

# **Erstellung und Evaluation eines Gamification-Konzeptes zur Verbesserung der Codequalität eingereichter Lösungen in INLOOP**

**Philipp Matthes**

Geboren am: 12. März 1997 in Chemnitz  
Studiengang: Diplom Informatik (PO 2010)  
Matrikelnummer: 4605459  
Immatrikulationsjahr: 2016

## **Großer Beleg**

Betreuer

Dipl.-Inf. Martin Morgenstern  
M.Sc. Julian Catoni

Betreuernder Hochschullehrer  
Prof. Dr. rer. nat. habil. Uwe Aßmann

Eingereicht am: 17. September 2020

## Aufgabenstellung für einen Großen Beleg

Studiengang: Diplom Informatik  
Name: Philipp Matthes  
Matrikelnummer: 4605459  
Immatrikulationsjahr: 2016  
Titel: Erstellung und Evaluation eines Gamification-Konzeptes zur Verbesserung der Codequalität eingereichter Lösungen in INLOOP

### Ziele der Arbeit

INLOOP<sup>1</sup> ist eine interaktive Programmierlernumgebung, die an der TU Dresden entwickelt wurde. Neben der Bereitstellung von Programmieraufgaben bietet INLOOP die Möglichkeit, die von den Studierenden erstellten Lösungen automatisiert bewerten zu lassen. Grundlegend wird damit das Ziel verfolgt, die Studierenden zur Vertiefung ihrer Programmierkenntnisse zu motivieren. Aktuell wird der Code primär auf korrekte Funktionsweise geprüft. Im Software Engineering spielt jedoch auch die Codequalität eine wichtige Rolle. Der Stand der Technik ist es, Codierungs-Richtlinien mit einem automatisierten Code-Review-System zu forcieren und Anti-Patterns durch Anwendung statischer Quellcodeanalysen frühzeitig zu erkennen.

Ziel dieser Arbeit ist es, ein Konzept für die automatisierte Bewertung der Codequalität eingereichter Lösungen in INLOOP zu entwickeln. Aufbauend auf dieser Grundlage soll weiterhin eine Gamification-Komponente konzipiert werden, die die Nutzer spielerisch zur Verbesserung der eigenen Codequalität motiviert. Dazu müssen zunächst geeignete Metriken und Werkzeuge für die automatisierte Bewertung von Codequalität sowie geeignete Gamification-Ansätze (z.B. Serious Refactoring Games<sup>2</sup>) in der wissenschaftlichen Literatur recherchiert werden. Das erstellte Konzept soll anschließend prototypisch implementiert sowie evaluiert werden.

### Schwerpunkte der Arbeit

- Recherche und Vergleich von Codequalitätsmetriken und -werkzeugen
- Recherche von Gamification-Ansätzen zur Verbesserung der Codequalität
- Anforderungsanalyse und Konzeption einer Gamification-Erweiterung für INLOOP
- Prototypische Implementierung der Gamification-Erweiterung für INLOOP
- Evaluation der Konzepte anhand der prototypischen Implementierung

Betreuer: Dipl.-Inf. Martin Morgenstern  
M.Sc. Julian Catoni

Ausgehändigt am: 30. April 2020  
Einzureichen am: 17. September 2020

Prof. Dr. rer. nat. habil. Uwe Aßmann  
Betreuer Hochschullehrer

---

<sup>1</sup><https://github.com/st-tu-dresden/inloop>

<sup>2</sup>Thorsten Haendler und Gustaf Neumann. „Serious Refactoring Games“. In: Jan. 2019. DOI: 10.24251/HICSS.2019.927.

### **Selbstständigkeitserklärung**

Hiermit versichere ich, dass ich die vorliegende Arbeit mit dem Titel *Erstellung und Evaluation eines Gamification-Konzeptes zur Verbesserung der Codequalität eingereichter Lösungen in INLOOP* selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in der Arbeit angegebenen Hilfsmittel und Quellen benutzt. Die wörtlichen und sinngemäß übernommenen Zitate habe ich als solche kenntlich gemacht. Es waren keine weiteren Personen an der geistigen Herstellung der vorliegenden Arbeit beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 17. September 2020



Philipp Matthes

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>VII</b>
<b>Tabellenverzeichnis</b>	<b>VIII</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Gegenstand und Motivation . . . . .	1
1.2 Problem- und Zielstellung . . . . .	1
1.3 Aufbau der Arbeit . . . . .	2
<b>2 Grundlagen</b>	<b>3</b>
2.1 INLOOP . . . . .	3
2.2 Gamification . . . . .	4
2.2.1 Begriffliche Einordnung . . . . .	4
2.2.2 Gamification-Elemente . . . . .	5
2.2.3 Gamification als Motivator . . . . .	6
2.2.4 Frameworks . . . . .	7
2.3 Softwarequalität . . . . .	8
2.3.1 Komponenten und Faktoren . . . . .	9
2.3.2 Codequalitätsmetriken . . . . .	10
2.3.3 Refaktorisierung . . . . .	10
2.3.4 Technische Schulden . . . . .	10
2.3.5 Codierungsrichtlinien . . . . .	11
2.4 Zusammenfassung . . . . .	13
<b>3 Related Work</b>	<b>14</b>
3.1 CleanGame: Gamifying the Identification of Code Smells . . . . .	14
3.1.1 Game-Design . . . . .	15
3.1.2 Ergebnisse . . . . .	16
3.2 Serious Refactoring Games . . . . .	17
3.2.1 Game-Design . . . . .	18
3.2.2 Ergebnisse . . . . .	19
3.2.3 Framework . . . . .	19
3.3 Teaching Clean Code . . . . .	19
3.4 Zusammenfassung . . . . .	20

<b>4 Anforderungsanalyse</b>	<b>22</b>
4.1 Rahmenbedingungen . . . . .	22
4.1.1 Istzustand von INLOOP . . . . .	22
4.1.2 Zielgruppe . . . . .	23
4.1.3 Anwendungsfälle und Akteure . . . . .	24
4.2 Spezifikation . . . . .	26
4.2.1 Funktionale Anforderungen . . . . .	26
4.2.2 Qualitätsanforderungen . . . . .	27
4.3 Zusammenfassung . . . . .	28
<b>5 Konzept</b>	<b>29</b>
5.1 Codeanalyse . . . . .	29
5.1.1 Regelwerk für Codierungsrichtlinien . . . . .	29
5.1.2 Integrationskonzept der statischen Codeanalyse . . . . .	30
5.1.3 Semantische Code-Smell-Extraktion . . . . .	31
5.1.4 Regelbasierte Erklärungsdatenbank . . . . .	33
5.2 Game-Design . . . . .	34
5.2.1 Leitmotive . . . . .	35
5.2.2 Narrativ . . . . .	35
5.2.3 Progressivität . . . . .	36
5.2.4 Soziale Interaktion . . . . .	39
5.2.5 Risiko und zeitliche Knappheit . . . . .	40
5.2.6 Dialogkonzept . . . . .	40
5.2.7 Einordnung nach Octalysis . . . . .	41
5.2.8 Einordnung nach Bloom . . . . .	42
5.3 Architekturentwurf der Gamification-Erweiterung . . . . .	43
5.3.1 Integrationspunkte und Komponenten . . . . .	43
5.3.2 Modelle und Persistenzschicht . . . . .	47
5.4 Zusammenfassung . . . . .	51
<b>6 Prototypische Implementation</b>	<b>52</b>
<b>7 Evaluation</b>	<b>57</b>
7.1 Expertenevaluation . . . . .	57
7.1.1 Evaluationsstrategie und Durchführung . . . . .	57
7.1.2 Ergebnisse . . . . .	58
7.2 Studierendenevaluation . . . . .	59
7.2.1 Evaluationsstrategie und Durchführung . . . . .	59
7.2.2 Ergebnisse . . . . .	60
7.3 Gefahren für die Validität . . . . .	62
7.4 Zusammenfassung . . . . .	64
<b>8 Zusammenfassung</b>	<b>66</b>
8.1 Ergebnisse . . . . .	66
8.2 Ausblick . . . . .	67
<b>Literatur</b>	<b>i</b>
<b>Anhang</b>	<b>v</b>

# Abbildungsverzeichnis

2.1	Varianten des Assessments anhand des iterativen Lernzyklus. Angelehnt an [HS12, S. 44] und basierend auf [Cri07, S. 41]. . . . .	4
2.2	Die von Yu-kai Chou entwickelte Gamification-Taxonomie „Octalysis“ als Oktagon mit beispielhaften Gamification-Elementen. Mit freundlicher Genehmigung [Cho19]. . . . .	8
2.3	Die Kaskade der Softwarequalitätskomponenten im Lebenszyklus von Software und deren Modellindikatoren nach ISO/IEC 9126:2001 [eng01]. . . . .	9
4.1	Architektur-Metamodell von INLOOP Entnommen aus [MD18]. . . . .	22
4.2	UML-Kompositionsstrukturdiagramm: Top-Level-Architektur der Webanwendung INLOOP . . . . .	23
4.3	UML-Use-Case-Diagramm: Anwendungsfälle einer Gamification-Erweiterung für INLOOP in Relation zu den jeweils beteiligten Akteuren. . . . .	25
5.1	Architektur des QualityReview Frameworks von Dietz et al. [Die+18]. . . . .	29
5.2	Integrationskonzept des QualityReview Frameworks für die TestRunner Komponente von INLOOP . . . . .	31
5.3	Eine Architektur zur vereinheitlichten, semantischen Code-Smell-Extraktion auf Grundlage der Ausgaben von Codeanalysetools. . . . .	33
5.4	Beispielabbildung für die Adaption der Identifikatoren von Detektionen auf Codeanalysetools zur Ergänzung von Code-Smell-spezifischen Erklärungen. . . . .	34
5.5	Ein Banner auf der Startseite, welches die „Code Doctors“ vorstellt. Ursprüngliche Gestaltung der Illustration von Freepik <sup>1</sup> . . . . .	36
5.6	Nutzer erhalten Rezepte von spezialisierten „Code Doctors“ zur Verbesserung einer eingereichten Lösung. . . . .	37
5.7	UI-Konzept eines progressiven Leaderboards in Form einer Tabelle, die Nutzer nach ihren erreichten Punktzahlen auflistet. . . . .	39
5.8	Konkrete beispielhafte Dialoglandkarte für die Integration der Gamification-Elemente in bestehenden Dialogen von INLOOP . . . . .	41
5.9	Einordnung der gewählten Gamification-Elemente nach dem Octalysis Gamification Framework von Chou [Cho19]. . . . .	42
5.10	Übersicht über mögliche Komponenten der Gamification-Erweiterung und deren Integrationspunkte in der bestehenden INLOOP-Architektur. . . . .	43
5.11	Notwendige Anpassungen (grau hinterlegt) im Ant Build-Prozess des über Continuous Publishing bezogenen Docker Image des TestRunners. . . . .	45
5.12	Architektur für die Bereitstellung von Notifikationen im Pull-Prinzip über eine Django-Middleware oder einen Django-Kontextprozessor im Request-Response-Prozess. . . . .	46

5.13 Architektur für die Bereitstellung von Notifikationen im Push-Prinzip über einen asynchronen Nachrichtenaustausch. . . . .	47
5.14 Eine mögliche Modellstruktur der inloop.medics.violations Komponente. . . . .	47
5.15 Eine mögliche Modellstruktur der inloop.medics.social Komponente. . . . .	48
5.16 Eine mögliche Modellstruktur der inloop.medics.rewards Komponente. . . . .	49
5.17 Eine mögliche Modellstruktur der inloop.medics.ranking Komponente. . . . .	50
 6.1 Täglich aktualisiertes Leaderboard mit Suchfunktion, Avataren und dynamischer Rangänderungsanzeige. . . . .	53
6.2 Navigationsleiste und Aufgabenansicht. . . . .	53
6.3 Lösungsansicht mit erreichten Punkten und Notifikation. . . . .	54
6.4 Wartezimmer mit dem Gesundheitszustand einer Lösung (roter Button). . . . .	54
6.5 Konsultation mit der Anzeige der Zeile und Beschreibung aus der Erklärungsdatenbank zu einer Detektion. . . . .	55
6.6 Freigeschaltete Badges, Avatare und Level in den Spielerdetails und die Anzeige von Kollegen. . . . .	55
6.7 Informationsseite über detektierbare Code Smells, deren Identifikator und Erklärung aus der Erklärungsdatenbank. . . . .	56
 7.1 Absolute Detektionsraten von auf Violations abgebildeten Checkstyle-Regeln anhand von analysierten INLOOP-Lösungen. . . . .	60
7.2 Umfrageergebnisse zur Berücksichtigung von Codequalität bei der Lösung von INLOOP-Aufgaben. . . . .	61
7.3 Umfrageergebnisse zur semantischen Code-Smell-Detektion. . . . .	61
7.4 Umfrageergebnis zur Attraktivität der Code-Smell-Konsultation. . . . .	62
7.5 Umfrageergebnisse zu Kernelementen des Game-Designs. . . . .	62
7.6 Diagramm über die Standardabweichungen der Antwortverläufe aus den erhobenen Umfragen. . . . .	63
7.7 Scatter-Plot über die Standardabweichungen und durchschnittlichen Bewertungen der Antwortverläufe aus den erhobenen Umfragen. Farbschema nach Abbildung 7.6. . . . .	63

# Tabellenverzeichnis

3.1 Kognitive Fähigkeiten bei der Refaktorisierung anhand von Blooms Taxonomie. Sinngemäß aus der tabellarische Darstellung in [HN19b, S. 3] abstrahiert. . . . .	17
4.1 Qualitätsanforderungen an die Gamification-Erweiterung analog zu den in [eng01] gezeigten Modellparametern von interner und externer Softwarequalität. Abkürzungen von links nach rechts: (+) normale Priorität, (++) hohe Priorität, (+++) sehr hohe Priorität, (++++) höchste Priorität. . . . .	27
A.1 Integrierte Code-Doktoren in der Gamification-Erweiterung. . . . .	vi
B.1 Erklärungen zu strukturellen Verstößen. . . . .	vii
B.2 Erklärungen zu stilistischen Verstößen. . . . .	ix
B.3 Erklärungen zu weiteren, generellen Verstößen. . . . .	x
C.1 Erreichbare Errungenschaften in der Gamification-Erweiterung. . . . .	xi
D.1 Erreichbare Level in der Gamification-Erweiterung. . . . .	xii

# 1 Einleitung

## 1.1 Gegenstand und Motivation

Im Rahmen der Lehrveranstaltung Softwaretechnologie an der Technischen Universität Dresden werden Methoden und Konzepte zur Entwicklung großer Softwaresysteme vermittelt. Als Teil des didaktischen Konzepts der Lehrveranstaltung wird das E-Learning-System INLOOP eingesetzt, in welchem Studierende die Möglichkeit haben, Programmieraufgaben aus unterschiedlichen Anwendungsdomänen interaktiv zu lösen. Die Studierenden sollen hierbei Konzepte aus der Softwareentwicklung verinnerlichen, zu denen auch die selbstständige Einschätzung der Softwarequalität gehört. Auf INLOOP eingereichte Lösungen werden mithilfe von Tests auf funktionale Korrektheit analog zu den Anforderungen der jeweiligen Aufgabe geprüft. Hierdurch erhält der Nutzer bereits ein interaktives Feedback, woraus er Rückschlüsse auf die Funktionsfähigkeit der Teilkomponenten seiner eingereichten Lösung ziehen kann. Zusätzlich hierzu könnte jedoch auch die Codequalität der jeweiligen Lösung über spezielle Metriken automatisiert ausgewertet werden, um dem Nutzer ein verbessertes Qualitätsfeedback zu geben. Auf Grundlage des verbesserten Feedbacks könnten Nutzer über eine sogenannte Gamification motiviert werden, die Codequalität der eigenen Lösungen selbstständig zu verbessern. Hierfür soll ein konkretes Gamification-Konzept entwickelt und prototypisch in INLOOP integriert werden.

## 1.2 Problem- und Zielstellung

Anhand eines Gamification-Konzepts und einer hieraus erstellten prototypischen Implementation soll diskutiert werden, inwiefern dies didaktisch zur Integration der Codequalität in der akademischen Lehre beitragen kann. Zur differenzierten Diskussion dieses zentralen Forschungsgegenstands sollen folgende konkrete Forschungsfragen eruiert werden.

**Forschungsfrage 1** *Welche Codequalitätsmetriken kommen hierfür in Frage und wie können diese im Kontext von INLOOP kombiniert und parametrisiert werden, um signifikante Fehlgestaltungen des Codes zu erkennen, welche die Codequalität der eingereichten Lösungen beeinträchtigen?*

**Forschungsfrage 2** *Welche Gamification-Elemente sind dafür geeignet, die durch Codequalitätsmetriken erkannten Fehlgestaltungen des Codes an die Nutzer zu kommunizieren, sodass diese motiviert werden, Fehlgestaltungen zu refaktorisieren, im Voraus zu vermeiden und damit die Codequalität der eingereichten Lösungen zu verbessern?*

**Forschungsfrage 3** *Wie können die Codequalitätsmetriken mit den Gamification-Elementen in einer Gamification-Erweiterung kombiniert werden?*

**Forschungsfrage 4** Welche Auswirkungen hat die Einführung der Gamification-Erweiterung auf die Motivation, die Codequalität eingereichter Lösungen zu verbessern?

Diese Fragen sollen beantwortet werden, indem zunächst eine extensive Literaturrecherche durchgeführt wird, auf deren Grundlage ein Prototyp einer Gamification-Erweiterung für INLOOP konzipiert und implementiert werden soll. Anhand des Prototyps sollen schließlich die vorangestellten Forschungsfragen im Kontext einer Evaluation beantwortet werden.

### 1.3 Aufbau der Arbeit

Zu Beginn der Arbeit werden zunächst grundlegende domänenspezifische Begriffe und Zusammenhänge erläutert. Hierzu wird INLOOP im Kontext der Lehrveranstaltung Softwaretechnologie betrachtet, grundlegende Begriffe und Zusammenhänge zur Gamification erklärt und Codequalitätsmetriken im Rahmen der Softwarequalität diskutiert. Hiernach wird ein systematischer Überblick über den Stand der Forschung zum Einsatz von Gamification mit dem Ziel der Verbesserung von Codequalität im akademischen Umfeld gegeben. Außerdem werden konkrete vergleichbare Konzepte vorgestellt und diskutiert. Mithilfe der Erkenntnisse aus den verwandten Konzepten werden anschließend Anforderungen an eine durch Codequalitätsmetriken gestützte Gamification-Erweiterung ermittelt. Hierzu wird nach der Beschreibung des Istzustands von INLOOP eine strukturelle Analyse durchgeführt, um die Schnittstellen des Systems zu finden, an denen eine durch Codequalitätsmetriken gestützte Gamification-Erweiterung anschließen kann. Weiterhin werden konkrete funktionale und nichtfunktionale Anforderungen an eine solche Erweiterung ermittelt und die hiermit zusammenhängenden externen Bibliotheken betrachtet. Aus der Anforderungsanalyse werden nachfolgend konkrete Konzepte für die Erweiterung erstellt. Nachdem die technischen Rahmenbedingungen festgelegt wurden, wird eine Lösungsstrategie entworfen, welche konkrete Gamification-Elemente und Codequalitätsmetriken auf Grundlage der gesammelten Evidenz auswählt und miteinander vereint. Hierzu wird unter anderem eine konkrete Softwarearchitektur entworfen und als Framework für eine darauf basierende Implementation beschrieben. Im sich hieran anschließenden Kapitel wird die entworfene Software prototypisch implementiert und gezeigt. Zur Vorbereitung der Evaluation werden weitere technische Vorkehrungen getroffen. Anhand der prototypischen Implementation wird die Gamification-Erweiterung schließlich evaluiert. Hierfür wird zunächst die Methodik der Evaluation konzeptioniert. Nach Durchführung der Evaluation werden die gesammelten Daten systematisch ausgewertet und mögliche Gefahren für die Validität betrachtet. Mithilfe dieser Daten sollen abschließend die Antworten auf die oben genannten Forschungsfragen determiniert werden.

# 2 Grundlagen

In diesem Kapitel werden Grundlagen zu den relevanten Begriffen der Domäne erläutert. Zunächst wird hierzu die Online-Plattform INLOOP beschrieben. Danach werden die Konzepte, Ziele und Probleme hinter Gamification und Codequalitätsanalyse diskutiert und schließlich zusammengefasst.

## 2.1 INLOOP

INLOOP ist eine Webanwendung, die an der Technischen Universität Dresden entwickelt wurde. Die Webanwendung wird im Rahmen der Lehrveranstaltung Softwaretechnologie genutzt, um deren didaktisches Konzept durch ein fakultatives Angebot von Online-Programmieraufgaben zu erweitern. Das didaktische Konzept beinhaltet als Lernziele dabei unter anderem, dass die Studierenden anhand der Programmiersprache Java objektorientierte Konzepte (Entwurfsmuster, Klassenbibliotheken, UML-Modelle) implementieren und diese Implementation einer Software-Qualitätssicherung unterziehen können [Aßm10]. In INLOOP können Programmieraufgaben veröffentlicht und in verschiedene Kategorien unterteilt werden. Die Kategorien (Basic, Lesson, Exam) sind hierbei nach steigender Komplexität und Schwierigkeit geordnet. Die in diese Kategorien eingegliederten Programmieraufgaben bestehen aus einer textuellen Beschreibung. In der textuellen Beschreibung der meisten Aufgaben sind außerdem Diagramme integriert, welche zum Beispiel die Klassenstruktur der zu implementierenden Software repräsentieren (in Form von UML-Analyse/Entwurf-Klassendiagrammen) oder die Reihenfolge und Art von zwischen Komponenten der Software ausgetauschten Informationen (in Form von UML-Sequenzdiagrammen oder UML-Zustandsdiagrammen) aufzeigen. Aufgaben können von Nutzern der Plattform, entweder in einer eigenen Entwicklungsumgebung oder in einem integrierten Online-Editor, von zuhause bearbeitet werden. Die eingereichten Lösungen werden in der Plattform auf funktionelle Korrektheit geprüft und hierdurch automatisiert ausgewertet. Zum Schluss eines jeden Bearbeitungsprozesses wird ein direktes Feedback zur funktionellen Korrektheit der Teilkomponenten der eingereichten Lösung präsentiert. Besteht eine Lösung nicht alle funktionellen Tests, so wird sie als „nicht bestanden“ gewertet. Als logische Konsequenz ist eine Lösung auch nur dann „bestanden“, wenn alle funktionellen Tests erfolgreich waren. INLOOP gliedert sich mit diesem didaktisch orientierten Grundkonzept in die so genannten E-Learning-Systeme ein und hat somit den primären Zweck, Lehrinhalte zu vermitteln und den Nutzern der Plattform beim Erlernen neuer Inhalte zu helfen. Da die Inhalte darüber hinaus auch abgefragt und bewertet werden können, repräsentiert INLOOP gleichzeitig ein so genanntes E-Assessment-System [HS12]. Abbildung 2.1 zeigt den im Fokus stehenden Lernprozess als Iterationszyklus, zu dessen Schluss das durch die automatisierte Beurteilung in INLOOP realisierte summative Assessment steht.

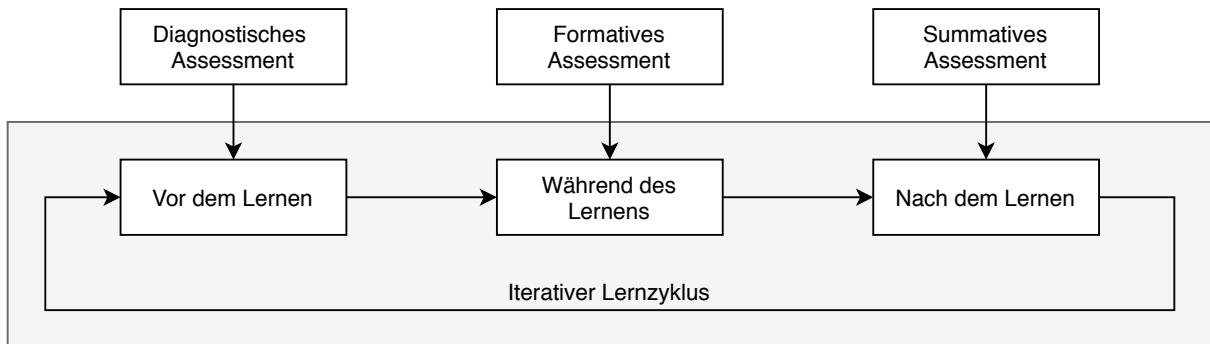


Abbildung 2.1 Varianten des Assessments anhand des iterativen Lernzyklus. Angelehnt an [HS12, S. 44] und basierend auf [Cri07, S. 41].

Die Aufgaben in INLOOP werden über das laufende Semester für die Studierenden in zeitlich aufeinanderfolgenden Abschnitten über ein eigenes Versionsmanagement freigegeben, sodass zu Anfang des Semesters noch keine der schwierigeren und komplexeren Exam-Aufgaben verfügbar sind. Die Studierenden können sich somit zu Beginn des Semesters auf die verhältnismäßig einfacheren und grundlegenderen Aufgaben konzentrieren. Hierbei ist zu beachten, dass Aufgaben als befristet definiert werden können, also nur bis zum Verfall einer bestimmten Deadline abgegeben werden können. Zur Motivation der Studierenden, das fakultative Angebot zu nutzen, werden als Anreiz Bonuspunkte für die Klausur verliehen. Die genauen Details und Rahmenbedingungen können die Studierenden dabei auf INLOOP einsehen<sup>1</sup>. Bonuspunkte werden hierbei nur vergeben, wenn die in Frage kommende Lösung nicht plagiert wurde. Für die Plagiatsprüfung der Java-basierten Lösungen wird JPlag<sup>2</sup> verwendet. Mit der Einführung von INLOOP und mithilfe der Bonuspunkte als Motivator konnte beobachtet werden, dass sich hierdurch vermehrt Studierende an den bereitgestellten Programmieraufgaben probierten und schließlich im Vergleich besser in der Abschlussprüfung der Lehrveranstaltung abschnitten [MD18].

## 2.2 Gamification

Analog zu den beschriebenen Bonuspunkten etablierte sich in den vergangenen Jahren ein als „Gamification“ bezeichnetes Prinzip als weitere Möglichkeit, die Motivation von Nutzern zu steigern. Durch den Einsatz dieses Prinzips konnte in verschiedenen Studien nachgewiesen werden, dass hierdurch die Leistungen der Nutzer innerhalb des jeweiligen Kontextes signifikant gesteigert werden konnten [Mek+13][SBK12][AS14]. Einige universitäre Online-Plattformen, beispielsweise das Auditorium-Forum der TU Dresden<sup>3</sup> oder die OUTPUT.DD-App<sup>4</sup>, nutzen Gamification bereits zur Motivation der Nutzer. Daher wird nach der begrifflichen Einordnung in dieser Sektion betrachtet, wie die motivierende Wirkung anhand des Einsatzes bestimmter Elemente interpretiert werden kann und auf welchen psychologischen Grundbedürfnissen dies fußt. Anschließend werden Gamification-Frameworks anhand des nutzerorientierten Octalysis-Frameworks betrachtet, mithilfe derer eine Gamification aus verschiedenen Perspektiven umgesetzt werden kann.

### 2.2.1 Begriffliche Einordnung

**Definition 1** *Gamification ist die Anwendung von aus Spielen bekannten charakteristischen Gestaltungselementen auf nicht spielbezogene Kontexte [Det+11, sinngemäß übersetzt].*

<sup>1</sup>Bonuspoint rules. <https://inloop.inf.tu-dresden.de/about/bonuspoint-rules/> (Abgerufen am 13.6.2020)

<sup>2</sup>JPlag. <https://jplag.ipd.kit.edu/> (Abgerufen am 13.6.2020)

<sup>3</sup>Auditorium. <https://auditorium.inf.tu-dresden.de/> (Abgerufen am 14.9.2020)

<sup>4</sup>OUTPUT.DD-App. <https://play.google.com/store/apps/details?id=de.tud.android.outputdd&hl=de> (Abgerufen am 14.9.2020)

Deterding et al. beschreiben die Ursprünge des Begriffs in der Industrie der digitalen Medien um das Jahr 2008, wobei sich der Begriff Gamification erst im Jahr 2010 weitläufig gegenüber koexistierenden Synonymen wie „Funware“ oder „Applied Gaming“ etabliert habe. Die Arbeit der Autoren schaffte eine kommunikative Grundlage, indem sie die domänenspezifischen Begriffe durch differenzierte Betrachtung taxonomisch einordnete [Det+11]. Die Autoren grenzen den Begriff Gamification hierbei von so genannten „Serious Games“ ab. Sie beschreiben Serious Games als Spiele, die einen ernsten Verwendungszweck haben, meist die Vermittlung von Lerninhalten. Im Unterschied hierzu sei Gamification interpretierbar als eine Verwendung von aus Spielen bekannten charakteristischen Gestaltungselementen auf *nicht spielbezogene* Kontexten. Gleichzeitig differenzieren die Autoren zwischen einem „Playful Design“ und einem „Gameful Design“. Dem Playful Design liegt eine „Playfulness“ zugrunde, die eine Denkweise beschreibt, bei der spielerische Tätigkeiten ohne Ernsthaftigkeit, klarem Ziel oder echten Konsequenzen ausgeführt werden [Luc+14]. Gameful Design wiederum als Grundlage der Gamification sei komplementär zum Playful Design durch bestimmte Regeln, Handlungsweisen, Akteure, Ziele und Ergebnisse („Gaming“) charakterisiert, wobei diese durch eine „Gameful Experience“ kombiniert und erfahrbar gemacht werden.

## 2.2.2 Gamification-Elemente

Eine wichtige Grundlage für das erfahrbar Machen einer „Gameful Experience“ besteht in der Verwendung von so genannten Gamification-Elementen.

**Definition 2** *Gamification-Elemente sind charakteristische Gestaltungselemente der Gamification, welche in den meisten (jedoch nicht zwangsläufig allen) Spielen gefunden und mit diesen assoziiert werden können und dort eine signifikante Rolle im Spielablauf einnehmen [Det+11, Sinngemäß übersetzt].*

Im Folgenden werden typische Gamification-Elemente beschrieben [Sai+17, angelehnt an die Kategorisierung von Sailer et al.].

**Punkte** repräsentieren den Fortschritt und Erfolg des Spielers, als quantitative Grundlage für verschiedene weitere Gamification-Elemente, wobei als Spieler die Nutzer der Anwendung im Rahmen der Gamification gemeint sind. Schließt ein Spieler eine Handlung innerhalb der Anwendung mit Erfolg ab, erhält dieser dafür eine Punktzahl. Die jeweils erreichten Punktzahlen werden in der Regel in einem Gesamtpunktestand aggregiert. Punkte können in verschiedenen Formen und Verwendungen auftreten, z.B. in Form von Erfahrungspunkten als Grundlage für die Bestimmung einer Erfahrungsstufe (Level), oder in Form einer Währung, welche für bestimmte Gegenleistungen innerhalb der Anwendung eingetauscht werden kann.

**Errungenschaften** sind visuelle Repräsentationen des Erreichens von im Voraus festgesetzten Zielen durch einen Spieler. Sie kommen häufig in der Form von Trophäen oder Badges vor und können vom Spieler durch die Erfüllung von bestimmten Zielen erhalten werden. Meist werden diese Errungenschaften danach für andere Spieler sichtbar auf dem jeweiligen Profil präsentiert.

**Ranglisten** führen Spieler sortiert nach ihrer Punktzahl oder nach einer ähnlichen Metrik auf, wie zum Beispiel die Dauer für das Erledigen einer Aufgabe. Somit hat jeder Spieler die Möglichkeit, sich bezüglich der jeweiligen Metrik mit anderen Spielern zu vergleichen.

**Leistungsgraphen** zeigen dem Spieler, wie sich seine Leistung bei einer Aufgabe im Laufe der Zeit oder im Vergleich zu einem vorigen Durchlauf verändert. Die visuelle Darstellung wird realisiert über die Darstellung der Leistung in Relation zur Zeit durch einen Graphen. Im Unterschied zu Ranglisten wird hierbei also der Spieler mit sich selbst verglichen und nicht mit anderen Spielern.

**Narrative** werden in den bestehenden Anwendungskontext eingebunden und beziehen sich im Vergleich zu den oben genannten Gamification-Elementen nicht auf die Leistung des Spielers. Das Grundkonzept hierbei ist, die im Rahmen der Anwendung zu lösenden Aufgaben oder auszuführenden Tätigkeiten in eine Geschichte zu integrieren. Dies kann beispielsweise über eine narrative Rahmenhandlung mit eigenen Charakteren geschehen.

**Avatare** zeigen die Spieler als Spielfigur innerhalb der Anwendung. In der Regel ist es dem Spieler möglich, den eigenen Avatar zu erstellen und zu modifizieren. Dies kann zum Beispiel in Form von simplen zweidimensionalen Grafiken geschehen.

**Teammitglieder** können echte oder durch den Computer gesteuerte Mitspieler sein, die zusammen mit dem Spieler in einer Gruppe bestimmte Aufgaben lösen.

### 2.2.3 Gamification als Motivator

Sailer et al. analysierten die beobachtete Leistungssteigerung bei der Anwendung von Gamification als Resultat aus der motivierenden Wirkung [Sai+17, p. 4, 5] auf die Nutzer. Die motivierende Wirkung sei ein Resultat daraus, dass die Einführung der charakteristischen Gestaltungselemente auf die Erfüllung von psychologischen Grundbedürfnissen abzielte, konkreter:

- das **Bedürfnis nach Kompetenz**, erfüllbar durch ein Gefühl der Effizienz und des Erfolges bei der Interaktion,
- das **Bedürfnis nach Freiheit bei der Wahl einer Aufgabe**,
- das **Bedürfnis nach Freiheit bei der Einschätzung der Sinnhaftigkeit einer Aufgabe** und der damit einhergehenden Freiheit, zu entscheiden, in welchem Maße die Aufgabe zu erfüllen ist sowie
- das **Bedürfnis nach sozialer Verbundenheit**.

Sailer et al. diskutierten die psychologischen Grundlagen der Gamification anhand der Wirkung der typischen Gamification-Elemente. Sie beschreiben, mithilfe von Punkten könne dem Spieler ein belohnendes direktes Feedback für die von ihm getätigten Aktionen übermittelt werden. Da hiermit bereits einzelne Aktionen oder Teile von diesen belohnt werden, bezeichnen Sailer et al. dies als granulares Feedback. Weiterhin erklären sie, Punkte könnten dem Spieler seinen eigenen Fortschritt visualisieren und damit das Bedürfnis nach Kompetenz erfüllen. Durch Errungenschaften in Form von Badges würde dem Nutzer weiterhin die Möglichkeit gegeben werden, bestimmte Ziele zu erreichen und dies jeweils nach außen als „virtuelles Statussymbol“ zeigen zu können, führen die Autoren weiterhin auf. Verweisend auf Wang und Sun [WS12] beschreiben die Autoren, dass gleichzeitig Spieler durch den Einsatz von Badges zur Erfüllung bestimmter Aufgaben motiviert werden könnten. Da Errungenschaften nach einer Reihe von bestimmten Handlungen vergeben werden, kategorisieren Sailer et al. anhand von Badges dies auch als kumulatives Feedback. Ähnlich zu Punkten bedienten Badges auch das Bedürfnis nach Kompetenz. Zum Gamification-Element Ranglisten fassen Sailer et al. zusammen, dass mithilfe dessen Spieler motiviert werden könnten, andere Mitspieler zu übertreffen. Ranglisten wirkten als soziales Druckmittel, schlussfolgern die Autoren. Es habe sich jedoch gezeigt, dass Ranglisten auch eine demotivierende Wirkung haben können, wenn sich beispielsweise der jeweilige Nutzer im unteren Bereich der Ranglisten wiederfindet. Sie empfehlen daher, Ranglisten so zu gestalten, dass die im Vergleich zum Spieler in der Rangliste gezeigten konkurrierenden Spieler eine ähnliche Punktzahl wie dieser haben. Sailer et al. ordnen Ranglisten, analog zu Punkten, der Erfüllung des Bedürfnisses nach Kompetenz in Form von einem kumulativen Feedback zu. Weiterhin diskutieren Sailer et al. zu dem Gamification-Element Leistungsgraphen, dass sich durch die Präsentation der Leistung über eine feste Zeitspanne ein Bedürfnis für den Spieler bilde, sich selbst zu verbessern. Leistungsgraphen zielten somit auch auf das Bedürfnis nach Kompetenz ab, indem sie dem Spieler ein nachhaltiges Feedback gäben. Als Ziel von sinnvollen Narrativen nennen die Autoren, dass Aufgaben

für den Nutzer gehaltvoller und weniger langweilig wirken sollen. Durch die gesteigerte Attraktivität der jeweiligen Aufgabe ziele dieses Gamification-Element ab auf das Bedürfnis des Spielers nach Freiheit, eine Aufgabe zu bearbeiten und dabei die Sinnhaftigkeit dieser einzuschätzen. Gleichzeitig könnten Narrative gezielt als Analogie zu Prozessen aus der „echten“ Welt gewählt werden. Sailer et al. erläutern Avatare als Möglichkeit, durch deren Wahl und Erstellung freie Entscheidungen innerhalb des Kontextes getroffen werden können. Sie richteten sich somit an die Erfüllung des Bedürfnisses der Entscheidungsfreiheit. Zum von Sailer et al. kategorisierten Gamification-Element der Teammitglieder beschreiben die Autoren schließlich, durch die Bearbeitung eines gemeinsamen Ziels im Team würde hierbei das Bedürfnis nach sozialer Verbundenheit erfüllt.

