Ruprecht-Karls-Universität Heidelberg
Institut für Informatik

Bachelorarbeit

Im Studiengang Angewandte Informatik

Automatisierte Empfehlung von Feature-Tags im Code

Maximilian Verclas
Matrikelnummer: 3079497

Betreuerin: Prof. Dr. Barbara Paech

Eingereicht am: 4. März 2019

Selbstständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.
Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht.
Die Arbeit ist in gleicher oder vergleichbarer Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Heidelberg, den 03.03.2019

____

Maximilian Verclas

Kurzbeschreibung

[Kontext und Motivation] Entwickler sind immer mehr auf Code-Empfehlungssysteme während der Programmierung der Software angewiesen. Code-Empfehlungssysteme wie zum Beispiel der Content Assist in Eclipse sind mittlerweile allgegenwärtig. Jedoch gibt es noch kein Empfehlungssystem für Feature-Tags im Code, welches Vorschläge bezüglich des Codekontextes generiert. Ein solches System wäre jedoch von Wichtigkeit, da Entwickler oft den passenden Feature-Tag für den Code nicht wissen oder Probleme haben, falls der Code mehr als ein Feature implementiert. **[Fragestellung]** Diese Arbeit analysiert zuerst verschiedene Ansätze bereits existierender Code-Empfehlungssysteme, auf welche Teile des Codes diese Ansätze angewendet werden und welche Ergebnisse für Precision und Recall diese dabei erzielen. **[Ansatz]** Es wird mit dem Ansatz der SimHash-Technik ein Eclipse Plug-In implementiert, welches dem Entwickler Empfehlungen für Feature-Tags vorstellt. Die Empfehlungen werden ihrer Relevanz bezüglich des Quellcodes absteigend sortiert angezeigt. **[Ergebnis]** Das Plug-In wird mittels zweier Studentenprojekte mit gegebenem Goldstandard evaluiert und zusätzlich bezüglich Precision und Recall optimiert. Die Ergebnisse mit 78,4% Precision beim Empfehlen des genau passenden Feature-Tags und mit 90,3% Recall beim Empfehlen der drei relevantesten Feature-Tags zeigen, dass bereits gute Werte für ein Code-Empfehlungssystem für Feature-Tags erzielt werden.

Abstract

[Context and motivation] Developers increasingly using code recommendation system when programming the software. Code recommendation systems such as the Content Assist in Eclipse are ubiquitous. However, is doesn´t exist a recommendation system for feature tags that generates recommendations about the code context. However, a recommendation system for feature tags would be important because developers often do not know the matching feature tag for the code or have problems if the code implements more than one feature. **[Question]** This paper analyses

different approaches of existing code recommender systems, which parts of the code these approaches are applied to, and what results are achieved for precision and recall. **[Approach]** An Eclipse plug-in based on the SimHash approach is implemented that recommends feature tags for the following source code. The recommendations are sorted in descending order by code relevance. **[Results]** The plug-in will be evaluated by two student projects with given gold standard and additional optimized for Precision and Recall. The results with 78.4% precision in recommending the exact feature tag and 90.3% recall by displaying the three most relevant feature tags demonstrate that the code recommender system for feature tags already achieves good results.

Inhalt

1 Einleitung und Motivation 7

1.1 Zielsetzung und Vorgehen 8

1.2 Aufbau der Arbeit 9

2 Grundlagen 10

2.1 Feature-Tags im Code 10

2.2 Code-Empfehlungssystem 12

2.3 Precision und Recall 13

2.4 F-Maß 14

2.5 Abstandmaß Hamming-Distanz 16

3 Literaturrecherche 17

3.1 Recherchefragen 17

3.2 Quellen und Rechercheverfahren 18

3.3 Durchführung der Recherche 18

3.4 Ergebnisse der Recherche 20

3.4.1 Verschiedene Ansätze 20

3.4.2 Beantworten der Recherchefragen 26

3.4.3 Vergleich verwandter Arbeiten 27

3.5 Auswahl eines Ansatzes für eigene Implementierung 28

4 Anforderungen und Entwurf 30

4.1 Grobanforderungen 30

4.2 Detailanforderungen 31

4.3 Qualitätsanforderungen 33

4.4 Klassendiagramm 34

4.5 Entwurf der Oberfläche 36

5 Implementierung 37
5.1 Vorüberlegungen.. 37
5.2 Entscheidungen 37
5.2.1 Technischer Rahmen 38
5.2.2 Algorithmus.. 38
5.2.3 Berechnung des Ähnlichkeitsmaßes zwischen zwei Codedateien 43
5.3 Qualitätssicherung 44
6 Evaluation 46
6.1 Studentenprojekte 46
6.2 Vorgehen 48
6.3 Ergebnisse 49
6.4 Zusammenfassung der Evaluation 51
7 Zusammenfassung 53
7.1 Fazit 53
7.2 Ausblick.. 54
Literaturverzeichnis.. 55
Anhang 57

1 Einleitung und Motivation

In Feature-orientierten Softwareprojekten wird die Software in ihre einzelnen enthaltenen Features zerlegt [1]. Das Ziel dabei ist es, eine strukturierte Software bezüglich dieser Features zu entwickeln. Ein Feature beschreibt eine funktionale oder nicht-funktionale Eigenschaft eines Softwareprojekts. Ein Feature wird durch sämtliche Dokumentationen, Anforderungen und Codes beschrieben. Um die Zugehörigkeit und Nachverfolgbarkeit all dieser Software Engineering Artefakte zu einem Feature zu gewährleisten, können diese auf unterschiedliche Arten verlinkt werden, zum Beispiel über Links zwischen den Artefakten [2] oder mittels Tags [3]. Da in den meisten Softwareunternehmen und Open-Source Projekten eine Kombination aus Issue-Tracking Systemen (ITS) und Versionskontrollsystemen (VCS) zur Unterstützung der Software Engineering Aktivitäten verwendet wird, ist eine Verlinkung zweier Artefakte zwischen den beiden Systemen IST und VCS schwierig und führt oft bei kleinen Änderungen zu einem signifikanten Mehraufwand [4]. Um nun die Nachverfolgbarkeit der Features zwischen den Issue-Tracking Systemen und den Versionskontrollsystemen eindeutig und effektiv zu gewährleisten, erfolgt dies über Feature-Tags [3]. Gegeben der Anforderungen wurde dieser Ansatz des Feature-Taggings in den Versionskontrollsystemen im Rahmen einer Studie mit Studenten evaluiert. Die Ergebnisse der Studie haben gezeigt, dass die Entwickler zwar in der Lage sind, richtige Feature-Tags für ihren Code zu vergeben, aber nur etwa 65% des Codes überhaupt mit Feature-Tags versehen sind. Daraus gingen zwei wesentliche Gründe hervor: Zum einen kennen die Entwickler oft den passenden Feature-Tag für den Code nicht und vergeben deshalb an diesen Codeabschnitt keinen. Und zum anderen tun sich die Entwickler schwer, passende Feature-Tags zu vergeben, falls der Code mehr als ein Feature implementiert. Um den Entwickler bei der Vergabe der Feature-Tags im Code zu unterstützen, kann ein Code-Empfehlungssystem dienen [5]. Basierend auf dem bereits existierenden Quellcode generiert dieses Code-Empfehlungssystem Vorschläge für Feature-Tags. Dabei muss herausgefunden werden, welche Teile des Quellcodes für die Ermittlung von Ähnlichkeiten zwischen bestimmten Codeabschnitten die besten Resultate erzielen. Die Feature-Tags der Codeabschnitte, die am ähnlichsten zu einer Codedatei sind, werden dem Entwickler

für diese Codedatei vorgeschlagen. Schwierigkeiten eines Code-Empfehlungssystems liegen jedoch darin, ob man eher Vollständigkeit oder Korrektheit braucht. Braucht der Entwickler eher Vollständigkeit, werden ihm mehr Feature-Tags auch mit weniger Ähnlichkeit vorschlagen. Wird hingegen eher Korrektheit benötigt, werden weniger Feature-Tags vorgeschlagen, welche jedoch mit höherer Wahrscheinlichkeit auch die passenden Feature-Tags sind. Da in der Regel jedoch beide Eigenschaften verlangt werden, muss hier ein Mittelmaß gefunden werden, welches beide Werte auf ein gleich hohes Niveau bringt.

1.1 Zielsetzung und Vorgehen

Das Ziel dieser Arbeit ist die Entwicklung eines Eclipse Plug-Ins zur automatisierten Empfehlung von Feature-Tags im Code. Das Plug-In unterstützt dabei den Entwickler bei der Vergabe von Feature-Tags innerhalb der Entwicklungsumgebung Eclipse. Um ein solches Code-Empfehlungssystem für Feature-Tags mit hoher Qualität implementieren zu können, wird zunächst eine Literaturrecherche durchgeführt, um bereits bestehende Code-Empfehlungssysteme aller Art zu analysieren. Dabei wird analysiert, wie die Relevanz der Empfehlungen berechnet wird, auf welche Teile des Codes dies angewendet wird und welche Precision und Recall diese Code-Empfehlungssysteme erzielen. Einer dieser Ansätze wird als Entwurf für eine eigene Implementierung des Plug-Ins ausgewählt. Dieses Plug-In wird mittels zweier Testprojekte mit gegebenen Goldstandards evaluiert und bereits während der Implementation auf Precision und Recall getestet, um diese Ergebnisse zu verbessern. Precision aus diesem Grund, dass wenn der Codeabschnitt nur ein Feature implementiert, auch dieses zugehörige Feature-Tag dem Entwickler als Erstes empfohlen werden soll. Allerdings muss auch der Recall in Betracht bezogen werden, da dem Entwickler alle passenden Feature-Tags empfohlen werden sollen, wenn der Code mehrere Features implementiert. Es muss also ein Mittelmaß gefunden werden, um für Precision und Recall gleichermaßen gute Resultate zu erzielen.

1.2 Aufbau der Arbeit

Kapitel 2 beschreibt die Grundlagen dieser Arbeit. Kapitel 3 behandelt die durchgeführte Literaturrecherche und begründet die Auswahl eines Code-Empfehlungssystems für die eigene Implementierung. Kapitel 4 stellt die an das zu entwickelte Plug-In gestellten Anforderungen und den Entwurf dazu vor. Kapitel 5 beschreibt die Implementierung des Plug-Ins und Kapitel 6 erklärt die durchgeführte Evaluation und präsentiert die Ergebnisse dieser Evaluation. Zuletzt schließt Kapitel 7 die Arbeit mit einem Fazit ab.

2 Grundlagen

Im folgenden Kapitel werden notwendige Grundlagen erläutert. In Abschnitt 2.1 wird der Begriff Feature-Tag im Code erklärt. Abschnitt 2.2 erläutert, was man unter einem Empfehlungssystem versteht. Abschnitt 2.3 beschreibt die Qualitätsmerkmale Precision und Recall für Empfehlungssysteme und Abschnitt 2.4 das F-Maß aus Precision und Recall. Zuletzt wird in Abschnitt 2.5 das Abstandsmaß Hamming-Distanz erläutert.

