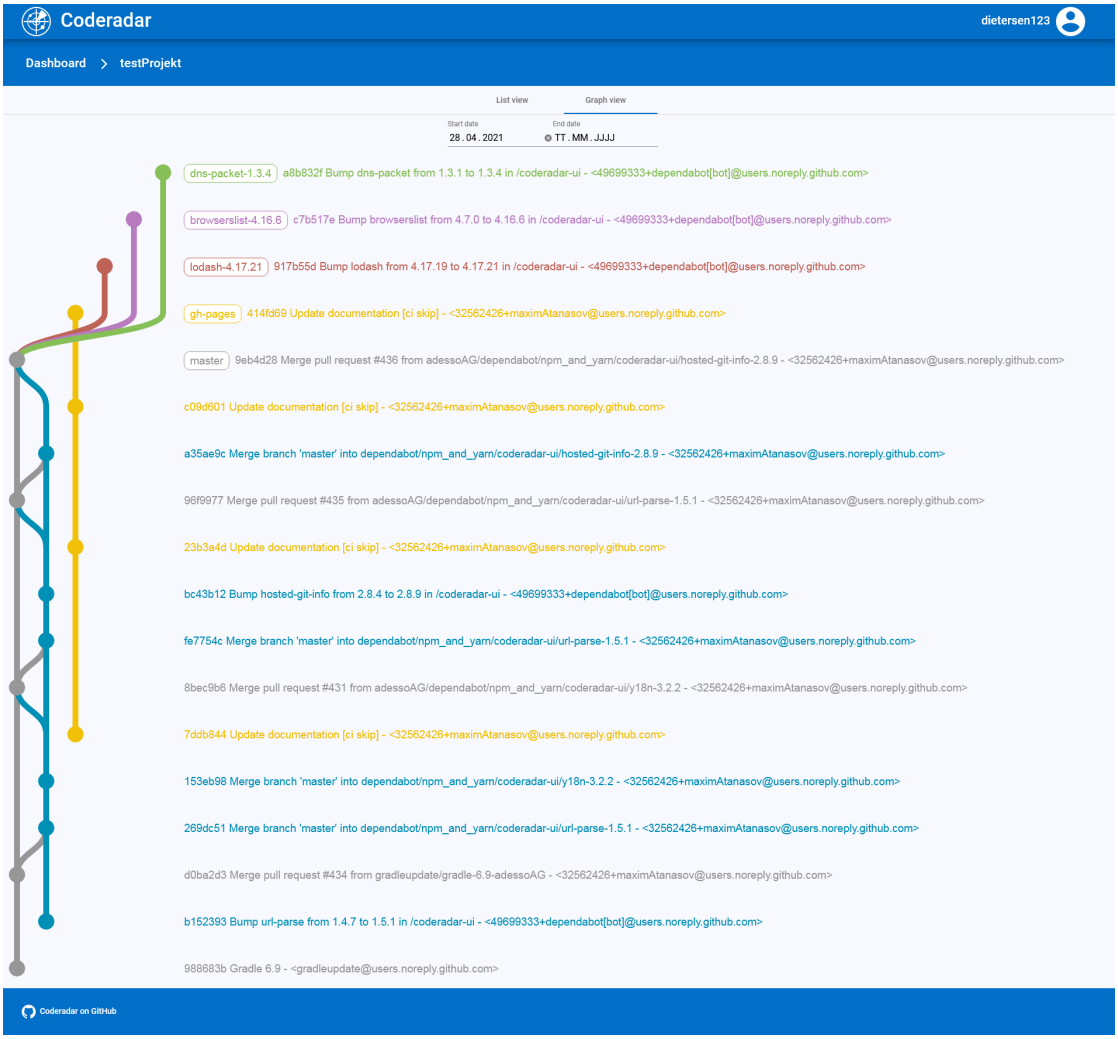

Abbildung 7.3: Die Baumansicht der Commits in coderadar.