## 2.2.4 Frameworks

Gamification-Ansätze können scheitern, wenn deren Grundlage eine ungeeignete Auswahl oder eine ungeeignete Kombination von Gamification-Elementen ist oder der zugrundeliegende Game-Design Prozess nicht stringent genug durchgeführt wird, beschreiben Mora et al. und analysieren daher eine Reihe von Gamification-Frameworks, welche den Gestaltungsprozess strukturieren sollen [Mor+15]. Im Folgenden wird das Gamification-Framework „Octalysis“ als Beispiel betrachtet. Der Gamification-Experte Yu-kai Chou illustriert die Notwendigkeit eines guten Game-Designs als Grundgerüst, welches einzelne Gamification-Elemente zusammenhält. Er karikiert dies anhand eines „schlechten“ Game-Designers, der bestimmte Elemente und Spielmechaniken allein anhand ihrer Popularität in anderen Anwendungen zusammenstellt, ohne auf ein gutes Game-Design zu achten [Cho19, S. 21ff]. Ein „guter“ Game-Designer würde sich nach Chous Interpretation zu Beginn fragen, welche Emotionen er beim Nutzer auslösen möchte und auf Grundlage dessen bestimmte Elemente und Spielmechaniken auswählen, mit denen diese Emotionen erzeugt werden könnten. In seinem „Octalysis“ Gamification Framework beschreibt Chou einen Gamification-Ansatz, bei dem die gewählten Methoden konzentrisch auf die Bedürfnisse und Ziele des Nutzers hinwirken. Chou analysierte hierfür verschiedene Spiele und warum einige dieser Spiele (teilweise gegenüber fast äquivalenten Kopien dieser) erfolgreich waren. Er fasst acht Kernantriebe einer Gamification zusammen:

1. **Epic Meaning and Calling:** Die erzählerische Bedeutsamkeit der dargestellten Inhalte und „Berufung“ des Spielers, indem dieser für die Lösung der Aufgabe auserwählt wird.
2. **Development and Accomplishment:** Die Möglichkeit, fortzuschreiten und dabei über Herausforderungen Errungenschaften zu erzielen.
3. **Empowering of Creativity and Feedback:** Die Möglichkeit, dass Spieler zur Lösung von Aufgaben kreativ sein können und hierbei Ergebnisse dessen beobachten können.
4. **Ownership and Possession:** Die Möglichkeit des Besitztums und der Akquise von weiterem Besitz.
5. **Social Influence and Relatedness:** Die Förderung der sozialen Interaktion.
6. **Scarcity and Impatience:** Die Reduktion der Verfügbarkeit von bestimmten Ressourcen oder die zeitliche Beschränkung von bestimmten Aktivitäten.
7. **Unpredictability and Curiosity:** Die Möglichkeit, Dinge zu entdecken oder überrascht zu werden.
8. **Loss and Avoidance:** Die Gefahr, etwas zu verlieren oder zu verpassen.

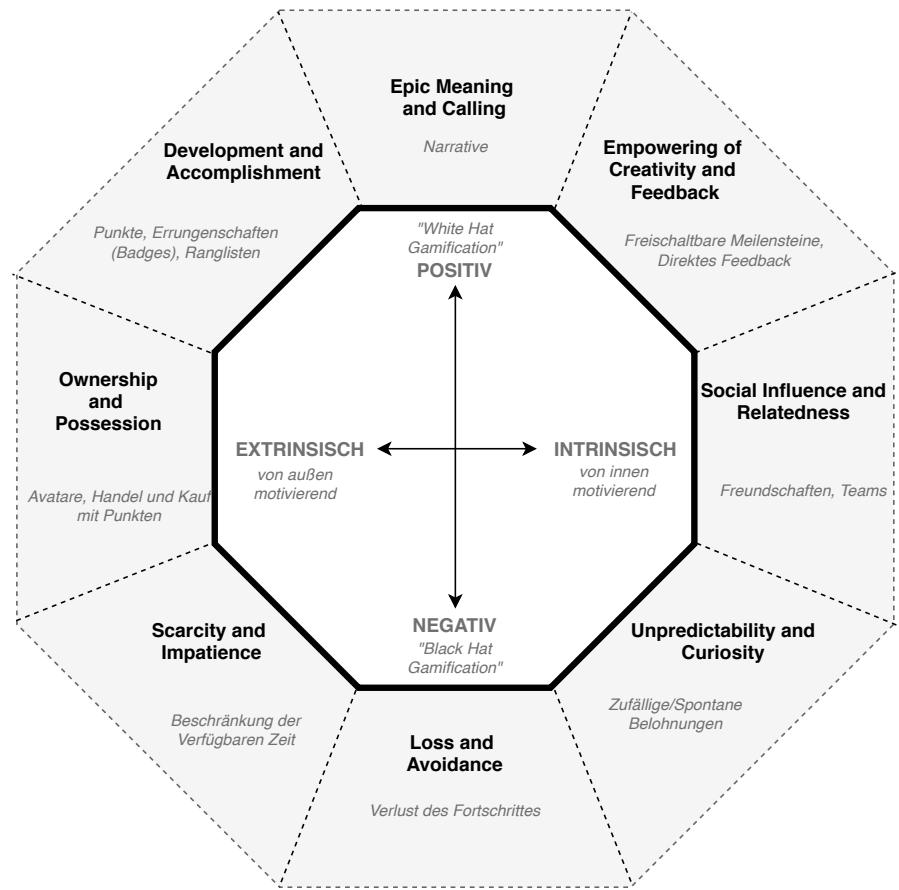


Abbildung 2.2 Die von Yu-kai Chou entwickelte Gamification-Taxonomie „Octalysis“ als Oktagon mit beispielhaften Gamification-Elementen. Mit freundlicher Genehmigung [Cho19].

Weiterhin ordnet Chou diesen Teilzielen die jeweiligen Gamification-Elemente zu. So seien die weit verbreiteten Gamification-Elemente Punkte, Ranglisten und Errungenschaften (bspw. in Form von Badges) zuzuordnen in die Kategorie „Development and Accomplishment“. Die in Abbildung 2.2 gezeigte Anordnung der acht Teilziele teilt Chou nochmals horizontal und vertikal. Die links in der Abbildung gezeigten Ziele zielen auf die durch äußere Reize hervorrufbare (extrinsische) Motivation des Nutzers ab, während die rechts gezeigten Ziele die von dem Nutzer selbst stammende (intrinsische) Motivation unterstützen. Außerdem seien die weiter oben angebrachten Ziele positiver als die weiter unten gezeigten Ziele, wie zum Beispiel „Loss and Avoidance“. Diese Teilung illustriert Chou als „White Hat Gamification“ und „Black Hat Gamification“, was eine Analogie zum legalen White Hat Hacking und zum illegalen Black Hat Hacking darstellt. Eine nach diesen Zielen orientierte Gamification bezeichnet Chou als „Level 1 Octalysis“. Unter „Level 2 Octalysis“ beschreibt er eine Gamification, welche die genannten Ziele der jeweiligen Phase des Spielers anpasst, wobei der Spieler das Spiel zunächst entdeckt, danach dessen Regeln lernt und Ziele erreichen möchte und schließlich nach erreichen aller Ziele nach weiteren Motivationen sucht. Zusätzlich ließe sich hieran nach Chou die „Level 3 Octalysis“ anschließen, die diese Faktorisierung der Spielphasen noch zusätzlich um eine Faktorisierung der Spielertypen erweitert.

## 2.3 Softwarequalität

In der folgenden Sektion werden die Grundlagen zum komplexen Begriff der Softwarequalität aufgeschlüsselt, um die Codequalität hierin schließlich systematisch einzuordnen und Möglichkeiten aufzuzeigen, diese in einem komplexen Softwaresystem abzubilden. Hierfür werden zunächst unterschiedliche Perspektiven erläutert, aus denen die Qualität einer Software betrachtet werden kann.

Nachfolgend wird die Codequalität in eine dieser Perspektiven eingeordnet, um die Konsequenzen aus einer guten oder schlechten Codequalität für die Gesamtqualität der Software ableiten zu können und die Relevanz der Codequalität als zentrale Qualitätskomponente in der Gesamtqualität von Software zu zeigen. Außerdem wird beschrieben, wie deren Analyse über Codequalitätsmetriken funktionieren kann. Als zentraler Prozess in der Verbesserung von Softwarequalität wird schließlich der Vorgang der Refaktorisierung erklärt und mit dem Begriff der technischen Schulden in Verbindung gebracht sowie aufgezeigt, inwiefern dies automatisierbar ist.

### 2.3.1 Komponenten und Faktoren

Die Norm ISO/IEC 9126:2001 [eng01], aktualisiert durch ISO/IEC 25010:2011 [Tec11] beschreibt die Softwarequalität als komplexes Resultat aus dem Lebenszyklus von Software. Die Autoren unterteilen folgende Komponenten der Gesamtqualität von Software.

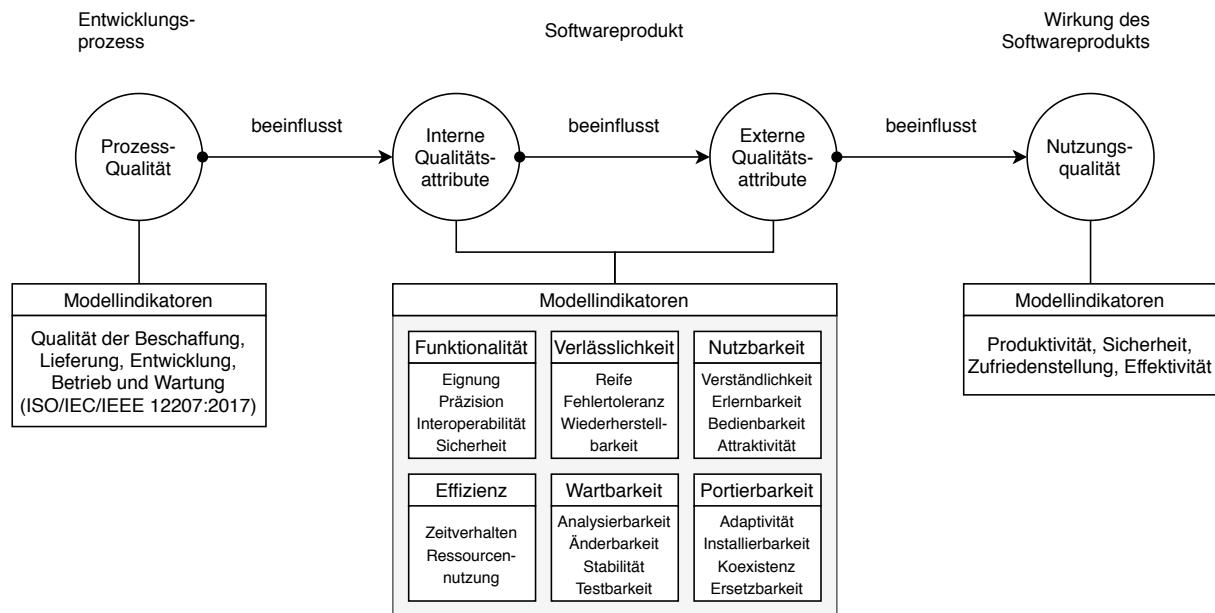


Abbildung 2.3 Die Kaskade der Softwarequalitätskomponenten im Lebenszyklus von Software und deren Modellindikatoren nach ISO/IEC 9126:2001 [eng01].

**Die Prozessqualität** beschreibt die Qualität der in ISO/IEC/IEEE 12207:2017 [eng17] normierten Prozesse (Beschaffung, Lieferung, Entwicklung, Betrieb und Wartung) des Lebenszyklus einer Software.

**Die interne und externe Qualität** wird von den Autoren beschrieben als die Charakterisierung der Software von einem internen bzw. externen Standpunkt, anhand der jeweiligen internen bzw. externen Anforderungen. Abbildung 2.3 zeigt die in ISO/IEC 9126:2001 hierfür vorgeschlagenen Modellindikatoren (Funktionalität, Verlässlichkeit, Nutzbarkeit, Effizienz, Wartbarkeit und Portierbarkeit). Die interne Qualität wird hierbei über die Analyse der inneren Details der Software (Codebasis, statische und dynamische Modelle oder Dokumentation) festgestellt. Vice versa repräsentiert die externe Qualität die ohne Kenntnis dieser internen Details der Software messbare Qualität, beispielsweise durch funktionale Tests.

**Die durch Nutzung der Software feststellbare Qualität** wird durch die Autoren definiert als die Fähigkeit der Software, die Bedürfnisse des Nutzers zu befriedigen. Sie separieren dies in die Effektivität, die Produktivität, die Sicherheit und die Zufriedenstellung bei der Nutzung.

Weiterhin wird von den Autoren illustriert, dass sich die oben genannten Komponenten in dieser Reihenfolge im Lebenszyklus der Software kaskadierend gegenseitig beeinflussen (siehe Abbildung 2.3). So ist die durch Nutzung feststellbare Qualität von der externen Qualität abhängig, diese hängt wiederum von der internen Qualität ab und die interne Qualität wird schließlich beeinflusst von der Prozessqualität.

### 2.3.2 Codequalitätsmetriken

Die Codequalität lässt sich analog zum oben beschriebenen Schema in die interne Qualität der Software einordnen und kann nach den Kriterien der ISO/IEC 9126:2001 (Funktionalität, Zuverlässigkeit, Nutzbarkeit, Effizienz, Wartbarkeit und Wiederverwendbarkeit) [eng01] beurteilt werden. Um die Konformität des Codes zu den Kriterien der internen Softwarequalität zu messen, werden typischerweise die Implementationen der einzelnen Teilkomponenten der Software anhand von bestimmten Metriken analysiert. Das grundlegende Prinzip solcher Metriken ist die Erkennung von Fehlgestaltungen verschiedener Art im Code und in dessen Dokumentation. Hierbei spielen oft der Umfang, die Komplexität und der Stil des Codes eine Rolle. Zusätzlich können Metriken eingesetzt werden, welche die Entwurfsqualität der Anwendung analysieren, zum Beispiel durch die Messung der Tiefe von Vererbungen in objektorientierten Anwendungen [RH]. Traditionelle Implementationen solcher Metriken umfassen hierzu beispielsweise die Berechnung der zyklomatischen Komplexität (McCabe-Metrik), die Bestimmung der Anzahl von Schachtelungen und Statements, die Kalkulation des Verhältnisses von Code und Kommentaren oder die Länge des Programms [Sta+02]. Die Wahl, Funktionsweise und Interpretation der Metriken ist vom Kontext, zum Beispiel von der verwendeten Programmiersprache, von den Anforderungen der Software oder von der Erfahrung des Entwicklers, abhängig. Für unterschiedliche Kontexte lassen sich somit unterschiedliche Modelle zur Codequalitätsanalyse entwerfen, deren Wirkung aus der Zusammensetzung und Parametrisierung resultiert.

### 2.3.3 Refaktorisierung

Genügt die Codequalität einer Software nicht mehr den Anforderungen der Softwarequalitätsziele, so bietet sich eine Refaktorisierung an. Händler und Neumann definieren den Begriff wie folgt:

**Definition 3** Unter Refaktorisierung versteht man die Verbesserung der internen technischen Qualität eines [Software]systems durch die Modifizierung und Restrukturierung des Quellcodes, ohne das von außen sichtbare Verhalten zu verändern. [Fow+99]

Das von außen sichtbare Verhalten ist hierbei vom Standpunkt und vom Kontext abhängig. Wird beispielsweise eine Software anhand ihrer nach außen verfügbaren Schnittstellen untersucht, so kann die interne Struktur mit Beibehaltung der äußeren Schnittstellen gänzlich verändert werden. Bei der Betrachtung einer einzelnen Komponente desselben Software-Systems, zum Beispiel anhand eines Unitests, muss die sichtbare Signatur der Komponente jedoch bei den Änderungen beibehalten werden, um nach der obigen Definition als Refaktorisierung zu gelten.

### 2.3.4 Technische Schulden

Um die Notwendigkeit der Refaktorisierung innerhalb einer jeweiligen Softwarekomponente zu quantifizieren, akkumulieren viele automatisierte Codeanalyseframeworks, wie zum Beispiel SonarQube<sup>5</sup>, anhand einer Auswahl von Codequalitätsmetriken und anderen Metriken einen so genannten TD-Score, wobei TD für Technical Debt (technische Schulden) steht. Der Score setzt sich hierbei metaphorisch zusammen aus der geschätzten Zeit, die zur Refaktorisierung der jeweiligen detektierten Fehlgestaltung notwendig wäre.

---

<sup>5</sup>SonarQube. <https://www.sonarqube.org/> (Abgerufen am 14.9.2020)

Die Art der Fehlgestaltungen kann hierbei jedoch stark variieren. Kruchten et al. unterteilen technische Schulden nochmals in Testschulden, menschliche Schulden, architekturelle Schulden, sich auf Abhängigkeiten beziehende Schulden, Dokumentationsschulden oder allumfassende amorphe Softwareschulden [KNO12]. Kruchten et al. erklären darauf aufbauend, warum der errechnete TD-Score nicht mit den tatsächlichen technischen Schulden der Software gleichgesetzt werden sollte [KNO12]. Die Gesamtheit von technischen Schulden sei nur schwierig durch statische Codeanalyseframeworks erfassbar, vor allem strukturelle und architekturelle Fehlgestaltungen. Zu beachten sei hierbei, dass sowohl die Codeanalyseframeworks als auch die zu analysierende Software dem technologischen Evolutions- und Alterungsprozess unterliegen.

### 2.3.5 Codierungsrichtlinien

Softwareentwickler können unterschiedliche Auffassungen von guter Codequalität haben. Hieraus lässt sich die Hypothese ableiten, dass Codequalitymetriken generell ungeeignet sind, um das Verständnis von guter Codequality in allen Facetten zu modellieren. Pantiuchina et al. zeigten hierzu anhand eines Experimentes, dass die durch Metriken modellierbare Repräsentation von guter Codequality nicht zwingend mit der Auffassung von Entwicklern übereinstimmt [PLB18].

Aus der Sicht des Entwicklers betrachtet, kann es, abhängig von der Zusammensetzung und Parametrisierung der Codequalitymetriken, zu falsch-positiven und falsch-negativen Detektionen kommen. Dies ist jedoch nicht immer von der individuellen Auffassung des Entwicklers abhängig. Händler und Neumann nennen hierzu den Fall, dass es auch bei bewusst auf eine bestimmte Art und Weise implementierten Strukturen, beispielsweise bei Entwurfsmustern, zu falsch-positiven Detektionen kommen kann. Sie begründen dies in der inhärenten Komplexität der Detektionsmechanismen [HN19b].

#### Konformität

Jan Rucks befasste sich im Rahmen seiner Diplomarbeit mit der Auswahl von geeigneten Codequalitymetriken für das sich an die Lehrveranstaltung Softwaretechnologie zeitlich anschließende Softwarepraktikum, wo die Codequality einiger Gruppen durch SonarQube analysiert wird [Ruc17]. Wegen der oben genannten Schwierigkeiten ist es nicht verwunderlich, dass es Rucks nicht gelang, die Qualität einer Software (halb)-automatisch anhand einer Zusammenstellung von Qualitätsmetriken in Form einer einzigen Metrik zu messen.

Rucks evaluierte hierzu mehrere Metriken, welche bestimmte Teilelemente der internen Softwarequalität modellieren sollen, darunter Flexibilität, Konformität, Sicherheit, Wiederverwendbarkeit, Testbarkeit, Modularität und Wartbarkeit. Rucks' Evaluation verglich dabei die Berechnungen der Metriken mit den Bewertungen der Tutoren zu der jeweiligen Gruppe sowie deren Selbsteinschätzung im Softwarepraktikum, um die Korrelation der Metriken mit der Softwarequalität zu bestimmen. Hierbei sei angemerkt, dass sowohl die Bewertungen der Tutoren als auch die Selbsteinschätzung der Gruppen gute Codequality nicht ideal repräsentieren, was die Aussagekraft der darauf fußenden Evaluation beschränkt. Rucks destillierte aus der Evaluation dennoch als einzige signifikant mit den jeweiligen Einschätzungen der Gruppe korrelierende Metrik die der Konformität, welche sich wie folgt berechnen lässt:

$$M_{CON} = 1 - \frac{N_{BLV} * 2 + N_{CRV} * 1,5 + N_{MAV} + N_{MIV} * 0,5}{N_{STA}} \quad (2.1)$$

wobei:

$M_{CON}$  = Konformitätsmetrik

$N_{BLV}$  = Anzahl der Verletzungen von „Blocker“-Regeln

$N_{CRV}$  = Anzahl der Verletzungen von „Critical“-Regeln

$N_{MAV}$  = Anzahl der Verletzungen von „Major“-Regeln

$N_{MIV}$  = Anzahl der Verletzungen von „Minor“-Regeln

$N_{STA}$  = Anzahl von Statements

Analog zu den Teilkomponenten der in Gleichung (2.1) beschriebenen Konformitätsmetrik berechnet sich diese aus dem Verhältnis der gewichteten Anzahl an Verletzungen von Codierungsrichtlinien zu der Gesamtanzahl an Statements. Die Gewichtung stammt vom durch SonarQube kategorisierten Schweregrad der jeweiligen Verletzung der Codierungsrichtlinien, von der schwerwiegendsten Art „Blocker“, über „Major“, „Minor“ bis hin zur Einstufung als „Info“. Die Beobachtung, dass die Konformität, also die Einhaltung von bestimmten Codierungsrichtlinien, die Codequality signifikant positiv beeinflusst, beschreiben auch Dietz et al [Die+18]. Durch die Einhaltung solcher Richtlinien würde die Lesbarkeit des Codes erhöht und hierdurch dessen Wartbarkeit verbessert. Als Konsequenz hieraus verringerten sich die Wartungskosten der Software.

## Detektion von Code Smells

Mithilfe einer fundierten Auswahl von Codierungsrichtlinien ist es möglich, eine gegebene Software auf potenzielle Verstöße gegen diese Codierungsrichtlinien zu prüfen und dem Entwickler verständlich aufzuzeigen. Die detektierten Verstöße werden hierbei oft als „Code Smells“, also sinngemäß „schlechte Gerüche“ bezeichnet. Als Synonym werden Code Smells auch einfach als „Smells“ oder präziser als Refaktorisierungskandidaten bezeichnet. Die Analyse der Code Smells läuft hierbei ohne einen tatsächlichen Start der Anwendung, weshalb diese Technik auch als statische Codeanalyse bezeichnet wird. Tools wie Checkstyle<sup>6</sup>, PMD<sup>7</sup> und FindBugs<sup>8</sup>/SpotBugs<sup>9</sup><sup>10</sup> verwenden Pattern Matching, um Code Smells anhand von bestimmten Mustern zu finden und so dem Entwickler zusätzlich zu den meist bereits vom Compiler bereitgestellten Warnungen Hinweise auf potenzielle Degradationen der Codequality zu geben. Wegen der großen Variabilität der zugrunde liegenden Codierungsrichtlinien gibt es eine Vielzahl von möglichen Unterteilungen von Code Smells. Fowler et al. unterteilen Code Smells beispielsweise in 22 Kategorien [Fow+99].

## Automatisierung

Mithilfe einer automatisierten Detektion von Refaktorisierungskandidaten können Degradationen der Codequality frühzeitig erkannt und behoben werden. In kontinuierlich evolvierenden Softwareprojekten und vor allem bei iterativen Projektmanagement-Strategien wie SCRUM [Glo16] oder dem Spiralmodell [Boe88] repräsentiert die Refaktorisierung als Entwicklungsschritt einen großen Zeitaufwand, wobei die einzelnen Refaktorisierungsmaßnahmen repetitiv erscheinen mögen. Zur Mitigation dieses Problems haben sich einige Ansätze etabliert, zum Beispiel Autofactor [Rou20], WalkMod [Pau20], Facebooks pfff [Fac20], Kadabra [Bis20] oder Leafactor [CAR17]. Diese Tools detektieren Refaktorisierungskandidaten und beheben diese soweit möglich. Zur Vereinfachung können die Tools dabei teilweise in Entwicklungsplattformen integriert oder über eigene Programmierschnittstellen angesprochen werden. Dennoch ist der Handlungsfreiraum dieser Tools durch die Komplexität der Detektionsmechanismen und der zu refaktorisierenden Software beschränkt. Eine automatisierte Refaktorisierung ist hierdurch mit weiteren Problemen behaftet, zum Beispiel durch die transitive Induzierung von weiteren Refaktorisierungskandidaten in den Programmcode durch die Behebung eines einzelnen Refaktorisierungskandidaten (für ein konkretes Beispiel siehe Abschnitt 3.1.1). Durch die Beschränktheit der automatisierten Refaktorisierung kann diese infolgedessen die manuelle Refaktorisierung nicht vollständig ersetzen, aber dennoch als hilfreiches Werkzeug bei der Entwicklung dienen.

---

<sup>6</sup>Checkstyle. <https://checkstyle.sourceforge.io/> (Abgerufen am 13.6.2020)

<sup>7</sup>PMD. <https://pmd.github.io/> (Abgerufen am 13.6.2020)

<sup>8</sup>FindBugs. <http://findbugs.sourceforge.net/> (Abgerufen am 13.6.2020)

<sup>9</sup>SpotBugs. <https://spotbugs.github.io/> (Abgerufen am 14.9.2020)

<sup>10</sup>FindBugs wird nicht mehr weiterentwickelt und wurde durch SpotBugs ersetzt.

## 2.4 Zusammenfassung

Im E-Learning-System INLOOP können Studierende interaktiv Programmieraufgaben lösen und erhalten ein direktes Feedback über die funktionelle Korrektheit der eingereichten Lösungen. Neben der funktionellen Korrektheit trägt auch die Codequality maßgeblich zur Gesamtqualität der Software bei. Dabei ist die eindeutige Erkennung von guter Codequality sowohl für Entwickler, als auch für automatisierte Metriken schwierig und komplex. Es wurde noch keine Metrik gefunden, die eine gute Codequality vollständig abbildet. Die Metrik der Konformität scheint allerdings signifikant mit der Codequality zu korrelieren. Durch die Verbesserung der Lesbarkeit und Wartbarkeit scheint Software, die einem festen Regelwerk mit Codierungsrichtlinien entspricht, in der Regel eine höhere Codequality zu besitzen, als Software, die diesem nicht entspricht. Für die Detektion von Verstößen gegen diese Richtlinien können Tools zur statischen Codeanalyse verwendet werden, um potenzielle Verstöße (Code Smells) gegen die festgelegten Codierungsrichtlinien zu detektieren und hierüber die Konformitätsmetrik gewichtet zu aggregieren. Studierende könnten zur Erhöhung der Codequality motiviert werden, die zugrundeliegenden Codierungsrichtlinien einzuhalten und Verstöße gegebenenfalls zu refaktorisieren. Als in unterschiedlichen Domänen erfolgreicher Motivator etablierte sich das Prinzip der Gamification, welches durch die Einführung von über ein Game-Design miteinander verknüpften Gamification-Elementen auf die Erfüllung von psychologischen Bedürfnissen abzielt. Anhand des Octalysis-Ansatzes wurde beschrieben, welche Faktoren bei einem nutzerzentrierten Game-Design zu beachten sind und wie diese wirken. In diesem Rahmen wurde gezeigt, wie unterschiedliche Game-Design-Elemente intrinsisch oder extrinsisch wirken können und wann diese als positiv oder negativ empfunden werden.

## 3 Related Work

Im folgenden Kapitel werden verschiedene verwandte Ansätze aus der Domäne der Gamification diskutiert, deren direktes oder indirektes Ziel die Verbesserung der Codequalität im universitären Kontext ist. Einen allgemeinen Überblick über die Gamification im Software Engineering zeigen die systematischen Studien von Pedreira et al. [Ped+14] und von Alhammad und Moreno [AM18], Letztere mit dem für diese Arbeit relevanten didaktischem Fokus. Obwohl Alhammad und Moreno anhand der systematischen Analyse Gamification als vielversprechendes Prinzip für die Verbesserung der Lehre der Softwareentwicklung einschätzen, gibt es zur Applikation von Gamification in diesem Kontext nur wenig empirische Forschung [AM18]. Campolina et al. analysierten die Gründe hinter der geringen Verbreitung von Gamification in diesem Kontext [CSF18]. Außerdem lässt sich beobachten, dass einige der Ansätze, Gamification in der Lehre der Softwareentwicklung einzusetzen, andere Schwerpunkte als die Codequalität behandeln, beispielsweise das Requirements Engineering [MA18][CSF18], den Scrum-Prozess [Net+19], die Integration von Peer-Review-Konzepten [ILD20][BN19], die Verbesserung von hochintensiven Programmierkursen [AS14] oder die Motivation von Softwaretests [RF16][SBK11]. Nennenswert ist auch der eher unkonventionelle Ansatz von Baars und Meester, die ein Java-Refaktorisierungsspiel in Minecraft integrierten [BM19]. Elezi et al. zeigen eine Anwendung von Gamification zur Motivation der Refaktorisierung von Code [Ele+16], allerdings nicht in einem akademischen Kontext, sondern mit praxiserfahrenen Softwareentwicklern.

Dos Santos et al. kamen daher 2019 nach deren Einschätzung zu dem Schluss, dass ihre Arbeit die erste wissenschaftliche Kontribution im Bereich der Motivation zur Refaktorisierung von Code Smells durch Gamification im akademischen Kontext ist [San+19]. Nachfolgend wird zunächst das von dos Santos et al. als „CleanGames“ bezeichnete Konzept diskutiert. Anschließend wird ein verwandter Ansatz von Händler und Neumann beschrieben, der die Motivation der Refaktorisierung von Code über Serious Games behandelt [HN19b]. Zur Erkennung von Refaktorisierungskandidaten eignet sich die statische Codeanalyse. Mit Fokus auf diese beschäftigen sich Dietz et al. mit der Frage, wie Codequalität gelehrt werden kann, um schließlich eine Grundlage für die automatisierte Codequalitätsanalyse in einem universitären Kontext zu konzeptionieren [Die+18]. Dieser Ansatz wird in der abschließenden Sektion diskutiert.

### 3.1 CleanGame: Gamifying the Identification of Code Smells

Wie in Abschnitt 2.3.3 bereits näher beschrieben, bietet die Refaktorisierung eine Möglichkeit, die Softwarequalität zu verbessern, indem interne Komponenten abgeändert werden, während die externe Funktionalität unverändert bleibt. Dieser Prozess inkludiert eine systematische Suche und Behandlung von Refaktorisierungskandidaten. In welchem Maße eine Refaktorisierung durchgeführt werden kann und wie erfolgreich diese der Softwarequalität zugutekommt, hängt somit auch von der Menge und der Sinnhaftigkeit der gefundenen Refaktorisierungskandidaten ab. Auf Grundlage

dieser Prämisse untersuchten dos Santos et al., wie die Suche von Refaktorisierungskandidaten in einem akademischen Kontext durch Gamification spielerisch motiviert werden kann und welche quantitativen Auswirkungen dies auf die Fähigkeit der Nutzer, Refaktorisierungskandidaten zu finden, haben kann [San+19].

### 3.1.1 Game-Design

Die von den Forschern hierzu entworfene Webanwendung trägt den Namen „CleanGame“ und besteht aus einem Quiz über Code Smells und einem Spiel, bei dem Refaktorisierungskandidaten in Codeschnipseln gefunden werden sollen. Über das Quiz sollen die Kernkonzepte der Refaktorisierung reflektiert werden, während durch das andere Modul die Suche nach Refaktorisierungskandidaten trainiert werden soll. Das Softwaretool lässt sich in die Kategorie der Serious Games einordnen.

#### Code-Smell-Quiz

Im Code-Smell-Quiz werden dem Nutzer Fragen zu ausgewählten Code-Smells gestellt. Die Forscher fokussieren sich hierbei auf die folgenden fünf konkreten Code-Smells, basierend auf einer Metaanalyse von Code Smells im akademischen und industriellen Kontext von Sharma und Spinellis [SS17]:

1. **Large Class:** Hierunter fallen Klassen, welche zu viele Verantwortlichkeiten tragen. Dieses Anti-Pattern ist auch bekannt unter den Namen „Blob“ oder „Gott-Klasse“ [Sou20].
2. **Long Method:** Dies bezeichnet Methoden, welche zu viele Zeilen von Code beinhalten.
3. **Divergent Change:** Hiermit beschreiben die Autoren eine Klasse, die sich im Verlauf der Entwicklung häufig und aus verschiedenen Gründen ändert.
4. **Feature Envy:** Unter diesem Namen werden Klassen kategorisiert, welche deren Funktion hauptsächlich durch die Kommunikation mit anderen Klassen realisieren, also kaum eigene Funktionalität bereitstellen. Häufig steht dieses Anti-Pattern in Verbindung mit der Auslagerung von Daten-Klassen [Ref20a].
5. **Shotgun Surgery:** Durch diese Bezeichnung beschreiben dos Santos et al. Modifikationen in Klassen, welche viele kleine Änderungen in anderen Klassen hervorrufen.

Innerhalb des Quiz werden dem Nutzer Fragen zu diesen fünf Kategorien gestellt, wobei die Antwortmöglichkeiten festgelegt sind. Um den Frageprozess zu illustrieren, zeigen dos Santos et al. folgende Frage inklusive deren Antwortmöglichkeiten:

Um den Shotgun Surgery Code Smell zu beseitigen, können Refaktorisierungen verwendet werden. Welche der folgenden Refaktorisierungen würden Sie nutzen, um dies zu realisieren?

- A Vererbung mit einer Delegation ersetzen
- B Inline Class
- C Methoden extrahieren
- D Klasse extrahieren

[San+19, sinngemäß übersetzt. S. 4, Abb. 1]

Um diese Frage, ohne zu raten, erfolgreich zu lösen, muss der Nutzer ein komplexes, vernetztes Fachwissen über Code Smells und Refaktorisierung besitzen. Zunächst muss der Nutzer wissen, worum es sich bei der „Shotgun Surgery“ handelt und, dass das zugrundeliegende Problem dieses Anti-Patterns die Zersplitterung einer Verantwortlichkeit über verschiedene Klassen ist [Ref20c]. Eine Möglichkeit der Refaktorisierung wäre die Extraktion von zur Verantwortlichkeit gehörenden

Methoden aus den verschiedenen Klassen und deren Reintegration in eine zentrale Klasse. Somit wäre Antwortmöglichkeit C bereits richtig. Durch die Migration der Methoden können jedoch auch Klassen entstehen, welche selbst keine Verantwortlichkeiten mehr besitzen. Diese Klassen werden als „Inline Class“ [Ref20b], meist auch als „Data Class“ bezeichnet und fallen unter die Kategorie „Feature Envy“. Richtig wäre also auch Antwortmöglichkeit B, um die möglichen transitiven Konsequenzen aus der initialen Refaktorisierung zu behandeln. Dieses Beispiel illustriert, wie über den Ansatz der Autoren über ein Quiz komplexes vernetztes Fachwissen zur Refaktorisierung abgefragt, reflektiert und erlernt werden kann.

### **Verwendete Gamification-Elemente**

Während des Quiz erhält der Nutzer Punkte für das richtige Beantworten einer Frage. Für das falsche Beantworten einer Frage bekommt der Nutzer Punkte abgezogen. Noch während des Quiz kann der Nutzer den durch seinen aktuellen Punktestand errechneten Rang in der Rangliste einsehen. Für eine Frage hat der Nutzer nur begrenzt Zeit zur Verfügung. Fragen können übersprungen werden, während Schnelligkeit bei deren Beantwortung durch einen Zeitbonus belohnt wird. Beantwortet der Nutzer einige aufeinander folgende Fragen richtig, so erhält er auch hierfür Bonuspunkte. Weiterhin sieht der Nutzer seinen eigenen Fortschritt über einen Fortschrittsbalken in der Webanwendung. Der Nutzer hat, nach [San+19, S. 4, Abb. 1] außerdem die Möglichkeit, sich zu registrieren und für den erstellten Account einen rudimentären Avatar anzulegen.

### **Code-Smell-Suche**

Bei der zweiten Kernkomponente von CleanGame handelt es sich um eine Code-Smell-Suche, bei der dem Nutzer Code-Schnipsel gezeigt werden, zu denen er den passenden Code Smell aus einer festen Auswahl von Code Smells selektieren soll. Die Code-Smell-Suche ähnelt dem Quiz in deren Aufbau und Funktionsweise, denn auch hier erhält der Nutzer Punkte für richtige Antworten und bekommt Punkte für falsche Antworten abgezogen. Dies wird erweitert durch die Fähigkeit des Nutzers, die Punkte als Währung gegen Tipps einzutauschen. Ein Tipp kann die Angabe von Regeln sein, die mit dem Code Smell in Verbindung stehen, oder eine Einschränkung des zu betrachtenden Codeabschnittes, oder die dem Code Smell zugrundeliegende Definition.