2.1 Feature-Tags im Code

Um den Begriff Feature-Tag erklären zu können, muss zuerst der Begriff Feature selbst und die Nachverfolgbarkeit der Features erläutert werden. Ein Feature in Feature-orientierten Softwareprojekten beschreibt eine Funktionalität oder ein Merkmal eines Softwareprojektes [3]. Softwareprojekte können mehrere Funktionalitäten enthalten. Features werden verwendet, um die verschiedenen Funktionalitäten oder Merkmale eines Softwareprojektes voneinander zu trennen und unterscheiden zu können. Des Weiteren können Features auch in Software Engineering Artefakten, welche diese Funktionalität des Feature beschreiben, vorkommen. Die Artefakte können jegliche Anforderungen, der implementierte Code selbst oder sonstige Dokumente, welche die Software beschreiben, sein.

Nachverfolgbarkeit ist die Möglichkeit, das Leben einer Anforderung vorwärts und rückwärts zu verfolgen [2]. Durch Nachverfolgbarkeit kann eingesehen werden, welche Artefakte zusammengehören. Zum Beispiel kann verfolgt werden, welche Codeabschnitte zu welchen Anforderungen, Dokumenten oder Features gehören.

Nachverfolgbarkeit kann entweder über Links zwischen den Artefakten [2] oder mithilfe von Tags hergestellt werden [3]. Als Grundlage dieser Arbeit dient die TAFT (Tagging Approach to support Feature Management) Methode von Seiler und Peach [3], mit der die Nachverfolgbarkeit zwischen den Artefakten geschaffen wird. Es werden alle Artefakte des Softwareprojekts mit dem entsprechendem Featurenamen versehen.

Innerhalb eines Issue-Tracking-Systems (ITS) wie zum Beispiel Jira[1] werden die Artefakte mittels Labels der entsprechenden Featurenamen versehen. Innerhalb eines Versionskontrollsystems (VCS) wie zum Beispiel git[2] erfolgt die Zuweisung der Features über Annotationen in der Codeebene. Eine solche Annotation im VCS oder ein Label im IST mit einem Feature ist ein Feature-Tag. Die Annotation im Code hat entweder die Form „@Feature(„ … ")", falls der folgende Codeabschnitt nur ein Feature implementiert oder die Form „@Feature({„ … ", „ … ", …})", falls der folgende Codeabschnitt mehrere Features implementiert.

Ein Beispiel für die Vergabe von Feature-Tags mit Annotationen im Code zeigt die Abbildung 1. Hier ist die Klasse *ElevatorManager* mit den Feature-Tags *RouteFinding* und *Filter* in Zeile 11 gekennzeichnet. Dies bedeutet, dass die folgende Klasse *ElevatorManager* Teile der Funktionalitäten von *RouteFinding* und *Filter* implementieren.

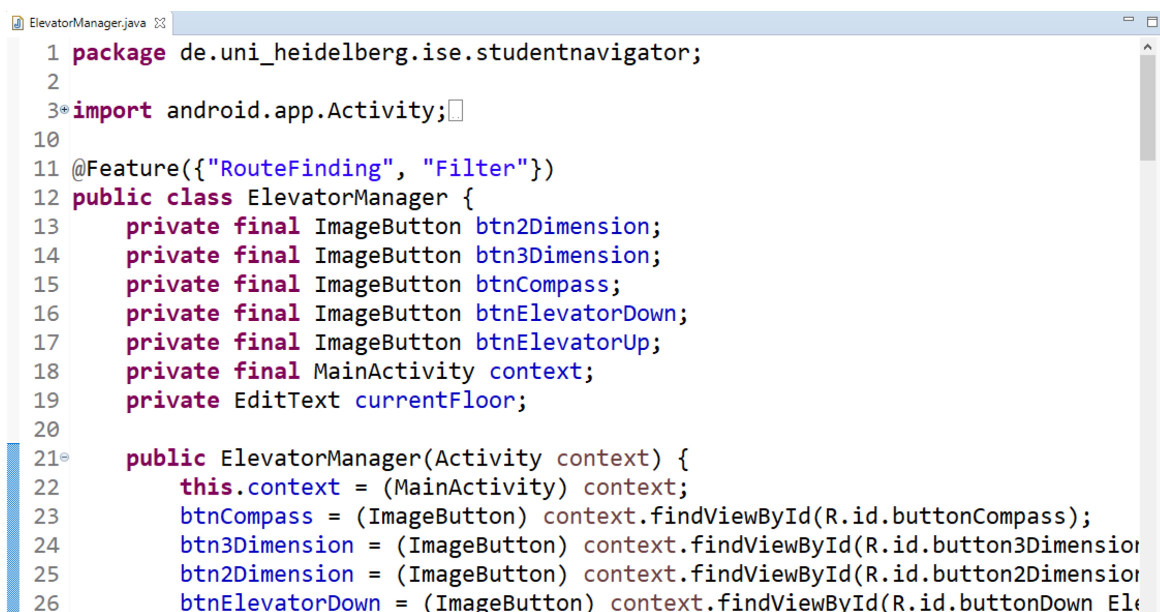


Abbildung 1: Beispiel aus dem Testprojekt *StudentNavigator* für die Vergabe von Feature-Tags in der Codeebene

[1] https://de.atlassian.com/software/jira
[2] https://git-scm.com/

2.2 Code-Empfehlungssystem

Ein Code-Empfehlungssystem ist eine Softwareanwendung, die dem Nutzer Informationen bereitstellt, die an einer bestimmten Stelle in einem bestimmten Kontext als wertvoll erachtet wird. Wenn die „Berechnung“ universell genau ist, ist das System kein Empfehlungssystem, sondern ein System zur Berechnung der richtigen Antwort [5]. Code-Empfehlungssysteme sind in der heutigen Programmierung ein gängiges Hilfsmittel beim Schreiben des Codes. Sie unterstützen den Entwickler bei der richtigen Auswahl von Methoden, Attributen etc. oder helfen bei der Verwendung von Methoden aus Bibliotheken, indem die Eingabe- und Ausgabewerte beschrieben werden. Ein gängiges Beispiel für ein Code-Empfehlungssystem ist der Content Assist in Eclipse[3]. Eclipse ist ein Programmierwerkzeug zur Entwicklung von Software verschiedenster Art. Es ist ein Open Source Projekt und bietet eine Vielzahl an Erweiterungen für den Entwickler wie zum Beispiel das Versionskontrollsystem git. Der Content Assist in Eclipse stellt dem Entwickler sämtliche Methoden, Attribute etc. vor, welche aufgrund der Syntax in einem bestimmten Kontext als verwendbar erachtet werden. Der Content Assist kann in allen beliebigen Situation mit der Tastenkombination „Strg + Leertaste“ aufgerufen werden. Des Weiteren wird der Content Assist in Eclipse automatisch aufgerufen, sobald der Entwickler durch das Schreiben eines Punktes einen Methodenaufruf oder eine Aktion auffordert. Der Content Assist stellt darauf alle Methoden alphabetisch sortiert zur Verfügung, welche auch aus syntaktischer Sicht in Frage kommen. Zusätzlich zu den Methoden selbst werden auch noch sämtliche Informationen dafür vom Content Assist bereitgestellt. Ein Beispiel dafür zeigt die Abbildung 2, bei welcher ein Methodenaufruf auf eine ArrayListe angefordert wird.

[3] https://www.eclipse.org/

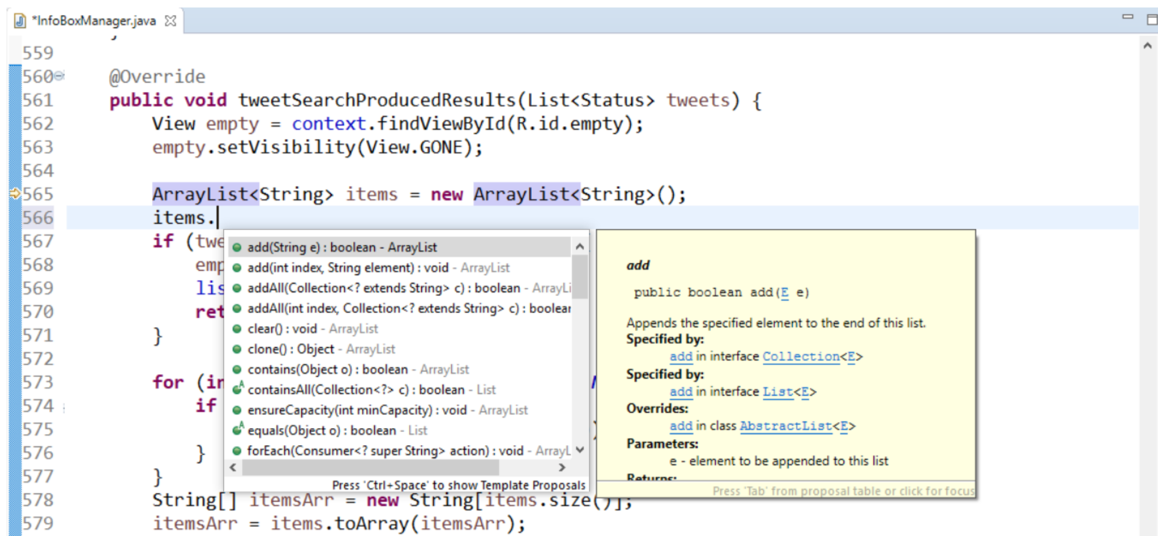


Abbildung 2: Beispiel des Code-Empfehlungssystems Content Assist in Eclipse

2.3 Precision und Recall

Ziel eines Code-Empfehlungssystems ist es, möglichst alle relevanten und dabei so wenig wie möglich irrelevante Empfehlungen wiederzugeben. Relevante Empfehlungen in Bezug auf Feature-Tags bedeutet, dass nur die Feature-Tags relevant sind, die auch zum aktuellen Codeabschnitt passen. Für eine gute Beurteilung der Qualität dieser Empfehlungen werden für Code-Empfehlungssysteme typischerweise Precision und Recall verwendet [5-8]. Sie werden wie folgt berechnet:

- Precision errechnet sich aus dem Verhältnis der vom Code-Empfehlungssystem angegebenen relevanten Empfehlungen zu der Gesamtanzahl der angegebenen Empfehlungen.

$$\text{Precision} = \frac{\text{Recommendations}_{\text{made} \cap \text{relevant}}}{\text{Recommendations}_{\text{made}}}$$

- Recall errechnet sich aus dem Verhältnis der vom Code-Empfehlungssystem angegebenen relevanten Empfehlungen zu der Anzahl der Empfehlungen, die es hätte geben sollen.

$$\text{Recall} = \frac{\text{Recommendations}_{\text{made} \cap \text{relevant}}}{\text{Recommendations}_{\text{relevant}}}$$

Aus den Code-Empfehlungssystemen der Literaturrecherche hat sich eine folgende Klassifikation ableiten können, in welche man die Werte für Precision und Recall einordnen kann [6-8]. Die Werte sind aus Code-Empfehlungssystemen für Methodenaufrufe hervorgegangen und dienen nur zur Eingliederung der eigenen Resultate:

- 80% oder höher (sehr guter Bereich): Für Precision schafften es drei der fünf Systeme nur knapp, einen Wert über 80% zu erzielen. Für Recall schaffte es lediglich nur ein System über 80%.
- 65% bis 80% (guter Bereich): Werte in diesem Bereich erzielen die meisten Code-Empfehlungssysteme, welche Empfehlungen aufgrund eines bestimmten Kontext berechnen.
- 50% bis 65% (mittelmäßiger Bereich): Werte in diesem Bereich zählen bereits zu den schlechteren Ergebnissen. Keines der Code-Empfehlungssysteme, welche den Kontext zur Bestimmung von Empfehlungen mit in Betracht zieht, ist in diesem Bereich oder schlechter für Precision und Recall gelandet.
- 50% oder niedriger (schlechter Bereich): Werte für Precision und Recall im Bereich knapp unter 50% schaffen es bereits Empfehlungssysteme, welche nicht einmal den aktuellen Kontext zur Berechnung von Empfehlungen miteinbeziehen.