Abkürzungsverzeichnis

REST Representational State Transfer

SOAP Simple Object Access Protocol

OWASP Open Web Application Security Project

CWE Common Weakness Enumeration

CLI Command Line Interface

JRE Java Runtime Environment

LOC Lines of Code

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[Ast00] Astleitner, Hermann: Designing emotionally sound instruction: The FEASP-approach. In: *Instructional science* 28 (2000), Nr. 3, S. 169–198

[Bal09] Balzert, Helmut: *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Heidelberg : Spektrum Akademischer Verlag, 2009. http://dx.doi.org/10.1007/978-3-8274-2247-7_3. http://dx.doi.org/10.1007/978-3-8274-2247-7_3. – ISBN 978–3–8274–2247–7

[Blo14] Blog, Oracle: *Upgrading major Java versions*. https://blogs.oracle.com/java-platform-group/upgrading-major-java-versions, 2014. – [Online; Zugriff am 01.07.2021 10:45]

[Che21] Checkstyle: *Standard Checks*. https://checkstyle.sourceforge.io/checks.html, 2021. – [Online; Zugriff am 30.06.2021 20:21]

[DDKN11] Deterding, Sebastian ; Dixon, Dan ; Khaled, Rilla ; Nacke, Lennart: From game design elements to gamefulness: defining "gamification". In: *Proceedings of the 15th international academic MindTrek conference: Envisioning future media environments*, 2011, S. 9–15

[Dij72] Dijkstra, Edsger W.: The Humble Programmer. In: *Commun. ACM* (1972), S. 859–866

[Fou20] Foundation, OWASP: *Owasp Top Ten*. https://owasp.org/www-project-top-ten/, 2020. – [Online; Zugriff am 04.06.2021 14:57]

[ISO11] International Organization for Standardization: System und Software-Engineering – Qualitätskriterien und Bewertung von System und Softwareprodukten (SQuaRE) – Qualitätsmodell und Leitlinien. 2011. – Standard

[KLME02] Kitchenham, Barbara ; Lawrence Pfleeger, Shari ; McColl, Beth ; Eagan, Suzanne: An empirical study of maintenance and development estimation accuracy. In: *Journal of Systems and Software* 64 (2002), Nr. 1, S. 57–77

[McC76] McCabe, Thomas J.: A complexity measure. In: *IEEE Transactions on software Engineering* (1976), Nr. 4, S. 308–320

[PNV12] Prause, Christian R. ; Nonnen, Jan ; Vinkovits, Mark: A Field Experiment on Gamification of Code Quality in Agile Development. In: *PPIG* Citeseer, 2012, S. 17

[Pra15] Prause, Matthias Christian und J. Christian und Jarke: Gamification for enforcing coding conventions, 2015, S. 649–660

[SHMK14] Sailer, Michael ; Hense, Jan ; Mandl, J ; Klevers, Markus: Psychological perspectives on motivation through gamification. In: *Interaction Design and Architecture Journal* (2014), Nr. 19, S. 28–37

[Son18] *SonarQuest - Mit Gamification zu besserer Softwarequalität.* `https://www.viadee.de/loesungen/java/sonarquest`, 2018. – [Online; Zugriff am 16.05.2021 16:06]

[Son21a] SonarSource: *SonarCloud.* `https://sonarcloud.io/`, 2021. – [Online; Zugriff am 30.06.2021 20:51]

[Son21b] SonarSource: *SonarQube Metric Definitions.* `https://docs.sonarqube.org/latest/user-guide/metric-definitions/`, 2021. – [Online; Zugriff am 30.06.2021 20:21]

[Spi11] Spinellis, Diomidis: elytS edoC. In: *IEEE Software* 28 (2011), März/April, Nr. 2, S. 104–103

[Wal03] Walther, Bo K.: Playing and gaming. In: *Game Studies* 3 (2003), Nr. 1, S. 1–20

[WCSP96] Wall, Larry ; Christiansen, Tom ; Schwartz, Randal L. ; Potter, Stephan: *Programming Perl (2nd Ed.).* USA : O Reilly and Associates, Inc., 1996. – ISBN 1565921496

[ZC11] Zichermann, Gabe ; Cunningham, Christopher: *Gamification by design: Implementing game mechanics in web and mobile apps*. Ö'Reilly Media, Inc.", 2011

Eidesstattliche Erklärung

ACHTUNG: Korrekten Text bitte unbedingt vorab mit Studienbüro klären!
Hiermit versichere ich gemäß § 18 Abs. 5 der Bachelor-Prüfungsordnung des Studiengangs Informatik aus dem Jahr 2013, dass ich die vorliegende Arbeit selbstständig angefertigt und mich keiner fremden Hilfe bedient, sowie keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Dortmund, Datum der Abgabe Johannes Teklote