#### **3.1.2 Ergebnisse**

Dos Santos et al. prüften ihr Konzept im akademischen Kontext anhand einer quantitativen Analyse über die Messung der durch die Probanden gefundenen Code Smells und einer qualitativen Analyse der persönlichen Einstellung der Nutzer gegenüber dem Game-Design. Bei der quantitativen Analyse zweier randomisierter Nutzergruppen, wobei die eine Nutzergruppe CleanGame und die andere Nutzergruppe eine IDE im Voraus zum Training der Identifikation von Code Smells nutzte, konnten die Nutzer der CleanGame-Gruppe nach dem Training doppelt so viele Code Smells identifizieren, wie Nutzer der IDE-Gruppe. Für die Selektion der Trainingsbeispiele und der Evaluationsbeispiele nutzten dos Santos et al. validierte Code Smells der Plattform „Landfill“ [Pal+15]. In einer qualitativen Analyse über eine Umfrage ermittelten dos Santos et al., dass die Attitüde der Probanden gegenüber dem evaluierten Serious Game überwiegend positiv war. Als Verbesserungswürdig schätzten Probanden unter anderem vor allem Probleme mit der Benutzeroberfläche ein, aber auch, dass kein direktes Feedback über die richtige Antwort einer Frage gegeben wurde und, dass die Punkte eines jeweiligen Nutzers öffentlich einsehbar waren. Als positiv schätzten die Probanden wiederum unter anderem vor allem die spielerische und interaktive Auseinandersetzung mit dem Thema ein sowie die Unterstützung beim Verstehen von Code Smells und die Kompetitivität. Die qualitative Evaluation der ästhetischen Wahrnehmung des Tools ergab weiterhin, dass durch den Ansatz der Autoren die Relevanz, die Lernwahrnehmung und die soziale Interaktion besonders betont würde.

Die Autoren interpretieren diese Ergebnisse als Unterstützung der Hypothese, dass Gamification in diesem Kontext geeignet ist. Zu berücksichtigen ist jedoch, dass sowohl die qualitative als auch die quantitative Analyse nur mit 18 Probanden durchgeführt wurden und somit nur bedingt aussagekräftig sind. Daher merken dos Santos et al. an, dass die Ergebnisse weiterhin validiert werden müssen, um eine hinreichende Konfidenz über die Wirksamkeit des Effekts der Gamification im universitären Kontext zur Motivation von Refaktorisierung zu erhalten.

### 3.2 Serious Refactoring Games

Die Refaktorisierung von Code mit Hinblick auf die Verbesserung der Codequalität ist ein komplexer Prozess. Ein zu dem vorgestellten „CleanGame“ Konzept ähnliches Serious Game konzipierten Händler und Neumann. Die Autoren betrachten die didaktische Komplexität der Refaktorisierung von Code anhand der Taxonomie von Bloom [Blo56][AKB00], welche Denkfähigkeiten geordnet nach der mentalen Beanspruchung, von „lower order thinking skills“ wie dem Merken von Informationen bis hin zu „higher order thinking skills“ wie der Evaluation und der Kreation kategorisiert. Händler und Neumann kommen dabei zu dem Ergebnis, dass die Refaktorisierung von Code alle der Kategorien in Blooms Taxonomie beansprucht [HN19b]. Diese Aufschlüsselung ist in Tabelle 3.1 gezeigt.

Tabelle 3.1 Kognitive Fähigkeiten bei der Refaktorisierung anhand von Blooms Taxonomie.  
Sinngemäß aus der tabellarische Darstellung in [HN19b, S. 3] abstrahiert.

Stufe in Blooms Taxonomie	Beispiel für eine benötigte Fähigkeit bei der Refaktorisierung
1. Wissen	Die Regeln und Schritte für die Planung und die Durchführung der Refaktorisierung kennen.
2. Verstehen	Die Regeln und Schritte für die Planung und die Durchführung der Refaktorisierung verstehen.
3. Applikation	Die Regeln und Schritte für die Planung und die Durchführung der Refaktorisierung anwenden.
4. Analyse	Den Quellcode eines Systems analysieren und hieraus Kandidaten für die Refaktorisierung ableiten.
5. Bewertung	Mehrere Möglichkeiten der Refaktorisierung von Refaktorisierungskandidaten miteinander vergleichen und sich für die bestmögliche Lösung entscheiden.
6. Kreation	Entwicklung und Verbesserung von Systemen und Arbeitsweisen, welche die Refaktorisierung von Softwaresystemen vereinfachen oder strukturieren.

Anhand dieser Erkenntnis kritisieren die Autoren, dass bisherige didaktische Lösungen für die Lehre des Refactorings nur künstliche, kleine Aufgaben beinhalten, die aber nicht ausreichend seien, um tiefergehende Kompetenzen wie analytisches Denken und die Bewertung von Lösungen zu vermitteln. Stattdessen schlagen sie ein Spiel vor, bei dem dem Nutzer ein größeres, in dessen Kontext funktionales, Codefragment präsentiert wird und es daraufhin des Nutzers Aufgabe ist, die in diesem Codefragment vorhandenen Code Smells zu beseitigen, ohne die von außen erkennbare Funktionalität abzuändern. Hierbei kann der Nutzer gegen einen Codequalitätsbenchmark (Technische Schulden) oder gegen echte Mitspieler antreten. In diesem Kontext beschreiben Händler und Neumann eine technische Grundlage für das Game-Design solcher „Serious Refactoring Games“ auf Basis von existierenden Codeanalyse-Tools (Test-Frameworks, Softwarequalitätsmetriken) und Regressionstests.

Die vorgeschlagene Lösung von Händler und Neumann hat das Ziel, insbesondere auch die höherliegenden Stufen 3-6 der Taxonomie von Bloom (Abschnitt 3.2) didaktisch zu vermitteln, da sich existierende Lösungen eher auf die darunter liegenden Stufen orientieren, analysieren die Autoren.

Durch die Konfrontation mit Codefragmenten aus „echten“ Anwendungen werde eine praxisnähere Situation simuliert, bei der Nutzer die Artefakte zunächst analysieren müssen und danach Alternativen evaluieren müssen, um die Code Smells zu beseitigen, wodurch höhere Lernziele in Blooms Taxonomie repräsentiert würden.

### 3.2.1 Game-Design

Der Spieler erhält den Quellcode der zu refaktorisierenden Datei(en). Des Spielers Aufgabe ist es hiernach, den Quellcode auf mögliche zu refaktorisierende Stellen (Refaktorisierungskandidaten) zu untersuchen und diese auszuwählen. Wenn der Spieler einen oder mehrere Kandidaten ausgewählt hat, muss er Modifikation(en) des Quellcodes planen und ausführen, die den jeweiligen ausgewählten Kandidaten behandelt. Signalisiert der Spieler, dass dieser die Modifikation(en) abgeschlossen hat, wird die editierte Datei im ursprünglichen System integriert und auf Funktionalität geprüft (Regressionstest). Nun kann es zwei Ergebnisse dieser Interaktion geben:

- Der Programmcode ist valid und alle Regressionstests waren erfolgreich. Infolgedessen wird die Qualität des refaktorisierten Codes analysiert und ein Score für die technischen Schulden ermittelt.
- Einer oder mehrere Regressionstest schlagen fehl oder der Programmcode ist anderweitig invalid. Die Qualität des refaktorisierten Codes wird nicht analysiert und es wird kein Score für die technischen Schulden ermittelt.

Das ermittelte Ergebnis wird dem Spieler präsentiert und dieser erhält, wenn vorhanden, den zum refaktorisierten Code berechneten Score. Der Spieler erhält nun die Option, weitere Modifikationen an den bereits ausgewählten Kandidaten vorzunehmen oder nach weiteren Kandidaten zu suchen. Das Spiel ist nach dem beschriebenen Ablauf in Spielzüge unterteilt, welche nach deren Abschluss als inkorrekt, korrekt oder erfolgreich gelten können:

- Inkorrekt: der Quellcode ist invalid und konnte nicht auf Funktionalität oder Qualität geprüft werden.
- Korrekt: der Quellcode ist valid, die technischen Schulden haben sich aber im Vergleich zum vorigen Spielzug nicht verringert.
- Erfolgreich: der Quellcode ist valid und die technischen Schulden haben sich im Vergleich zum vorigen Spielzug verringert.

Händler und Neumann schlagen mehrere erweiterte Spielmodi vor, darunter ein Einzelspielermodus, wobei der Spieler gegen die Uhr spielt sowie mehrere Mehrspielermodi (parallel vs. alternierend, kompetitiv vs. kollaborativ).

### Ziele eines Serious Refactoring Games

Das oberste Ziel des Spielers besteht darin, das Codefragment auf eine solche Weise zu refaktorisieren, sodass die Funktionsfähigkeit der Software weiterhin durch die Regressionstests verifiziert werden kann, sich in diesem Kontext also nach außen nicht verändert, während die technischen Schulden minimiert werden.

### Technische Realisation

Das von Händler und Neumann konzipierte Softwaresystem ist in die folgenden Komponenten unterteilt:

- Die **Game Control** koordiniert die Interaktion von Spielern und dem Testframework bzw. der Qualitätsanalyse.

- Das **Test Framework** führt Regressionstests auf der eingereichten Software aus.
- Der **Quality Analyzer** analysiert die Qualität des Codefragments.
- Die **GUI Component** kapselt das visuelle Feedback und die gesamte visuelle Interaktion.

Als Grundlage schlagen Händler und Neumann eine Messung der internen technischen Qualität über die Akkumulation eines TD-Scores (Abschnitt 2.3.4) vor.

### 3.2.2 Ergebnisse

Händler und Neumann planen die Evaluation des vorgestellten Konzeptes im universitären Kontext im Rahmen zukünftiger Arbeit [HN19b, S. 9]. Zu dem vorgestellten Konzept der Serious Refactoring Games wurden noch keine experimentellen Resultate publiziert (Stand: 30. Juni 2020), so auch nicht in der in [HNS19] von Händler et al. auf Grundlage dessen fortgeführten Arbeit, welche sich im Kontrast zu den vorgestellten Serious Refactoring Games auf die Realisation der Interaktivität über ein direktes Feedback im Rahmen eines Tutoring-Systems fokussiert.

### 3.2.3 Framework

In [HN19a] diskutieren Händler und Neumann die Serious Refactoring Games und das oben genannte Tutoring-System erneut und konzipieren anhand dessen ein Framework, welches die Lernziele in Blooms Taxonomie möglichst gut abbildet. Hierbei soll das vorgestellte Serious Refactoring Game durch folgende Konzepte aus dem Tutoring-System komplementiert werden:

- Die Bereitstellung von direktem, interaktiven Feedback durch die Präsentation der Resultate von Regressionstests.
- Der interaktive Vergleich des durch den Code zu reflektierenden Referenz-UML-Diagramms (Soll-Zustand) mit einem aus dem eingereichten Code ermittelten UML-Diagramm (Ist-Zustand).

Ähnlich zu den in Abschnitt 3.1 beschriebenen CleanGames soll hierdurch ein weiterer Spielmodus umgesetzt werden, dessen Ziel die Eliminierung von Code Smells aus einem gegebenen Codefragment ist, während die Regressionstests auf eine fortexistierende Funktionalität und damit die Validität der Refaktorisierung prüfen. Als eine weitere Möglichkeit zur Ergänzung des Frameworks nennen Händler und Neumann die Integration eines Quiz über Refaktorisierung, ähnlich dessen in CleanGames, welches in Abschnitt 3.1 näher beschrieben wurde.

## 3.3 Teaching Clean Code

Dietz et al. befassten sich mit den Forschungsfragen, wie die Programmierung von „Clean Code“, also Code mit hoher Konformität zu Codierungsrichtlinien, in einem universitären Kontext gelehrt werden kann und wie dies mit einer zunehmenden Anzahl an Studierenden skalierbar umsetzbar ist. Dazu gehen die Autoren auf Probleme ein, die in der Lehre von Clean Code im universitären Kontext bestehen. Über die Integration eines didaktischen Konzeptes auf Grundlage von persönlichem Feedback sammelten Dietz et al. Erkenntnisse über die Strukturierung und Auswahl von Codierungsrichtlinien. Mithilfe dieser empirischen Auswahl wird schließlich eine Grundlage für deren Integration in einem E-Learning-System wie ArTEMiS [KS18], einem INLOOP ähnlichen E-Learning-System für Softwaretechnologie der Technischen Universität München, mit Fokus auf der Verbesserung der Codequalität gegeben.

### Herausforderungen in der universitären Lehre

Dietz et al. beschreiben, dass Studierende der Informatik häufig aufgrund der Lehrstruktur des Studiengangs nicht ausreichend motiviert würden, sich über die praktische Applikation von theoretischen

Konzepten hinweg autodidaktisch mit der qualitativen Verbesserung des eigenen Codes zu befassen, weil dieser oft nach der Entwicklung vergessen oder „weggeworfen“ würde. Motiviert hierdurch analysierten Dietz et al., wie die Einhaltung von Codierungsrichtlinien in einem universitären Kontext, ähnlich dem dieser Arbeit, gelehrt werden kann. So integrierten sie ein didaktisches Konzept in den Präsenzbetrieb einer Lehrveranstaltung mit den folgenden drei Kernelementen:

- Interaktive Diskussion von Refaktorisierungskandidaten in der Vorlesung zwischen den Studierenden und dem Vorlesenden anhand von Lösungen zu Aufgaben
- Detaillierte Code Reviews von Lösungen der Übungsgruppen durch Lehrpersonal
- Integration einer Aufgabenstellung in die mündliche Prüfung, bei der Refaktorisierungskandidaten in einem unbekannten Code-Schnipsel erkannt werden sollen

Dieses didaktische Konzept erprobten die Autoren an der Universität Bamberg mehrere Jahre und evaluierten schließlich mithilfe von statischer Codeanalyse die messbaren Änderungen in der Konformität des eingereichten Codes zu einem festen Regelwerk. Sie stellten als Tendenz fest, dass sich die allgemeine Struktur des Codes verbesserte, während andere Regelverstöße zunahmen, wobei die Autoren diese Zunahmen teilweise der Art der Aufgabe attribuieren [Die+18, S. 3].

Bei diesem Konzept spielen die personellen Ressourcen eine große Rolle, so dass bei der Umsetzung jedes der drei Kernelemente Lehrpersonal vonnöten ist. Mit einer steigenden Teilnehmerzahl in den Informatik-Studiengängen steigt damit auch der Bedarf an den personellen Ressourcen zur Umsetzung eines solchen Konzeptes, wodurch der Bedarf einer skalierbareren Lösung entsteht.

### Auswahl von Codierungsrichtlinien

Um die Skalierbarkeit des Konzepts zu verbessern, schlagen Dietz et al. vor, die Erfahrungen aus der Umsetzung des Konzepts in eine Wissensbasis zu überführen [Die+18] und auf Grundlage dieser eine automatisierte statische Codeanalyse durchzuführen. Die Autoren verfassten die Wissensbasis als Buch unter dem Titel „Java by Comparison: Become a Java Craftsman in 70 Examples.“ mit den am weitesten verbreiteten Problemen inklusive Codebeispielen und Erklärungen [HLD18].

Anhand der universitären Wissensbasis erstellten die Autoren ein Meta-Tool, welches in Summe etwas mehr als 100 Regeln der Codeanalyseframeworks Checkstyle, PMD und FindBugs/SpotBugs kombiniert und genutzt werden kann, um Codierungsrichtlinien, welche sich für einen universitären Kontext eignen, automatisiert in E-Learning-Systeme zu integrieren. Bei der Zusammenstellung der Richtlinien verfolgen Dietz et al. die Maxime, diese möglichst sinnvoll für alle Lerntypen auszuwählen und die Anzahl der falsch-positiven Detektionen gering zu halten. Bestimmte Programmierstile sollen darüber hinaus nicht forciert werden. Das entwickelte Meta-Tool ist unter dem Namen QualityReview auf GitHub<sup>1</sup> verfügbar und beinhaltet Regelsätze für die einzelnen statischen Analysetools.

## 3.4 Zusammenfassung

Die selbstständige Refaktorisierung von Code mit Hinblick auf die Verbesserung der Codequalität scheint in den praktischen Teilen der Informatik-Studiengänge oft unterrepräsentiert oder nicht hinreichend motiviert zu sein. Dietz et al. zeigten, wie „Clean Code“ als didaktisches Konzept in die Präsenzlehre integriert werden kann [Die+18]. Die mit steigenden Studierendenzahlen an Relevanz gewinnenden E-Learning-Umgebungen für Softwareentwicklung können koexistierend zur Präsenzlehre jedoch weitere Wege erschließen, Studierende zu motivieren, nicht nur funktionell einwandfreien, sondern auch hochqualitativen Code zu schreiben. Um dies skalierbar zu erreichen, können Studierende motiviert werden, ausgewählte Codierungsrichtlinien einzuhalten. Eine sorgfältige Auswahl von solchen Richtlinien, für einen universitären Kontext mit Studierenden aus frühen Semestern, entwickelten Dietz et al. über mehrere Jahre iterativ zu den Erfahrungen aus der Integration von

---

<sup>1</sup>QualityReview. <https://github.com/LinusDietz/QualityReview> (Abgerufen am 13.6.2020)

Codequality als didaktisches Konzept in die universitäre Lehre. Das entwickelte Regelwerk bietet eine gute Grundlage für eine im akademischen Kontext geeignete Detektion von Code Smells und damit auch eine gute Grundlage für die zu entwickelnde, hierauf basierende Gamification-Erweiterung.

Zur Motivation der Refaktorisierung als Mittel der qualitativen Verbesserung von Code im universitären Kontext wurden die Serious-Game-Konzepte CleanGame [San+19] und Serious Refactoring Games [HN19b] diskutiert. Händler und Neumann zeigen im Rahmen der Serious Refactoring Games, wie höhergestellte Lernziele nach Blooms Taxonomie bedient werden können. Dos Santos et al. zeigen, wie komplexe Lerninhalte über ein zweikomponentiges Quiz vermittelt werden können. Die hieraus extrahierbaren Teilkonzepte können für die zu erstellende Gamification-Erweiterung verwendet werden, um höherwertige Lernziele abzudecken. Beide Arbeiten diskutieren hierzu ein eigenes Game-Design, welches jedoch nur in der Arbeit zu CleanGame durch dos Santos et al. experimentell evaluiert wird. Dos Santos et al. schildern, dass es weiterer Forschung bedarf, um konfidente Aussagen über die Wirksamkeit von Gamification im universitären Kontext als Motivator zur Verbesserung der Codequality treffen zu können.

Sowohl dos Santos et al. als auch Händler und Neumann und Dietz et al. zeigen Ansätze, wie Studierenden ein besseres Verständnis von Codequality vermittelt werden kann und wie sie motiviert werden können, dieses Verständnis im Rahmen der Refaktorisierung praktisch anzuwenden. Dos Santos et al. sowie Händler und Neumann fokussierten sich konzentrisch hierauf, wobei deren Arbeiten hierfür ein eigenständiges Serious-Game-Konzept entwickelten. Die vorgestellten Serious Games können, ähnlich zu INLOOP, als fakultatives Angebot neben einer Lehrveranstaltung für die Motivation der Refaktorisierung von Code in Form von Einzelanwendungen eingesetzt werden. Im Unterschied zu den Serious Games jedoch werden in INLOOP darüber hinaus gehende Lehrinhalte vermittelt, zum Beispiel die Interpretation von UML-Modellen. Die in den folgenden Kapiteln zu konzipierende Gamification-Erweiterung unterscheidet sich somit vor allem in dem nachfolgenden Punkt von den vorgestellten Arbeiten. Der primäre Fokus auf dem Erlernen von objektorientierten Konzepten anhand der Programmiersprache Java soll in INLOOP nicht durch das Erlernen von Codequality-Richtlinien verdrängt werden. Vielmehr muss sich die zu konzipierende Gamification-Erweiterung nahtlos in das Gesamtsystem von INLOOP integrieren und dieses subtil genug ergänzen, um im Lernprozess nicht als hinderlich wahrgenommen zu werden. Die von Händler und Neumann im Rahmen von [HNS19], [HN19b] und schließlich [HN19a] systematisch vorgestellten Konzepte dienen dennoch zusammen mit der Arbeit von dos Santos et al. [San+19] als gute Grundlage für die Analyse, Konzeption und Implementation der Gamification-Erweiterung in den folgenden Kapiteln.

# 4 Anforderungsanalyse

In diesem Kapitel sollen konkrete Anforderungen an die zu konzipierende Erweiterung ermittelt werden. Hierzu werden zunächst die generellen Rahmenbedingungen beschrieben und anschließend eine konkrete Spezifikation der Anforderungen vorgenommen.

## 4.1 Rahmenbedingungen

Da die Konzeption der Erweiterung von INLOOP abhängig ist, soll nachfolgend zunächst dessen Iststand beschrieben, eine Zielgruppenbeschreibung ermittelt sowie konkrete Anwendungsfälle gezeigt werden.

### 4.1.1 Istzustand von INLOOP

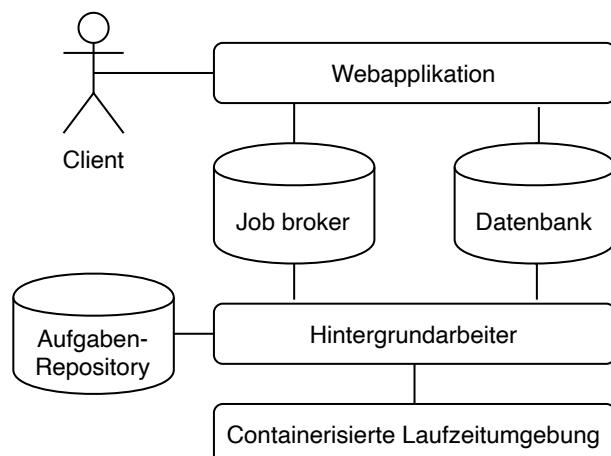


Abbildung 4.1 Architektur-Metamodell von INLOOP. Entnommen aus [MD18].

INLOOP ist eine auf Python<sup>1</sup>, Git<sup>2</sup>, dem Django Framework<sup>3</sup> und Docker<sup>4</sup> basierende Webanwendung. In Abbildung 4.1 ist die äußere Struktur der Anwendung gezeigt. Der Client kann direkt auf die Webanwendung zugreifen, wobei diese mit weiteren Komponenten zur Realisation der einzelnen

<sup>1</sup>Python. <https://www.python.org/> (Abgerufen am 15.9.2020)

<sup>2</sup>Git. <https://git-scm.com/> (Abgerufen am 15.9.2020)

<sup>3</sup>Django. <https://www.djangoproject.com/> (Abgerufen am 15.9.2020)

<sup>4</sup>Docker. <https://www.docker.com/> (Abgerufen am 15.9.2020)

Funktionalitäten kommuniziert. Zur persistenten Datenspeicherung wird eine Datenbank verwendet. Um asynchrone Prozesse zu planen und auszuführen, wird ein Job Broker verwendet, welcher Hintergrundarbeiter initiiert. Zu diesen asynchronen Aufgaben gehören beispielsweise die kontinuierliche Integration von Aufgaben über ein externes Aufgaben-Repository, aber auch die Steuerung des Aufgaben-Testprozesses in dedizierten containerisierten Laufzeitumgebungen.

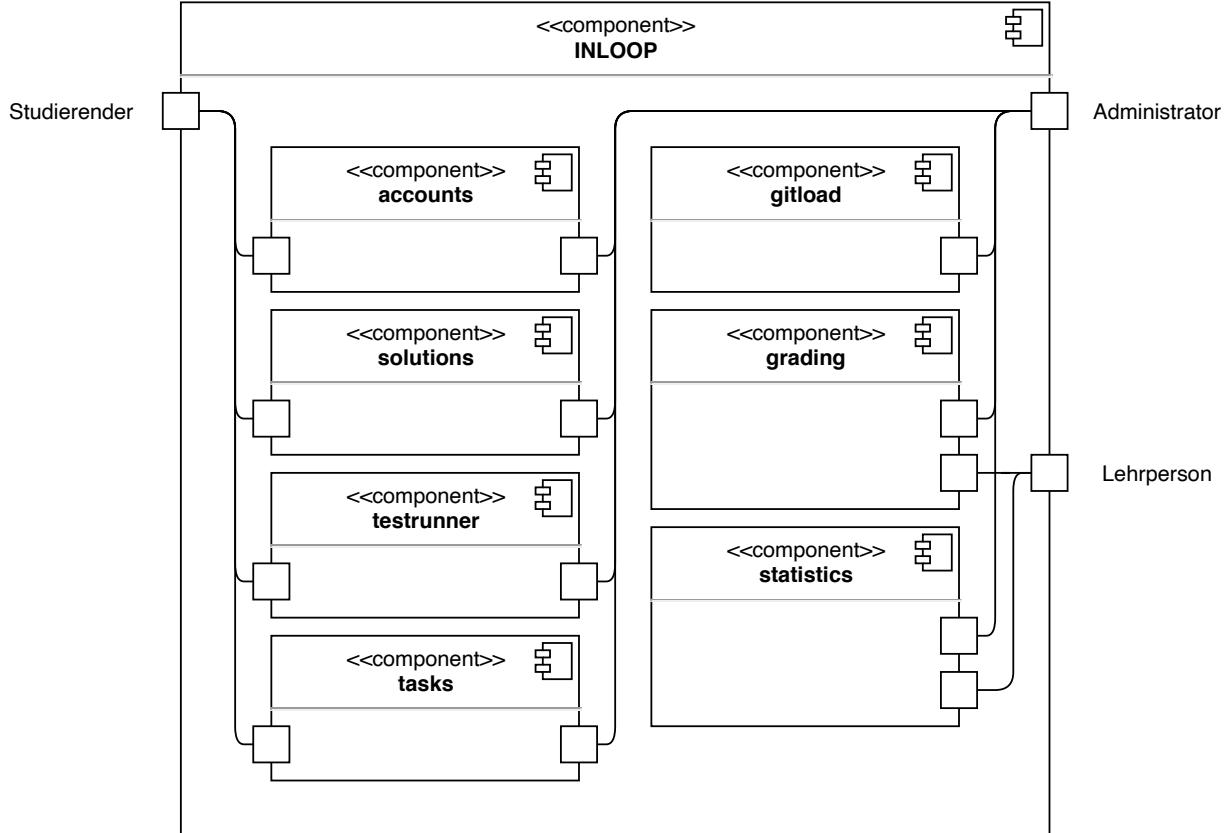


Abbildung 4.2 UML-Kompositionssstrukturdiagramm: Top-Level-Architektur der Webanwendung INLOOP

Das Kompositionssstrukturdiagramm in Abbildung 4.2 zeigt die Komponenten der Webanwendung, wobei vor allem drei Akteure auf die einzelnen Teilkomponenten zugreifen können. Die Komponente „accounts“ ist für das grundlegenden Account-Management zuständig, sodass ein rollenbasierter Zugriff auf andere Teilkomponenten gewährleistet werden kann. Über die Komponente „gitload“ kann ein Administrator die unter der Komponente „tasks“ gehandhabten Aufgabenstellungen in das System dynamisch integrieren. Lösungen dieser Aufgaben werden in der „solutions“ Komponente behandelt und durch die „testrunner“ Komponente getestet. Als Metaanalyssetool steht eine „statistics“ Komponente zur Verfügung. Für die Plagiatsprüfung der Lösungen und die Evaluation der Bonuspunkte ist schließlich die „grading“ Komponente zuständig. Die Teilkomponenten sind teilweise untereinander gekoppelt und greifen auf gemeinsame Ressourcen zu.

#### 4.1.2 Zielgruppe

Um ein besseres Verständnis für die Interessen, Probleme, Fähigkeiten und Motivationen der Nutzer zu erhalten, soll in der folgenden Sektion auf die Zielgruppe von INLOOP näher eingegangen werden. Die primäre Zielgruppe von INLOOP setzt sich zusammen aus Studierenden verschiedener Studiengänge. Die Studierenden in der Lehrveranstaltung Softwaretechnologie kommen in der Regel aus den Studiengängen der Informatik (Bachelor Informatik, Bachelor Medieninformatik, Diplom Informatik), wo die Lehrveranstaltung regulär im zweiten Semester des jeweiligen Studienablaufplans eingegliedert

ist. Einige Studierende kommen jedoch auch aus anderen fachfremden Studiengängen im Rahmen eines Nebenfachs oder aus der Schüleruniversität. Die Studierenden besitzen daher unterschiedliche Vorkenntnisse und Sichtweisen bezüglich der Softwareentwicklung.

**Fähigkeiten und Probleme.** Da es sich bei der Plattform um ein fakultatives Ergänzungsangebot zum Lernen von Vorlesungsinhalten handelt, ist den Studierenden auch weitestgehend überlassen, zu welchem Zeitpunkt sie die INLOOP-Aufgaben bearbeiten. Somit kann nicht pauschal davon ausgegangen werden, dass Konzepte aus der Vorlesung zu einem bestimmten Zeitpunkt schon verinnerlicht wurden, auch wenn die Vorstellung in der dazugehörigen Übung oder Vorlesung schon weit zurückliegt. Es stellt sich daher als ein zentrales Problem heraus, die Lerninhalte möglichst angepasst an die unterschiedlichen Lerntypen und Wissensstände anzupassen. Es kommt vor, dass Studierende mit den Aufgaben oder der Fehlersuche überfordert sind und sich hierdurch Frust entwickelt. Die in Abschnitt 2.1 genannte Unterteilung der Aufgaben nach Schwierigkeitsgrad in Kategorien gibt den Studierenden daher eine metaphorische Leiter, an deren Stufen sie sich Schritt für Schritt zu den komplexesten Aufgaben emporarbeiten können. Analog hierzu sollte schließlich auch bei der Gamification-Erweiterung berücksichtigt werden, dass den Studierenden während der Verwendung zum Teil grundlegende Kompetenzen aus der Softwareentwicklung fehlen. Die auf der grafischen Nutzeroberfläche dargestellten Informationen sollten demnach eine angemessene Komplexität besitzen oder sich hierbei gegebenenfalls graduell an die Fähigkeiten des Nutzers anpassen.

**Interessen und Motivationen.** Es obliegt der Entscheidung der Studierenden, ob diese das fakultative Angebot INLOOP nutzen möchten. Die bei der erfolgreichen Lösung der Exam-Aufgaben erreichbaren Bonuspunkte für die Klausur spielen hierbei eine nicht unwesentliche Rolle als eine der zentralen Motivationen. Durch die Bonuspunkte kann die Modulnote bei Bestehen der Klausur signifikant verbessert werden. Außerdem ermöglicht es INLOOP den Studierenden, die im fortgeschrittenen Verlauf der Übung vorgestellten Konzepte, beispielsweise Entwurfsmuster oder UML-Entwurfsdiagramme, nochmals auf die Implementierungsebene zu transkribieren und hierbei iterativ zu lernen. Mit einer ersten Lösung für eine Aufgabe mag der eingereichte Code noch keine der Tests bestehen, aber durch das direkte interaktive Feedback kann der Code sukzessiv modifiziert werden, bis dieser schließlich zum vollständigen Erfolg der Unitests führt. Auf dem Weg hierhin kann der Studierende viele kleine Erfolgsergebnisse erhalten, durch kleine Fortschritte in der Lösungsfindung. Die hierbei oft notwendige Fehlersuche in den Lösungen kann frustrieren, aber durch den Rätselcharakter auch Spaß bereiten und unterstützt in diesem Fall nochmals den Lernprozess. Schließlich ist noch eine weitere Motivation zu beschreiben, welche in der Lösung der Aufgaben liegt, denn hierdurch kann sich der Studierende außerdem selbst kontrollieren und mehr Sicherheit über die eigenen Kompetenzen erhalten.

#### 4.1.3 Anwendungsfälle und Akteure

In der folgenden Sektion sollen eine Reihe von Anwendungsfällen konzipiert werden, welche durch die Gamification-Erweiterung erfüllt werden sollen. Diese Anwendungsfälle sollen in den nachfolgenden Kapiteln weiter diskutiert, bei der Konzeption und der Implementation umgesetzt und hierbei weiter verfeinert werden.

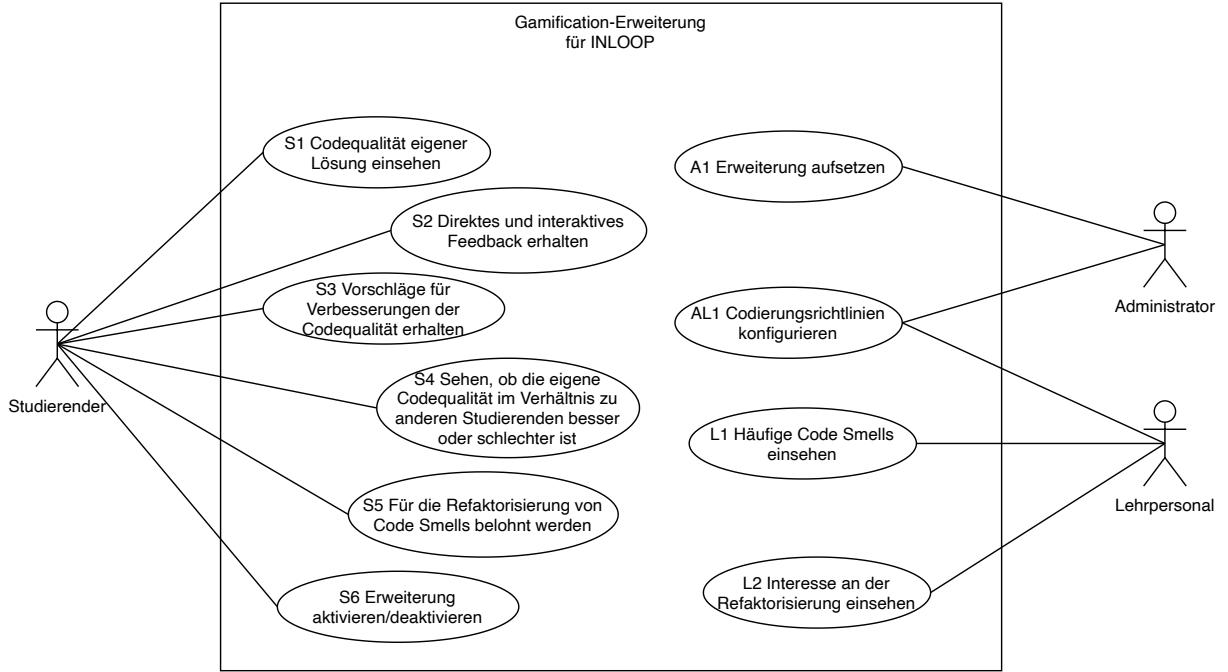


Abbildung 4.3 UML-Use-Case-Diagramm: Anwendungsfälle einer Gamification-Erweiterung für INLOOP in Relation zu den jeweils beteiligten Akteuren.

In Abbildung 4.3 ist die bidirektionale Zuordnung zwischen Akteuren und deren Anwendungsfällen zu sehen. Als primäre Akteure wurden der Studierende, der Administrator und das Lehrpersonal aus der Kontextanalyse übernommen. Der Studierende soll flexibel entscheiden können, ob er die Gamification-Erweiterung aktivieren oder deaktivieren möchte (S6). Aktiviert ein Studierender die Gamification-Erweiterung, so soll er die Codequality seiner eigenen Lösung einsehen können (S1), wobei ein direktes und interaktives Feedback gegeben werden soll (S2). Dieses Feedback soll Vorschläge für die Verbesserung der Codequality enthalten (S3). Außerdem soll der Studierende einsehen können, ob die eigene Codequality im Verhältnis zu anderen Studierenden besser oder schlechter ist (S4). Als zentraler Anwendungsfall des Studierenden soll dieser Spaß und Interesse an der Refaktorisierung entdecken können, indem sie für die Refaktorisierung von Code Smells belohnt werden (S5). Aus der Sicht des Administrators soll die Gamification-Erweiterung schnell und bequem aufsetzbar sein (A1). Zusammen mit dem Lehrpersonal teilt sich der Administrator den Anwendungsfall, dass die dem Bewertungsprozess zugrundeliegende Codierungsrichtlinien konfigurierbar sein sollen (AL1). Schließlich besitzt das Lehrpersonal zwei isolierte Anwendungsfälle, genauer die Möglichkeit, häufig auftretende Code Smells (L1) sowie das generelle Interesse an der Refaktorisierung und der Nutzung der Erweiterung (L2) einzusehen. Hieraus soll das Lehrpersonal Rückschlüsse über die Güte der Konfiguration (AL1) und die Nutzung der Erweiterung (S6) schließen können.

**Einordnung.** Insgesamt wird durch das Anwendungsfalldiagramm sichtbar, dass vor allem der Studierende als Hauptnutzer der Anwendung Nutzen an der Erweiterung finden soll. Dies reflektiert den Grundgedanken, dass die Gamification-Erweiterung als Hauptziel verfolgt, den Studierenden zur Verbesserung der Codequality zu motivieren. Die Anwendungsfälle A1, AL1, L1, L2 nehmen hierbei nur eine periphere Rolle ein. Durch die Anwendungsfälle AL1, L1 und L2 wird zusätzlich zu den Anwendungsfällen des Studierenden das Bedürfnis des Lehrpersonals, die Interaktion der Studierenden mit der Erweiterung zu beobachten und zu beeinflussen, berücksichtigt.