2.4 F-Maß

Das F-Maß ist ein Mittel, welches Precision und Recall kombiniert. Das allgemeine F-Maß F_β ist definiert als (für reelle, nicht negative Werte von β) [5]:

$$F_\beta = \frac{(1+\beta^2) * (\text{Precision} * \text{Recall})}{(\beta^2 * \text{Precision} + \text{Recall})}$$

Oft wird β auf den Wert 1 gesetzt, wodurch das F_1-Maß erzeugt wird. Dadurch werden Precision und Recall gleichgewichtet. Das F_1-Maß ist definiert als [5]:

$$F_1 = \frac{2 * (\text{Precision} * \text{Recall})}{(2 * \text{Precision} + \text{Recall})}$$

Für die Evaluation des eigenen Code-Empfehlungssystems für Feature-Tags werden zusätzlich das F_2-Maß und das $F_{0,5}$-Maß benötigt. Diese sind wie folgt definiert [5]:

$$F_2 = \frac{5 * (\text{Precision} * \text{Recall})}{(4 * \text{Precision} + \text{Recall})}$$

Beim F_2-Maß wird der Recall im Vergleich zur Precision viermal so stark gewichtet. Dies wird bei der Evaluation beim Anzeigen von nur einem Feature-Tag benötigt, da hier die Precision mehr gewichtet, jedoch der Recall nicht ganz vernachlässigt werden soll.

$$F_{0,5} = \frac{1{,}25 * (\text{Precision} * \text{Recall})}{(0{,}25 * \text{Precision} + \text{Recall})}$$

Beim $F_{0,5}$-Maß wird die Precision im Vergleich zum Recall viermal so viel gewichtet. Dies wird bei der Evaluation beim Anzeigen von mehreren Feature-Tags benötigt, da hier der Recall mehr gewichtet werden soll, um zu zeigen, dass so viele Feature-Tags wie möglich getroffen wurden. Aber auch hier soll die Precision auch mit in das Ergebnis einfließen.

Zur Beurteilung der F-Maße konnte die gleiche Klassifizierung wie bei der Beurteilung der Werte für Precision und Recall aus der Literaturrecherche abgeleitet werden [6-8]. Auch hier sind die Werte aus Code-Empfehlungssystemen für Methodenaufrufe hervorgegangen und dienen nur zur Eingliederung der eigenen Resultate:

- 80% oder höher (sehr guter Bereich): Einen Wert über 80% für das F1-Maß schafft nur ein Empfehlungssystem.
- 65% bis 80% (guter Bereich): Werte in diesem Bereich erzielen die meisten Code-Empfehlungssysteme, welche Empfehlungen aufgrund eines bestimmten Kontext berechnen.
- 50% bis 65% (mittelmäßiger Bereich): Keines der Code-Empfehlungssysteme, welche den Kontext zur Bestimmung von Empfehlungen mit in Betracht zieht, hat Werte in diesem Bereich oder schlechter für das F_1-Maß erzielt.
- 50% oder niedriger (schlechter Bereich): Ergebnisse im Bereich unter 50% liegen nur Empfehlungssysteme, welche nicht einen bestimmten Kontext zur Berechnung von Empfehlungen miteinbeziehen.

2.5 Abstandmaß Hamming-Distanz

Die Hamming-Distanz ist ein Maß für die Unterschiedlichkeit von Zeichenketten. Sie ist die Anzahl der unterschiedlichen Stellen zweier Zeichenketten mit fester Länge. Oft, wie auch in dieser Arbeit, handelt es sich bei den Zeichenketten um Binärcodes. Also ist die Hamming-Distanz zweier Binärcodes die Anzahl der unterschiedlichen Stellen von Nullen und Einsen [7].
Beispiele:

00110	und	00100	->	Hamming-Distanz = 1
12345	und	13342	->	Hamming-Distanz = 2
Haus	und	Baum	->	Hamming-Distanz = 2

Die Hamming-Distanz wird im Laufe der Arbeit dazu verwendet, zwei Namen, welche zuvor in 8-Bit lange Binärcodes übersetzt wurden, auf ihre Gleichheit zu überprüfen. Ist die Hamming-Distanz beider Namen gleich Null, sind auch die Namen selbst identisch.
Ein weiteres Maß wäre zum Beispiel die Levenshtein-Distanz. Die Levenshtein-Distanz zwischen zwei Zeichenketten errechnet sich aus der minimalen Anzahl an Einfügungs-, Löschungs- und Ersetzungs-Operationen, um die erste Zeichenkette in die zweite Zeichenkette umzuwandeln [9]. Beispielsweise ist die Levenshtein-Distanz zwischen „Tier“ und „Tor“ gleich 2: Aus Tier wird Toer (Ersetze i durch o) und aus Toer wird Tor (Lösche e).
Mit der Levenshtein-Distanz können, wie im Beispiel gezeigt, auch Abstände von Zeichenketten mit unterschiedlichen Längen berechnet werden. Da im Laufe der Arbeit und in der entsprechenden Literaturrecherche nur gleich lange Binärcodes verglichen werden, reicht die Verwendung der Hamming-Distanz aus [6].

3 Literaturrecherche

Ziel dieser Arbeit ist die Entwicklung eines Ansatzes zur automatisierten Empfehlung von Feature-Tags im Code. Dies erfordert zunächst eine Recherche nach verschiedenen Ansätzen von Code-Empfehlungssystemen. Abschnitt 3.1 führt dazu die zu beantwortenden Recherchefragen auf. Abschnitt 3.2 beschreibt die verwendeten Quellen und das darauf basierende Rechercheverfahren. Abschnitt 3.3 stellt die resultierenden Ergebnisse daraus vor, und Abschnitt 3.4 beschreibt und vergleicht die verschiedenen Ansätze. Zuletzt wird in Abschnitt 3.5 basierend auf den Resultaten zuvor eine Auswahl eines Ansatzes für eine eigene Implementierung getroffen.

3.1 Recherchefragen

RF1: ***Welche inhaltlichen und strukturellen Aspekte von Codedateien werden zur* Generierung von Empfehlungen verwendet?** Diese Frage soll Antworten geben, welche Teile des Codes verwendet werden können, um eine Ähnlichkeit zwischen Codeabschnitten zu berechnen. Aus den inhaltlichen und strukturellen Aspekten, bei denen die Ähnlichkeit am besten berechnet werden konnte, sollen auch die Empfehlungen für Feature-Tags generiert werden.

RF2: ***Welche Ähnlichkeiten zwischen Codedateien werden zur Generierung von Empfehlungen verwendet?*** Dabei soll beantwortet werden, ob es verschiedene Ansätze zur Berechnung von Ähnlichkeit gibt und welche am besten zur automatisierten Empfehlung für Feature-Tags geeignet ist.

RF3: ***Welche Precision und welchen Recall bieten diese Ansätze?*** Diese Frage bezieht sich auf die Qualität der Code-Empfehlungssysteme. Dabei geben diese Werte Antworten auf die Qualität der verschiedenen Methoden der ersten beiden Fragen. Das zu entwickelnde Empfehlungssystem für Feature-Tags soll natürlich bestmögliche Ergebnisse ausgeben.

3.2 Quellen und Rechercheverfahren

Als erste Quellen für das weitere Rechercheverfahren dienen die Paper *Learning from examples to improve code completion systems* [6] von Bruch et al. (2009) und *CSCC: Simple, Efficient, Context Sensitive Code Completion* [7] von Asaduzzaman et al. (2014). Der Artikel von Bruch et al. [6] beinhaltet zwei verschiedene Ansätze, und der Artikel von Asaduzzaman et al. [7] einen Ansatz für Code-Empfehlungssysteme für Methodenaufrufe. Beide Publikationen geben dabei erste Antworten auf die Recherchefragen bezogen auf deren Verfahren. Als nächstes sind ausgehend von den beiden genannten Veröffentlichungen mittels Vorwärts- und Rückwärtssuche weitere wissenschaftliche Publikationen in Bezug auf die Recherchefragen zu finden und zu analysieren. Vorwärtssuche bedeutet, dass alle wissenschaftlichen Veröffentlichungen in Betracht bezogen werden, die mindestens einen der beiden Artikel referenzieren. Dagegen bedeutet Rückwärtssuche, dass alle Publikationen untersucht werden, die von einem der beiden gegebenen Paper referenziert wird.

3.3 Durchführung der Recherche

Die Durchführung der Vorwärts- und Rückwärtssuche erbrachte folgende Anzahlen an Publikationen (Stand Dezember 2018):

	Bruch et al. [6]	**Asaduzzaman et al. [7]**
Vorwärtssuche	79	11
Rückwärtssuche	27	31

Aufgrund dieser hohen Anzahl der zur Verfügung stehenden Artikel, werden diese durch die Kurzbeschreibung und danach, falls diese vielversprechend sind, durch das ganze Paper auf folgende Kriterien überprüft:

- **Methode für Code-Empfehlung:** Das Paper muss mindestens einen Ansatz vorstellen, welcher für ein Code-Empfehlungssystem eingesetzt wird.
- **Neuer Ansatz:** Die beschriebene Methode sollte einen neuen Ansatz im Vergleich zu den beiden gegebenen Artikeln bieten oder zumindest einen existierenden Ansatz erweitern bzw. verbessern.

Dieses Verfahren brachte zwei weitere relevante Veröffentlichungen, welche einen neuen Ansatz für Code-Empfehlung vorstellen (siehe Tabelle 1). *Intelligent Code Completion with Bayesian Networks* von Proksch et al. [10] und *Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion* von Nguyen et al. [8] referenzieren den Artikel von Bruch et al. [6]. Außerdem wird zusätzlich *Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion* von Nguyen et al. [8] von Asaduzzaman et al. [7] referenziert.

Tabelle 1: Übersicht der relevanten Veröffentlichungen

Nr.	Quelle	Titel der Veröffentlichung	Autoren	Jahr	Vorgehen
(1)	ACM Digital Library	Learning from examples to improve code completion systems[6]	M. Bruch et al.	2009	gegebener Artikel
(2)	IEEE Xplore	CSCC: Simple, Efficient, Context Sensitive Code Completion[7]	M. Asaduzzaman et al.	2014	gegebener Artikel
(3)	ACM Digital Library	Intelligent Code Completion with Bayesian Networks[10]	S. Proksch et al.	2015	Vorwärtssuche [6]
(4)	IEEE Xplore	Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion[8]	Nguyen et al.	2012	Vorwärtssuche [6] & Rückwärtssuche [7]

Insgesamt sind es damit fünf verschiedene Ansätze für Code-Empfehlungssysteme, da Bruch et al. [6] in seiner Publikation zwei verschiedene Ansätze vorstellt (vergleiche Tabelle 2). Dabei spezialisieren sich alle Ansätze auf Code-Empfehlungssysteme für Methodenaufrufe.

3.4 Ergebnisse der Recherche

Im Folgenden werden die verschiedenen Ansätze kurz beschrieben. Danach werden mittels Tabelle 2 die Recherchefragen beantwortet und zuletzt die Ansätze miteinander verglichen.