## 4.2 Spezifikation

Um die Anforderungen an die Gamification-Erweiterung zu konkretisieren, werden anhand den gezeigten Anwendungsfällen nachfolgend genaue funktionale Anforderungen aufgelistet und konkrete Qualitätsanforderungen beschrieben.

### 4.2.1 Funktionale Anforderungen

In der nachfolgenden Sektion sollen konkrete funktionale Anforderungen organisiert werden, also notwendige Funktionalitäten, welche durch die Gamification-Erweiterung unterstützt werden sollen. Insbesondere die abstrakten Anwendungsfälle und Akteure aus Abschnitt 4.1.3 sollen hierzu als kontextbezogene Grundlage dienen.

1. Studierende sollen die Möglichkeit haben, die Codequalität der eigenen Lösungen automatisiert auf Grundlage eines Regelwerks für Codierungsrichtlinien analysieren zu lassen.
2. Durch die Analyse von Lösungen soll es für Studierende möglich sein, detektierte Code Smells in den eigenen Lösungen zu erkennen.
3. Die Detektionen sollen für Studierende optisch ansprechend und möglichst leicht verständlich aufbereitet werden.
4. Für die Initialisierung des automatisierten Analyseprozesses und die Auflistung der Detektionen muss die Erweiterung eigene interaktive Komponenten in der grafischen Oberfläche der Gesamtanwendung integrieren und bereitstellen.
5. Die integrierten und bereitgestellten visuellen Komponenten müssen ein direktes und interaktives Feedback ermöglichen.
6. Ein Studierender soll die Möglichkeit haben, die eigene Codequalität mit anderen Studierenden zu vergleichen, sodass die Kompetitivität der Refaktorisierung gefördert wird.
7. Um den Spaß und das Interesse an der Refaktorisierung zu vermitteln, soll der Prozess der Refaktorisierung im Rahmen eines eigenen Game-Designs erlebbar gemacht werden.
8. Dem Studierenden soll es freistehen, ob er die Gamification-Erweiterung nutzen möchte oder nicht.
9. Die Gamification-Erweiterung soll eine hinreichende architekturelle und visuelle Trennung von der Hauptanwendung realisieren, so dass der Aspekt des eigentlichen E-Assessments nicht in den Hintergrund rückt.
10. Die Detektionen der Code Smells sollen mit durch den Studierenden interpretierbaren Lösungsvorschlägen für die Behebung des Code Smells ausgestattet sein und diese auch visuell repräsentieren.
11. Hierzu soll die Erweiterung Beschreibungen zu den im Regelwerk festgesetzten Codierungsrichtlinien bereitstellen.
12. Die der automatisierten Codequalitätsanalyse zugrundeliegenden Regeln sollen, wenn dies auf die jeweilige Regel applizierbar ist, durch einen Administrator oder Lehrpersonal über die Bereitstellung von Konfigurationsdateien konfigurierbar, aktivierbar und deaktivierbar sein.
13. Dem Lehrpersonal soll es ermöglicht werden, eine Übersicht in Form einer Statistik oder in Form einer ähnlichen visuellen Darstellung zu erhalten, welche es ermöglicht, häufig auftretende Code Smells einzusehen.

14. Über eine ähnliche Ansicht soll das Lehrpersonal anhand einer Statistik oder einer ähnlichen visuellen Darstellung einsehen können, wie groß das Interesse der Studierenden an der Refaktorisierung ist.
15. Der Administrator soll die Gamification-Erweiterung möglichst ohne zusätzlichen Aufwand zusammen mit der bestehenden Basisanwendung aufsetzen können.

#### 4.2.2 Qualitätsanforderungen

Analog zu den in Abbildung 2.3 gezeigten Modellparametern sollen auch für die zu entwickelnde Gamification-Erweiterung bestimmte Qualitätsanforderungen spezifiziert werden. Hierzu werden die übergeordneten Qualitätsattribute zunächst priorisiert.

Tabelle 4.1 Qualitätsanforderungen an die Gamification-Erweiterung analog zu den in [eng01] gezeigten Modellparametern von interner und externer Softwarequalität. Abkürzungen von links nach rechts: (+) normale Priorität, (++) hohe Priorität, (+++) sehr hohe Priorität, (++++) höchste Priorität.

Qualitätsattribut	Priorisierung			
	+	++	+++	++++
Funktionalität			x	
Verlässlichkeit	x			
Nutzbarkeit				x
Effizienz		x		
Wartbarkeit		x		
Portierbarkeit		x		

Wie in Tabelle 4.1 gezeigt, ist die Nutzbarkeit von höchster Priorität. Diese Festlegung beruht auf der Annahme, dass die Erfahrungen des Nutzers, wie in Abschnitt 2.2.4 und in Abschnitt 2.2.3 beschrieben, eine außerordentlich große Rolle spielen. Außerdem wurde für die Funktionalität eine sehr hohe Priorität angesetzt, worunter nicht nur die Qualität der Erfüllung der funktionalen Anforderungen zählt, sondern auch die Interoperabilität, welche eine größere Rolle bei der Integration in das Basissystem spielt sowie die Sicherheit der Erweiterung, welche insbesondere durch die Speicherung von personenbezogenen Daten und kopierrechtlich geschützten Source-Codes motiviert wird. Bei der Nutzbarkeit spielt auch die Effizienz eine Rolle, weil aus dem Zeitverhalten und der Skalierbarkeit der Gamification-Erweiterung direkte Auswirkungen auf die Interaktivität resultieren können. Ein denkbares Szenario, welches diesen qualitativen Effekt induziert, wäre, dass ein Studierender sehr lange auf die Auswertung der statischen Analyse des eigenen Codes warten muss. Dies würde sich unmittelbar auf die Nutzbarkeit auswirken. Da die Anwendung nicht für die Installation auf einem eingebetteten System mit stark beschränkten Hardwareressourcen gedacht ist, sondern auf einem Webserver mit verhältnismäßig großem Speicherplatz und ausreichender Rechenleistung laufen soll, wurde die Ressourcennutzung der Anwendung mit geringerer Priorität eingeschätzt. Daher wurde insgesamt die Priorität dieses Attributs Effizienz als hoch und nicht als sehr hoch eingestuft. Als weiteres Qualitätsattribut mit hoher Priorität wurde die Wartbarkeit selektiert, insbesondere durch die Relevanz der dahinter liegenden Teilateilattribute, wie die Änderbarkeit (betont durch Anwendungsfall AL1 in Abbildung 4.3) sowie die Analysierbarkeit (repräsentiert durch die Anwendungsfälle L1 und L2 in Abbildung 4.3). Ebenso wurde die Priorität der Portierbarkeit als hoch eingestuft, motiviert durch die Installierbarkeit (gezeigt durch Anwendungsfall A1 in Abbildung 4.3) und die kontextsensitive Adaptivität, wobei, wie in Abschnitt 4.1.2 bereits erwähnt, vor allem der persönliche Kontext und die Fähigkeiten des Nutzers relevant für die Adoptionsmechanismen der Anwendung sind. Weiterhin kontribuiert zur Portierbarkeit auch die Koexistenz der Erweiterung neben anderen Anwendungen, insbesondere INLOOP selbst, wobei dies unmittelbar mit der Interoperabilität zusammenhängt. Ein

indikatives Qualitätsszenario hierfür bestünde in der gemeinsamen Nutzung einer Datenbank, sodass die Annahmen und Prozesse der Basisanwendung nicht gestört werden. Die Erweiterung soll entsprechend durch ihre Softwarearchitektur soweit möglich abgekapselt werden. Abschließend, da die Reife der Gamification-Erweiterung im Rahmen der prototypischen Implementation eine untergeordnete Rolle spielt und das durch einen Ausfall des Systems abschätzbare Risiko wegen der dynamischen Ausschaltbarkeit (illustriert durch Anwendungsfall S6 in Abbildung 4.3) der Funktionalitäten gering ist, wird der Verlässlichkeit eine normale Priorität zugewiesen.

## 4.3 Zusammenfassung

Im Rahmen einer kurzen Analyse des Istzustands von INLOOP wurden die relevanten Komponenten von INLOOP gezeigt, an welche die Gamification-Erweiterung nachfolgend gekoppelt werden soll. Um außerdem hierauf ein nutzerorientiertes Gamification-Konzept zu erstellen, wurde der Studierende als Hauptnutzer der Anwendung anhand dessen Fähigkeiten, Problemen, Interessen und Motivationen analysiert. Auf Grundlage einer Anwendungsfallanalyse, welche zusätzlich den Administrator und das weitere Lehrpersonal als Akteure inkludiert, wurden konkrete funktionale Anforderungen und Qualitätsanforderungen an die Gamification-Erweiterung spezifiziert.

# 5 Konzept

In diesem Kapitel soll ein Konzept erstellt werden, welches der Anforderungsspezifikation aus Kapitel 4 gerecht wird und Nutzer motiviert, die Codequalität eingereichter Lösungen zu verbessern. Dazu wird nachfolgend zunächst betrachtet, wie die statische Codeanalyse architekturell in INLOOP eingegliedert werden kann, um als Grundlage für die Detektion und Kommunikation von Code Smells zu dienen. Danach wird ein Game-Design entwickelt, welches Gamification-Elemente gezielt auswählt und auf den Zielkontext anwendet, um zur Verbesserung der Codequalität zu motivieren. Auf Grundlage dieses Game-Designs wird anschließend diskutiert, welche strukturellen und funktionellen Änderungen in INLOOP für die Umsetzung eines solchen Game-Designs notwendig sind und schließlich eine konkrete Modellstruktur konzipiert, auf deren Basis die Umsetzung der Gamification-Erweiterung durchgeführt werden kann.

## 5.1 Codeanalyse

### 5.1.1 Regelwerk für Codierungsrichtlinien

Grundlage der Codeanalyse soll das in Abschnitt 3.3 beschriebene, im akademischen Kontext entwickelte QualityReview Framework von Dietz et al. [Die+18] sein. Im Folgenden soll beschrieben werden, wie das Framework für diese Arbeit wiederverwendet werden kann.

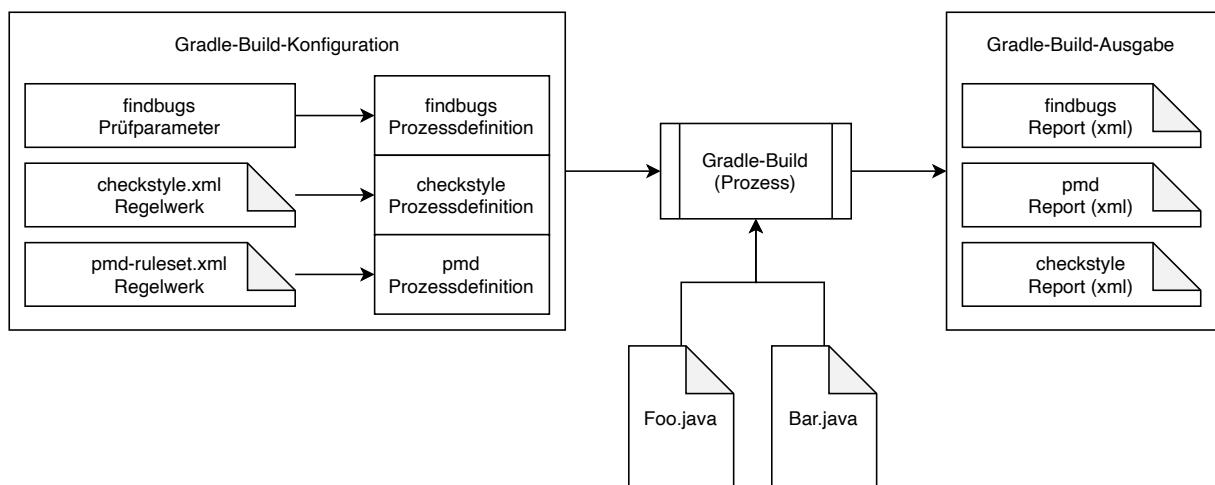


Abbildung 5.1 Architektur des QualityReview Frameworks von Dietz et al. [Die+18].

In Abbildung 5.1 ist die Architektur des Analyseframeworks gezeigt. Um eine beliebige Java-Codebasis

(in Abbildung 5.1 illustriert durch Foo.java und Bar.java) mithilfe des Frameworks zu analysieren, werden die darin enthaltenen Java-Dateien in einen Gradle<sup>1</sup> Buildprozess gegeben. Für jedes der drei unterstützten Codeanalysetools (PMD, FindBugs/SpotBugs, Checkstyle) wird hierbei eine Spezifikation der Eingaben und der Ausgaben vorgenommen. Die eingegebenen Java-Dateien werden entsprechend der Spezifikation vom jeweiligen Codeanalysetool analysiert und das Analyseergebnis schließlich in einer standardisierten Form wie zum Beispiel XML ausgegeben.

Teil der Spezifikation ist das Regelwerk, nach dem die Codeanalysetools Code Smells detektieren. Das EingabefORMAT des Regelwerks unterscheidet sich hierbei je nach Codeanalysetool (siehe Abbildung 5.1). Checkstyle und PMD werden mit einem eigenen XML-Regelwerk konfiguriert, während FindBugs/SpotBugs lediglich mit Prüfparametern wie dem „reportLevel“ konfiguriert wird. Diese Konfigurationen sind das Kernelement des Frameworks und sollen im Folgenden so adaptiert werden, dass diese in INLOOP integriert werden können und die in Abschnitt 4.2 erläuterten Anforderungen erfüllen.

### 5.1.2 Integrationskonzept der statischen Codeanalyse

Die Prüfung auf funktionelle Korrektheit von Lösungen in INLOOP über Unit Tests wird durch eine besondere Komponente durchgeführt, welche sich *TestRunner* nennt. Wird eine Lösung in INLOOP eingereicht, so initialisiert dieser TestRunner eine virtualisierte Laufzeitumgebung, in der die Dateien der Lösung dynamisch durch jUnit<sup>2</sup> Tests getestet werden. Hierzu wird innerhalb eines individuellen Docker Containers ein Ant<sup>3</sup> Buildprozess durchgeführt, welcher die hierzu notwendigen Schritte und die zu inkludierenden Dateien spezifiziert. Die Ant Buildkonfiguration wird hierbei zusammen mit der Aufgabe über Continuous Publishing [MD18] dynamisch von einem externen Git-Repository bezogen. Die Ausgabe der jUnit Testsuite wird wiederum im XML-Format gespeichert und schließlich vom TestRunner geparsst, so dass die in der virtualisierten Laufzeitumgebung temporär abgelegten Ausgabedateien schließlich in der Datenbank von INLOOP persistiert werden.

Da sowohl die dynamische Codeanalyse, als auch die statische Codeanalyse relativ (im Vergleich zur Bereitstellung anderer Funktionalitäten in INLOOP) rechenintensiv sind, sollten diese nicht blockierend ausgeführt werden. Genauer sollen diese Prozesse vom normalen Request-Response-Prozess der Webanwendung entkoppelt werden. Ein weiteres Problem ist die Ausführung von hochgeladenem, potenziell schadhaftem Code. Der TestRunner bietet für beide Probleme eine hinreichend gute Lösung durch die Virtualisierung des Betriebssystems über Docker Container und die entkoppelte Ausführung dieser über eigene Service-Worker-Prozesse. Daher soll im Folgenden gezeigt werden, wie die statische Codeanalyse zusätzlich zur dynamischen Codeanalyse im TestRunner konzeptuell integriert werden kann, um schließlich zu zeigen, wie hierbei das QualityReview Framework eingesetzt werden kann.

---

<sup>1</sup>Gradle. <https://gradle.org/> (Abgerufen am 11.8.2020)

<sup>2</sup>jUnit 5. <https://junit.org/junit5/> (Abgerufen am 11.8.2020)

<sup>3</sup>Ant. <https://ant.apache.org/> (Abgerufen am 11.8.2020)

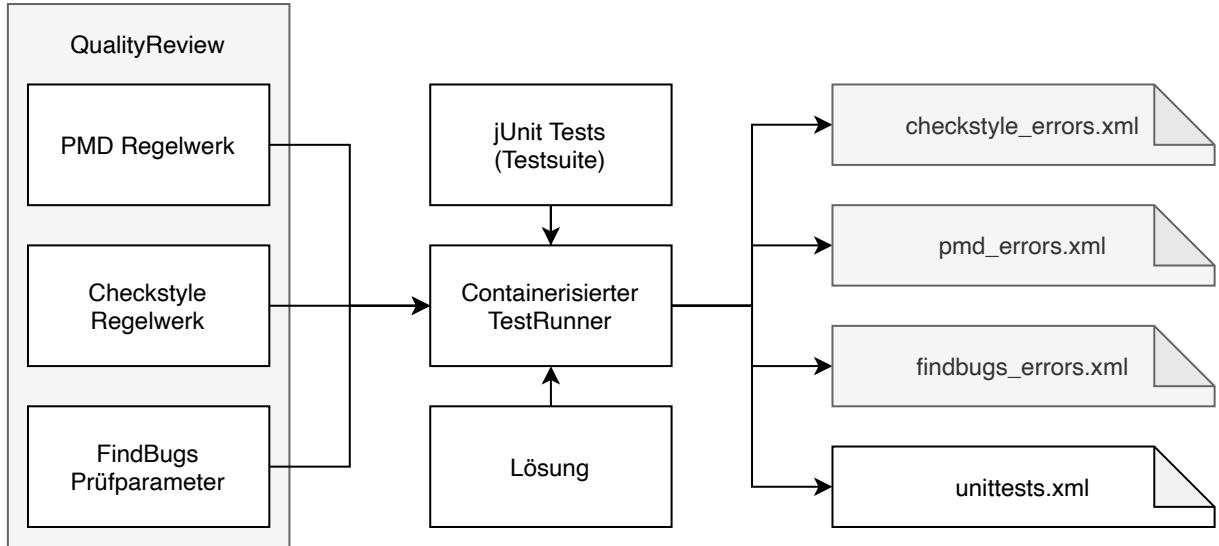


Abbildung 5.2 Integrationskonzept des QualityReview Frameworks für die TestRunner Komponente von INLOOP

Das in Abbildung 5.2 gezeigte Integrationskonzept ermöglicht die nahtlose Ergänzung der statischen Codeanalyse auf Grundlage des QualityReview Frameworks. Im Zentrum steht die bereits beschriebene TestRunner Komponente. Der durch diese gesteuerte Ant Buildprozess wird durch die Ausführung der Codeanalysetools PMD, Checkstyle und FindBugs/SpotBugs erweitert, wobei die Konfigurationen aus dem QualityReview Framework wiederverwendet werden. Die Spezifikation des Gradle Builds aus QualityReview wird hierzu auf eine Ant Spezifikation überführt. Nach der Ausführung des Ant Buildprozesses stehen die Ausgabedateien der statischen Codeanalysetools analog zum Gradle Buildprozess in standardisierter Form (XML) auf dem Dateisystem des virtualisierten Docker Containers zur Verfügung und können von hieraus für die weitere Verarbeitung verwendet werden.

### 5.1.3 Semantische Code-Smell-Extraktion

Mithilfe der im vorigen Abschnitt vorgestellten Architektur können Lösungen in INLOOP auf Verletzungen von Codierungsrichtlinien (Detektionen) statisch geprüft werden. Die Detektionen der Codeanalysetools sind dabei mehr oder weniger deskriptiv, beispielsweise beinhalten die Detektionen aus dem Checkstyle Tool (in der Version 7.6.1 aus dem QualityReview Framework) oder dem PMD Tool meist nur eine sehr kurze, prägnante Beschreibung des eigentlichen Fehlers und nie eine Beschreibung der zugrundeliegenden Maximen. Ein Studierender soll später jedoch nicht nur eine sehr kurze und potenziell missverständliche Meldung über die Detektion sehen (analog zur Anforderungsbeschreibung aus Abschnitt 4.2), sondern eine Erklärung erhalten, weshalb die zugrundeliegende, verletzte Codierungsrichtlinie möglicherweise eine Degradation der Codequalität zur Folge hat. Zur Illustration dessen soll folgender Java-Codeabschnitt aus einer hypothetischen Datei „Example.java“ dienen.

---

```

49 // ...
50 public void do_something() {
51     if (conditionIsTrue)
52         callFunction();
53 }
54 // ...

```

---

Die Analyse durch Checkstyle mit der Checkstyle-Konfiguration aus dem QualityReview Framework detektiert hierbei unter anderem einen Verstoß gegen Codierungsrichtlinien in Zeile 51 und gibt die folgende XML-Datei aus.

---

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <checkstyle version="7.6.1">
3     <file name="/checker/input/Example.java">
4         <error line="51"
5             severity="error"
6             message="if construct must use {}s."
7             source="[URI].NeedBracesCheck"/>
8     </file>
9 </checkstyle>
```

---

Bei der Analyse durch PMD wiederum wird unter anderem in Zeile 50 ein Verstoß gegen die Benennungskonventionen detektiert.

---

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <pmd xmlns="http://pmd.sourceforge.net/report/2.0.0"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="[URL]/report_2_0_0.xsd"
5     version="6.2.0" timestamp="2020-08-12T09:23:06.378">
6     <file name="Example.java">
7         <violation beginline="51" endline="51"
8             begincolumn="12" endcolumn="24"
9             rule="MethodNamingConventions" ruleset="Code Style"
10            class="Example" method="do_something"
11            externalInfoUrl="[URL]#methodnamingconventions"
12            priority="1">
13             Method names should not contain underscores
14         </violation>
15         ...
16     </file>
17 </pmd>
```

---

Zu sehen ist, dass die Nachrichten der Detektionen zwar jeweils eine kurze Beschreibung des Code Smells geben, jedoch nicht, aus welchem Grund dies die Codequalität potenziell mindert. Im illustrierten Beispiel müssten bei einer Ergänzung einer Instruktion in Zeile 53 geschweifte Klammern ergänzt werden, um diese Instruktion zusammen mit der Instruktion in 52 konditionell auszuführen. Wird dies vergessen, könnte dies zu Fehlern aufgrund einer nichtkonditionellen Ausführung der Instruktion kommen. Außerdem sollten Methoden in „lowerCamelCase“ statt, wie in Zeile 50 in „snake\_case“, geschrieben werden, um zu den Konventionen der Programmiersprache konsistent zu bleiben und die Lesbarkeit zu verbessern. Solche semantische Beschreibungen der Detektionen soll der Studierende erhalten, um einerseits die Verständlichkeit zu verbessern, aber auch, um die zugrundeliegenden Maximen von „Clean Code“ zu lehren. Dies dient als Mensch-Computer-Schnittstelle für die weitere Kommunikation der Code Smells, um schließlich die Codequalität eingereichter Lösungen nachhaltig verbessern zu können.

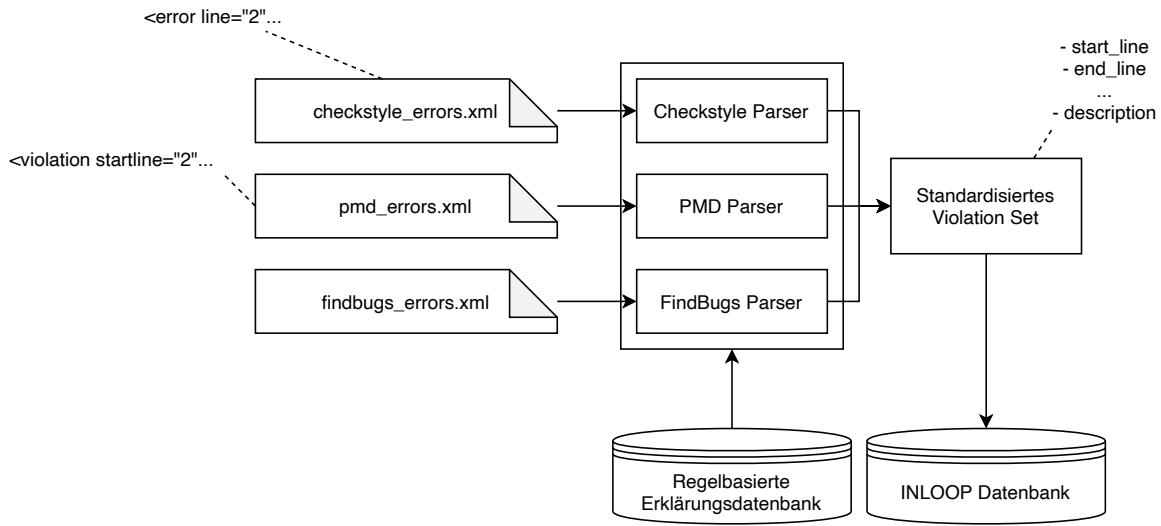


Abbildung 5.3 Eine Architektur zur vereinheitlichten, semantischen Code-Smell-Extraktion auf Grundlage der Ausgaben von Codeanalysetools.

Die Codeanalysetools PMD, Checkstyle und FindBugs/SpotBugs inkludieren in deren standardisierten Ausgaben jeweils pro Detektion einen Identifikator für die Art des ausgelösten Detektionsmechanismus oder der zugrundeliegenden Codierungsrichtlinie. Auf Grundlage dessen ermöglicht die in Abbildung 5.3 gezeigte Architektur die Erweiterung der Detektionen um Erklärungen zu den Maximen. Da die Ausgabegrammatiken der Codeanalysetools stark voneinander abweichen, werden die detektierten Code Smells zunächst über einen jeweiligen, spezialisierten Parser in eine in der Datenbank von INLOOP persistierbare Form (ein „Set“ von „Violations“) überführt. Hierzu wird die Wichtigkeit der Detektion in die drei Kategorien „information“ für rein informative Detektionen, „warning“ für mittelwichtige Detektionen und „error“ für die wichtigsten Detektionen aufgeteilt. Die entsprechenden Parser bilden die Formate der Codeanalysetools hierauf ab. Beispielsweise werden die PMD-Prioritäten 1 und 2 auf die Wichtigkeit „error“, 3 und 4 auf „warning“ und 5 auf „information“ entsprechend zur Dokumentation der Prioritäten<sup>4</sup> abgebildet. Die Detektionen werden schließlich ausgestattet mit einer jeweiligen textuellen Erklärung aus einer regelbasierten Erklärungsdatenbank, welche im Folgenden näher beschrieben werden soll.

#### 5.1.4 Regelbasierte Erklärungsdatenbank

Um eine „Violation“, genauer die vereinheitlichte und persistierbare Abbildung der Codeanalysetool-Ausgaben, mit diesen textuellen Erklärungen zu ergänzen, werden die Identifikatoren (URIs) der Detektionen auf einen Identifikator der Erklärungsdatenbank und die dazugehörige textuelle Erklärung abgebildet.

<sup>4</sup>PMD. [https://pmd.github.io/latest/pmd\\_userdocs\\_extending\\_rule\\_guidelines.html](https://pmd.github.io/latest/pmd_userdocs_extending_rule_guidelines.html) (Abgerufen am 11.8.2020)

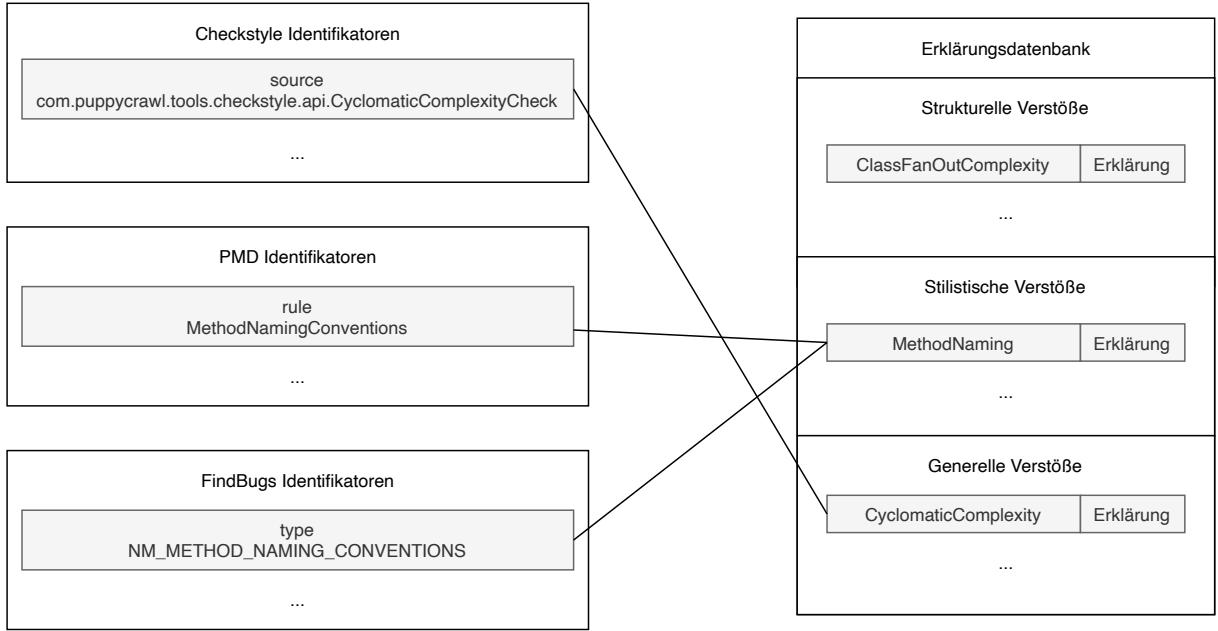


Abbildung 5.4 Beispielabbildung für die Adaption der Identifikatoren von Detektionen auf Codeanalysetools zur Ergänzung von Code-Smell-spezifischen Erklärungen.

Abbildung 5.4 zeigt die Adaption der Identifikatoren, um die Detektionen mit in der Datenbank hinterlegten Erklärungen zu ergänzen. Zu sehen ist, dass die Erklärungsdatenbank zwischen drei Kategorien unterscheidet, den strukturellen, stilistischen und generellen Verstößen. Prinzipiell können diese Kategorien bei der Umsetzung dieses Konzepts frei gewählt werden, beispielsweise nach einer anderen Unterteilung, zum Beispiel in „Bad Practice“, „Correctness“, „Vulnerability“ oder „Performance“, analog zu den Detektionskategorien in FindBugs<sup>5</sup>/SpotBugs<sup>6</sup>. Die in Abbildung 5.4 gezeigte Kategorisierung hängt zusammen mit der späteren Aufbereitung in Form eines Narratives, welches in Abschnitt 5.2.2 gezeigt wird. Außerdem frei wählbar sind die Identifikatoren der Erklärungen aus der Erklärungsdatenbank und damit die Granularität der Abbildung. Da die Abbildung zwischen den Identifikatoren der Codeanalysetools und den Identifikatoren der Erklärungen von der Parser-Infrastruktur (siehe Abbildung 5.3) realisiert wird, können gemeinsame Schnittmengen im Regelsatz gebildet werden, die jeweils auf dieselbe Erklärung abbilden. Illustriert wird dies exemplarisch in Abbildung 5.4 durch den PMD-Identifikator „MethodNamingConventions“ und den FindBugs-Identifikator „NM\_METHOD\_NAMING\_CONVENTIONS“, welche beide auf die Erklärung hinter dem Erklärungsdatenbank-Identifikator „MethodNaming“ abgebildet werden können, da den beiden Detektionsmechanismen dieselbe Maxime zugrunde liegt. Außerdem können in der Erklärungsdatenbank verhältnismäßig wenige oder viele Erklärungen hinterlegt sein, auf welche die Identifikatoren der Codeanalysetools abgebildet werden. Direkt abhängig davon ist jedoch die Spezifität der Erklärungen. Ein Nachteil des Konzeptes der Erklärungsdatenbank ist die Notwendigkeit der manuellen Erstellung der Abbildung in der Parser-Infrastruktur sowie die manuelle Erstellung der textuellen Erklärungen. Auf eine mögliche Strategie in der konkreten Implementation wird in Kapitel 6 eingegangen.

## 5.2 Game-Design

Auf Grundlage der in den vorigen Sektionen vorgestellten Konzepte zur Integration der statischen Codeanalyse in INLOOP kann nun ein Gamification-Konzept erstellt werden, welches dazu dienen soll, die semantischen Detektionen weiter aufzubereiten und den Nutzer dazu zu motivieren, die

<sup>5</sup> FindBugs. <http://findbugs.sourceforge.net/bugDescriptions.html> (Abgerufen am 12.8.2020)

<sup>6</sup> SpotBugs. <https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html> (Abgerufen am 14.9.2020)

Detektionen zu lesen, zu verstehen und schließlich selbst umzusetzen. Gemäß dem in Abschnitt 2.2.4 vorgestellten Octalysis-Ansatz soll dafür im Folgenden ein nutzerorientiertes Game-Design erstellt werden. Hierzu wird zunächst eruiert, welche Bedürfnisse und Ziele des Nutzers motiviert werden sollen (Leitmotive). Anschließend wird hierum ein Narrativ entwickelt, um hierzu schließlich gezielt Gamification-Elemente auszuwählen.

### 5.2.1 Leitmotive

Als wesentlicher Teil der funktionalen Anforderungen aus Abschnitt 4.2.1 soll der Nutzer Spaß und Interesse an der Durchführung von Refaktorisierungen am eigenen Code haben. Eine Frustration des Nutzers durch wiederholtes Scheitern soll möglichst vermieden werden und kleine Fortschritte belohnt werden, um Frust zu vermeiden. Rätsel können die Neugier des Nutzers wecken und das Interesse an der Refaktorisierung amplifizieren. Der Octalysis-Ansatz bietet hierbei eine gute Richtlinie für die Auswahl von Gamification-Elementen und die Einschätzung von deren Wirkungsweisen auf den Nutzer.

### 5.2.2 Narrativ

Beispielsweise können witzige und interessante Narrative dabei helfen, dass Nutzer sich noch mehr in die Thematik der Refaktorisierung vertiefen und dabei durch zwischenzeitliches Schmunzeln erfrischt werden. Ein solches Narrativ soll im Folgenden konzipiert werden. Wie in Abschnitt 2.2.2 beschrieben, eignen sich sinnvolle Narrative für den Einsatz als Gamification-Element, um die im Game-Design zusammengestellten Aktionen miteinander zu verknüpfen und ihnen dabei Tiefe und Relevanz zu verleihen. Im ausgewählten Narrativ der Gamification-Erweiterung sollen die auf INLOOP eingereichten Lösungen als „Patienten“ behandelt werden, wobei die Patienten krank oder gesund sein können und der Gesundheitszustand eines Patienten der Codequalität der dazugehörigen Lösung entspricht. Lösungen mit vielen detektierten schwerwiegenden Code Smells sind somit „todkranke“ Patienten, während Lösungen mit hoher Konformität zu den festen Codierungsrichtlinien „gesund“ sind. Der Nutzer nimmt hierbei die Rolle eines Arztes ein und versucht, seine eigenen Patienten (seine eingereichten Lösungen) von den Verletzungen der Codierungsrichtlinien zu heilen. Das Narrativ soll dabei aber keinesfalls leidvoll, sondern vor allem humoristisch orientiert sein, gestützt durch witzige Parallelen zur Realität.

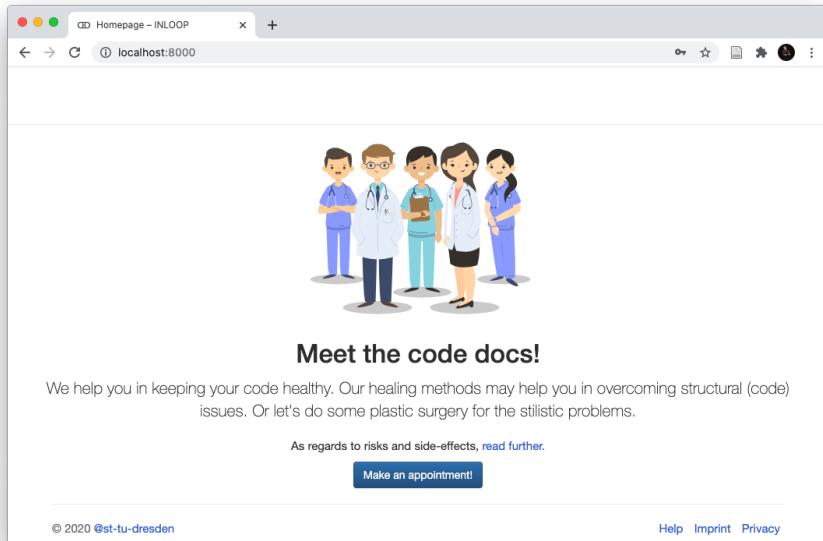


Abbildung 5.5 Ein Banner auf der Startseite, welches die „Code Doctors“ vorstellt. Ursprüngliche Gestaltung der Illustration von Freepik<sup>7</sup>.

Eine dieser Parallelen soll durch die „Code Doctors“, also „Code-Ärzte“ (nicht zu verwechseln mit Code-Dokumentation) geschaffen werden. Jeder Code Doc hat seine eigene, individuelle Profession. So gibt es einen allgemeinen Chirurgen, der sich zum Beispiel um gebrochene Knochen eines Patienten, also um die groben strukturellen Code Smells kümmert. Außerdem gibt es einen Schönheitschirurgen, welcher sich um die Schönheitsprobleme des Patienten, repräsentiert durch stilistische Code Smells, kümmert und einen Allgemeinmediziner, der verschiedene andere Probleme behandelt. Krankenschwestern helfen bei allem anderen, zum Beispiel beim Zurechtfinden in der Praxis, also die Erklärung von den einzelnen Gamification-Elementen, oder bei der Terminfindung, genauer der Initiierung einer „Untersuchung“ (Code-Smell-Analyse) durch den Arzt. Jeder Code Doc hat dabei seinen eigenen Namen und einen kurzen Steckbrief, um dem gesamten Narrativ Glaubwürdigkeit zu verleihen.

Nutzer haben nun auf Grundlage dessen die Möglichkeit, ihre Lösungen einer ärztlichen Untersuchung zu unterziehen. Hierbei sollen sie sich in ein Wartezimmer begeben können, um die anwesenden Patienten zu sehen, wobei dies die jeweils aktuellsten Lösungen zu einer jeden Aufgabe sind. Gab es bereits eine „Voruntersuchung“ zu einer vorherigen Lösung einer Aufgabe, so soll hierbei der Gesundheitszustand der vorherigen Variante angezeigt werden. Der Nutzer soll nun die Aufgabe übernehmen, den Lösungen zur „Genesung“ zu verhelfen, indem er sich den gesundheitlichen Problemen (Code Smells) annimmt.

### 5.2.3 Progressivität

Während der Nutzung von INLOOP soll Studierenden das Gefühl des Fortschrittes und der Verbesserung der eigenen Kompetenzen als psychologisches Grundbedürfnis (beschrieben in Abschnitt 2.2.3) vermittelt werden, um die intrinsische Motivation bei der Verbesserung der Codequalität zu amplifizieren. Hierzu sollen im Folgenden mögliche Gamification-Elemente vorgestellt werden, welche auf unterschiedliche Weise eine Progressivität vermitteln und verschiedene Faktoren aus dem nutzerorientierten Octalysis-Framework einbeziehen.

<sup>7</sup>Freepik. [https://www.freepik.com/free-vector/flat-nurse-team\\_4387619.htm](https://www.freepik.com/free-vector/flat-nurse-team_4387619.htm) (Abgerufen am 14.9.2020)

## Konsultationen

Zur Einsicht der Code Smells, die den Gesundheitszustand einer Lösung beeinträchtigen, kann der Nutzer in der Rolle eines beobachtenden (angehenden oder lernenden) Doktors im Wartezimmer einen Patienten in eine Konsultation „herein bitten“. In dieser Konsultation erhält der Nutzer eine Übersicht über den Code der Lösung und an welcher Stelle Code Smells detektiert wurden. Je nach der Art des jeweiligen Code Smells wird dieser nun dem Nutzer vom Code Doc übermittelt, der sich auf diesen (entsprechend zur Profession) spezialisiert hat. Dabei zeigt der Code Doc dem Nutzer, an welcher Stelle ein Code Smell detektiert wurde und wie er diesen beheben kann, narrativ repräsentiert durch ein „Rezept“.

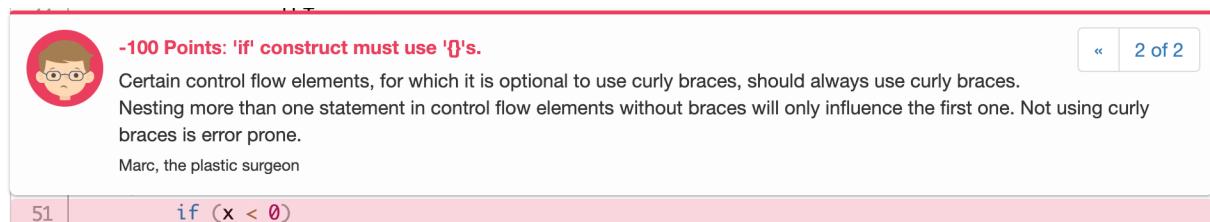


Abbildung 5.6 Nutzer erhalten Rezepte von spezialisierten „Code Doctors“ zur Verbesserung einer eingereichten Lösung.

Abbildung 5.6 zeigt, wie ein solches Rezept dem Nutzer präsentiert werden kann. Das Rezept nutzt die Nachricht aus der adaptierten Detektion des Codeanalysetools sowie die genaue Zeile und die Wichtigkeit der Detektion. Außerdem beinhaltet die Präsentation die aus der Erklärungsdatenbank bezogene Erklärung zu dieser Detektion. Da es sich hierbei um einen stilistischen Code Smell handelt, welcher als solcher in der Erklärungsdatenbank klassifiziert wird, ist der zuständige Code Doctor im Rahmen des Narratives der plastische Chirurg, da sich ein solcher vorrangig um das „Aussehen“ des Patienten kümmert. Grobe strukturelle Verstöße werden hingegen vom Chirurgen behandelt, alle weiteren generellen Verstöße vom Allgemeinmediziner.

**Code-Smell-Suche nach [San+19].** Durch das Aufzeigen des konkreten Fehlers erhält der Studierende die Möglichkeit, diesen direkt im Code zu beheben und damit seinen Patienten zu heilen. Die Konsultation kann jedoch bei Bedarf noch durch unterschiedliche Darstellungsstufen der Code Smells erweitert werden. Beispielsweise kann die Sichtbarkeit der Zeilennummer oder der konkreten Erklärung eingeschränkt werden. Hierdurch soll der Studierende, ähnlich zu dem in Abschnitt 3.1 vorgestellten Konzept, selbst herausfinden, an welcher Stelle eine Lösung Code Smells beinhaltet oder welche Art von Code Smell sich hinter einer konkreten Zeile verbirgt. Zusammen hiermit kann nach Octalysis der Gamification-Faktor „Unpredictability and Curiosity“ (siehe Abbildung 2.2) einbezogen werden. Um die Nutzererfahrung zu verstärken, kann so der Zufall auch visuell durch ein „Glücksrad“ symbolisiert werden, wobei von mehreren möglichen Darstellungsvarianten des Code Smells eine Konkrete zufällig ausgewählt wird. Für eine genauere Auflistung könnte der jeweilige Code Doctor nochmals aufgefordert werden, genauere Untersuchungen auszuführen, sodass dann die jeweiligen Zeilennummern oder der Typ des Code Smells gezeigt werden, beispielsweise durch den Einsatz von Punkten.

## Punkte

Um die Code Docs zu bitten, eine genauere Aufschlüsselung der Zeilennummern (durch weitere Untersuchungen) oder den konkreten Typ des Code Smells (abhängig vom Zufall) zu zeigen, muss der Nutzer hierfür Punkte ausgeben. Dies bedient die Gamification-Faktoren „Development and Accomplishment“ sowie „Ownership and Possession“ nach Octalysis. Die Punkte erhält ein Nutzer hierbei durch folgende Interaktionen:

- Durch das Einreichen von Lösungen, welche die Unit Tests der jeweiligen Aufgabe bestehen, abzüglich der Punkte, die als „Gesundheitskosten“ aufgrund der vorliegenden Code Smells je nach Wichtigkeit hiervon abgezogen werden
- Durch „Rückerstattung von der Krankenkasse“, wenn ein Rezept eingelöst wurde und der Code Smell eines Patienten beseitigt wurde
- Durch das Freischalten von Badges, die in Abschnitt 5.2.3 weiter beschrieben sind

Die Möglichkeiten, Punkte zu erreichen, müssen für den Nutzer hierbei klar ersichtlich sein, um Verwirrung zu vermeiden. Erreicht ein Nutzer Punkte, so kann dies über entsprechende Notifikationen in INLOOP realisiert werden.

## **Badges**

Badges sind eine Möglichkeit, Punkte zu erhalten und werden analog zum Game-Design bei bestimmten Handlungen vergeben, die den Zielen der Gamification dienen und den zugrundeliegenden Spielregeln (Gamefulness) folgen. Im Folgenden sind mögliche Handlungen aufgelistet, die beispielsweise durch Badges weiter motiviert werden könnten:

- Das Betrachten bestimmter Informationsansichten, wie zum Beispiel einer Informationsansicht zu den Code Doctors anhand den von diesen behandelten Code Smells
- Das Einreichen einer besonders guten Lösung, die keine oder nur wenige Code Smells beinhaltet
- Das erstmalige Ausprobieren einer Funktionalität des Game-Designs, wie zum Beispiel das erstmalige Starten einer Konsultation
- Die Nutzung von Möglichkeiten zur sozialen Interaktion mit anderen Studierenden

Im Zentrum steht hierbei vor allem die Belohnung der Auseinandersetzung mit besonderen Aktivitäten, die vom regulären Einreichen von Lösungen abweichen und auf die Kommunikation der Codierungsrichtlinien oder die generelle gegenseitige Motivation an der Bearbeitung von Aufgaben hinwirken. Gleichzeitig können die Aufgabenstellungen auch so vage formuliert sein, dass der Nutzer zum Rätseln animiert wird und herausfinden möchte, wie diese entsprechenden Badges erhalten werden können.

## **Level und Fortschrittsbalken**

Ein weiteres progressives Gamification-Element kann durch Level repräsentiert werden. Je mehr Punkte ein Nutzer hat, desto höher ist sein Level. Fortschrittsbalken können dieses Level repräsentieren und auf das nächsthöhere hinweisen, um ein Bedürfnis zu erzeugen, dieses durch weitere Verbesserungen der Codequalität zu erreichen. Dabei können die erreichbaren Level gut in das Narrativ integriert werden. Entsprechend der Rahmenhandlung kann der Nutzer als Studierender anfangen und sich über die Ernennung als Doktor über Karrierestufen vom Facharzt über den Oberarzt bis hin zum Chefarzt oder ärztlichen Direktor verbessern.

## **Progressive Leaderboards**

Level, Punkte und Badges sollen die Errungenschaften und den Fortschritt des Studierenden nach außen zeigen. In INLOOP ist es jedoch bisher nicht möglich, Informationen über andere Nutzer einzusehen.

Rank	Avatar	User	Points	Level
1 ± 0		admin	3000	2 - Specialist
2 ± 0		testuser   <a href="#">Add as colleague</a>	800	0 - Student

Abbildung 5.7 UI-Konzept eines progressiven Leaderboards in Form einer Tabelle, die Nutzer nach ihren erreichten Punktzahlen auflistet.

Über progressive Leaderboards wird der Fortschritt des Studierenden für jeden anderen Nutzer der Plattform sichtbar. In Ergänzung zum bereits in anderen Gamification-Konzepten umgesetzten, rangbasierten Leaderboard wird dieses um eine progressive Komponente ergänzt, die in Abbildung 5.7 zu sehen ist und aus Rennspielen entnommen ist. Genauer ist hiermit die Rangänderung gemeint, die neben dem Rang im Leaderboard angezeigt wird. Sie ermöglicht nicht nur eine Momentanaufnahme des eigenen Rangs, sondern motiviert den Fortschritt nochmals, indem die Rangänderung (beispielsweise pro Tag) zusätzlich angezeigt wird.

### 5.2.4 Soziale Interaktion

Bei der Betrachtung des Nutzungskonzeptes von INLOOP fällt auf, dass jeder Nutzer isoliert von anderen Nutzern Aufgaben bearbeitet und dadurch keine direkte soziale Interaktion auf der Plattform stattfindet. Die in der vorigen Sektion vorgestellten progressiven Leaderboards brechen diese Isolation auf, indem sie gezielt Informationen anderen Nutzern zugänglich machen und erstmalig eine soziale Interaktion auf INLOOP ermöglichen. Diese soziale Interaktion soll durch weitere Gamification-Elemente gestärkt werden und zu einem der zentralen intrinsischen Motivatoren nach dem Octalysis-Framework werden. Hierfür werden im Folgenden einige weitere Gamification-Elemente in das Konzept eingebracht.

#### Avatare

In Abschnitt 2.2.2 wurde das Gamification-Element Avatare bereits vorgestellt. Nutzer sollen eine Auswahl an Avataren erhalten, wobei die Komplexität der Nutzerschnittstelle und der Auswahlmöglichkeiten variieren kann. Avatare sollen den Nutzer repräsentieren und es diesem ermöglichen, zum virtuellen Profil und dessen Fortschritt eine persönliche Beziehung aufzubauen. Eine weitere Möglichkeit wäre die Konfigurierbarkeit des Profils mit einem echten Profilbild, wobei allerdings sichergestellt werden muss, dass keine urheberrechtlich geschützten oder anstößigen Bilder hochgeladen werden können. Außerdem wahren Avatare die Privatsphäre des Nutzers, aus welchem Grund diese im Rahmen der Umsetzung zu präferieren sind.

#### Auswahl von Kollegen

Auf Grundlage von Avataren können weitere soziale Elemente integriert werden, beispielsweise eine Freundeslisten-Funktion. Im professionellen Kontext als Arzt im Rahmen des Narratives werden diese Freunde als „Kollegen“ bezeichnet. Ein Nutzer hat hierbei die Möglichkeit, nach anderen Nutzern zu suchen und diese in seinem eigenen Profilbereich, in dem auch sein Avatar, Level, Badges und Punkte sichtbar sein können, als Kollegen aufzuführen und deren Fortschritt nachzuverfolgen. Hierbei kann dies so umgesetzt werden, dass der gewählte Kollege dies zunächst bestätigen muss, oder, so dass das Hinzufügen direkt geschieht (vgl. Abschnitt 5.3.2). Dies soll auch die Kompetitivität fördern, indem Nutzer dazu motiviert werden, ihre Kollegen, welche sie möglicherweise persönlich kennen, in ihrem Fortschritt oder Rang zu übertreffen.

## **Fragen zu Lösungen anderer Nutzer**

Weiteres Erkenntnispotenzial verbirgt sich hinter der Isolation des eingereichten Codes selbst. Bedingt dadurch, dass Nutzer Lösungen anderer nicht plagiieren sollen, erhalten Nutzer lediglich Zugang zum Quellcode der *eigenen* Lösungen. Analog zu den Code-Smell-Quizzes aus [San+19] könnte es von Vorteil sein, wenn Nutzer Code anderer lesen und verstehen müssen, um so zu verstehen, wie Code Smells die Lesbarkeit von Lösungen verringern können. Um zu vermeiden, dass Nutzer Code aus fremden Lösungen verwenden, um selbst Aufgaben zu bestehen, könnte ein Nutzer beispielsweise nur Zugang zu Lösungen bekommen, die er selbst schon bestanden hat. Analog zu den in Abschnitt 3.1 beschriebenen CleanGames könnte ein Code-Smell-Quiz zu diesen Lösungen integriert werden. Im Rahmen des Narratives könnten hierbei Patienten untersucht werden, die aufgrund ihres schlechten Gesundheitszustands gestorben sind (vgl. Abschnitt 5.2.5).

### **5.2.5 Risiko und zeitliche Knappheit**

Wie in der vorigen Sektion erwähnt, sollen Patienten auch „sterben“ können, wenn diese zu viele Code Smells beinhalten oder über eine längere Zeit nicht geheilt werden konnten. Stirbt ein Patient zu einer Aufgabe, so hat ein Nutzer nicht mehr die Möglichkeit, die Punktzahl zu dieser Aufgabe zu verbessern. Als Bedingung, ob eine Lösung als „gestorben“ gekennzeichnet wird, kann die Punktzahl herangezogen werden. Erreicht eine Lösung keine Punkte, also werden von der erreichbaren Punktzahl so viele Punkte für die Code Smells abgezogen, dass keine Punkte übrig bleiben, so könnte sie als gestorben gekennzeichnet werden. Dieses Konzept repräsentiert die Gamification-Komponente „Loss and Avoidance“ aus dem Octalysis Framework, indem ein Nutzer nun nicht mehr unbegrenzt Zeit hat, einen Patienten zu heilen, oder der Patient auch sofort sterben kann, wenn der Nutzer nicht auf eine hinreichende Konformität achtet. Entsprechend müssen die erreichbaren Punktzahlen der Aufgaben so gewählt werden, dass dieser Sonderfall nicht zu häufig auftritt, um mögliche Frustrationen bei den Nutzern zu vermeiden. Beispielsweise könnte die Punktzahl für Exam-Aufgaben in INLOOP zunächst auf 3000 bis 4000 Punkte festgesetzt und anschließend anhand der statistischen Daten nachjustiert werden. Außerdem könnte der Punktabzug für wiederholte Code Smells desselben Typs, zum Beispiel das wiederholte Vergessen von Klammern, graduell verringert werden, um zu vermeiden, dass das häufige Auftreten eines Fehlers zu übermäßigem Punktabzug führt.

### **5.2.6 Dialogkonzept**

Um die konzeptuellen Elemente der vorigen Sektionen anzuwenden und hierfür in konkrete bestehende Dialoge von INLOOP zu integrieren, wurde eine Dialoglandkarte angefertigt.

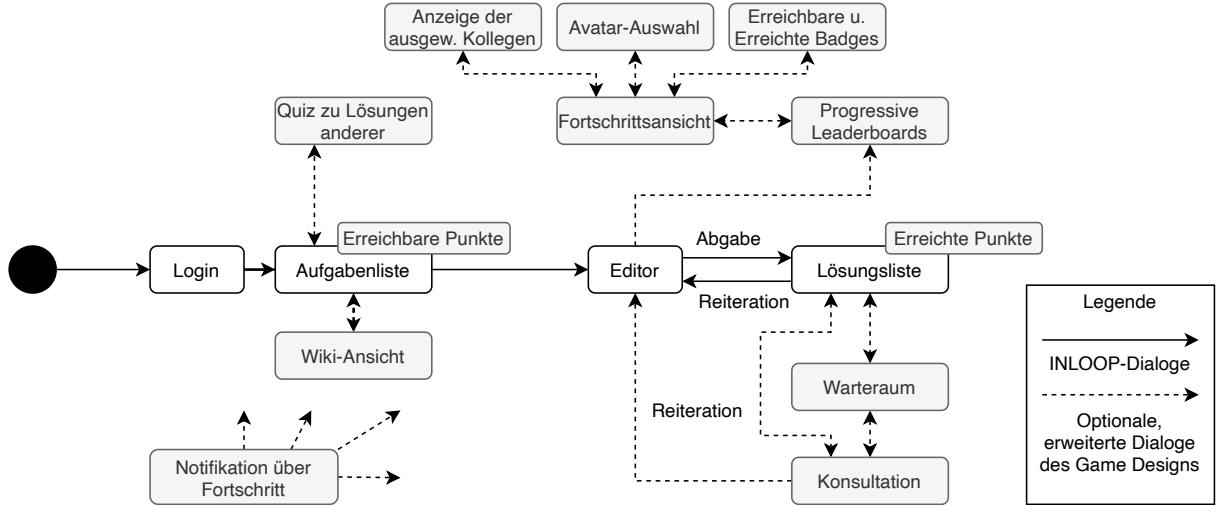


Abbildung 5.8 Konkrete beispielhafte Dialoglandkarte für die Integration der Gamification-Elemente in bestehenden Dialogen von INLOOP

Die in Abbildung 5.8 gezeigte Dialoglandkarte zeigt den bestehenden zentralen INLOOP-Workflow, bei dem ein Nutzer sich zunächst einloggt, dann eine Aufgabe über die Aufgabenliste wählt, diese in einem Editor mit der Aufgabenbeschreibung bearbeitet und einreicht, wonach er seine eingereichten Lösungen sehen und diese gegebenenfalls reiterieren kann, beispielsweise, wenn die Unit-Tests oder der Kompilierprozess nicht erfolgreich waren. Hinzu kommen nun weitere, optionale Dialoge der Gamification-Erweiterung, beginnend mit einer Darstellung der erreichbaren Punkte in der Aufgabenliste und der Möglichkeit, in einer Wiki-Ansicht bestimmte Informationen über die Gamification-Erweiterung zu erhalten, zum Beispiel die Möglichen Verstöße inklusive deren Erklärungen. Außerdem können von hieraus auch die in Abschnitt 5.2.4 erläuterten Quizzes gestartet werden. Führt ein Nutzer hierbei bereits eine Aktion aus, die dazu führt, dass er eine Errungenschaft erhält, wird über einen modularen Notifikationsdialog die dazugehörige Notifikation eingeblendet. Dies ist jedoch unabhängig von der aktuellen Ansicht, so dass Notifikationen prinzipiell in jeder Ansicht eingeblendet werden können. Gibt ein Nutzer eine Aufgabe ab, so hat er zusätzlich die Möglichkeit, die erreichten Punkte der Abgabe zu sehen. Nun hat er die Möglichkeit, die Lösung direkt zu überarbeiten, beispielsweise, wenn der Kompilierungsprozess gescheitert ist. Zusätzlich jedoch, wenn alle Tests erfolgreich waren, kann der Nutzer seine Lösung über einen gesonderten Warteraum oder direkt in einer Konsultation bezüglich der detektierten Code Smells analysieren und von dort aus direkt zur Überarbeitung der Lösung gelangen, um mehr Punkten zu erhalten. Seinen Gesamtfortschritt kann der Nutzer in einer Fortschrittsansicht einsehen. Hierin gelangt der Nutzer direkt über die Navigationsleiste oder über die progressiven Leaderboards, in denen sich der Nutzer mit anderen vergleichen kann. In der Fortschrittsansicht selbst kann der Nutzer seine (in den Leaderboards) gewählten Kollegen anzeigen lassen und selbst einen Avatar auswählen beziehungsweise auch seine erreichten und erreichbaren Errungenschaften anzeigen lassen. Die unterschiedlichen Dialoge haben unterschiedliche Intentionen, so dass zum Beispiel die Wiki-Ansicht zur Information über Codequalität beiträgt, die Komponenten der Progressivität zusätzliche Kompetitivität über das Punktesystem induzieren und schließlich Konsultationen über eigene Verstöße aufklären sollen und gleichzeitig über eine Reiteration der Lösung zum Gewinn von mehr Punkten (adaptiert aus [HN19b]) und dadurch zur Motivation der Verbesserung der Codequalität beitragen sollen.

### 5.2.7 Einordnung nach Octalysis

Die beschriebenen Gamification-Elemente des Game-Designs wurden so ausgewählt, dass die Gamification-Faktoren nach Octalysis hiervon vollständig abgedeckt werden, um eine möglichst ausgewogene Gamification zu erreichen.

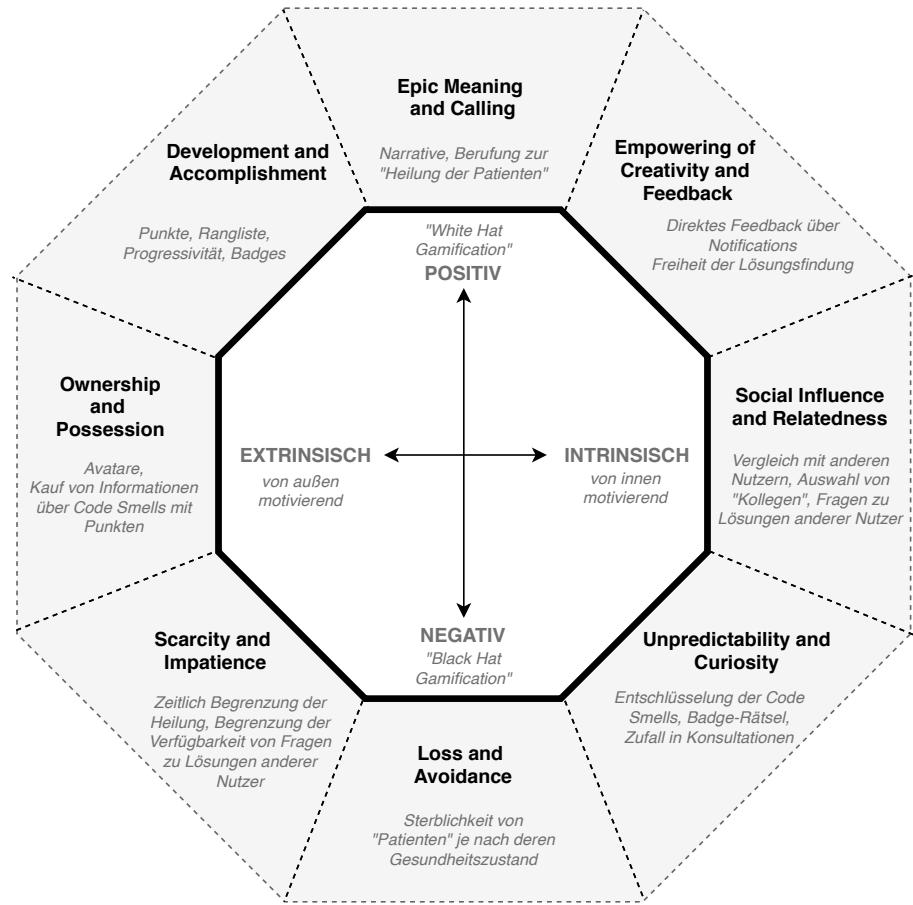


Abbildung 5.9 Einordnung der gewählten Gamification-Elemente nach dem Octalysis Gamification Framework von Chou [Cho19].

In Abbildung 5.9 wird dies sichtbar. Das konzipierte Game-Design verwendet ausgewählte Gamification-Elemente, die sowohl extrinsisch (Avatare, Kauf mit Punkten) als auch intrinsisch (Soziale Interaktion) wirken. Außerdem sind sowohl negativ wirkende Elemente (Risiko und zeitliche Knappheit) als auch positiv wirkende Elemente, wie das allgegenwärtige Narrativ, enthalten. Das Narrativ dient als verbindendes Element zwischen den einzelnen Gamification-Elementen und leitet den Nutzer durch die Anwendung, mit Hinblick auf die Motivation der Refaktorisierung über die Erkennung und Behebung von Code Smells.

### 5.2.8 Einordnung nach Bloom

Wie in Tabelle 3.1 und der dazugehörigen Sektion diskutiert, liegt ein zentraler Aspekt in der Kreation einer didaktisch wirkungsvollen Anwendung in der Evaluation der notwendigen Kompetenzen. Anhand der von Bloom et al. hierfür entwickelten Taxonomie soll auch für das vorliegende Game-Design berücksichtigt werden, dass dieses auch die höheren Lernziele (wie der Ansatz von Händler und Neumann) erfüllen kann. Die Möglichkeit, Refaktorisierungskandidaten zunächst selbst zu suchen und dann gegebenenfalls Hilfe (gegen den Handel von Punkten) zu erhalten, Code anderer Nutzer zu lesen und zu verstehen sowie selbst Clean Code zu entwickeln und dafür Punkte zu erhalten, zielt ab auf unterschiedliche Lernziele nach Blooms Taxonomie, so dass der Nutzer selbst zunächst die Fakten und Zusammenhänge zwischen Clean Code und Code Smells verstehen müssen, diese anhand seiner eigenen Lösungen und der Lösung anderer analysieren und schließlich diese Prinzipien auf die weitere Programmierung selbstständig anwenden soll.

## 5.3 Architekturentwurf der Gamification-Erweiterung

Auf Grundlage der Architektur zur Integration der statischen Codeanalyse sowie der Ergänzung von Code Smells um semantische Beschreibungen und dem vorgestellten Game-Design soll nun die Gamification-Erweiterung architekturell entworfen werden.

### 5.3.1 Integrationspunkte und Komponenten

Da die Gamification-Erweiterung in das bestehende INLOOP-System integriert werden soll, muss zunächst eine Konzeption der konkreten Integrationspunkte durchgeführt werden, um besser zu verstehen, an welchen Stellen die Erweiterung ansetzt und die Erweiterung bestmöglich von der Basisanwendung hinsichtlich der Modularität und der Wartbarkeit abzukapseln.

#### Komponentenansicht

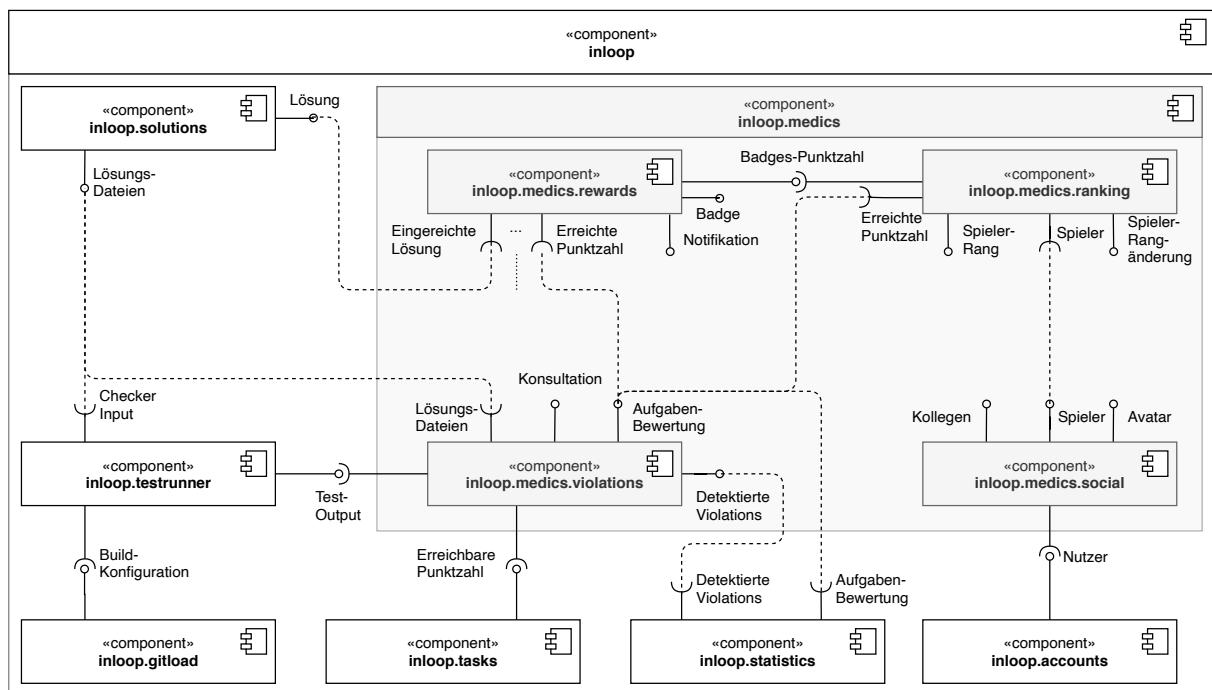


Abbildung 5.10 Übersicht über mögliche Komponenten der Gamification-Erweiterung und deren Integrationspunkte in der bestehenden INLOOP-Architektur.

Anhand von Abbildung 5.10 werden die existierenden Komponenten in INLOOP sichtbar, an welche die Teilkomponenten der Gamification-Erweiterung, welche unter dem Metamodul **inloop.medics** zusammengefasst sind, anknüpfen:

- **inloop.solutions** ist für die Prozesse der Lösungseinreichung und -darstellung verantwortlich und stellt Lösungsdateien bereit.
- **inloop.testrunner** beinhaltet die TestRunner Architektur und erhält von **inloop.solutions** Lösungsdateien, welche durch dieses Modul getestet werden.
- **inloop.gitload** stellt die Continuous-Publishing-Funktionalitäten bereit und die darüber bezogenen Docker Images, von denen die vorkonfigurierten Docker Container zur Durchführung der Tests im TestRunner instanziiert werden.
- **inloop.tasks** ist als Modul verantwortlich für die in INLOOP angebotenen Aufgabenstellungen.
- **inloop.statistics** ermöglicht die Integration von Statistiken für den Administrator.

- **inloop.accounts** stellt die grundlegenden Registrierungs- und Login-Funktionalitäten bereit und erweitert diese um weitere Funktionalitäten, wie beispielsweise das Speichern einer Matrikelnummer.
- **inloop.grading** wird genutzt für die Vergabe von Bonuspunkten, welche für diese Erweiterung nicht vorgesehen ist. Daher ist dieses Modul in Abbildung 5.10 nicht aufgeführt.

Im Folgenden sollen die neuen Komponenten anhand deren Aufgaben beschrieben werden.

**inloop.medics.violations:** Diese Komponente beinhaltet die vorgestellte Parser-Architektur aus Abbildung 5.3 und realisiert hierüber das Einlesen des Test-Outputs der Komponente inloop.testrunner und die Abbildung der Detektionen auf Violations. Die Komponente stellt diese Violations zur Verfügung, indem die dazugehörigen Lösungsdateien (durch inloop.solutions bereitgestellt) im Rahmen einer Konsultation bereitgestellt werden. Des Weiteren ist diese Komponente verantwortlich für die Bewertung der Lösungen. Dazu bezieht sie von der inloop.tasks Komponente die jeweils erreichbaren Punkte für eine Aufgabe, wobei dies im Rahmen der Integration hinzuzufügen ist. Die erreichbaren Punkte können zum Beispiel vom Schwierigkeitsgrad der Aufgabe abhängen. Die Aufgabenbewertung wird als Schnittstelle der Komponente nach außen verfügbar gemacht. Außerdem stellt die Komponente erkannte Violations für die Analyse der häufig auftretenden Code Smells in der inloop.statistics Komponente zur Verfügung, analog zur Anwendungsfall- und Anforderungsspezifikation aus Abschnitt 4.1.3 und Abschnitt 4.2.

**inloop.medics.rewards:** Mithilfe dieser Komponente erhalten Nutzer die Möglichkeit, Errungenschaften zu erhalten, zum Beispiel in Form von Badges. Die Isolation dieser Komponente ist besonders schwierig, weil die Ereignisse, welche zur Freischaltung einer Errungenschaft führen können, in verschiedenen anderen Komponenten erzeugt werden können. In Abbildung 5.10 ist die Einreichung einer Lösung in inloop.solutions und das Erreichen einer Punktzahl in inloop.medics.violations als Beispiel hierfür aufgeführt. Die Komponente muss entsprechende Schnittstellen bereitstellen, so beispielsweise die Prüfung der Freischaltung einer Badge möglichst selbstständig durchführen können. Hierfür kann das Observer[Ull20, S. 448ff] Entwurfsmuster oder eine Abwandlung genutzt werden. Django stellt als Implementation dessen das event-basierte Signals<sup>8</sup> Framework zur Verfügung. Wird eine Badge durch diesen Prozess freigeschaltet, so stellt die Komponente eine entsprechende Notifikation zur Verfügung sowie die hierbei gutgeschriebene Punktzahl.

**inloop.medics.ranking:** Die Punktzahlen von Lösungen und von Badges werden Nutzern über diese Komponente gutgeschrieben und in einem Leaderboard präsentiert, wobei die Komponente den individuellen Rang präsentiert. Dabei werden die Punkte für jeden Nutzer aggregiert und daraus eine Rangfolge erstellt. Da diese Aufgabe rechenintensiv ist, sollte sie gesondert ausgeführt werden (siehe Abschnitt 5.3.1). Die Aggregation kann hierbei in diskreten Zeitintervallen geschehen. Um zusätzlich die Rangänderungen zu aggregieren, bietet sich beispielsweise ein Zeitintervall von 24 Stunden an, sodass sich Ränge und Rangänderungen täglich aktualisieren.

**inloop.medics.social:** Diese Komponente ermöglicht die Bereitstellung der sozialen Funktionalitäten und der Assoziation des Rangs aus der inloop.medics.ranking Komponente mit einem konkreten Spieler. Hierzu wird der jeweils zu einem Spieler gehörende Nutzer von der inloop.accounts Komponente bezogen und mit weiteren Merkmalen ausgestattet. Beispielsweise erhält ein Spieler die Möglichkeit, einen konkreten Avatar auszuwählen. Außerdem stellt diese Komponente die Funktionalität bereit, dass Spieler andere Spieler verfolgen können, indem jeder Spieler im Verantwortungsbereich dieser Komponente andere Spieler zu seinen Kollegen ernennen kann.

---

<sup>8</sup>Django Signals. <https://docs.djangoproject.com/en/3.1/topics/signals/> (Abgerufen am 16.9.2020)

Gemeinsam bilden die oben genannten Komponenten die Architektur der Gamification-Erweiterung. Durch die Komponentenansicht konnten allerdings noch nicht alle Integrationspunkte hinreichend erläutert werden. Hierzu werden in den folgenden Sektionen nochmals konkrete Teile der Architektur aufgegriffen, die eine besondere Rolle im Integrationskonzept spielen.

### Anpassung der Build-Konfiguration

Um die Integration der Codeanalysetools in den containerisierten Buildprozess zu ermöglichen, ist es notwendig, das über Continuous Publishing bezogene Docker Image mit den entsprechenden Build-Schritten für die auszuführenden Codeanalysetools zu ergänzen.