3.4.1 Verschiedene Ansätze

Frequenzbasiertes Code-Empfehlungssystem

Bei diesem System von Bruch et al. [6] wird ein ganz einfacher Ansatz verwendet, der nur die Häufigkeit der Verwendung einer Methode in Betracht zieht. Das System wird immer dann automatisch ausgeführt, wenn der Entwickler während der Programmierung einen Methodenaufruf durch einen Punkt in Eclipse anfordert. Dabei zählt nur die Anzahl der Verwendungen einer Methode. Je häufiger eine Methode verwendet wurde, desto wahrscheinlicher ist es, dass diese wiederverwendet wird. Die Top 3 werden dann in absteigender Reihenfolge als Ergebnis über den Empfehlungen von Eclipse angezeigt. Da dieser Ansatz nicht auf das Umfeld eines Methodenaufrufs, auch Methodenkontext genannt, eingeht, fällt dementsprechend die Qualität der Empfehlungen aus. Mit nur 34% Precision und 49% Recall und dem errechneten F1-Wert von 40,5% bildet das frequenzbasierte Code-Empfehlungssystem das Schlusslicht im Qualitätsranking.

Nr.	Titel der Methode	Inhaltlicher Aspekt	Struktureller Aspekt	Ähnlichkeiten zwischen den Codedateien	Trigger der Methode	Ausgabe/ Ergebnis	Precision, Recall & F_1-Maß
(1)	Frequenzbasiertes Code-Empfehlungssystem [6]	Alle Methodenaufrufe	Namen der Methodenaufrufe	Anzahl der Verwendungen eines Methodenaufrufs	Automatisch	Top 3	P: 34% R: 49% F_1: 40,5%
(2)	BMN: Das am besten passende Nachbarn Code-Empfehlungssystem [6]	Alle Methodenaufrufe	Namen der Methodenaufrufe	Abstand von Vektoren	Automatisch	Mind. 30% Ähnlichkeit/ Relevanz	P: 80% R: 79% F_1: 79,5%
(3)	CSCC: Einfache, effiziente, kontextsensitive Code-Empfehlung [7]	Alle Methodenaufrufe, Typen und Java-Schlüsselwörter	Namen der Methodenaufrufe, Typnamen und die Namen der Java-Schlüsselwörter als 8-Bitcodes	Hamming-Distanz von 8-Bit langen Binärcodes	Automatisch	Top 3	P: 83% R: 99% F_1: 90%
(4)	Intelligente Code-Empfehlung mit Bayes'sches Netz [10]	Alle Methodenaufrufe	Namen der Methodenaufrufe gebündelt in Objektverwendungen	Anzahl der gleichen Methodenaufrufe von Objektverwendungen	Automatisch	Mind. 30% Ähnlichkeit/ Relevanz	Nicht angegeben
(5)	GraPacc: graphenbasiertes, musterorientiertes, kontextsensitives Code-Empfehlungstool [8]	Alle Methodenaufrufe, Typen und Java-Schlüssel-wörter	Namen und Position der Methodenaufrufe, Typen und Java-Schlüsselwörter als Graph	Anzahl an Überreinstimmungen von Knoten im Graph	Semiautomatisch durch ein Tool	Top 5	P: 84% R: 71% F_1: 77%

Tabelle 2: Übersicht der Ansätze bezüglich der Recherchefragen

BMN: Das am besten passende Nachbarn Code-Empfehlungssystem

Dieses System ist der Hauptbeitrag von Bruch in seinem Artikel [6]. Es basiert auf einer Modifikation des maschinellen Lernalgorithmus „k-nächste-Nachbarn-Algorithmus“ [11]. Der Algorithmus wird auch immer dann automatisch angewendet, wenn der Entwickler einen Methodenaufruf auf ein Objekt anfordert. Basierend darauf durchsucht das System das Projekt auf alle Codeabschnitte, in denen dieses Objekt verwendet wurde. Für jeden dieser Codeabschnitte wird ein Vektor mit allen Methodenaufrufen auf dieses Objekt erstellt. Mit diesen Vektoren wird eine Matrix erzeugt, in der jede Zeile einen dieser Vektoren darstellt und jede Spalte eine Methode, welche auf diesem Objekt aufgerufen wurde. Die Matrix wird nun mit Einsen und Nullen befüllt, je nachdem ob in dem Codeabschnitt die jeweilige Methode aufgerufen wurde oder nicht. Als nächstes wird die Matrix basierend auf dem Vektor, der zum aktuellen Methodenaufruf geführt hat gefiltert, indem nur die Vektoren in Betracht gezogen werden, die den gleichen Vektor aufweisen wie der aktuelle Codeabschnitt (siehe Abbildung 3: Vektor besteht in diesem Fall nur aus Text.<init>).

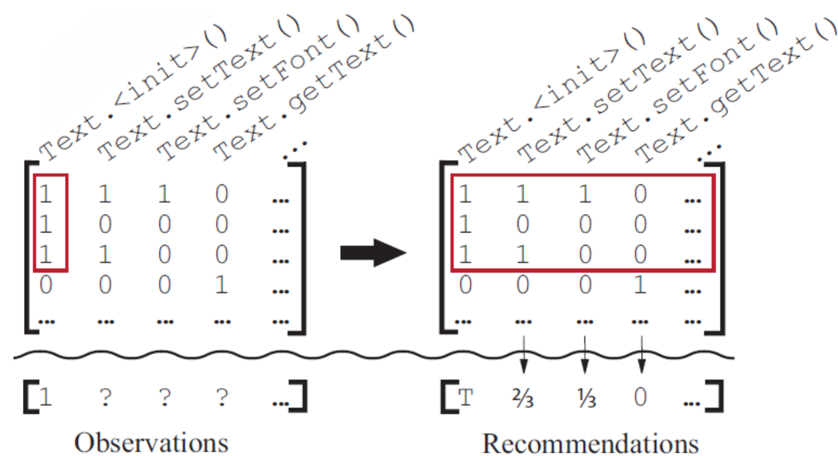


Abbildung 3 (Quelle: Bruch et al.: *Learning from examples to improve code completion systems* [6], leicht bearbeitet)

Nun werden mit den restlichen Vektoren die Wahrscheinlichkeiten für die Methodenaufrufe erstellt, indem die Anzahl der Vorkommen der Methodenaufrufe durch die Anzahl der Codeabschnitte geteilt wird (siehe Recommendations in

Abbildung 3). Als Resultat werden alle Methoden mit Wahrscheinlichkeiten über 30% absteigend empfohlen.
Mit dieser Methode erreichen Bruch et al. [6] und Asaduzzaman et al. [7] in ihren Evaluationen eine Precision von 80% und einen Recall von 79%. Daraus ergibt sich einen F_1-Wert von 79,5%, welcher schon deutlich höher ist als beim frequenzbasierten Code-Empfehlungssystem.

CSCC: Einfache, effiziente, kontextsensitive Code-Empfehlung

Das Code-Empfehlungssystem von Asaduzzaman et al. [7] wird immer dann automatisch ausgeführt, wenn der Benutzer während der Programmierung einen Methodenaufruf in Eclipse durch einen Punkt nach einem Objekt anfordert. Daraufhin werden aus den vier Zeilen zuvor alle Methodenaufrufe, Typnamen und Java Schlüsselwörter extrahiert. Das gleiche Verfahren wird mit allen Methodenaufrufen im Projekt ausgeführt, um den aktuellen Methodenkontext damit zu vergleichen. Der Vergleich wird wie folgt ausgeführt: Jedes einzelne Wort wird mit der SimHash-Technik [12] in einen 8-Bit langen Binärcode übersetzt, indem jeder einzelne Buchstabe des Wortes mittels der ASCII-Tabelle als Binärcode gespeichert und daraus die Summe gebildet wird (vergleiche Abbildung 4)

T	->	01010100
e	->	01100101
s	->	01110011
t	->	01110100
Test	->	10100001

Abbildung 4: Beispiel Übersetzen eines Worts mittels der SimHash-Technik [12]

Daraufhin werden alle Namen von Methodenaufrufen, Typen und Java Schlüsselwörtern des aktuellen Methodenkontextes mit jedem anderen Kontext verglichen. Dabei wird die Hamming-Distanz zwischen jedem Wort mit allen Wörtern des zu vergleichenden Kontextes berechnet und die geringste Distanz wird gespeichert. Die Summe der kleinsten Hamming-Distanzen aller Wörter bilden die

Ähnlichkeit zu diesem zu vergleichenden Methodenkontext. Je kleiner die Distanz, desto ähnlicher sind die Codeabschnitte. Zuletzt werden die Top 3 Methodenaufrufe in absteigender Relevanz angezeigt. Durch dieses Verfahren schafft es Asaduzzaman et al. [7] auf eine Qualität von 83% Precision und 99% Recall. Dies ergibt einen Wert für das F_1-Maß von ca. 90%, welcher der Spitzenwert der vorgestellten Methoden ist.

Intelligente Code-Empfehlung mit Bayes´sches Netz

Proksch et al. [10] stellt in seinem Dokument einen Ansatz zur intelligenten Code-Empfehlung mit Hilfe eines Bayes´schen Netzes vor. Das System wird *Pattern-based Bayesian Network (PBN)* [10] genannt. Das Verfahren verwendet ein Bayes´sches Netz, um zu ermitteln, wie wahrscheinlich bestimmte Methodenaufrufe sind, da Kontextinformationen und möglicherweise andere Methodenaufrufe bereits beobachtet wurden. Abbildung 5 zeigt ein Beispiel eines Bayes´schen Netzes für ein Objekt, welches dreimal auf unterschiedliche Weise verwendet wurde.

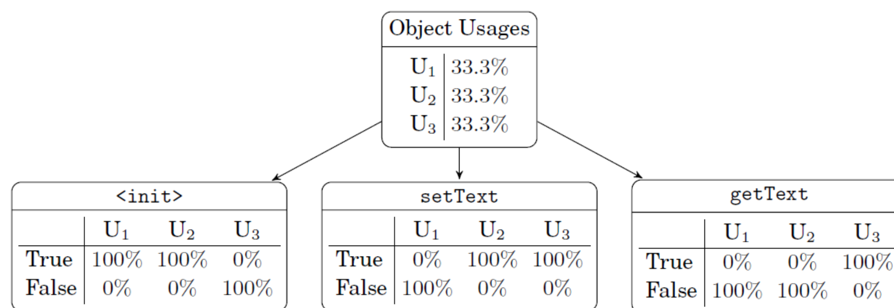


Abbildung 5: Beispiel Bayes´sches Netz (Quelle: Proksch et al.: Intelligent Code Completion with Bayesian Networks [10])

Es existieren in diesem Beispiel also drei verschiedene Objektverwendungen, welche die Zustände des Objektverwendungsknoten repräsentieren. Jede davon existiert genau einmal. Also besitzen alle eine Wahrscheinlichkeit von 33,3%. Die drei verbleibenden Knoten repräsentieren die Methodenaufrufe. Die Zustände jedes Methodenaufrufknotens lauten „True“ und „False“, die angeben, ob die Methode in einer Objektverwendung aufgerufen wird oder nicht. Die bedingten Wahrscheinlichkeiten für die Methoden sind dann zum Beispiel P(<init> | U_1) = 100%.