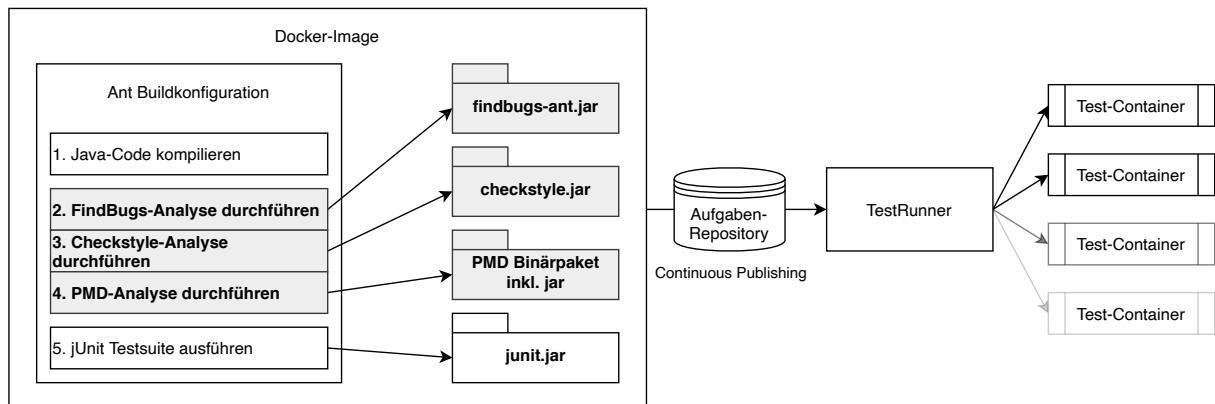


Abbildung 5.11 Notwendige Anpassungen (grau hinterlegt) im Ant Build-Prozess des über Continuous Publishing bezogenen Docker Image des TestRunners.

Diese Schritte sind in Abbildung 5.11 als grau hinterlegt markiert. Hinzu kommen die hier nicht explizit gezeigten Konfigurationen aus dem QualityReview Framework. Die Änderungen müssen im durch Continuous Publishing bezogenen Aufgaben-Repository realisiert werden. Hierfür ist es notwendig, die Archive der Codeanalysetools im Docker Image (grau hinterlegt) zu integrieren, damit diese im vom TestRunner instanziierten Docker Container ausgeführt werden können.

### Auslagerung rechenintensiver und periodisch wiederkehrender Aufgaben

INLOOP nutzt Service Workers, also von der eigentlichen Webanwendung separierte Prozesse, um rechenintensive und periodisch wiederkehrende Aufgaben wie das Durchführen eines Tests durch den TestRunner vom klassischen Request-Response Zyklus oder das Löschen von ungültigen Nutzern auszugliedern. Die Gamification-Erweiterung inkludiert selbst rechenintensive und periodisch wiederkehrende Prozesse, welche aus dem Request-Response Zyklus ausgegliedert werden müssen. Für die Berechnung der Leaderboards beispielsweise sind komplexe Aggregationen für jeden Nutzer notwendig, um den insgesamten Punktestand zu ermitteln. Außerdem ist dieser Prozess unabhängig von einer konkreten Anfrage an die Webanwendung. Für die Integration bietet sich daher an, diesen Prozess an einen Service Worker Prozess auszulagern. Idealerweise werden diese Berechnungen auch nur zu einem Zeitpunkt ausgeführt, wenn die Webanwendung nur eine geringe Last besitzt, zum Beispiel um Mitternacht.

### Bereitstellung von Notifikationen

Zur Ermöglichung eines direkten Feedbacks, sollen Nutzer Notifikationen zu bestimmten ausgeführten Aktionen erhalten, beispielsweise die Änderung ihres Punktestandes. Das im Django Framework, auf welchem die webanwendungsspezifischen Funktionalitäten INLOOPs basieren, bereitgestellte und bereits in INLOOP genutzte Messages-Framework ist hierzu nicht applizierbar, weil dies nur im

nicht unterbrochenen Request-Response-Zyklus eingebunden werden kann. Bei ausgelagerten Prozessen oder bei von Django Signals initiierten Prozessen, welche jeweils eine Notifikation zur Folge hätten, könnte dies nicht genutzt werden. Daher ist die Einführung einer weiteren Komponente notwendig, welche zuvor persistierte Notifikationen in die Webanwendung integriert. Für diese Komponente sollen nun zwei konkrete Konzepte vorgestellt werden.

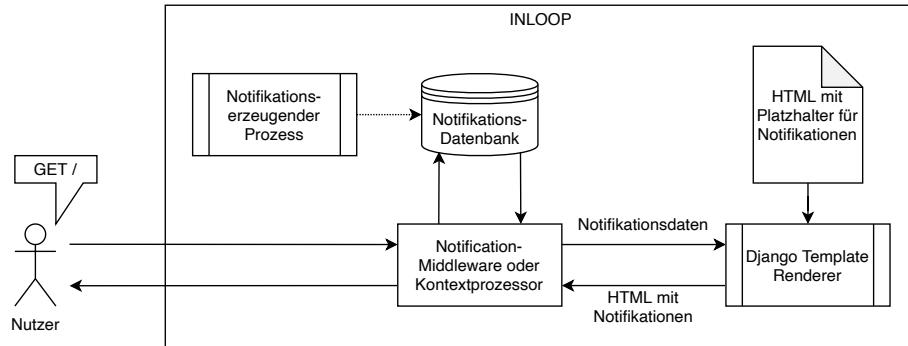


Abbildung 5.12 Architektur für die Bereitstellung von Notifikationen im Pull-Prinzip über eine Django-Middleware oder einen Django-Kontextprozessor im Request-Response-Prozess.

Das erste Konzept ist in Abbildung 5.12 gezeigt und beinhaltet die Einführung einer Komponente, die ähnlich wie ein Proxy agiert. Browser-Anfragen eingeloggter Nutzer an den Server werden über diesen Proxy geleitet, wobei die INLOOP-Datenbank nach ungelesenen Notifikationen für einen Nutzer abgefragt wird. Diese Notifikationen können hier zuvor von anderen Prozessen erzeugt worden sein und sind nicht an den Request-Response-Prozess gebunden. Geladene, ungelesene Notifikationen werden als gelesen markiert und in Metadaten der Anfrage des Nutzers geschrieben. Diese Metadaten werden von der angeforderten Ansicht eingelesen und können nun verwendet werden, um die Notifikationen in das zurückzugebende HTML zu rendern. Der Nutzer erhält die Notifikation in der angeforderten Seite. Zu exkludieren sind hierbei Anfragen, die kein HTML, sondern beispielsweise Ressourcen anfordern, beispielsweise AJAX Anfragen. Dieses Prinzip ist sehr einfach integrierbar über die Nutzung einer Django Middleware<sup>9</sup>, welche es ermöglicht, im Request-Response Zyklus zusätzliche Operationen zu ergänzen, oder über das Verwenden eines Django Kontextprozessors<sup>10</sup> für die Ergänzung von zusätzlichen Umgebungsinformationen, in diesem Fall die Notifikationen. Leider erhält der Nutzer auf diese Weise Notifikationen immer erst beim Laden einer Seite, nicht schon asynchron, während die Seite geöffnet ist. Daher soll noch ein zweites Konzept vorgestellt werden.

<sup>9</sup>Django Middleware. <https://docs.djangoproject.com/en/3.1/topics/http/middleware/> (Abgerufen am 16.9.2020)

<sup>10</sup>Django Template API. <https://docs.djangoproject.com/en/3.1/ref/templates/api/> (Abgerufen am 16.9.2020)

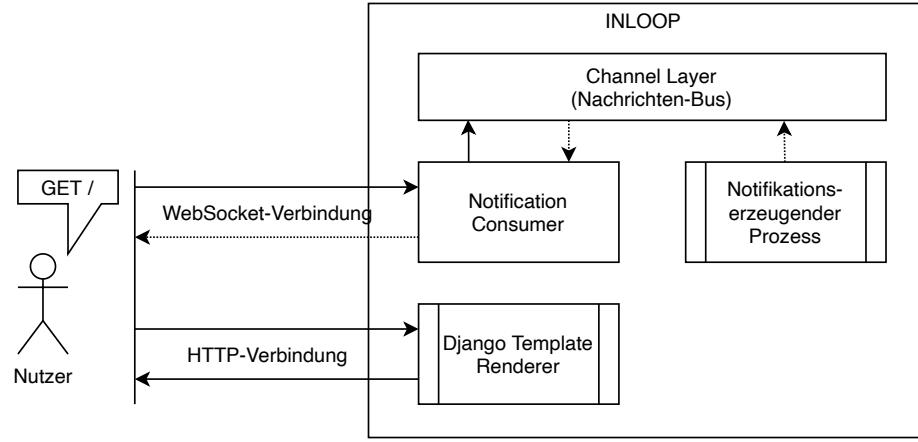


Abbildung 5.13 Architektur für die Bereitstellung von Notifikationen im Push-Prinzip über einen asynchronen Nachrichtenaustausch.

Bei dem in Abbildung 5.13 vorgestellten zweiten Konzept zur Bereitstellung von Notifikationen wird zur Bereitstellung von Notifikationen ein Seitenkanal benutzt, der von dem Django-Channels-Framework bereitgestellt werden kann. Hierbei registriert sich der Nutzer bei dem Besuch der Seite über eine WebSocket Verbindung in einem von Django bereitgestellten individuellen „Consumer“. Sobald ein nebenläufiger Prozess eine Notifikation erzeugt, wird diese über den Nachrichten-Bus an den Consumer übermittelt, und schließlich über die aktive WebSocket-Verbindung an den Nutzer. Eine Persistierung ist nicht zwingend notwendig, jedoch sinnvoll, um Notifikationen nicht zu verlieren. Dieser Ansatz ermöglicht eine Benachrichtigung des Nutzers in Echtzeit, jedoch ist der Ansatz technisch limitiert und erfordert den Einsatz eines ASGI<sup>11</sup>-unterstützenden Server-Backends, wie zum Beispiel Daphne<sup>12</sup>, und damit eine Umstrukturierung des Deployments.

### 5.3.2 Modelle und Persistenzschicht

Zusätzlich zur Strukturierung der Gamification-Erweiterung in Teilkomponenten soll nun auch eine konkrete jeweilige Modellstruktur entworfen werden, als Grundlage für die Implementation im nächsten Kapitel.

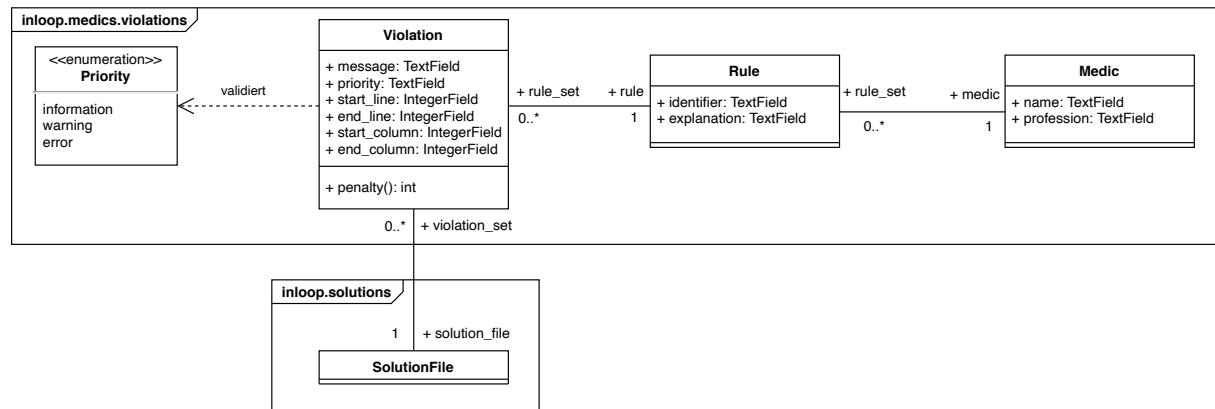


Abbildung 5.14 Eine mögliche Modellstruktur der inloop.medics.violations Komponente.

Abbildung 5.14 zeigt eine Modellstruktur, auf deren Grundlage die Funktionalitäten der inloop.medics.violations Komponente umgesetzt werden können. Hierbei werden, wie in den vorigen Sektionen erläutert, Violations durch das Parsing der Reports von Codeanalysetools aus dem TestRunner erstellt

<sup>11</sup>ASGI. <https://asgi.readthedocs.io/en/latest/> (Abgerufen am 13.8.2020)

<sup>12</sup>Daphne. <https://github.com/django/daphne> (Abgerufen am 13.8.2002)

und zu einem jeweiligen SolutionFile aus der inloop.solutions Komponente zugewiesen, um dieses später in der Nutzeroberfläche zusammen anzeigen zu können. Außerdem erhält eine Violation eine Priorität in Form einer zu validierenden Zeichenkette, wobei auf Grundlage dieser später der Punktabzug für Lösungen über die Bereitstellung einer Penalty errechnet werden muss sowie eine Rule, welche die Abbildung der Erklärungsdatenbank symbolisiert und die jeweiligen Erklärungen beinhaltet. Eine Rule wird analog zum Narrativ einem Medic zugeordnet, welcher einen Namen und eine spezielle Profession (zum Beispiel Chirurg) hat. Zu beachten ist, dass die Rule-Objekte und die Medic-Objekte nicht dynamisch vom Nutzer erzeugt werden, sondern zum Start der Anwendung über bestimmte Migrationsmechanismen bereitgestellt werden müssen. Außerdem ist anzumerken, dass, wie bei allen in dieser Sektion gezeigten bidirektionalen Assoziationen die Gegenseite durch objektrelationales Datenbankmapping automatisch inferiert wird und somit in der Integration implizit gegeben ist, aber in der Modellstruktur dennoch explizit notiert ist. Der Gesundheitszustand einer Lösung bestimmt sich durch die Anzahl an Violations und der durch die Aufgabe der Lösung erreichbare Punktzahl, von der diese abgezogen werden. Erreicht eine Lösung die volle mögliche Punktzahl, so ist sie gesund. Erhält sie Punkte, jedoch nicht alle möglichen Punkte, so ist die Lösung krank. Eine Lösung wird als gestorben gekennzeichnet, wenn sie keine Punkte erreicht hat. Auf Grundlage dessen können außerdem die im Game-Design erwähnten Konsultationen (Abschnitt 5.2.3) oder Fragen zu Lösungen anderer (Abschnitt 5.2.4) umgesetzt werden, wozu eine Konsultation oder ein Quiz zu mehreren Violations erstellt und mit den zur jeweiligen Violation sichtbaren Informationen ausgestattet werden kann.

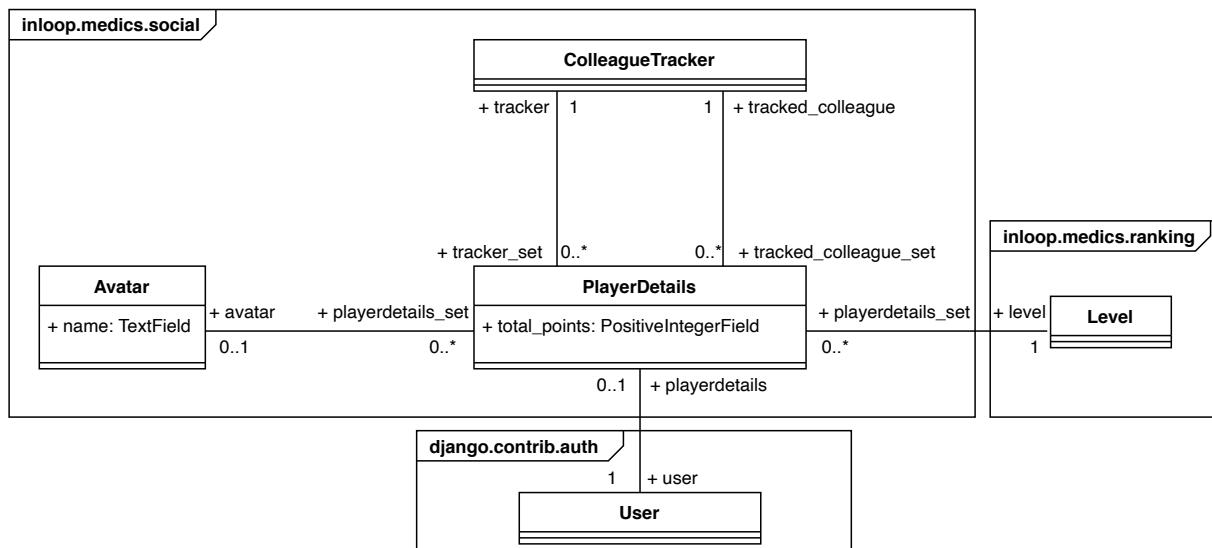


Abbildung 5.15 Eine mögliche Modellstruktur der inloop.medics.social Komponente.

Die in Abbildung 5.15 gezeigte inloop.medics.social Komponente erweitert durch das PlayerDetails Modell den bereits gegebenen User, wobei dieser durch die erreichten Punkte, dem direkt damit zusammenhängenden Level und einem auswählbaren (jedoch optionalen) Avatar ergänzt wird. Außerdem ermöglicht die Komponente das Erstellen von so genannten ColleagueTrackers, welche die Kollegen-Beziehung eines Nutzers zu einem anderen abbildet. Hierbei gibt es immer einen „tracker“, also der Nutzer, welcher die Kollegenbeziehung mit dem hierbei passiven „tracked\_colleague“ auswählt. Wählt ein Nutzer einen anderen Nutzer als Kollegen aus, so bedeutet dies nicht, dass diese Beziehung auch umgekehrt gelten muss. Somit wird verhindert, dass Nutzer gegen ihren Willen auf dem eigenen Profil als Kollege eines anderen, möglicherweise ihnen unbekannten Nutzer, hinzugefügt werden. Für eine implizite Umkehrbarkeit wäre die Konzeption und Integration von Freundschaftsanfragen sinnvoll.

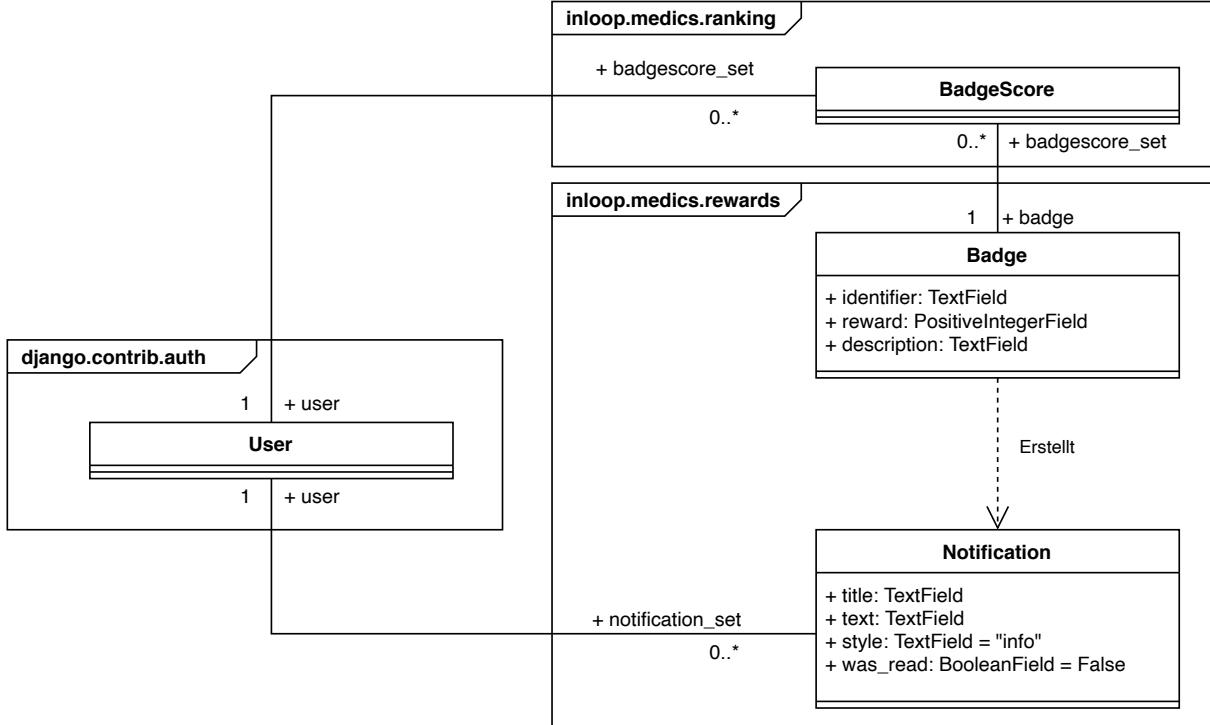


Abbildung 5.16 Eine mögliche Modellstruktur der inloop.medics.rewards Komponente.

Eine zentrale Rolle in der inloop.medics.rewards Komponente spielen die Badges, welche durch einen Spieler freigeschaltet werden können. Badges werden hierbei durch einen Identifikator gekennzeichnet, durch welchen diese später in der Anwendungslogik referenziert werden können. Über eine Schnittstelle, die für einen Nutzer eine Badge mit einem bestimmten Identifikator durch eine spezielle Interaktion freischaltet, ist eine hinreichende Entkopplung vom Kontext der anderen Komponente möglich. Hierbei ist zu unterscheiden zwischen Badges, welche unabhängig vom Nutzer über den initialen Migrationsprozess geladen werden können (repräsentiert durch das Badge Modell) sowie der Markierung, dass ein bestimmter Nutzer eine Badge freigeschaltet hat (BadgeScore). Bei der Freischaltung einer Badge wird über eine der in Abschnitt 5.3.1 diskutierten Architekturen eine Notifikation erstellt und an den Nutzer ausgegeben.

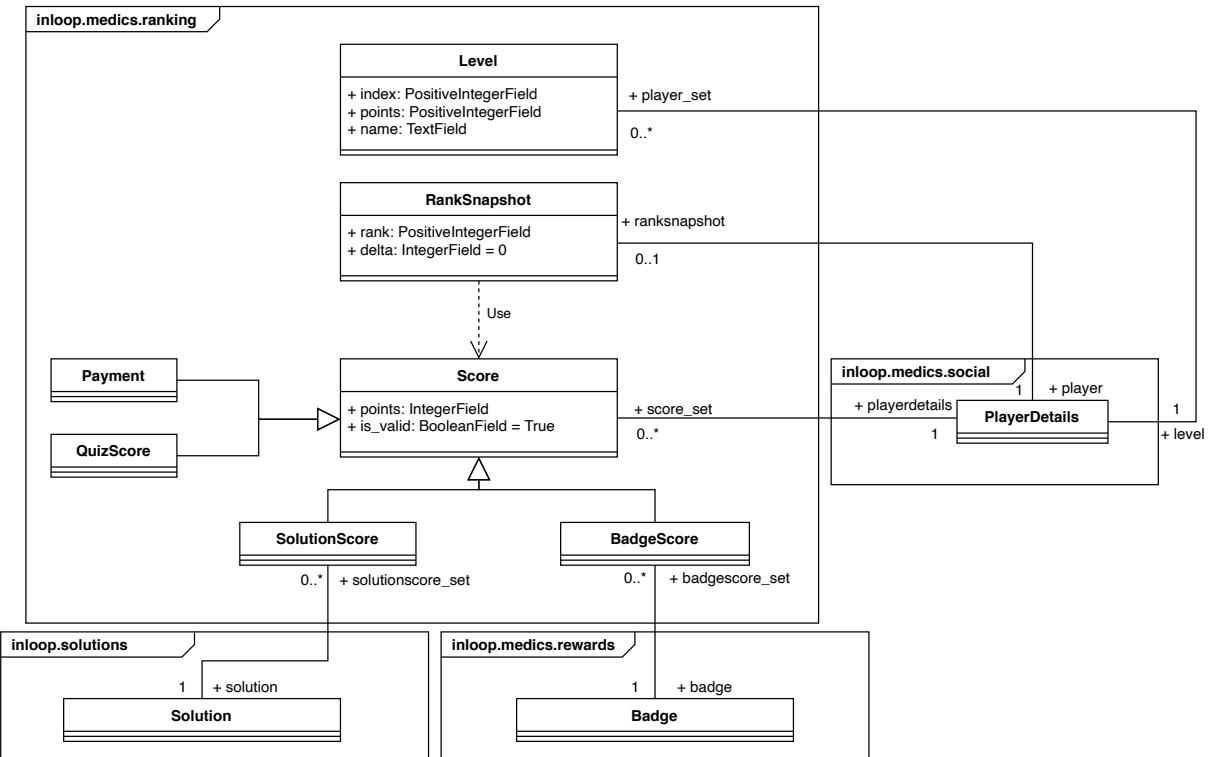


Abbildung 5.17 Eine mögliche Modellstruktur der `inloop.medics.rankin` Komponente.

Das BadgeScore-Modell stammt hierbei aus der `inloop.medics.rankin` Komponente, wie in Abbildung 5.17 gezeigt. Diese beinhaltet verschiedene Modelle, die als Datenpunkte für die Persistierung der erreichten Punkte eines Nutzers dienen. Das Metamodell, Score, referenziert hierfür die dazugehörigen PlayerDetails und speichert einen ganzzahligen Punktwert. Da sowohl die Abgabe von Lösungen, als auch das Freischalten von Badges dem Nutzer Punkte gutschreibt, kann ein Score sowohl ein SolutionScore (referenziert Lösung), als auch ein BadgeScore (referenziert Badge) sein. Diese Score-Objekte werden durch die Interaktionen des Nutzers erzeugt und mit den entsprechenden Punktwerten versehen. Da beispielsweise eine Lösung auch aktualisiert werden kann und die Punktwerte nur für die aktuellste, bestandene Lösung gutgeschrieben werden sollen, wurde dem Metamodell Score noch ein boolsches Feld hinzugefügt, um die Validität zu setzen. SolutionScore Objekte überholter Lösungen können auf invalid gesetzt werden und werden infolgedessen nicht mehr für die Punkteberechnung herangezogen. Des Weiteren können Payments eingeführt werden, für die Umsetzung der Kauf-Mechanik in Konsultationen, um eine Detektion aufzudecken. Diese Payments würden dann von der Punktzahl eines Nutzers abgezogen werden, realisierbar durch einen negativen Punktwert. Die in Fragerunden zu Lösungen anderer Nutzer (Abschnitt 5.2.4) richtig beantworteten Fragen erzielen Punkte als QuizScore. Die Punkte eines Nutzers können somit errechnet werden, indem die Summe aller Punkte der validen Score Objekte eines Nutzers gebildet wird. Anhand dessen wird auch das Level des Nutzers bestimmt. Für die Realisation der dynamischen Leaderboards wird außerdem ein RankSnapshot Modell eingeführt, welches für einen Nutzer in diskreten Zeitintervallen den Rang berechnet und den Unterschied zur letzten Position abspeichert.

**Skalierbarkeit:** Die Erstellung der RankSnapshots erfordert in festen Zeitäbständen wiederholte und rechnenintensive Aggregationsoperationen über der Datenbank. Dies ist bei der Implementation zu beachten und gesondert an Service-Worker-Prozesse auszulagern. Außerdem ist eine konsistente Zwischenspeicherung der Punkte eines Nutzers sinnvoll, wenn Leaderboards nach diesem Kriterium sortiert werden sollen.

## 5.4 Zusammenfassung

Die vorgestellte Architektur zur Integration des QualityReview Framework und Vereinheitlichung von Code Smells, zusammen mit dem Einsatz einer Erklärungsdatenbank, schafft eine konzeptionelle Grundlage für die Implementation der Gamification-Erweiterung. Hierzu werden die aufbereiteten Detektionen im Rahmen einer Konsultation oder eines Quiz an den Nutzer vermittelt, wobei dies Teil eines in eine Rahmenhandlung integrierten Game-Designs ist. Zur weiteren Motivation des Nutzers wurden konkrete Gamification-Elemente auf Grundlage des Octalysis-Frameworks ausgewählt und konzeptuell in das Narrativ integriert. Unter Berücksichtigung der psychologischen Wirkung werden hierdurch die acht Kernantriebe der Octalysis genutzt, um eine ausgewogene Gamification mit Hinblick auf die Motivation von Clean Code zu realisieren. Nutzer können zum Beispiel Punkte, Errungenschaften und Level durch das aktive Lernen der Charakteristika von Clean Code, zum Beispiel durch die Minimierung der Code Smells in eigenen Lösungen, erreichen und sich mit anderen Nutzern in einem sozialen Kontext vergleichen. Das direkte Feedback wird durch die Bereitstellung von Notifikationen erweitert. Nachfolgend wurde konzeptionell gezeigt, wie die Gamification-Elemente des Game-Designs in INLOOP integriert werden können und welche Teilkomponenten hierbei separierbar sind. Für die einzelnen Teilkomponenten wurden konkrete Modelle konzipiert, welche nun für die prototypische Implementation der Gamification-Erweiterung verwendet werden können.

# 6 Prototypische Implementation

Für die Implementation der Gamification-Erweiterung wurden für die Hauptanwendung<sup>1</sup> und das Beispielaufgaben-Repository<sup>2</sup> jeweils einen öffentlichen Fork auf GitHub erzeugt, in welchem die Integration der Erweiterung stattfand. Hierzu wurden zunächst im später per Continuous Publishing einbezogenen Beispiel-Aufgabenrepository die in Abbildung 5.2 gezeigten Änderungen im Buildprozess des Aufgabenrepositories umgesetzt. Hierbei wurde unter anderem der Buildschritt für das Checkstyle-Framework zusammen mit dem Regelwerk aus QualityReview ergänzt. Das Checkstyle-Regelwerk wurde nochmals leicht modifiziert, indem eine Regel entfernt wurde, welche in Java-Dateien auf das Vorhandensein von Package-Statements prüft. Dies könnte sonst möglicherweise mit dem TestRunner inferieren<sup>3</sup>. Für die integrierten Regeln des Checkstyle Frameworks wurde anschließend anhand deren Dokumentation<sup>4</sup> eine Erklärungsdatenbank ermittelt, welche die Checkstyle-Regeln des Quality-Review Frameworks auf zusätzliche Erklärungen abbildet. Diese Erklärungsdatenbank ist in Textform als Abschnitt 8.2 beigefügt und wird in die INLOOP-Anwendung per Datenbank-Migration dynamisch integriert. Im Umsetzungsrahmen des Prototyps wurde Checkstyle ausgewählt, um das Konzept der Erklärungsdatenbank und der Code-Smell-Detektion in INLOOP zu evaluieren. Zu einem späteren Zeitpunkt können PMD und FindBugs/SpotBugs<sup>5</sup> mit relativ geringem Zeitaufwand ergänzt werden. Um die Abbildung der Code-Smells auf die Erklärungsdatenbank zu realisieren, wurde ein Checkstyle-Parser implementiert, welcher die XML-Baumstruktur der Checkstyle-Ausgabe auf ein persistierbares Violation-Modell analog zu Abbildung 5.14 überführt. Für eine Ergänzung von PMD und FindBugs/SpotBugs müssten analog eigene Parser implementiert werden, welche die genutzten Regeln auf bestehende Erklärungen in der Datenbank abbilden, wobei diese gegebenenfalls durch weitere Erklärungen ergänzt werden müssten.

Als weitere Funktionsgrundlage wurden die in Abschnitt 5.3.2 beschriebenen weiteren Modelle der Persistenzschicht auch in INLOOP integriert. Hierbei wurden die Teilkomponenten in einer Komponente „medics“ entsprechend der Komponentenansicht in Abbildung 5.10 zusammengefasst. Außerdem wurde die Notifikationsarchitektur nach Pull-Prinzip (Abbildung 5.12) mithilfe eines Kontextprozessors integriert. In der „rewards“ Teilkomponente der Gamification-Erweiterung wurde eine Schnittstelle zur Verfügung gestellt, mithilfe derer Nutzer bestimmte „Badges“ erhalten können. Die daraus resultierende Neuberechnung der Punkte und Level eines Spielers wurde als rechenintensiver Prozess an einen Worker-Prozess analog zu Abschnitt 5.3.2 ausgelagert. Auf Grundlage dieser Backend-Funktionalitäten konnten nun die Nutzerschnittstellen implementiert werden, welche im nachfolgenden näher beschrieben werden sollen.

<sup>1</sup>GitHub. <https://github.com/inloop-gamified/inloop> (Abgerufen am 1.9.2020)

<sup>2</sup>GitHub. <https://github.com/inloop-gamified/inloop-java-repository-example> (Abgerufen am 1.9.2020)

<sup>3</sup>GitHub. <https://github.com/st-tu-dresden/inloop/issues/288> (Abgerufen am 1.9.2020)

<sup>4</sup>Checkstyle. <https://checkstyle.sourceforge.io/checks.html> (Abgerufen am 1.9.2020)

<sup>5</sup>Da FindBugs durch SpotBugs ersetzt wurde, müsste dies gegebenenfalls zusätzlich migriert werden.

**Anmerkung.** Bei der Konzeption und Gestaltung der Webseiten wurden bestimmte Prinzipien der Nutzbarkeit befolgt. So wurde insbesondere auf die Einfachheit der Nutzeroberfläche und eine nahtlose Integration der Erweiterung in die Basis-Anwendung geachtet. Außerdem wurde das weit verbreitete Prinzip der „magischen Nummer Sieben“ nach Miller [Mil56] verwendet, um Nutzer bei der Bedienung nicht zu überfordern und eine möglichst gute Perzeption der Funktionsweisen des Game-Designs zu ermöglichen. Das Prinzip besagt, dass Nutzer unterschiedliche Informationen in Nutzeroberflächen nur solange ideal wahrnehmen, wie deren Anzahl nicht  $7 \pm 2$  überschreitet.

Rank	Avatar	User	Points	Level
1 ± 0		admin	3000	2 - Specialist
2 ± 0		testuser   Add as colleague	800	0 - Student

Note: Today's ranks are updated at midnight.

Abbildung 6.1 Täglich aktualisiertes Leaderboard mit Suchfunktion, Avataren und dynamischer Rangänderungsanzeige.

In Abbildung 6.1 ist ein Screenshot der UI-Komponente des Leaderboards gezeigt. Die direkt aus den Punkten eines Spielers resultierende Rangänderung wird hierbei nächtlich berechnet, als rechenintensiver Prozess wiederholt ausgelagert an einen der Worker-Prozesse. Ein Nutzer kann somit seinen eigenen täglichen Fortschritt anhand der aufgestiegenen oder abgestiegenen Ränge erkennen. Mit der angebrachten Suchleiste kann ein Nutzer nach anderen Nutzern suchen und diese bei Bedarf als seine Kollegen hinzufügen.

Abbildung 6.2 Navigationsleiste und Aufgabenansicht.

Wie in Abbildung 6.2 sichtbar, integriert die Gamification-Erweiterung auch ihre eigene Navigationskomponente, mit welcher die gesonderten Seiten der Gamification-Erweiterung (Wartezimmer, Ranglisten, Profilseite, Errungenschaften, Wiki über Code-Doktoren) besucht werden können. Hieran gut sichtbar ist die weitgehende Separierung der Funktionalitäten und Webseiten der Erweiterung von der Basis-Anwendung durch eine Trennung der Kontexte. Eine der Stellen, bei der die Integration der Gamification-Erweiterung direkt in der Schnittstelle der Basis-Anwendung stattfinden musste,

ist die Aufgabenliste. Hier werden die erreichbaren Punkte einer Aufgabe dem Nutzer entsprechend angezeigt.

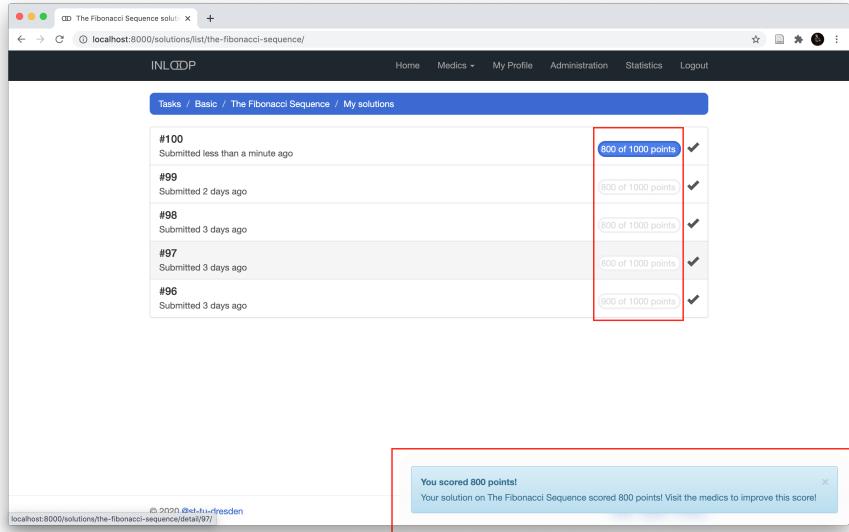


Abbildung 6.3 Lösungsansicht mit erreichten Punkten und Notifikation.

Reicht ein Nutzer nun eine Lösung ein, erhält er zunächst eine Notifikation über das Erreichen von Punkten durch die Abgabe sowie eine Information über die erreichten Punkte in der Liste der Lösungen (Abbildung 6.3). Hierbei ist nur die oberste Lösung hervorgehoben, um zu signalisieren, dass nur die aktuellste Lösung zu den erreichbaren Punkten beiträgt. Im gezeigten Beispiel erreichte der Nutzer 800 von 1000 Punkten.

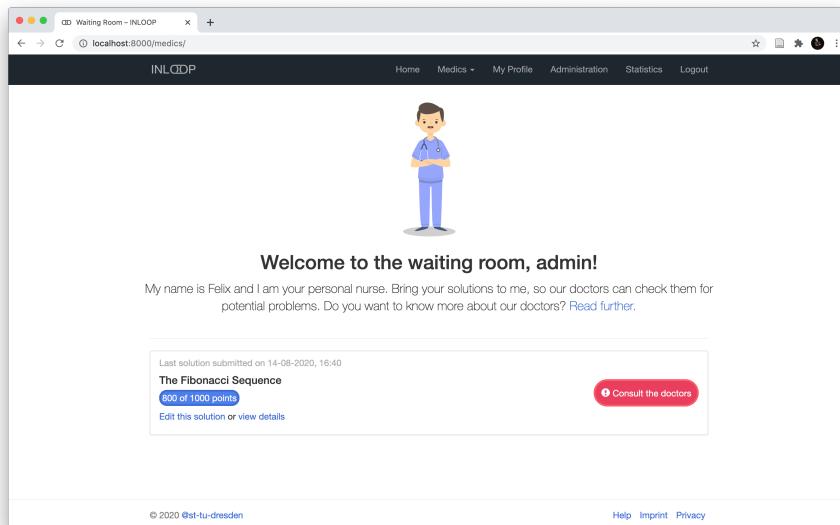


Abbildung 6.4 Wartezimmer mit dem Gesundheitszustand einer Lösung (roter Button).

Um nun die Punktzahl seiner Lösung zu verbessern, kann ein Nutzer ein Wartezimmer besuchen, in dem er seine aktuellen Lösungen sieht sowie deren Gesundheitszustand. Dieses Wartezimmer ist in Abbildung 6.4 gezeigt.

```

1 import java.lang.*;
2 import java.math.BigDecimal;
3 import java.math.BigInteger;
4
5 /**
6 * This class has some style issues, which should be detected by a code checker.
7 */
8
9 public final class Fibonacci {
10     private static final BigDecimal GOLDEN_RATIO = new BigDecimal((1 + Math.sqrt(5)) / 2);
11     private static final BigDecimal GOLDEN_RATIO_MINUS_1 = GOLDEN_RATIO.subtract(BigDecimal.ONE);
12     private static final BigDecimal SQRT_5 = new BigDecimal(Math.sqrt(5));
13     private static final int DECIMAL_LIMIT = 45;
14     private static final int FIVE = 5;
15
16     public Fibonacci() {}
17
18     private static BigInteger fib_version_1(int n) {
19         BigInteger nthTerm = new BigInteger("0");
20         if (n == 2) {
21             nthTerm = BigInteger.ONE;
22         } else {
23

```

Abbildung 6.5 Konsultation mit der Anzeige der Zeile und Beschreibung aus der Erklärungsdatenbank zu einer Detektion.

Vom Wartezimmer gelangt der Nutzer in eine Konsultation mit den Code-Doktoren. Wie in Abbildung 6.5 gezeigt, erhält der Nutzer hier ein visuelles Feedback der gefundenen Violations, indem die dazugehörige Codezeile gezeigt wird sowie die zur Violation aus der Erklärungsdatenbank ermittelte Erklärung des Problems. Je nach dem Schweregrad der Violation wird diese mit 100 Punkten (Schweregrad error), 50 Punkten (Schweregrad warning) oder 0 Punkten (Schweregrad information) von der erreichbaren Punktzahl abgezogen. Somit wird ein faires Punktesystem gewährleistet, bei dem Nutzer, welche mit besserer Codequalität entwickeln, am meisten durch Punkte belohnt werden.

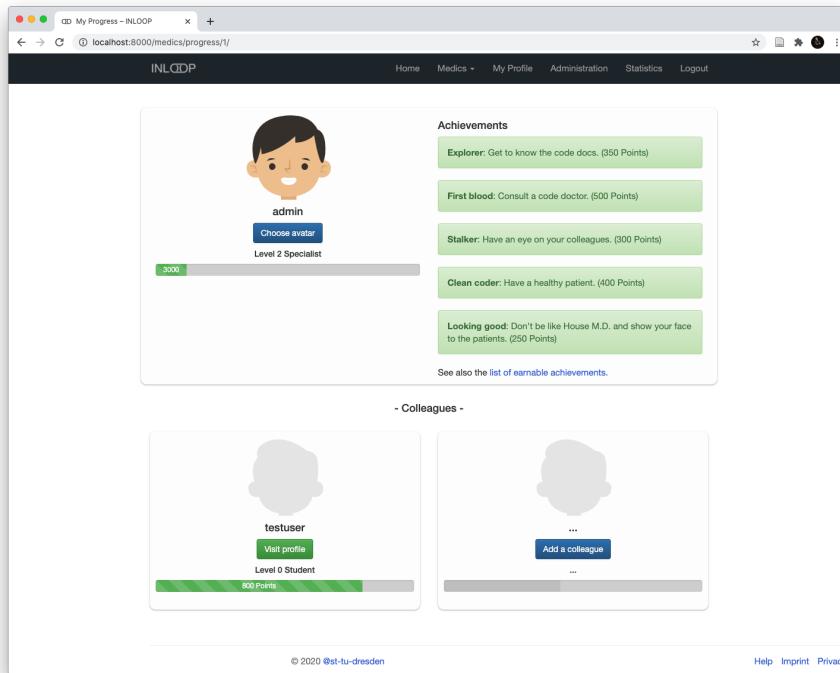


Abbildung 6.6 Freigeschaltete Badges, Avatare und Level in den Spielerdetails und die Anzeige von Kollegen.

In der Fortschrittsansicht eines Nutzers (Abbildung 6.6) sieht dieser seine Errungenschaften, sein erreichtes Level, seinen durch eine gesonderte Ansicht auswählbaren Avatar sowie seine im Leaderboard ausgewählten Kollegen. Die Beschreibungen der erreichbaren Errungenschaften sind hierbei als einfache Rätsel gedacht, beigelegt unter Tabelle C.1. Die erreichbaren Level sind außerdem an das Narrativ angelehnt und unter Tabelle D.1 aufgelistet. Die Level repräsentieren den Fortschritt des Nutzers und sind an den Werdegang eines ambitionierten Code-Doktors angelehnt. So beginnt der Nutzer als „Student“ und kann über verschiedene Level bis zum „Medical Director“ aufsteigen.

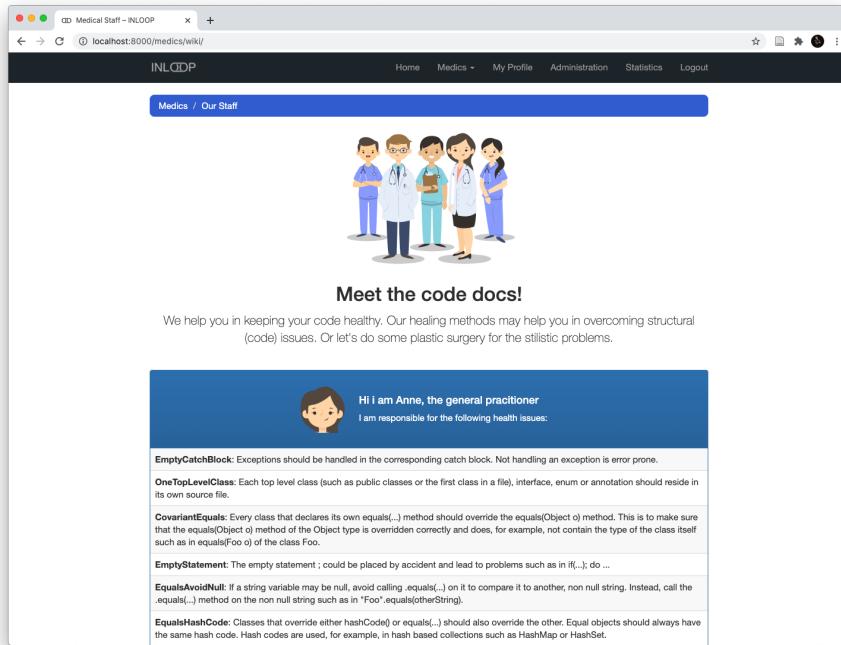


Abbildung 6.7 Informationsseite über detektierbare Code Smells, deren Identifikator und Erklärung aus der Erklärungsdatenbank.

Möchte ein Nutzer mehr über die Code-Doktoren erfahren oder sich über die einzelnen möglichen Violations aus der Erklärungsdatenbank informieren, so kann er eine Wiki-Seite (gezeigt in Abbildung 6.7) besuchen, auf der dies aufgelistet ist. Analog zu den Beschreibungen des Konzepts zur Kategorisierung von Violations wird jedem Code-Doktor eine spezielle Profession zugeordnet sowie den dazugehörigen Violations. Die konkreten Professionen sind unter Tabelle A.1 beigelegt. Dies konkludiert die prototypische Implementation von Kernelementen des Konzeptes aus Kapitel 5.

**Selektivität und Repräsentativität des Prototyps.** Nicht alle im Konzept vorgeschlagenen Gamification-Elemente wurden hierbei integriert, so zum Beispiel die Möglichkeit, dass ein Patient „sterben“ und danach nicht mehr geheilt werden kann, oder das Code-Smell-Quiz und die Fragen zu Code Smells anderer Nutzer, welche jeweils die nun implementierte, einfache Konsultation noch erweitern können. Für eine vollständige Integration des Konzeptes müssten diese Elemente noch ergänzt werden. Eine statistische Auswertung der aufgetretenen Violations ist im Statistiken-Modul analog zur Anforderungsspezifikation nahtlos auf Grundlage der persistierten Modelle möglich, um eine Analyse der Nutzerinteraktion der Gamification-Erweiterung in Produktion durch das Lehrpersonal zu realisieren. Um generell festzustellen, inwiefern die Ziele dieser Arbeit anhand des Konzeptes erreicht werden konnten, kann eine Evaluation anhand der prototypischen Implementation dennoch mit hinreichender Aussagekraft und Fokus auf die Erschließung von verschiedenen Gamification-Elementen durchgeführt werden.

# 7 Evaluation

Zur Validierung des Konzeptes sowie der prototypischen Umsetzung wurde ein zweikomponentiges Evaluationskonzept erstellt, welches sich zusammensetzt aus einer am Prototypen durchgeführten Expertenevaluation sowie einer umfragenbasierten Studierendenevaluation. Die Expertenevaluation ist hierbei qualitativ orientiert, während über die Studierendenevaluation durch Umfragen eine quantitative Analyse durchgeführt wurde. Im Zentrum der Evaluation stand hierbei die Klärung der in Abschnitt 1.2 formulierten Forschungsfragen, auf welche zum Schluss dieses Kapitels eingegangen werden soll. Zunächst sollen jedoch nachfolgend die einzelnen Teilevaluationen sowie deren Ergebnisse erläutert und auf potenzielle Gefahren für die Validität hingewiesen werden.

## 7.1 Expertenevaluation

### 7.1.1 Evaluationsstrategie und Durchführung

Für die Expertenevaluation wurden drei Experten akquiriert, welche die fachliche Expertise vorwiesen und INLOOP selbst bereits genutzt haben, darunter zwei Tutoren der Softwaretechnologie mit mehrjähriger didaktischer Erfahrung sowie ein angestellter Webentwickler mit Abschluss als B.Sc. an der Fakultät Informatik der TU Dresden. Die Expertenevaluation wurde in individuellen Videokonferenzen durchgeführt, bei denen der Experte jeweils auf das erweiterte INLOOP per Webbrowser zugreifen konnte. Hierzu wurde die erweiterte Anwendung über einen Netzwerk tunnel verfügbar gemacht. Anhand dessen wurde anschließend eine Evaluation anhand von Methoden des Discount Usability Engineerings angewandt [Nie94, S. 17 ff][Nie95]. Hierbei wurde zunächst eine Einführung in den didaktischen Hintergrundgedanken der Erweiterung gegeben und nachfolgend eine Simplified-Thinking-Aloud Analyse durchgeführt. Dem Experten wurden mehrere Aufgaben anhand des Szenarios der Nutzung des Prototypen als Studierender gegeben, nach denen er bestimmte Interaktionen in der Webanwendung ausführen sollte. Konkreter waren die Aufgaben so gewählt, dass hierdurch die Verwendung der Gamification-Elemente in den Vordergrund gestellt wurde sowie welche Gefühle, Motivationen und sonstige Gedanken mit Fokus auf die Refaktorisierung und Erkennung von Refaktorisierungskandidaten vom Experten geäußert wurden. Als initialer Schritt der Expertenevaluation wurden die Experten hierbei gebeten, sich in der Anwendung anzumelden und sich mit der Startseite vertraut zu machen. Hierfür wurden im Voraus individuelle Accounts vorbereitet, welche für die Anmeldung verwendet werden konnten. Außerdem wurden bereits Lösungen zu einer Beispielaufgabe vorbereitet, welche bestimmte Code Smells beinhalteten. Nach der Anmeldung in der Anwendung wurden nun, beginnend mit einer Infografik auf der Startseite, individuelle Interaktionswege der Experten durch die Anwendung gewählt und diskutiert. Hierzu wurde der jeweilige Experte gebeten, alle Interpretationen und Gedanken während der Durchführung möglichst ausführlich zu äußern. Nach Abschluss einer Aufgabe wurden dem jeweiligen Experten konkrete

Nachfragen zu den einzelnen Komponenten gestellt, um den gedanklichen Fokus von der eigenen Benutzung der Anwendung hin auf die didaktische Sinnhaftigkeit für Studierende zu richten und dies hierdurch besser nachvollziehen zu können. Nach Durchführung des Simplified-Thinking-Aloud wurde eine an Heuristiken orientierte Analyse durchgeführt, indem konkrete abschließende Fragen zur Erweiterung gestellt wurden. Die Ergebnisse der Expertenevaluation sollen in der nachfolgenden Sektion eruiert werden.

### 7.1.2 Ergebnisse

#### Evaluation des Gesamteindrucks

Das Game-Design der Anwendung wurde von den Experten als „gut im Workflow integriert“, „stilistisch etwas spielerisch“, daher „ein wenig anders als INLOOP“, aber auch von einem der Experten anfänglich als etwas überfordernd wahrgenommen. Bei Letzterem lag dies vor allem an der Menge der möglichen Code Smells, welche in der Wiki-Komponente der Erweiterung nach den Code-Doktoren aufgelistet sind. In der Lösungsansicht erwarteten zwei der drei Experten eine Möglichkeit, die Konsultation zu starten, welche jedoch nicht gegeben war, so dass zunächst über die Navigationsleiste und das Wartezimmer auf die Konsultation gewechselt werden musste. Ein Experte merkte an, dass die Darstellung der möglichen Punkte in der Aufgabenansicht, auch bedingt durch ein fehlendes Tutorial, nur wenig Mehrwert besäße und durch die erreichte Punktzahl ergänzt werden könnte. Das Narrativ wurde von den Experten als „witzig“ und „gut rüberkommend“ empfunden, vereinzelt wurden Elemente des Narratives jedoch nicht richtig interpretiert, so zum Beispiel, dass Lösungen „Patienten“ darstellen, oder, dass der Nutzer selbst die Rolle eines angehenden Doktors übernimmt. Von einem der Experten wurde anfänglich fehlinterpretiert, dass beim Start einer Konsultation möglicherweise andere Nutzer einbezogen werden. Als weitere Verbesserungsmöglichkeit wurde daher vorgeschlagen, die Erweiterung durch ein Tutorial zu ergänzen, welches die Grundprinzipien des Game-Designs erklärt und mögliche Missverständnisse mitigt. Nach erstmaliger Benutzung der Gamification-Komponenten war es den Experten jedoch auch so möglich, die Spielprinzipien weitestgehend nachzuvollziehen und intuitiv zwischen den Ansichten zu wechseln.

#### Evaluation der Detektion und der Konsultation

Zur Evaluation der Code-Smell-Detektion wurden die Experten zunächst gebeten, in einer vorbereiteten Lösung nach potenziellen Fehlern zu suchen. Die Experten nannten zu der Lösung nach kurzer Zeit mehrere Code Smells. Nach Abschicken der Lösung öffneten die Experten die Konsultationsansicht und informierten sich über die von den Code-Doktoren angemerktten Code Smells. Das hierbei angewandte Punktesystem und, dass der Punktabzug von den Code Smells stammt, wurde von den Experten richtig interpretiert. Die detektierten Code Smells überschnitten sich hierbei mit den vorher genannten Code Smells, teils wurden vom Experten weitere Anmerkungen gemacht (zum Beispiel zur Einrückung), welche nicht detektiert wurden. Bestimmte Detektionen wurden von den Experten nicht im Voraus erkannt, so zum Beispiel das Vorhandensein eines öffentlichen Konstruktors in einer Klasse ohne Objektfunktionalitäten (Utility-Klasse). Manche Detektionen wurden als „abhängig von der persönlichen Meinung zum Stil“ eingeschätzt. Dies stellt ein Problem dar, welches bereits durch Dietz et al. bei der Auswahl der Codierungsrichtlinien aus QualityReview versucht wurde, zu mitigieren [Die+18]. Bei der konkreten Anmerkung des Experten handelte es sich um die Prüfung auf das Vorhandensein von geschweiften Klammern um ein if-Statement, welche jedoch nicht nur stilistisch begründet ist, sondern auch in der potenziellen Degradation der Wartbarkeit (siehe Abschnitt 5.1.3). Die hierfür durch die Erklärungsdatenbank ergänzten Erklärungen der Code Smells wurden von den Experten als vor allem für unerfahrene Studierende sinnvolle Ergänzungen zu den Titeln der Detektionen bewertet. Im oben genannten konkreten Beispiel bei der Detektion der geschweiften Klammern um ein if-Statement klärte sich der Grund hinter der Detektion auch durch das Lesen der Erklärung. Das Verständnis der Erklärungen ist laut den Experten auch abhängig von

der Programmiererfahrung, da bestimmte Fachbegriffe unter Umständen nicht bekannt seien. Als weitere Verbesserungsmöglichkeit der Erklärungen schlugen die Experten daher die Ergänzung von konkreten Code-Beispielen zu den jeweiligen Regeln vor, welche die Grundgedanken hinter diesen, vor allem für unerfahrene Programmierer, besser illustrieren könnten.

### Evaluation der Motivationswirkung

Nach Ende der anfänglichen Kennenlernphase bestätigten die Experten das Vorhandensein des Bedürfnisses, die Codequalität der Beispielaufgabe zu verbessern und zu perfektionieren. Einer der Experten führte selbstständig (ohne Instruktion von außen) mehrere Iterationen zwischen der Konsultation und der Bearbeitungsansicht durch, um für die Beispielaufgabe eine „gesunde“ Lösung zu erhalten. Die Motivation für diese kontinuierliche Reiteration der Lösung begründete sich laut des Experten auf dem kompetitiven Charakter des Game-Designs und darauf, dass er hierbei „perfektionistisch“ sei. Der Experte wollte „im Leaderboard erster sein“ und durch die Verbesserung der Lösung möglichst viele „Punkte“ und „Level“ erreichen. Auch die anderen Elemente der Gamification kontribuierten hierbei, so wurde zum Beispiel den erreichbaren Errungenschaften im positiven Sinne ein „gewisser Sammelcharakter“ zugesprochen. Die Benennung der Errungenschaften wurde als witzig und auflockernd empfunden. Die Experten versuchten intuitiv, die rätselhaft formulierten Beschreibungen der Errungenschaften zu entschlüsseln und zeigten dadurch wiederum verstärktes Interesse an diesen. Zum Schluss wurden die Experten befragt, ob die Erweiterung bei Studierenden motivierend wirken könnte, so dass diese selbstständig Code Smells identifizieren und refaktorisieren. Die Experten stimmten dem zu und bewerteten das Gamification-Konzept als sinnvolle Erweiterung für INLOOP, um die Codequalität mehr in den didaktischen Fokus zu integrieren. Gleichzeitig äußerten die Experten die Vermutung, dass dies möglicherweise nicht gleich auf alle Studierende wirken würde. Studierende, welche „lediglich Bonuspunkte für die Klausur erreichen“ wollten, oder generell ein geringes Interesse an der Bearbeitung der INLOOP-Aufgaben zeigten, würden vermutlich im Gegensatz zu engagierteren und „an Knobelei interessierten“ Studierenden eher weniger durch die Gamification-Erweiterung motiviert werden oder diese nur selten nutzen, so die Experten. Zusammenfassend befanden die Experten die Gamification-Erweiterung jedoch „definitiv [für] eine Verbesserung“, um die laut den Experten wichtige Codequalität mehr in den Vordergrund zu rücken und die „Fehleranfälligkeit zu reduzieren“.

## 7.2 Studierendenevaluation

### 7.2.1 Evaluationsstrategie und Durchführung

Die Studierendenevaluation wurde auf Basis von Fragebögen umgesetzt. Die Fragebögen enthielten drei Sektionen, darunter zunächst allgemeine Fragen zum Interesse an der Berücksichtigung einer guten Codequalität bei dem Einreichen von Lösungen, danach die Bewertung der Sinnhaftigkeit einer Violation sowie deren Erklärung anhand eines Codebeispiels, und schließlich konkrete Fragen zur Motivation zentraler Elemente des Game-Designs. Um im mittleren Teil eine hinreichende Aussagekraft über die Qualität der Beschreibungen aus der Erklärungsdatenbank zu erreichen, wurden zu öffentlich verfügbaren INLOOP-Lösungen verschiedener Nutzer mithilfe der integrierten Analyse- und Parser-Infrastruktur verschiedene Violations und deren dazugehörigen Erklärungen ermittelt, um diese im Rahmen der Fragebögen Nutzern individuell zu präsentieren. Die gefundenen Violations wurden jeweils einer Umfrage zugeordnet, so dass 1006 mögliche Umfragen generiert wurden. Die Umfragen wurden anschließend auf der Plattform fragenautom.at<sup>1</sup> integriert<sup>2</sup>, jeweils versehen mit einem individuellen Zugangsschlüssel. Die individuellen, anonymisierten Umfragelinks wurden per E-Mail auf Grundlage von §18 Absatz 2 der IT-Ordnung der TU Dresden an 243 INLOOP-Nutzer verteilt. Bei

<sup>1</sup>fragenautom.at. <https://fragenautom.at/> (Abgerufen am 1.9.2020)

<sup>2</sup>GitHub. <https://github.com/inloop-gamified/website> (Abgerufen am 14.9.2020)

der Selektion der Nutzer wurde das Kriterium angewandt, dass diese im laufenden Sommersemester 2020 mindestens eine INLOOP-Aufgabe des „Exam“ Schwierigkeitsgrads gelöst haben mussten, um sicherzugehen, dass sich diese Nutzer bereits ausreichend mit der Basisplattform und deren Prinzipien auskennen. Die Ergebnisse sind in den Folgenden Sektionen visuell aufbereitet.

### 7.2.2 Ergebnisse

#### Evaluation der Detektion

Zur Detektion der Violations wurden öffentlich verfügbare oder für diesen Zweck zur Verfügung gestellte Lösungen von vier verschiedenen Nutzern mithilfe der in QualityReview bereitgestellten Checkstyle-Regeln analysiert.

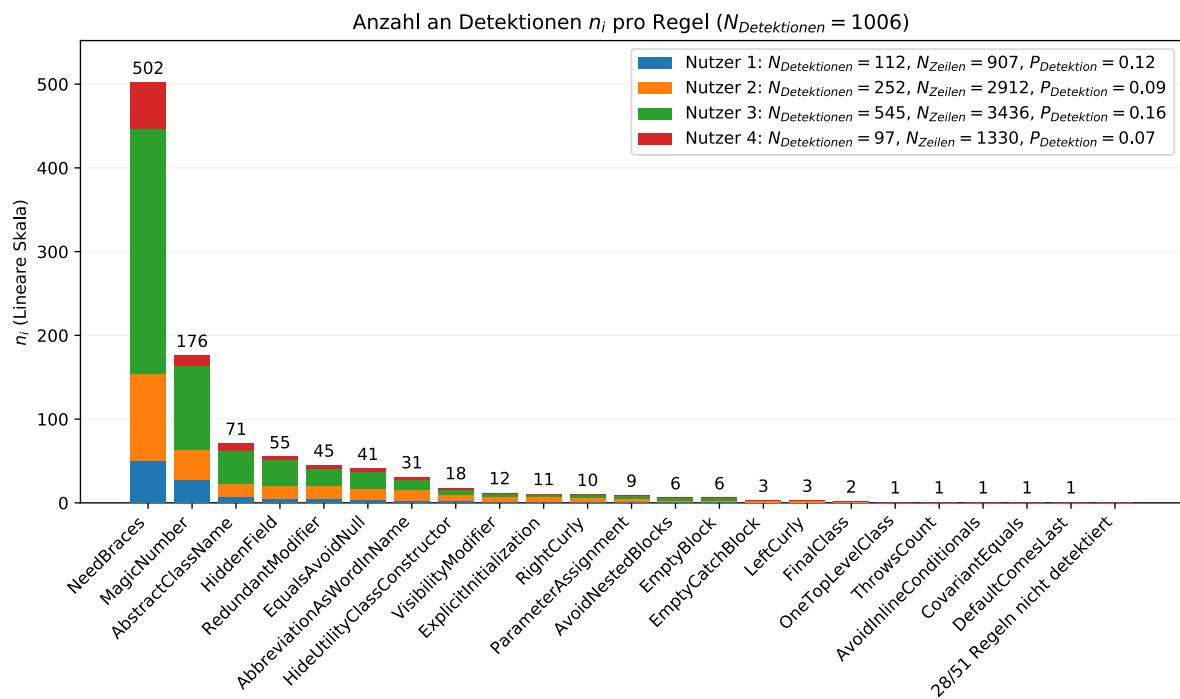


Abbildung 7.1 Absolute Detektionsraten von auf Violations abgebildeten Checkstyle-Regeln anhand von analysierten INLOOP-Lösungen.

In Abbildung 7.1 sind die absoluten Anzahlen der hierbei detektierten Violations anhand deren Identifikator gezeigt. Die Gesamtanzahl der analysierten Codezeilen betrug 8585 Zeilen, wobei insgesamt 1006 Detektionen erzeugt wurden. Damit ermittelt sich eine Sensitivität von einer Detektion pro 8,5 Codezeilen, oder eine Detektionswahrscheinlichkeit von 11,7%. Anhand der vertikalen Skala lässt sich erkennen, dass die mit Abstand am häufigsten erkannte Violation den Identifikator „NeedBraces“ besitzt. Diese Violation prüft auf das Vorhandensein von geschweiften Klammern um bestimmte Kontrollstrukturen. Bemerkenswert ist auch, dass mehr als die Hälfte der möglichen Violation-Identifikatoren in der Stichprobe nicht detektiert wurden. Im Kontext der Umfragegenerierung wurden die Detektionen nachfolgend zufällig für eine möglichst anwendungsnahe Evaluation ausgewählt.

#### Evaluation des Interesse an der Codequalität

**Anmerkung.** Die nachfolgend gezeigten Umfrageergebnisse wurden nach Heiberger und Robbins [HR14] visuell aufbereitet. Hierbei werden die Ergebnisse der Likert-Skalen in drei Sektionen unterteilt

(eher negativ, neutral, eher positiv), als Bar-Chart geplottet und entsprechend eingefärbt. Die Skala und Position kann hierbei variieren, im Rahmen dieser Evaluation wurden die Daten so skaliert, dass diese in der horizontalen Ausdehnung insgesamt 100% der Befragten ergeben. Diese Darstellung eignet sich besonders gut, da sich so an der 50% Marke direkt der Median als statistischer Mittelwert ablesen lässt.

Im einleitenden Teil der Umfragen wurde analysiert, inwiefern die Codequality bei der bisherigen Bearbeitung der INLOOP-Aufgaben bereits berücksichtigt wurde.



Abbildung 7.2 Umfrageergebnisse zur Berücksichtigung von Codequality bei der Lösung von INLOOP-Aufgaben.

Die statistische Aufbereitung in Abbildung 7.2 zeigt, dass die Codequality bei der Mehrheit der Befragten häufiger berücksichtigt wurde, während sieben Befragte selten und zwei Befragte noch nie auf ihre Codequality geachtet haben. Zum Empfinden der Wichtigkeit von Codequality beim Bearbeitungsprozess antworteten die Befragten differenzierter, wobei auch hier ein überwiegender Teil der Nutzer Codequality als eher wichtig einschätzen. Drei der Befragten antworteten, dass für sie Codequality unwichtig ist, während fünf der Befragten sie eher als unwichtig einschätzen würden.

### Evaluation der Erklärungsdatenbank

Für die Auswertung der Erklärungsdatenbank erhielt jeder Umfrageteilnehmer eine, im Voraus anhand von INLOOP-Lösungen generierte, Violation zur Bewertung der Sinnhaftigkeit der Detektion und der dazugehörigen Erklärung. Dazu wurde für die visuelle Repräsentation der Violation die entsprechende UI-Komponente der Konsultation aus der Gamification-Erweiterung nachempfunden, so dass sowohl Code, Code-Doktor, die in Checkstyle generierte Meldung sowie die Erklärung aus der Erklärungsdatenbank gezeigt wurden.

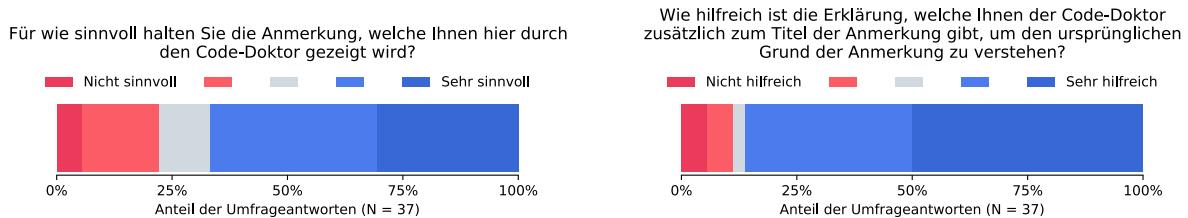


Abbildung 7.3 Umfrageergebnisse zur semantischen Code-Smell-Detektion.

In Abbildung 7.3 ist das kumulative Ergebnis dieser Analyse gezeigt, wobei etwas weniger als ein Viertel der Befragten die Checkstyle-Anmerkung als nicht sinnvoll oder eher nicht sinnvoll befanden. Die aus der Erklärungsdatenbank stammende Erklärung wurde hingegen überwiegend als hilfreich oder sehr hilfreich bewertet.

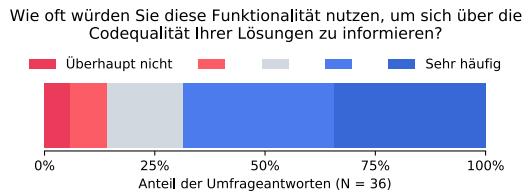


Abbildung 7.4 Umfrageergebnis zur Attraktivität der Code-Smell-Konsultation.

Nach Analyse der Attraktivität der Code-Smell-Konsultation in Abbildung 7.4 würden circa zwei Drittel der Nutzer diese Funktionalität sehr häufig oder eher häufig nutzen. Nur zwei der Nutzer würden diese Funktionalität überhaupt nicht nutzen wollen.

### Evaluation des Game-Designs

Im dritten Teil der Umfrage wurden abschließend wesentliche Elemente des Game-Designs evaluiert. Die Umfrageteilnehmer erhielten hierbei eine textuelle Einführung und kurze Erklärung und bewerteten hierzu anschließend einzelne Kernelemente des Game-Designs.

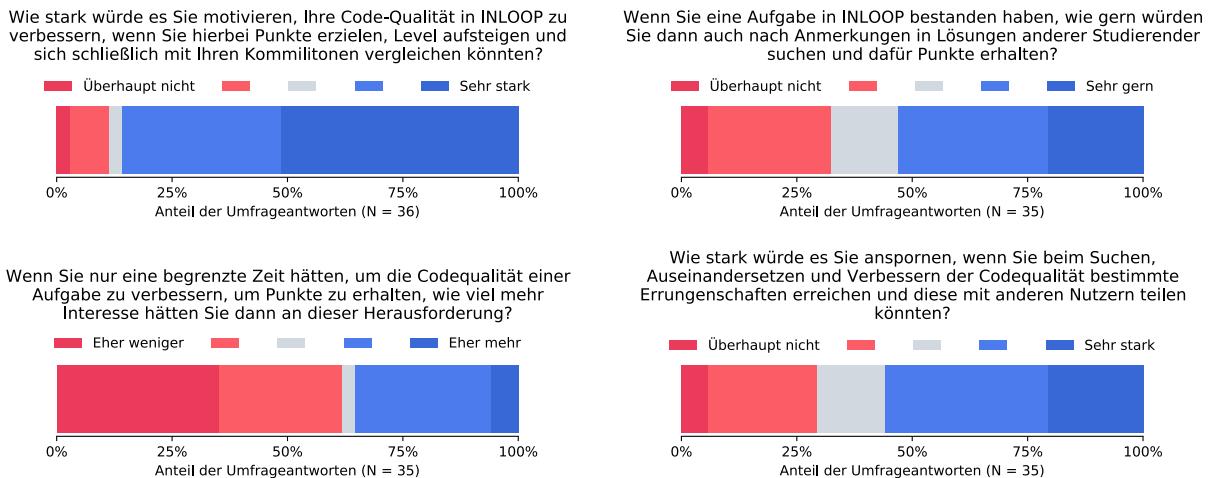


Abbildung 7.5 Umfrageergebnisse zu Kernelementen des Game-Designs.

In Abbildung 7.5 sind die Umfrageergebnisse hierzu gezeigt. Als besonders positiv und stark bis eher stark motivierend bei der Refaktorisierung schätzen die Befragten klassische Gamification-Elemente (Punkte, Level und Vergleichbarkeit) im Rahmen des Konzeptes ein, wobei nur ein Befragter antwortete, dass dies überhaupt nicht motivierend wirkte. Als überwiegend motivierend schätzen die Befragten auch das konzipierte Code-Smell-Quiz ein. Auf eine potenzielle Zeitbegrenzung bei der Refaktorisierung reagierten die Befragten hingegen überwiegend ablehnend, lediglich ein Drittel der Befragten schätzen dies als Möglichkeit ein, das Interesse hieran zu steigern. Als weiteres Element im Game-Design schätzen die Befragten schließlich auch die Errungenschaften und deren Teilbarkeit mit anderen Nutzern als überwiegend motivierend ein.

### 7.3 Gefahren für die Validität

Um die ausgewerteten Ergebnisse der an Likert-Skalen [Jos+15] angelehnten Fragen besser einschätzen zu können, wurden weitere Metaanalysen über dem Umfrageverhalten der Befragten durchgeführt, welche im Folgenden erläutert werden sollen. Insbesondere wurde hierbei analysiert,

inwiefern bei der jeweiligen Beantwortung eine tendenzielle Meinungsverzerrung hätte vorliegen können, beispielsweise durch eine allgemein sehr negative oder sehr positive Einstellung gegenüber der Softwaretechnologie-Lehre oder andere Verzerrungen, beispielsweise durch die im Zeitraum kurz vor der Umfrage geschriebene Klausur Softwaretechnologie an der TU Dresden, aus der möglicherweise Frust hätte entstehen können.

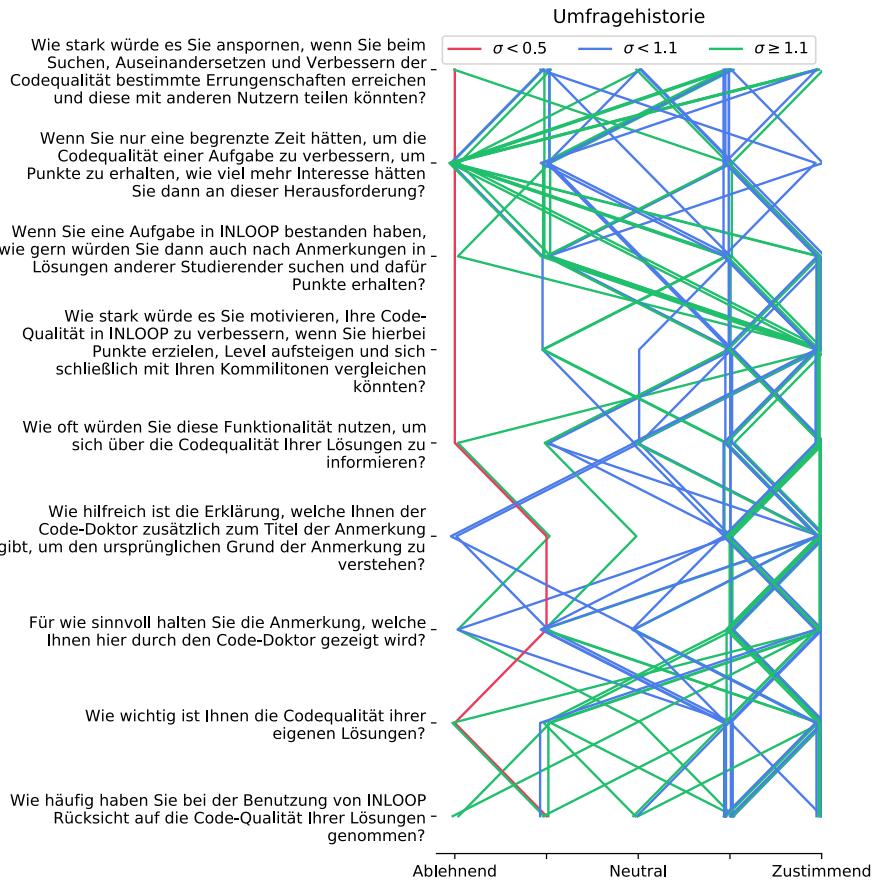


Abbildung 7.6 Diagramm über die Standardabweichungen der Antwortverläufe aus den erhobenen Umfragen.