Die Wahrscheinlichkeit für P(<init>) ist dann die Summe aller bedingten Wahrscheinlichkeiten und somit 66,7% (vergleiche Abbildung 5).
Ziel dieses Verfahren ist es, den Entwickler dabei zu unterstützen, Methodenaufrufe zu finden, die noch fehlen, nachdem er schon eine oder mehrere Methoden auf ein Objekt aufgerufen hat. Wenn er zum Beispiel bereits die Methode <init> ausgeführt hat, würde die Methode setText unter dieser Bedingung eine Wahrscheinlichkeit von 50% betragen, da bei einer der zwei Objektverwendungen, bei denen <init> ausgeführt wurde, auch die Methode setText ausgeführt wurde.
Auf diese Art und Weise wird jede Methode auf ihre Relevanz überprüft, und alle Methodenaufrufe mit mindestens 30% Wahrscheinlichkeit werden absteigend angezeigt. Berechnungen für Precision und Recall hat Proksch et al. [10] in seinem Paper leider nicht angegeben.

GraPacc: graphenbasiertes, musterorientiertes, kontextsensitives Code-Empfehlungsstool

Nguyen et al. [8] verwendet in seinem Code-Empfehlungsstool Graphen mit Knoten und Kanten, die den Weg mittels aller Anweisungen zum Methodenaufruf beschreiben. Die Knoten können Methodenaufrufe, Typen (Objekte, Variablen) und Java-Schlüsselwörter (if, while) beschreiben. Dabei wird Typ, Name und Position des Knotens gespeichert. Wird nun ein Methodenaufruf durch einen Punkt angefordert, vergleicht das System den Graphen, der den aktuellen Weg zum Methodenaufruf führt, mit allen anderen Graphen der Methodenaufrufe des Projekts. Dabei werden die Typen und Namen verglichen und, falls sie übereinstimmen, auch die Position im Graphen. Je mehr dieser Elemente übereinstimmen, desto relevanter ist der gefundene Methodenaufruf im aktuellem Codesegment.
Mit diesem Resultat zeigt Nguyen et al. [8] in seinem Tool für Code-Empfehlung die Top 5 Ergebnisse an, da er so Precision mit 84% und Recall mit 71% maximieren konnte. Dies ergibt einen F_1-Wert von ca. 77%.

3.4.2 Beantworten der Recherchefragen

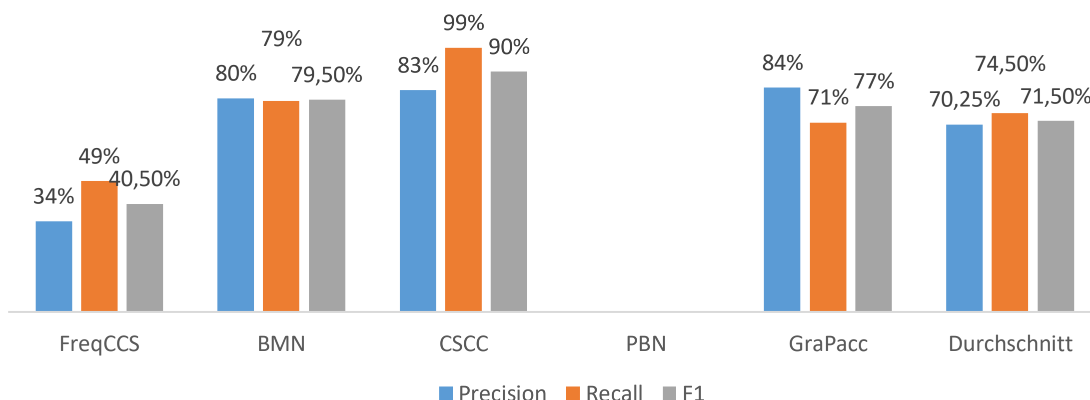
Im Folgenden werden die zu Beginn aufgestellten Recherchefragen beantwortet (vergleiche Tabelle 2):

RF1: Bei drei der fünf Empfehlungssysteme werden die Methodenaufrufe eines Codeabschnittes mit den Methodenaufrufen der restlichen Codeabschnitte des Projekts verglichen. Dafür werden immer die Namen der Methoden verwendet [6,10]. Bei den beiden anderen Verfahren werden zusätzlich die Namen von Typen und Schlüsselwörter zum Vergleich miteinbezogen [7,8]. In nur einem Fall spielen dabei die Positionen der Aktionen und Aufrufe eine Rolle [8].

RF2: Die Ähnlichkeit zweier Codeabschnitte wird bei allen Empfehlungssystemen über die Anzahl an Übereinstimmungen der Methodenaufrufe berechnet. Diese Anzahl wird allerdings durch viele verschiedene Ansätze berechnet, entweder durch den Abstand von Vektoren der Methodenaufrufe [6], durch die Hamming-Distanz der Methodenaufrufe [7] oder mittels Graphen [8,10].

RF3: Der Durchschnitt der vier Ansätze, da von Proksch et al. [10] leider keine Angaben zu Precision und Recall gegeben wurden, liegt bei 70,25% für Precision, 74,5% für Recall und 71,75% für das F_1-Maß. Dabei liegt das Spektrum der Precision von 34% bis 84%, des Recall von 49% bis 99% und des F_1-Maßes von 40,5% bis 90%. (siehe Tabelle 3)

Tabelle 3: Qualität der Ansätze



3.4.3 Vergleich verwandter Arbeiten

Folgende allgemeine Ergebnisse haben sich aus den Publikationen ergeben (vergleiche Tabelle 2):

- **Inhaltliche und strukturelle Aspekte**: Da sich alle gefundenen Code-Empfehlungssysteme auf Empfehlungen von Methodenaufrufen beziehen, werden diese natürlich auch bei allen zur Generierung von Vorschlägen mit einbezogen. Darüber hinaus werden in *CSCC: Simple, Efficient, Context Sensitive Code Completion* [7] und in *Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion* [8] noch Typen und Java-Schlüsselwörter in Betracht bezogen. Für den strukturellen Aspekt sind folgende Gemeinsamkeiten zu erkennen.
  Bruch et al. [6], M. Asaduzzaman et al. [7] und Proksch et al. [10] nehmen ausschließlich die Namen der Methoden zum Vergleich. Nguyen et al. [8] nehmen zusätzlich für ihre Generierung der Empfehlungen auch die Positionen der Methoden, Typen und JAVA-Schlüsselwörter im Graphen.

- **Ähnlichkeiten zwischen den Codedateien**: Bei drei der fünf Ansätzen wird die Relevanz von Methodenaufrufen über die Anzahl berechnet. Das *Frequenzbasierte Code-Empfehlungssystem* [6] nimmt die Anzahl der Verwendungen der Methodenaufrufe im Projekt. Bei der *intelligenten Code-Empfehlung mit Bayes'sches Netz* [10] wird die Anzahl von Methodenaufrufen zweier Objektverwendungen zur Ähnlichkeit verwendet. Zuletzt verwendet *GraPacc* [8] die Anzahl an Knoten, welche übereinstimmen. Einen Unterschied bei der Berechnung von Relevanz zeigt BMN [6] und CSCC [7] auf. Bei BMN [6] wird die Ähnlichkeit über Abstände von Vektoren berechnet, bei CSCC [7] über die Hamming-Distanz von 8-Bitcodes.

- **Trigger der Methode**: Bei vier der fünf Methoden wird das System durch eine Methodenaufforderung, wie zum Beispiel in Java durch einen Punkt

nach einem Objekt, automatisch gestartet [6,7,10]. Bei GraPacc [8] muss das Tool manuell gestartet werden.

- **Ausgabe/ Ergebnis**: Es werden zwei unterschiedliche Arten der Ausgabe verwendet. Die erste Variante ist, die Anzahl der angegebenen Empfehlungen auf eine Zahl zu beschränken. Das *Frequenzbasierte Code-Empfehlungssystem* [6] und *CSCC* [7] empfehlen die „Top 3“ und *GraPacc* [8] die „Top 5“. Die Anzahl wurde bei den Systemen so gewählt, dass Precision und Recall maximiert wurden. Eine hohe Zahl wie zum Beispiel die „Top 5“ ergibt sich daraus, da es für Code-Empfehlungssysteme für Methodenaufrufe mehrere relevante Methoden gibt, anstatt wie bei Feature-Tags nur die eine richtige Auswahl.
  Die zweite Variante wird beim *am besten passenden Nachbarn Code-Empfehlungssystem* [6] und bei der *intelligenten Code-Empfehlung mit Bayes'sches Netz* [10] verwendet. Diese beinhaltet, dass alle Methoden vorgeschlagen werden, die eine gewisse Ähnlichkeit aufweisen. Bruch et al. [6] und Proksch et al. [10] wählen dabei für deren Algorithmen eine Relevanz von mindestens 30%. Dieser Wert wurde bei beiden wieder so gesetzt, dass Precision und Recall für deren Testergebnisse maximiert wurden.

3.5 Auswahl eines Ansatzes für eigene Implementierung

Im folgenden Abschnitt wird nun einer dieser Ansätze ausgewählt, um ihn für eine eigene Implementierung eines Code-Empfehlungssystems für Feature-Tags zu konzeptionieren. Um den optimalen Ansatz für das eigene System zu finden, wird zunächst das Ausschlussverfahren angewendet. Wie der lautende Titel dieser Arbeit „Automatisierte Empfehlung von Feature-Tags im Code“ sagt, sollte das entwickelte System automatisch Empfehlungen generieren. Bei diesem Gedanken fällt sofort auf, dass das Code-Empfehlungstool GraPacc von Nguyen et al. [8] manuell getriggert werden muss (siehe Tabelle 2). Also wird dieser Ansatz für eine eigene

Implementierung zunächst hintenangestellt, da die restlichen Verfahren eine bereits automatisierte Methode vorstellen.

Da die eigene Implementierung natürlich bestmögliche Ergebnisse empfehlen soll, wird als nächstes die Qualität der verschiedenen Ansätze in Betracht bezogen. Zu diesem Punkt haben Proksch et al. [10] leider keine Angaben zu Precision und Recall und fallen dadurch ebenfalls zunächst aus der Auswahl heraus. Des Weiteren weist das *Frequenzbasierte Code-Empfehlungssystem* von Bruch et al. [6] deutlich schlechtere Werte für Precision und Recall auf im Vergleich zu BMN [6] und CSCC [7] (vergleiche Tabelle 3).

Nun stehen nur noch das *am besten passende Nachbarn Code-Empfehlungssystem BMN* von Bruch et al. [6] und die *einfache, effiziente, kontextsensitive Code-Empfehlung CSCC* von Asaduzzaman et al. [7] zur Auswahl der eigenen Implementierung fest. Beide Ansätze würden sich wahrscheinlich auch gut für automatisierte Empfehlungen für Feature-Tags eignen. Hierfür fällt die letzte Wahl auf den Ansatz von Asaduzzaman et al. [7]. Dieser bietet im Vergleich zum BMN [6] nochmals einen verbesserten Wert für den Recall, der nahe an Perfektion gleicht (vergleiche Tabelle 3).

4 Anforderungen und Entwurf

Das folgende Kapitel beschreibt die Anforderungen und den Entwurf der automatisierten Empfehlung von Feature-Tags im Code. Dafür werden zunächst die Grobanforderungen in Abschnitt 4.1 beschrieben, und die Detailanforderungen als User Task in Abschnitt 4.2. Abschnitt 4.3 erläutert die nicht-funktionalen Anforderungen. Der Abschnitt 4.4 stellt das UML Klassendiagramm dar. Basierend auf den Anforderungen wird im letzten Abschnitt 4.5 ein Abbild der Benutzeroberfläche erstellt.

4.1 Grobanforderungen

Der zu entwickelnde Ansatz wird in dem Eclipse Prototypen „FRAGRANCE“ umgesetzt, um die Entwickler bei der Vergabe von Feature-Tags im Code innerhalb der Entwicklungsumgebung zu unterstützen. Dafür werden Vorschläge für Feature-Tags in Abhängigkeit des Codes automatisiert generiert und dem Entwickler in der Entwicklungsumgebung Eclipse zur Vergabe angeboten (vergleiche Aufgabenstellung).

R1 Der Ansatz schlägt basierend auf den bereits vorhandenen Feature-Tags in einem Java-Projekt für eine beliebige Codedatei Feature-Tags vor.

R2 Der Ansatz nutzt die folgenden zwei Quellen zum Generieren von Feature-Tag-Vorschlägen für eine Codedatei X:

- Quelle 1: Inhalt und Struktur der Codedatei X, d.h. die Vorschläge für Feature-Tags werden durch den Ansatz anhand des Inhalts und der Struktur der Codedatei X generiert.
- Quelle 2: Ähnlichkeit der Codedatei X zu anderen Codedateien, d.h. die Vorschläge für Feature-Tags werden durch den Ansatz anhand der Feature-Tags ähnlicher Codedateien generiert.

R3 Der Prototyp „FRAGRANCE“ setzt den Ansatz innerhalb der Entwicklungsumgebung Eclipse um und unterstützt den Entwickler bei der Vergabe von Feature-Tags durch die Anzeige der vorgeschlagenen Feature-Tags für Code.

R4 Die vorgeschlagenen Feature-Tags von „FRAGRANCE“ werden dem Entwickler in übersichtlicher Form in Eclipse angezeigt, wobei die Vorschläge nach Relevanz sortiert sind.

R5 Die Generierung und die Anzeige der Feature-Tag-Vorschläge werden mit Hilfe von Eclipse Content Assist in Eclipse eingebunden.

R6 Der von „FRAGRANCE“ verwendete Ansatz zur automatisierten Empfehlung von Feature-Tags ist austauschbar und insbesondere für weitere Quellen erweiterbar.

4.2 Detailanforderungen

Im Folgenden werden die Detailanforderung als User Task nach Lausen [13] beschrieben. Da der Prototyp „FRAGRANCE“ nur eine Funktion erfüllen soll, dem Entwickler automatisch Feature-Tags zu generieren, gibt es zu dieser User Task auch nur eine Sub-Task. Dies wird wie folgt beschrieben:

Tabelle 4: User Task zu dem Prototyp „FRAGRANCE“

<table>
<tr><td colspan="2">Task: Dokumentation der Features im Code
Zweck: Vollständigkeit und Korrektheit der Features im Code sicherstellen</td></tr>
<tr><td>Sub-Tasks:</td><td>Lösung:</td></tr>
<tr><td>Dokumentiere Features im Code

Problem: Entwickler kennt passende Features nicht</td><td>Das System ermöglicht die Dokumentation von Features im Code, indem Empfehlungen von Feature-Tags vorgeschlagen werden. Diese Empfehlungen werden absteigend ihrer Relevanz angezeigt.
Systemfunktionen:
• Inhalt und Struktur von Codedateien einlesen (R2, R6)
• Ähnlichkeit zwischen Codedateien berechnen (R2)
• Empfehlungen von Features für Codedatei generieren (R2)
• Empfehlungen von Features anhand Relevanz sortieren (R4)
• Empfehlungen von Features für Codedateien anzeigen (R1, R3, R4, R5)</td></tr>
</table>

Sub-Task: Dokumentation der Features im Code:

Der Entwickler soll die passenden Feature-Tags im Code vergeben können. Um dem Entwickler das passende Feature vorzuschlagen, muss es in den zuvor eingelesenen Codedateien bereits als Feature-Tag vergeben worden sein. Durch das Schreiben der Annotation „@Feature“ will der Entwickler ein oder mehrere Feature-Tags an eine Klasse oder Codedatei vergeben. Da der Entwickler die passenden Feature-Tags nicht kennt, lässt er sich Vorschläge durch die Aufforderung des Content Assist generieren. Standardgemäß erfolgt dies in Eclipse durch die Tastenkombination „Strg + Leertaste“. Dadurch werden dem Entwickler die Feature-Tags vorgeschlagen, welche nach Relevanz sortiert sind.

Systemfunktion: Zum Subtask *Dokumentation der Feature im Code* gehören folgende Systemfunktionen:

- *Inhalt und Struktur von Codedateien einlesen* (Teil von R2 und R6):
  Die benötigten Inhalte und Strukturen der Codedateien werden eingelesen. Durch Hinzufügen, Bearbeiten oder Entfernen von Codedateien werden die Inhalte und Strukturen bei der nächsten Verwendung neu eingelesen.
- *Ähnlichkeit zwischen Codedateien berechnen* (R2):
  Die Ähnlichkeit zwischen zwei Codedateien wird berechnet, um daraus die Relevanz der Feature-Tags zu bestimmen.
- *Empfehlungen von Features für Codedatei generieren* (R2):
  Die Relevanz aller vorhandenen Feature für eine Codedatei wird berechnet, um die passenden Feature-Tags für eine Codedatei zu finden.
- *Empfehlungen von Features anhand Relevanz sortieren* (R4):
  Die Features werden anhand ihrer errechneten Relevanz für eine Codedatei absteigend sortiert, um diese in eine nach Relevanz sortierten Reihenfolge anzeigen zu können.
- *Empfehlungen von Features für Codedateien anzeigen* (R1, R3, R5):
  Die Features werden dem Entwickler über den Content Assist in Eclipse angezeigt. Der Entwickler muss nur die passenden Feature-Tags auswählen.

4.3 Qualitätsanforderungen

Die folgenden Qualitätsanforderungen an das Plug-In haben sich aus den Grobanforderungen ergeben:

- *Benutzbarkeit*: Dies lässt sich aus R4 und R5 herauslesen. Es soll eine einfache Bedienbarkeit sichergestellt werden, indem die Feature-Tags in einer übersichtlichen Form für den Entwickler angezeigt werden. Darüber hinaus werden diese nach ihrer Relevanz sortiert, sodass es dem Entwickler leichtfällt, das relevante Feature-Tag zu erkennen (R4). Zuletzt werden die

Feature-Tags mit Hilfe des Eclipse Content Assist in Eclipse eingebunden, welches ein gängiges Tool dieser Entwicklungsebene ist (R5).

- *Änderbarkeit und Erweiterbarkeit*: Diese beiden Qualitätsmerkmale gehen aus R6 hervor. Die Quellen für den Ansatz sollen sich zum einen austauschen und zum anderen erweitern lassen. Dies soll sichergestellt werden, indem die Berechnung der Relevanz Schnittstellen verwendet, sodass der implementierte Ansatz leicht austauschbar und erweiterbar ist.

4.4 Klassendiagramm

In der folgenden Abbildung 6 werden die Klassen mittels eines UML Klassendiagramms festgehalten. Durch die Klasse *Activator* wird das Plug-In gestartet. Die Klasse *FeatureProposalComputer* mit dem Interface *IJavaCompletionProposalComputer* des Content Assist in Eclipse stellt dafür die generierten Feature-Tags zur Verfügung, welche in der Klasse *FeatureProposal* erstellt werden. Zur Generierung der Feature-Tags wird die Klasse *CodeProposalGenerator* mit dem Interface *FeatureProposalGenerator* verwendet. Der *CodeProposalGenerator* liest den Inhalt und die Struktur mittels der Klasse *ActiveProject* ein und wendet darauf den Algorithmus zur Generierung der Ähnlichkeiten mit der Klasse *SimHash* an. Die Ergebnisse werden der Klasse *FeatureProposalSorter* übergeben, welche die Feature-Tags anhand ihrer Relevanz sortiert. Die Klasse *FeatureProposalSorter* erbt dabei von der Klasse *AbstractPropsoalSorter* des Content Assist in Eclipse.
Die Klassen *Activator*, *FeatureProposalComputer*, *FeatureProposal*, *FeatureProposalGenerator* und *FeatureProposalSorter* sind dabei in einem existierenden Plug-In bereits enthalten. Die Klassen CodeProposalGenerator, ActiveProject und SimHash werden dem Plug-In hinzugefügt und die Klasse *FeatureProposalSorter* wird für die Argumente des *CodeProposalGenerator* entsprechend erweitert.

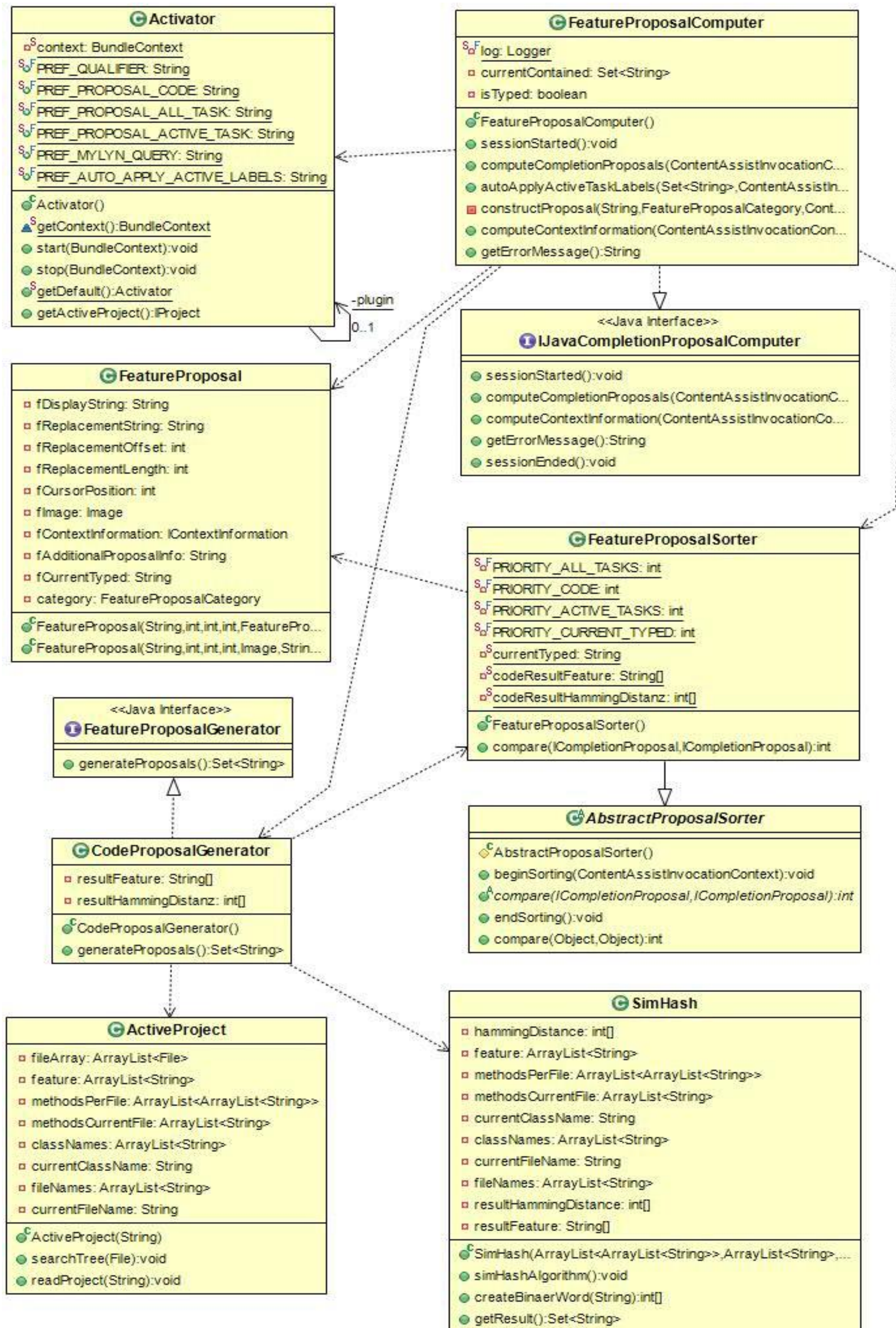


Abbildung 6: UML Klassendiagramm

4.5 Entwurf der Oberfläche

Basierend auf den Anforderungen wurde ein Entwurf der Oberfläche angefertigt. Als Grundlage der Benutzeroberfläche dient natürlich Eclipse, da der Prototyp als Plug-In für die Entwicklungsebene implementiert werden soll (siehe Editor in Abbildung 7). Des Weiteren sollen die Empfehlungen der Feature-Tags mittels des Content Assist von Eclipse dargestellt werden. Dies wird im rot umrandeten Feld in Abbildung 7 dargestellt.