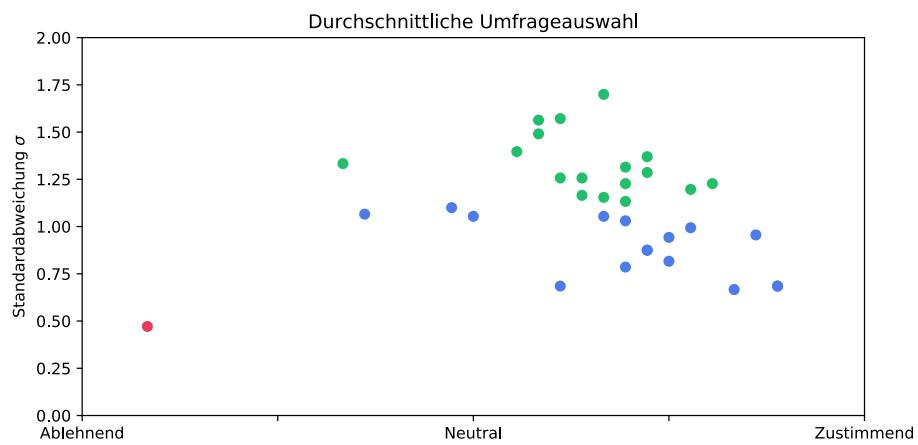


Abbildung 7.7 Scatter-Plot über die Standardabweichungen und durchschnittlichen Bewertungen der Antwortverläufe aus den erhobenen Umfragen. Farbschema nach Abbildung 7.6.

Um dies genauer nachvollziehen zu können, wurde das in Abbildung 7.6 gezeigte Diagramm aus den erhobenen Daten erstellt, welches die Umfragehistorien der Teilnehmer sowie die Standardabweichungen der Antworthistorien als Streuungsmaß um den Mittelwert der jeweiligen Historie darstellt. Eine der Umfragen sticht hierbei besonders heraus, mit einer Standardabweichung  $\sigma < 0.5$  von  $\sigma \approx 0.47$ . Diese Historie ist in Abbildung 7.6 rot markiert und zeigt eine durchgängig ablehnende oder eher ablehnende Antwortauswahl. Der Median aller Standardabweichungen liegt bei  $\tilde{\sigma} \approx 1.1$ . Im Scatter-Plot aus Abbildung 7.7 sind die Standardabweichungen anhand der durchschnittlichen Umfrageauswahlen (arithmetisches Mittel) gezeigt. Hierbei zeigt sich nochmals die durchgängig negativ bewertete Umfrage (rot markiert) als äußerst distanziert von einer klar erkennbaren Hauptgruppe. In Anbetracht dessen wäre eine denkbare Verbesserung des Umfragedesigns eine gegenüber solchen Meinungsverzerrungen robustere Fragenstellungsreihenfolge gewesen, in der die ablehnende und zustimmende Seite jeweils durch die Fragestellung gegeneinander alterniert worden wäre.

Im Allgemeinen ist zu beachten, dass die durchgeföhrte Umfrage als statistischer Indikator für die Motivationswirkung der Elemente aus dem Game-Design zu betrachten ist, nicht als Evidenz für eine kausale Wirkung. Im Kontext der Evaluation wurden die Gamification-Elemente zwar kurz erklärt, dies gibt den Nutzern jedoch eine gewisse Freiheit in der konkreten imaginären Vorstellung des Game-Designs. Hierfür wurde die Expertenevaluation anhand des konkreten Prototypen durchgeführt, welche die Motivationswirkung des Game-Designs anhand der benutzbaren Nutzeroberfläche bestätigen konnte. Die eingeladenen Experten haben jedoch einen anderen Kenntnisstand und andere Motivationen als Studierende, so dass bestimmte Teile des Game-Designs, wie die Erklärung der Code Smells, vermutlich bei Studierenden eine höhere Relevanz tragen. Daher wurde in der Konzeption der Umfragen besonders auf die konkrete Darstellung eines Teils einer Code-Smell-Konsultation (analog zum Prototypen) geachtet, um über die Qualität und Sinnhaftigkeit dieser Komponente eine qualifiziertere Aussage treffen zu können. Da die motivative Ausrichtung des Game-Designs hochkomplexe intrinsische, extrinsische, negative und positive Wirkungen erzeugt, ist nicht auszuschließen, dass bestimmte psychologische Effekte oder weitere mögliche Probleme nicht modelliert werden konnten. Ideal wäre hierfür der Einsatz der Erweiterung in der Produktionsumgebung und (hierauf basierend) die Durchführung entsprechend konzipierter Studien.

## 7.4 Zusammenfassung

Im Rahmen der Expertenevaluation konnte die prototypische Implementation des Gamification-Konzeptes zeigen, dass bei der Nutzung der über das Game-Design verbundenen Komponenten ein intuitives Bedürfnis entstehen kann, Code Smells in Lösungen selbstständig zu suchen und diese gegen Belohnungen wie Punkte, Level, Errungenschaften und Rangaufstieg in den Lösungen zu refaktorisieren. Hierbei entwickelte sich die Hypothese, dass diese Motivationswirkung möglicherweise nicht gleich stark auf Studierende mit verschiedenen Mentalitäten auswirken könnte. Bei der Expertenevaluation konnte darüber hinaus festgestellt werden, dass die Sicherstellung einer guten Nutzbarkeit der Erweiterung eine zentrale Komponente in der Wahrnehmung des Game-Designs repräsentiert. Über die Simplified-Thinking-Aloud Analyse konnten hierfür konkrete Elemente der Nutzerschnittstelle identifiziert werden, welche zur Verbesserung der Nutzbarkeit, zum Beispiel durch die Ergänzung von Informationen, Links oder eines (optionalen) Tutorials, beitragen könnten. Die Ergänzung von Beschreibungen im Rahmen der semantischen Code-Smell-Extraktion wurde von den Experten als sinnvoll bewertet, um die Hintergründe der Detektionen besser nachzuvollziehen. Als mögliche Verbesserung dieses Ansatzes wurde die Ergänzung von konkreten Code-Beispielen zu den jeweiligen Regeln identifiziert.

Durch die quantitativ orientierte Studierendenevaluation wird die Hypothese gestützt, dass die Erklärungen dazu beitragen können, Code Smells besser zu verstehen und hierdurch ein besseres Gesamtverständnis für gute Codequalität didaktisch zu prägen. Die erhobenen Daten suggerieren, dass Nutzer durch die Einführung bestimmter Gamification-Elemente motiviert werden könnten und prinzipiell bereits daran interessiert sind, die Codequalität eigener Lösungen in der Konsultation zu

verbessern. Die Befragten interessieren sich hierbei anhand der Umfrageergebnisse vor allem für das Erreichen von Punkten, Leveln und das Vergleichen mit anderen Nutzern. Auch gegenüber dem nicht prototypisch implementierten „Code-Smell-Quiz“ zu Lösungen anderer Nutzer sind die Befragten überwiegend positiv eingestellt. Lediglich auf zeitliche Limitationen bei der Refaktorisierung einer Lösung reagierten die Befragten überwiegend negativ. Insgesamt unterstützen auch die quantitativen Umfrageergebnisse die Hypothese, dass sich die in der Expertenevaluation festgestellten Motivationseffekte auch bei Studierenden einstellen würden. Die Validierung dieser Hypothese anhand der Finalisierung des Prototypen, dessen Integration und Produktionseinführung über einen zur Validation sinnvollen Zeitraum (zum Beispiel einem Semester) kann die in dieser Arbeit durchgeführte Evaluation als Gegenstand zukünftiger Forschung weiter fortsetzen.

# 8 Zusammenfassung

## 8.1 Ergebnisse

Um die Codequalität der eingereichten Lösungen in INLOOP zu verbessern, wurde eine Erweiterung entworfen, welche INLOOP um ein Gamification-Konzept ergänzt.

**Forschungsfrage 1** *Welche Codeequalitätsmetriken kommen hierfür in Frage und wie können diese im Kontext von INLOOP kombiniert und parametrisiert werden, um signifikante Fehlgestaltungen des Codes zu erkennen, welche die Codeequalität der eingereichten Lösungen beeinträchtigen?*

Bedingt durch die Komplexität und Subjektivität von Codeequalität existieren viele verschiedene Metriken, von denen die Konformität des Codes zu bestimmten Codierungsrichtlinien als vielversprechend korrelierend mit der wahrgenommenen Codeequalität aus den bisherigen wissenschaftlichen Analysen hervorgegangen ist [Ruc17][Die+18]. Die Konformitätsmetrik berechnet sich hierbei aus nach deren Schwere gewichteten Detektionen von Fehlgestaltungen („Code Smells“). Das Ziel des entworfenen Gamification-Konzeptes ist die Motivation der Nutzer, in eingereichten Lösungen konkrete Code Smells zu identifizieren und zu refaktorisieren, um die Codeequalität über die gesteigerte Konformität zu Codierungsrichtlinien und die hiermit zusammenhängenden Qualitätsfaktoren, vor allem die der Lesbarkeit und der Wartbarkeit, zu verbessern [eng01].

**Forschungsfrage 2** *Welche Gamification-Elemente sind dafür geeignet, die durch Codeequalitätsmetriken erkannten Fehlgestaltungen des Codes an die Nutzer zu kommunizieren, sodass diese motiviert werden, Fehlgestaltungen zu refaktorisieren, im Voraus zu vermeiden und damit die Codeequalität der eingereichten Lösungen zu verbessern?*

Die Gamification-Elemente wurden nach dem Octalysis Gamification-Framework [Cho19] im Rahmen eines Game-Designs durch ein nutzerzentriertes Konzept kombiniert. Nutzer können für eingereichte Lösungen eine bestimmte Anzahl an Punkten erreichen, von welcher je nach Schweregrad und Anzahl der aufgetretenen Code Smells Punkte abgezogen werden. Mithilfe von individuellen Code-Smell-Konsultationen wird die Konformität und hierdurch gleichzeitig auch potenzielle Degradationen der Codeequalität an Nutzer kommuniziert. Gemeinsam mit progressiven Ranglisten, erreichbaren Erfahrungsstufen, Errungenschaften, Avataren, dem Hinzufügen von Kollegen und dem direkten Feedback über ein Notifikationssystem wurden die gewählten Gamification-Elemente in einem Narrativ integriert und nach dem Octalysis-Framework gewichtet, sodass hierbei zur Ausgeglichenheit sowohl positive/negative, als auch extrinsische und intrinsische Elemente vertreten sind. Hierbei wurden Game-Design-Elemente und Konzepte verwandter Arbeiten, insbesondere der Arbeit von

Händler und Neumann [HN19b] sowie Dos Santos et al. [San+19] an verschiedenen Stellen wieder verwendet, beispielsweise in Form von interaktiven Fragen zu Lösungen anderer Nutzer. Im Fokus des nutzerzentrierten Ansatzes stand bei der Konzeption stets die Motivation des Nutzers, sich über potenzielle oder tatsächliche Degradationen der Codequalität seiner Lösungen und den Lösungen anderer zu informieren, und selbstständig Wissen zu akquirieren und schließlich auf die eigenen Lösungen anzuwenden.

**Forschungsfrage 3** *Wie können die Codequalitätsmetriken mit den Gamification-Elementen in einer Gamification-Erweiterung kombiniert werden?*

Als technische Grundlage der Gamification-Erweiterung dient eine didaktisch durch Erklärungen gestützte semantische Code Smell Detektion im Zusammenspiel mit der statischen Codeanalyse mithilfe von im akademischen Kontext entwickelten Regelwerken für Codierungsrichtlinien [Die+18]. Auf Verstöße gegen diese Richtlinien wird beim Einreichen einer Aufgabe automatisiert geprüft. Die hierbei entstandenen Detektionen werden in einem Violation-Modell vereinheitlicht und durch textuelle Erklärungen ergänzt, um diese schließlich in individuellen Code-Smell-Konsultationen sinnvoll darstellen zu können.

**Forschungsfrage 4** *Welche Auswirkungen hat die Einführung der Gamification-Erweiterung auf die Motivation, die Codequalität eingereichter Lösungen zu verbessern?*

Mithilfe einer zweikomponentigen Evaluation wurde das Gamification-Konzept anhand einer prototypischen Implementation auf dessen Motivationswirkung untersucht. Durch eine Expertenevaluation nach Prinzipien des Discount Usability Engineering konnte gezeigt werden, dass die Benutzung des Prototypen ein intuitives Interesse am Entdecken, Lernen und zur Refaktorisierung von Code Smells anregen kann. Die erhobenen Umfragedaten der Studierendenevaluation unterstützen diese Hypothese insgesamt, insbesondere zusätzlich auch die Sinnhaftigkeit und Notwendigkeit der Ergänzung von Erklärungen zu den detektierten Code Smells, um die Hintergründe der individuellen Detektionen besser nachvollziehen zu können.

## 8.2 Ausblick

Die erhobenen Evaluationsergebnisse suggerieren, dass eine Finalisierung des Prototypen sowie dessen Produktionseinführung zur Integration der Codequalität zum fakultativ orientierte didaktische Konzept von INLOOP beitragen und die Codequalität der eingereichten Lösungen signifikant verbessern kann. Hierzu wurden in der durchgeführten Expertenevaluation weitere Verbesserungsmöglichkeiten der Gamification-Erweiterung identifiziert, beispielsweise eine Ergänzung der semantischen Code-Smell-Detektion um konkrete Codebeispiele. Eine Studie über die Effekte und die Nutzung der sich in Produktion befindlichen Gamification-Erweiterung wäre denkbar und könnte zum wissenschaftlichen Verständnis der didaktischen Wirksamkeit von Gamification in der universitären Lehre mit Bezug auf Codequalität beitragen, zu der bisher nur wenige Konzepte und kaum Evidenz vorliegen [San+19]. Außerdem könnte die Gamification-Erweiterung auf andere Bereiche der Softwarequalität ausgedehnt werden, beispielsweise auf die Motivation von besonders performantem Code.

Ein weiteres Problem stellt die für diese Arbeit grundlegende Detektion von Code Smells über statische Codeanalysetools dar, welche durch eine verhältnismäßig hohe falsch-positive Detektionsrate geprägt ist [Yan+19]. Yang et al. schildern, 35 bis 91 Prozent der Detektionen in Codebeispielen würden von erfahrenen Entwicklern als „nicht zu behandeln“ eingestuft werden. Die stichprobenartige Ermittlung von Code Smells zur Generierung von Umfragen im Rahmen der Studierendenevaluation zeigt, dass die Sensitivität der Detektion in INLOOP-Lösungen bereits bei der Einbeziehung von nur einem statischen Codeanalysetool (in diesem Fall Checkstyle) recht hoch ist. Um die Genauigkeit zu verbessern, bietet sich damit eine geeignete Filterung nach bestimmten Heuristiken an, welche

falsch-positive Detektionen erkennen. Ein etablierter Ansatz hierfür ist die Einstufung der Detektionen nach bestimmten statistischen Ranking-Systemen [Eng+01][All+12]. Yang et al. schlagen als Lösung des Problems einen Machine-Learning-Algorithmus zur binären Klassifikation der Detektionen in die Kategorien „unberechtigt“ und „berechtigt“ vor, wobei der Algorithmus zunächst automatisiert auf Änderungshistorien von bekannten Bibliotheken vortrainiert und danach durch einen Experten iterativ mit Klassifikationen angepasst wird [Yan+19]. Anhand dieser Erkenntnisse könnte ein durch Machine Learning getriebenes Ranking-Framework für die Code-Smell-Detektion weiter dazu beitragen, Code Smells kontextsensitiv vorauszuwählen und somit die Präzision der statischen Codeanalyse zu verbessern.

# Literatur

- [AKB00] Lorin W. Anderson, David R. Krathwohl und Benjamin S. Bloom. „A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom’s Taxonomy of Educational Objectives“. In: 2000.
- [All+12] Simon Allier u. a. „A Framework to Compare Alert Ranking Algorithms“. In: Okt. 2012, S. 277–285.
- [AM18] Manal M. Alhammad und Ana M. Moreno. „Gamification in software engineering education: A systematic mapping“. en. In: *Journal of Systems and Software* 141 (Juli 2018), S. 131–150.
- [AS14] Bilal Sercan Akpolat und Wolfgang Slany. „Enhancing software engineering student team engagement in a high-intensity extreme programming course using gamification“. In: *2014 IEEE 27th Conference on Software Engineering Education and Training (CSEE T)*. Apr. 2014, S. 149–153.
- [Aßm10] Uwe Aßmann. *Modulbeschreibung Softwaretechnologie, INF-D-240*. de. 2010.
- [Bis20] João Bispo. <https://github.com/specs-feup/kadabra>. Juni 2020.
- [Blo56] Benjamin S. Bloom. *Taxonomy of Educational Objectives - The Classification of Educational Goals*. en. 1956.
- [BM19] Simon Baars und Sander Meester. „CodeArena: Inspecting and Improving Code Quality Metrics using Minecraft“. In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. Mai 2019, S. 68–70.
- [BN19] Kay Berkling und Katja Neubehler. „Presenting an Open-Source Platform for Supporting Gamified Class Teaching with Peer Reviews“. In: *2019 IEEE Global Engineering Education Conference (EDUCON)*. Apr. 2019, S. 245–252.
- [Boe88] B. W. Boehm. „A spiral model of software development and enhancement“. In: *Computer* 21.5 (Mai 1988), S. 61–72.
- [CAR17] Luís Cruz, Rui Abreu und Jean-Noel Rouvignac. „Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring“. In: Mai 2017, S. 205–206.
- [Cho19] Yu-kai Chou. *Actionable Gamification: Beyond Points, Badges, and Leaderboards*. en. Packt Publishing Ltd, Dez. 2019.
- [Cri07] Geoffrey Crisp. *E-Assessment Handbook*. Englisch. London ; New York: Bloomsbury Academic, Juli 2007.
- [CSF18] Pedro Campolina, Maurício Souza und Eduardo Figueiredo. „Games and Gamification in Software Engineering Education: A Survey with Educators“. In: Okt. 2018.

- [Det+11] Sebastian Deterding u. a. „From game design elements to gamefulness: defining "gamification"“. In: *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*. MindTrek '11. Tampere, Finland: Association for Computing Machinery, Sep. 2011, S. 9–15.
- [Die+18] Linus W. Dietz u. a. „Teaching Clean Code“ In: *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018 (SE 2018), Ulm, Germany, March 06, 2018*. Hrsg. von Stephan Krusche u. a. Bd. 2066. CEUR Workshop Proceedings. CEUR-WS.org, 2018, S. 24–27.
- [Ele+16] Leonard Elezi u. a. „A game of refactoring: Studying the impact of gamification in software refactoring“ In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. XP '16 Workshops. Edinburgh, Scotland, UK: Association for Computing Machinery, Mai 2016, S. 1–6.
- [Eng+01] Dawson Engler u. a. „Bugs as deviant behavior: a general approach to inferring errors in systems code“ In: *ACM SIGOPS Operating Systems Review* 35.5 (Okt. 2001), S. 57–72.
- [eng01] Technical Committee ISO/IEC JTC 1/SC 7 Software and systems engineering. *ISO/IEC 9126-1:2001*. en. 2001.
- [eng17] Technical Committee ISO/IEC JTC 1/SC 7 Software and systems engineering. *ISO/IEC/IEEE 12207:2017*. en. 2017.
- [Fac20] Facebook. <https://github.com/facebookarchive/pfff>. Juni 2020.
- [Fow+99] Martin Fowler u. a. *Refactoring: Improving the Design of Existing Code*. Englisch. 1. Aufl. Reading, MA: Addison Wesley, Juni 1999.
- [Glo16] Boris Gloer. *Scrum: Produkte zuverlässig und schnell entwickeln*. de. Carl Hanser Verlag GmbH Co KG, Juni 2016.
- [HLD18] Simon Harrer, Jorg Lenhard und Linus Dietz. *Java by Comparison: Become a Java Craftsman in 70 Examples*. Englisch. Raleigh, North Carolina: O'Reilly UK Ltd., Apr. 2018.
- [HN19a] Thorsten Haendler und Gustaf Neumann. „A Framework for the Assessment and Training of Software Refactoring Competences“ en. In: *Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*. Vienna, Austria: SCITEPRESS - Science und Technology Publications, 2019, S. 307–316.
- [HN19b] Thorsten Haendler und Gustaf Neumann. „Serious Refactoring Games“ eng. In: Jan. 2019.
- [HNS19] Thorsten Haendler, Gustaf Neumann und Fiodor Smirnov. „An Interactive Tutoring System for Training Software Refactoring“ en. In: *Proceedings of the 11th International Conference on Computer Supported Education*. Heraklion, Crete, Greece: SCITEPRESS - Science und Technology Publications, 2019, S. 177–188.
- [HR14] R. Heiberger und Naomi B. Robbins. „Design of Diverging Stacked Bar Charts for Likert Scales and Other Applications“ In: (2014).
- [HS12] Jürgen Handke und Anna Maria Schäfer. *E-Learning, E-Teaching und E-Assessment in der Hochschullehre: Eine Anleitung*. de. Walter de Gruyter, Dez. 2012.
- [ILD20] Theresia Devi Indriasari, Andrew Luxton-Reilly und Paul Denny. „Gamification of student peer review in education: A systematic literature review“ en. In: *Education and Information Technologies* (Mai 2020).
- [Jos+15] Ankur Joshi u. a. „Likert Scale: Explored and Explained“ en-US. In: *Current Journal of Applied Science and Technology* (Feb. 2015), S. 396–403.
- [KNO12] Philippe Kruchten, Robert L. Nord und Ipek Ozkaya. „Technical Debt: From Metaphor to Theory and Practice“ In: *IEEE Software* 29.6 (Nov. 2012), S. 18–21.

- [KS18] Stephan Krusche und Andreas Seitz. „ArTEMiS: An Automatic Assessment Management System for Interactive Learning“. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE ’18. Baltimore, Maryland, USA: Association for Computing Machinery, Feb. 2018, S. 284–289.
- [Luc+14] Andrés Lucero u. a. „Playful or Gameful?: creating delightful user experiences“. en. In: *interactions* 21.3 (Mai 2014), S. 34–39.
- [MA18] Soo Mei Teng und Hazleen Aris. „Game-Based Learning in Requirements Engineering: An Overview“. In: Nov. 2018.
- [MD18] Martin Morgenstern und Birgit Demuth. „Continuous Publishing of Online Programming Assignments with INLOOP“. In: *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018 (SE 2018), Ulm, Germany, March 06, 2018*. Hrsg. von Stephan Krusche u. a. Bd. 2066. CEUR Workshop Proceedings. CEUR-WS.org, 2018, S. 32–33.
- [Mek+13] Elisa D. Mekler u. a. „Do points, levels and leaderboards harm intrinsic motivation? an empirical analysis of common gamification elements“. In: *Proceedings of the First International Conference on Gameful Design, Research, and Applications*. Gamification ’13. Toronto, Ontario, Canada: Association for Computing Machinery, Okt. 2013, S. 66–73.
- [Mil56] George A. Miller. „The magical number seven, plus or minus two: some limits on our capacity for processing information“. In: *Psychological Review* 63.2 (1956), S. 81–97.
- [Mor+15] Alberto Mora u. a. „A Literature Review of Gamification Design Frameworks“. In: *2015 7th International Conference on Games and Virtual Worlds for Serious Applications (VS-Games)*. Sep. 2015, S. 1–8.
- [Net+19] Pedro Santos Neto u. a. „Case study of the introduction of game design techniques in software development“. In: *IET Software* 13.2 (2019), S. 129–143.
- [Nie94] Jakob Nielsen. *Usability Engineering*. en. Morgan Kaufmann, Okt. 1994.
- [Nie95] J. Nielsen. „Applying discount usability engineering“. In: *IEEE Software* 12.1 (Jan. 1995), S. 98–100.
- [Pal+15] Fabio Palomba u. a. „Landfill: An Open Dataset of Code Smells with Public Evaluation“. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. Mai 2015, S. 482–485.
- [Pau20] Raquel Pau. <http://walkmod.com/>. Juni 2020.
- [Ped+14] Oscar Pedreira u. a. „Gamification in software engineering – A systematic mapping“. In: *Information and Software Technology* 57 (Jan. 2014).
- [PLB18] Jevgenija Pantiuchina, Michele Lanza und Gabriele Bavota. „Improving Code: The (Mis) Perception of Quality Metrics“. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Sep. 2018, S. 80–91.
- [Ref20a] Refactoring.guru. *Data Class*. en. Juni 2020.
- [Ref20b] Refactoring.guru. *Inline Class*. en. Juni 2020.
- [Ref20c] Refactoring.guru. *Shotgun Surgery*. en. Juni 2020.
- [RF16] José Miguel Rojas und Gordon Fraser. „Code Defenders: A Mutation Testing Game“. In: *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2016, S. 162–167.
- [RH] Dr Linda H Rosenberg und Lawrence E Hyatt. „Software Quality Metrics for Object-Oriented Environments“. en. In: (), S. 6.
- [Rou20] Jean-Noël Rouvignac. <http://autorefactor.org>. en. Juni 2020.

- [Ruc17] Jan Rucks. *Erstellung eines (teil-)automatisierten Bewertungssystems für studentische Projekte im Softwarepraktikum*. Deutsch. 2017.
- [Sai+17] Michael Sailer u. a. „How gamification motivates: An experimental study of the effects of specific game design elements on psychological need satisfaction“. en. In: *Computers in Human Behavior* 69 (Apr. 2017), S. 371–380.
- [San+19] Hoyama Maria dos Santos u. a. „CleanGame: Gamifying the Identification of Code Smells“. en. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering - SBES 2019*. Salvador, Brazil: ACM Press, 2019, S. 437–446.
- [SBK11] Swapneel Sheth, Jonathan Bell und Gail Kaiser. „HALO (highly addictive, socially optimized) software engineering“. In: *Proceedings of the 1st International Workshop on Games and Software Engineering*. GAS ’11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, Mai 2011, S. 29–32.
- [SBK12] Swapneel Kalpesh Sheth, Jonathan Schaffer Bell und Gail E. Kaiser. „Increasing Student Engagement in Software Engineering with Gamification“. en. In: (2012).
- [Sou20] Sourcemaking.com. *Design Patterns and Refactoring*. en. Juni 2020.
- [SS17] Tushar Sharma und Diomidis Spinellis. „A Survey on Software Smells“. In: *Journal of Systems and Software* 138 (Dez. 2017).
- [Sta+02] Ioannis Stamelos u. a. „Code quality analysis in open source software development“. en. In: *Information Systems Journal* 12.1 (2002), S. 43–60.
- [Tec11] Technical Committee ISO/IEC JTC 1/SC 7 Software and systems engineering. *ISO/IEC 25010:2011*. en. 2011.
- [Ull20] Christian Ullenboom. *Java ist auch eine Insel: Das Standardwerk für Programmierer. Über 1.000 Seiten Java-Wissen. Mit vielen Beispielen und Übungen, aktuell zu Java 14*. Deutsch. 15. Aufl. Rheinwerk Computing, Juni 2020.
- [WS12] Hao Wang und Chuen-Tsai Sun. „Game Reward Systems: Gaming Experiences and Social Meanings“. In: (Mai 2012).
- [Yan+19] Xueqi Yang u. a. „An Expert System for Learning Software Engineering Knowledge (with Case Studies in Understanding Static Code Warning)“. In: *arXiv:1911.01387 [cs]* (Nov. 2019).

# **Anhang**

# A Code-Doktoren

Name	Fachrichtung (Englisch)	Zuständigkeit
John	Surgeon	Zuständig für grobe strukturelle Verstöße aus Tabelle B.1
Marc	Plastic Surgeon	Zuständig für stilistische Verstöße aus Tabelle B.2
Anne	General Practitioner	Zuständig für generelle Verstöße aus Tabelle B.3
Emily	Nurse	Erklärung des Game-Designs (Tutorial)
Felix	Nurse	Erklärung des Game-Designs (Tutorial)

Tabelle A.1 Integrierte Code-Doktoren in der Gamification-Erweiterung.

## B Erklärungsdatenbank

Identifikator	Erklärung (Englisch)
AvoidNestedBlocks	Nested blocks are often leftovers from the debugging process, they confuse a reader.
EmptyBlock	Empty blocks occur, when a code block is created but never used. They should be removed.
FinalClass	Classes with private constructors should be declared as final, because they are not initializable from an outside context.
HideUtilityClass	Utility classes (classes that contain only static methods or fields in their API) should not have a public constructor, because they are only accessed by their type and not by an object.
Constructor	
ThrowsCount	Throwing too many (different) exceptions makes the handling of these very complex. Apply polymorphism to the exception types (where possible) and reduce the amount of throw statements.
VisibilityModifier	Class members should be private, unless they are static final, immutable or specifically annotated, to enforce encapsulation.
ClassFanOutComplexity	Reduce the number of classes a given class relies on. Many classes from the standard library are excluded from this restriction, such as ArrayList or NullPointerException. A class with too many dependencies on other classes should be refactored.

Tabelle B.1 Erklärungen zu strukturellen Verstößen.

Identifikator	Erklärung (Englisch)
NeedBraces	Certain control flow elements, for which it is optional to use curly braces, should always use curly braces. Nesting more than one statement in control flow elements without braces will only influence the first one. Not using curly braces is error prone.
LeftCurly	Code blocks should always include left curly braces.
RightCurly	Right curly braces should be placed in the same line of a potentially following control flow elements, such as if-else or try-catch. Do not place statements in the same line after right curly braces.
AvoidInlineConditionals	Inline conditionals should be avoided, since they are hard to read.

DefaultComesLast	Java allows the default statement to be placed at any position in the switch statement. Placing it at the end makes the switch statement more readable.
HiddenField	Local variables should not shadow a field that is defined in the same class.
IllegalInstantiation	For certain classes, it is preferred to use a factory method instead of a constructor for performance reasons. For example, in the java.lang.Boolean class, constructor invocations should be replaced with the factory method Boolean.valueOf().
IllegalToken	Certain statements should not be used, because they make the code less readable, may lead to confusion or are deprecated in the given context.
InnerAssignment	All assignments should occur in their own isolated statement to increase readability. Inner assignments such as Integer.toString(i = 2) should be avoided.
MagicNumber	Numeric literals (except -1, 0, 1 and 2) should be defined as constants, i.e. as a variable or field with the final modifier such as static final int SECONDS_PER_DAY = 24 * 60 * 60".
MissingSwitchDefault	Switch statements should contain a default to account for cases that get introduced by future revisions. Default cases can also throw exceptions if they are never expected to be executed.
MultipleVariable Declarations	Each variable declaration should reside in its own statement and line. This increases readability.
OneStatementPerLine	Avoid multiple statements per line. It's very difficult to read multiple statements in one line.
OverloadMethods DeclarationOrder	Overloaded methods should be placed next to each other to increase readability. Overloaded methods are methods, which have the same name but different sets of parameters.
ModifierOrder	The order of modifiers should conform to the Java language specification. Declare modifiers in the following order: public, protected, private, abstract, default, static, final, transient, volatile, synchronized, native, strictfp. Annotations should be placed before any modifier.
RedundantModifier	In certain contexts, modifiers should be not explicitly specified, because they are already applied by the programming language. For example, interface definitions should not contain the public and abstract modifiers for method declarations.
AbbreviationAsWordIn Name	If any abbreviation (consecutive capital letters) is used in a specification, it should not exceed a certain length, to make the specification more readable.
AbstractClassName	Abstract classes should conform to naming conventions, such as being named Abstract...". Vice versa, classes named in this way should include the abstract modifier.
ClassTypeParameter Name	Class type parameters of generic classes should conform to naming conventions. For example, naming a type parameter "Tis ok, whereas naming it äbcßuch as in MyClass<abc> {} violates the conventions. Type parameters should match the regular expression "^[a-zA-Z]\$".

ConstantName	Constants (static and final fields or interface/annotation fields) should conform to naming conventions. For example, naming a constant "fooör" "BARis ok, whereas naming it "fooBARßsuch as in final static int fooBAR violates the conventions. Constants should match the regular expression "^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*\$".
MethodNaming	Methods should be named in "lowerCamelCase" to conform to the Java naming conventions. This improves readability and avoids confusion.

Tabelle B.2 Erklärungen zu stilistischen Verstößen.

Identifikator	Erklärung (Englisch)
EmptyCatchBlock	Exceptions should be handled in the corresponding catch block. Not handling an exception is error prone.
OneTopLevelClass	Each top level class (such as public classes or the first class in a file), interface, enum or annotation should reside in its own source file.
CovariantEquals	Every class that declares its own equals(...) method should override the equals(Object o) method. This is to make sure that the equals(Object o) method of the Object type is overridden correctly and does, for example, not contain the type of the class itself such as in equals(Foo o) of the class Foo.
EmptyStatement	The empty statement ; could be placed by accident and lead to problems such as in if(...); do ...
EqualsAvoidNull	If a string variable may be null, avoid calling .equals(...) on it to compare it to another, non null string. Instead, call the .equals(...) method on the non null string such as in "Foo".equals(otherString).
EqualsHashCode	Classes that override either hashCode() or equals(...) should also override the other. Equal objects should always have the same hash code. Hash codes are used, for example, in hash based collections such as HashMap or HashSet.
ExplicitInitialization	Java initializes variables to default values (such as false for boolean or null for object types) before performing any initialization specified in the code. Therefore, if you explicitly initialize variables to their internal default value, Java will run this computation twice, resulting in a minor inefficiency.
FallThrough	In Java switch statements, when cases do not specify a break, return, throw or continue statement, they fall through to the case specified below them.
IllegalCatch	Certain exceptions should not be caught, such as a java.lang.Exception, because they are too general and/or may shadow problems in the control flow.
IllegalThrows	Certain exceptions should not be thrown, such as a java.lang.Exception, because they are too general and/or do not represent the error cause specifically enough.
ModifiedControlVariable	Control variables such as i in for(int i = 0;...) should not be modified manually inside the according control flow block.
NestedForDepth	Extensive nesting of for blocks decreases readability and may exponentially increase computational efforts.

NestedIfDepth	Extensive nesting of if blocks decreases readability and may lead to confusion about which branches will be executed in certain situations.
NestedTryDepth	Extensive nesting of try blocks decreases readability and may lead to confusion about which exceptions will be handled in certain situations.
ParameterAssignment	Assignments to parameters of a function is often considered poor programming practice, because the parameters may be mutated without knowledge of the outside caller and reused in a potentially erroneous context. Parameters can be declared final.
SimplifyBooleanExpression	Over complicated boolean expressions are hard to understand and should be simplified.
SimplifyBooleanReturn	Over complicated boolean return statements are hard to understand and should be simplified.
StringLiteralEquality	Always use .equals(...) instead of == or != when comparing strings, since the latter will only compare the object references instead of the actual values of the strings.
VariableDeclarationUsageDistance	Variables should be declared as near to their first usage, as possible.
BooleanExpressionComplexity	Boolean expressions with too many operators are hard to read, maintain and debug. Too complex expressions may be split into separate expressions.
CyclomaticComplexity	The control flow of a component should not have too many possible decision paths. Reduce the number of decision points (such as if, for, switch, ... statements).
JavaNCSS	The control flow of a component should not have too many executable statements. Reduce the number of executable statements.
NPathComplexity	The control flow of a component should not have too many possible decision paths. Reduce the number of decision points (such as if, for, switch, ... statements and complex boolean clauses).

Tabelle B.3 Erklärungen zu weiteren, generellen Verstößen.

# C Errungenschaften

Identifikator	Belohnung	Hinweis (Englisch)	Erklärung
First blood	500 Punkte	Consult a code doctor.	Belohnung für das erstmalige Konsultieren eines Code-Doktors über die Konsultationsfunktion.
Looking Good	250 Punkte	Don't be like House M.D. and show your face to the patients.	Belohnung für das Auswählen eines Avatars.
Clean Coder	400 Punkte	Have a healthy patient.	Belohnung für das erstmalige Einreichen einer Lösung ohne erkannte Violations.
Stalker	300 Punkte	Have an eye on your colleagues.	Belohnung für das erstmalige Hinzufügen eines Kollegen.
Explorer	350 Punkte	Get to know the code docs.	Belohnung für das erstmalige Besuchen der Wiki-Webseite der Code-Doktoren.

Tabelle C.1 Erreichbare Errungenschaften in der Gamification-Erweiterung.

## D Level

Level	Zu erreichende Punkte	Name (Englisch)
0	0	Student
1	1000	Resident
2	2500	Specialist
3	4250	Physician
4	6500	Senior Physician
5	8500	Chief Doctor
6	10000	Medical Director

Tabelle D.1 Erreichbare Level in der Gamification-Erweiterung.