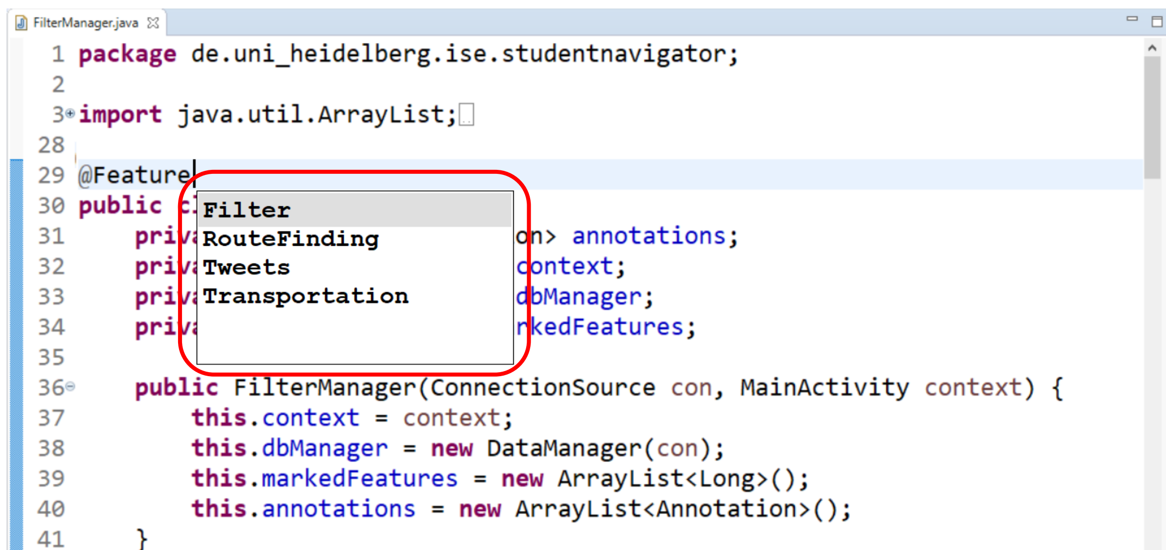


Abbildung 7: Prototyp der Benutzeroberfläche in Eclipse. (Beispiel aus Java-Testprojekt Studentnavigator). Der Content Assist für die Feature-Tags wird durch die Tastenkombination „Strg + Leertaste" aufgerufen, sofern in der Zeile die Annotation „@Feature" enthalten ist. Für die Klasse *FilterManager* werden dem Entwickler die Feature-Tags *Filter, RouteFinding, Tweets* und *Transportation* absteigend bezüglich ihrer Relevanz angezeigt.

5 Implementierung

Das folgende Kapitel beschreibt die Implementierung des Prototyps „FRAGRANCE". Der in Kapitel 4 beschriebene Entwurf dient dabei als Vorlage. Abschnitt 5.1 erläutert erste Vorüberlegungen zum Vorgehen der eigenen Implementierung. Danach werden in Abschnitt 5.2 Entscheidungen erläutert, die im Laufe der Implementierung entstanden sind. Zuletzt wird in Abschnitt 5.3 die Qualitätssicherung beschrieben.

5.1 Vorüberlegungen

Die gegeben Grobanforderungen an den Prototyp „FRAGRANCE" geben sehr wenig Spielraum für die eigene Implementierung der Benutzeroberfläche. Der Grund dafür ist zum einen, dass der Prototyp als Plug-In in Eclipse umgesetzt werden soll und zum anderen, dass die Empfehlungen der Feature-Tags mittels des integrierten Content Assist von Eclipse dargestellt werden.

Der Fokus der Implementierung liegt demnach im Wesentlichen auf dem Algorithmus der Generierung der Empfehlungen für Feature-Tags. Dieser Algorithmus soll von bestmöglicher Qualität sein, sodass Precision und Recall maximiert werden. Um die Qualität des Ansatzes am einfachsten testen zu können, wird dieser zuerst als Kommandozeilenwerkzeug implementiert. Sobald dieser die gewünschten Resultate erzielt, wird der Ansatz als Plug-In für Eclipse umgesetzt.

5.2 Entscheidungen

Im folgenden Abschnitt werden Entscheidungen erläutert, die während der Implementierung des Prototyps „FRAGRANCE" getroffen wurden.

5.2.1 Technischer Rahmen

Der Algorithmus zur Generierung von Empfehlungen für Feature-Tags wird als ein Plug-In in Eclipse umgesetzt. Ein Plug-In, welches Feature-Tags aus dem Code ausliest und für den Content Assist von Eclipse bereitstellt, existiert bereits. Dieses Plug-In wird also durch den Algorithmus erweitert, der die Relevanz der Feature-Tags berechnet und entsprechend sortiert.
Da es sich hierbei um ein Eclipse-Plug-In handelt, welches in Java geschrieben wurde, wird der Algorithmus ebenfalls in Java geschrieben. Zum Ausführen des Plug-Ins ist ein installiertes JRE der Version JavaSE-1.8 oder höher erforderlich.

5.2.2 Algorithmus

Wie in Kapitel 3 zur Literaturrecherche entschieden wurde, wird der Ansatz *CSCC: Simple, Efficient, Context Sensitive Code Completion* von Asaduzzaman et al. [7] für eine eigene Implementierung verwendet. Dieser sieht zu Anfang zunächst wie folgt aus:

- Zuerst wird zu jeder Codedatei, sofern sie mindestens einen Feature-Tag aufweist, pro Methode, Methodenaufruf und Kontrollpunkt, wie *if*, *while* usw., ein Eintrag des Namens in einer Liste mit dem zugehörigen Feature-Tag gespeichert. Sind in einer Codedatei mehrere Feature-Tags enthalten, wird pro Feature-Tag eine solche Liste mit jeweils einem Feature-Tag erstellt.
- Die gleichen Namen werden auch für die aktuelle Codedatei, für welche Feature-Tags vergeben werden sollen, in einer weiteren Liste gespeichert.
- Als nächstes werden alle Einträge der aktuellen Codedatei zu allen Einträgen aller anderen Codedateien verglichen, um die „ähnlichste“ Datei zu ermitteln. Dieser Vergleich sieht wie folgt aus:
  - Jeder Eintrag der aktuellen Codedatei wird mit allen Einträgen der anderen Codedatei verglichen.

- Der Vergleich wird anhand von 8-Bit langen Binärcodes getätigt, die aus den Namen mittels der SimHash-Technik [12] erstellt werden. Ein Beispiel, wie Binärcodes aus Wörtern erstellt werden, zeigt Abbildung 4. Vergleich heißt in diesem Fall, dass die Hamming-Distanz zweier Binärcodes berechnet wird.
- Aus dem Vergleich eines Eintrags der aktuellen Datei zu allen Einträgen der anderen Datei wird die geringste Hamming-Distanz gespeichert.
- Die Summe aller Hamming-Distanzen zu der anderen Codedatei bildet die Gesamt-Hamming-Distanz. Ein Beispiel zur Berechnung zeigt Abbildung 8.
- Je geringer die Hamming-Distanz ist, desto „ähnlicher“ sind zwei Codedateien.

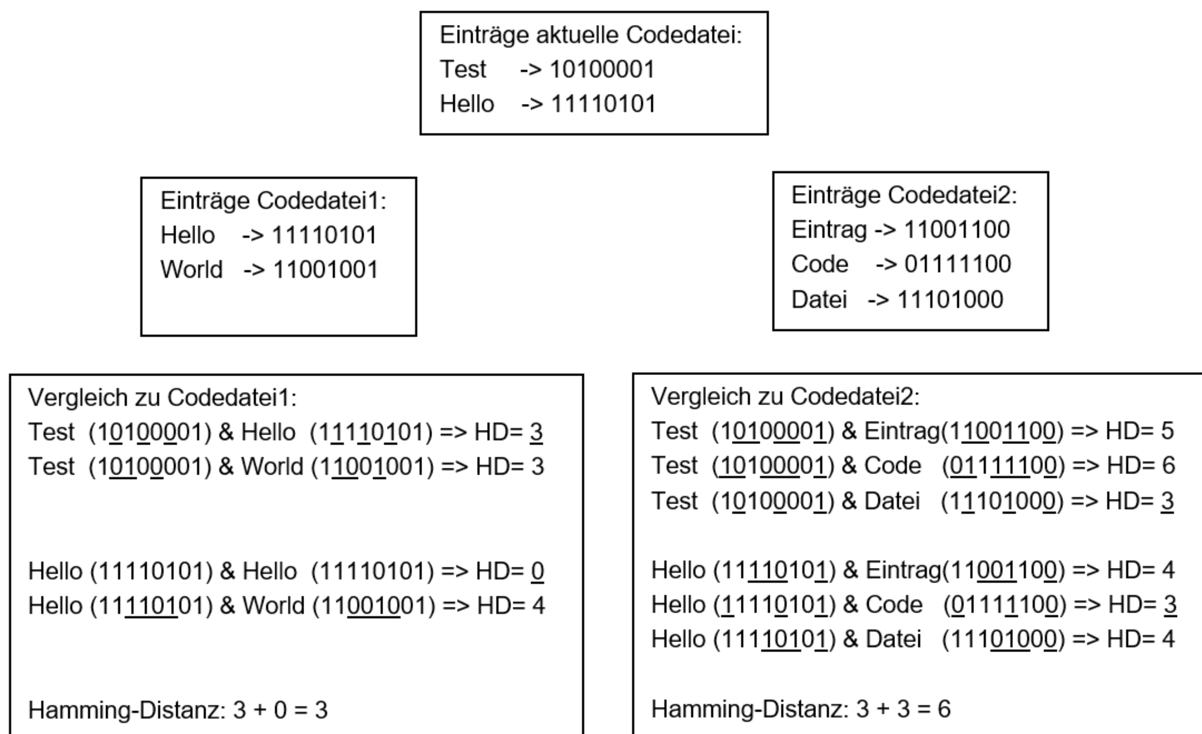


Abbildung 8: Beispiel zur Berechnung der Hamming-Distanzen einer Codedatei zu zwei weiteren Codedateien.

Durch diesen ersten Ansatz wurden folgende Werte für Precision und Recall erzielt. Dabei wurden die Werte für die Top 1, Top 2 und Top 3 zum Anzeigen der Feature-Tags ermittelt. (Eine detaillierte Beschreibung der Berechnung von Precision und Recall erfolgt in der Evaluation in Kapitel 6):

Anzahl Feature-Tags	**3**	**2**	**1**
Precision	34%	37,3%	52,9%
Recall	83,9%	61,3%	43,5%

Tabelle 5: Precision und Recall beider Studentenprojekte *MasterPatientIndex* und *Studentnavigator* im Durchschnitt nach erstem Ansatz

In den gegebenen Grobanforderungen, die in Kapitel 4 dargelegt wurden, wurde festgelegt, dass als erste Quelle der Inhalt und die Struktur der aktuellen Codedatei und als zweite Quelle die Ähnlichkeit zu den anderen Codedateien zur Generierung von Vorschlägen für Feature-Tags verwendet wird. Der beschriebene Ansatz von Asaduzzaman et al. [7] verwendet bisher nur die Quelle Zwei, also die Ähnlichkeit zu den anderen Codedateien. Um auch Quelle Eins in die Implementierung zu integrieren, wird folgende eigene Überlegung mit eingebracht:

> Wenn der Name der aktuellen Codedatei, zu welcher Feature-Tags vorgeschlagen werden sollen, Ähnlichkeiten zu einem Feature-Tag aufweist, wird dieses Feature-Tag als „relevanter“ gewichtet. Die Umsetzung der Überlegung sieht wie folgt aus: Bei allen Listen der Codedateien, bei denen das mitgespeicherte Feature-Tag keine Ähnlichkeit aufweist, wird die Hamming-Distanz potenziert. Der Grund des Potenzierens liegt darin, dass im folgendem noch zwei ähnliche Überlegungen implementiert werden, welche aber einer geringeren Priorität unterliegen. Bei deren Umsetzung wird die Hamming-Distanz verdoppelt. Um nun diese Umsetzung zu priorisieren, wird bei diesem Verfahren die Hamming-Distanz potenziert. Die folgende Tabelle 6 zeigt die Verbesserung von Precision und Recall.

Anzahl Feature-Tags	3	2	1
Precision	35% (+1%)	41,2% (+3,9%)	59,8% (+6,9%)
Recall	86,3% (+2,4%)	67,7% (+6,4%)	49,2% (+5,7%)

Tabelle 6: Precision und Recall beider Studentenprojekte *MasterPatientIndex* und *Studentnavigator* im Durchschnitt nach Implementation der ersten Überlegung

Um die Ähnlichkeit eines Dateinamens und eines Feature-Tags zu berechnen, werden zunächst die Dateinamen und die Feature-Tags in ihre einzelnen Wörter aufgeteilt. Die einzelnen Wörter entstehen dabei, wenn in den Namen Großbuchstaben enthalten sind. An diesen Stellen wird der Name in zwei Wörter aufgeteilt (Beispiel: Aus „FilterManager“ wird „Filter“ und „Manager“). Nun werden wieder alle Wörter mittels der SimHash-Technik [12] miteinander verglichen. Weist nun ein Wort des Dateinamens zu einem Wort des Feature-Tags eine Hamming-Distanz von Null auf, sind diese Wörter identisch und das Feature-Tag wird als „relevanter“ gewichtet.

Mit dieser Überlegung ist nun auch die Quelle Eins in die Implementierung integriert worden. Durch diesen implementierten Zusatz ist noch eine weitere Überlegung aus eigenen Erfahrungen entstanden, welche Precision und Recall nochmals erhöht hat. Diese lautet wie folgt:

Oftmals besitzen Dateien, deren Namen Ähnlichkeiten aufweisen, auch die gleichen Feature-Tags. Diese Namen werden wie zuvor bei der Ähnlichkeitsberechnung zum Feature-Tag miteinander verglichen. Wird beim Vergleich zweier Dateinamen wieder eine Hamming-Distanz von Null erkannt, werden die Feature-Tags bei dieser Codedatei als „relevanter“ gewichtet. Die Umsetzung lautet auch hier, dass bei allen Listen der Codedateien, bei denen diese Ähnlichkeit nicht erkannt wird, die Hamming-Distanz erhöht wird. Allerdings in diesem Fall nicht potenziert, sondern verdoppelt. Dadurch wird die erste Überlegung deutlich mehr gewichtet, und Precision und Recall haben sich auf diese Weise folgendermaßen verbessert (siehe Tabelle 7):

Anzahl Feature-Tags	3	2	1
Precision	36,9% (+1,9%)	46,1% (+4,9%)	68,6% (+8,8%)
Recall	91,1% (+4,8%)	75,8% (+8,1%)	56,5% (+7,3%)

Tabelle 7: Precision und Recall beider Studentenprojekte *MasterPatientIndex* und *Studentnavigator* im Durchschnitt nach Implementation der zweiten Überlegung

Aus weiteren Beobachtungen von Projekten mit verwendeten Feature-Tags und durch eigene Erfahrungen bei der Projektentwicklung ist eine letzte Überlegung entstanden. Diese bezieht sich auf beide Quellen Eins und Zwei:

> Oftmals besitzen Dateien, die einem gleichen Ordner zugehören, auch die gleichen Feature-Tags. Die Umsetzung dieser Überlegung lautet auch hier, dass bei allen Listen der Codedateien, die nicht den gleichen Pfad aufweisen, die Hamming-Distanz wieder verdoppelt wird. Durch diese Implementierung haben sich Precision und Recall vor allem beim Anzeigen des Top-Features nochmals erhöht (vergleiche Tabelle 8). Beim Anzeigen der Top Drei Feature ist dagegen ein relevantes Feature von Position drei auf vier gefallen, was die minimalen in Rot dargestellten schlechteren Ergebnisse erklärt. Darüber kann aber hinweggesehen werden, da die grün markierten Positivergebnisse beim Anzeigen des Top-Feature deutlich höher sind.

Anzahl Feature-Tags	3	2	1
Precision	36,6% (-0,3%)	49% (+2,9%)	78,4% (+9,8%)
Recall	90,3% (-0,8%)	80,6% (+4,8%)	64,5% (+8%)

Tabelle 8: Precision und Recall beider Studentenprojekte *MasterPatientIndex* und *Studentnavigator* im Durchschnitt nach Implementation der letzten Überlegung

Bei allen drei Umsetzungen der Überlegungen wird die Hamming-Distanz verdoppelt oder potenziert. Dabei ist ein Problem während der Qualitätssicherung in Abschnitt 5.3 aufgetreten. Wenn die Hamming-Distanz bis zu diesem Zeitpunkt des Verdoppelns oder Potenzierens genau null beträgt, wird der Wert durch das Verdoppeln oder Potenzieren nicht erhöht. Deshalb wird vor jeder dieser Aktion die Anzahl an

verschiedenen Feature-Tags hinzuaddiert. Damit wurde der Wert an die Größe des Softwareprojekts angepasst.

5.2.3 Berechnung des Ähnlichkeitsmaßes zwischen zwei Codedateien

Zusammenfassend aus dem Abschnitt zuvor wird das Ähnlichkeitsmaß zweier Codedateien wie folgt berechnet. Entsprechend dazu zeigt Abbildung 9 den Pseudocode des Algorithmus zur Berechnung des Ähnlichkeitsmaßes:

- Jede Methode der ersten wird mit jeder Methode der zweiten Codedatei mittels der SimHash-Technik [12] über die Hamming-Distanz verglichen.
- Pro Methode der ersten Codedatei wird der kleinste Wert der gesamten Hamming-Distanz hinzuaddiert. Sobald eine Methode der ersten Codedatei denselben Namen hat wie eine Methode der zweiten Codedatei, wird die Hamming-Distanz dieser Methode gleich null sein.
- Danach werden die Dateinamen miteinander verglichen. Sofern nicht zwei gleiche Namen enthalten sind, wird die gesamte Hamming-Distanz erst mit der Anzahl an verschiedenen Feature addiert und danach verdoppelt.
- Danach wird überprüft, ob beide Codedateien den gleichen Pfad besitzen. Ist dies nicht der Fall, wird wiederum die gesamte Hamming-Distanz erst mit der Anzahl an verschiedenen Feature addiert und danach verdoppelt.
- Zuletzt wird überprüft, ob der Name der Codedatei Eins Ähnlichkeiten zum Namen des Feature-Tags aus Codedatei Zwei aufweist. Wenn nicht zwei gleiche Namen enthalten sind, wird die gesamte Hamming-Distanz erst mit der Anzahl an verschiedenen Features addiert und danach potenziert. Der Grund dafür, dass diese Abfrage zuletzt durchgeführt wird und sogar die Hamming-Distanz potenziert und nicht nur verdoppelt, ist, dass dadurch diese Abfrage deutlich mehr gewichtet wird als die anderen beiden. Wenn der Name der Codedatei bereits Ähnlichkeiten zu einem Feature-Tag aufweist, ist es sehr wahrscheinlich, dass diese Codedatei auch dieses Feature implementiert.

```
Eingabe: 2 Codedateien A und B (B mit einem vergebenen Feature-Tag)

1  ghd = 0                              // ghd: Gesamt-Hamming-Distanz
2  for i = 0 to A.länge                 // A[]: Namen der Methoden der Datei A
3      hd = 8
4      for j = 0 to B.länge             // B[]: Namen der Methoden der Datei B
5          sh = SimHash(A[i], B[j])     // SimHash: Berechnung Hamming-Distanz
6          if sh < hd
7              hd = sh
8      ghd += hd
9
10 hd = 8
11 for i = 0 to C.länge                 // C[]: Teile des Namens der Datei A
12     for j = 0 D.länge                // D[]: Teile des Namens der Datei B
13         sh = SimHash(C[i], D[j])
14         if sh < hd
15             hd = sh
16 if hd > 0
17     ghd += f                         // f: Anzahl verschiedener Feature-Tags
18     ghd = ghd * 2
19
20 if pfadA != pfadB                    // pfad…: Pfad der Dateien A und B
21     ghd += f
22     ghd = ghd * 2
23
24 hd = 8
25 for i = 0 to C.länge
26     for j = 0 F.länge                // F[]: Namen des Feature-Tags
27         sh = SimHash(C[i], F[j])
28         if sh < hd
29             hd = sh
30 if hd > 0
31     ghd += f
32     ghd = ghd * ghd

Ausgabe: Hamming-Distanz zu dem vergebenen Feature-Tag der Codedatei B
```

Abbildung 9: Pseudocode zur Berechnung der Hamming-Distanz zweier Codedateien. Wenn die Codedatei (hier Datei B) mehrere Feature-Tags enthält, wird dies für jedes einzelne Feature-Tag durchgeführt.

5.3 Qualitätssicherung

In diesem Abschnitt wird auf die Qualitätssicherung des Plug-Ins eingegangen. Dabei wird ein Komponententest der eigenen implementierten Klassen und deren Operationen und ein Systemtest zu allen Systemfunktionen, sowie der Einhaltung der Qualitätsanforderungen getestet.

Beim Komponententest wurden die beiden eigenen Klassen ActiveProject und SimHash mittels durchgeführten JUnit[4]-Tests getestet. Insgesamt wurden 16 JUnit-Tests durchgeführt, um die Anweisungen der verschiedenen Operationen zu überdecken.

Beim Systemtest wurden zu den definierten Systemfunktionen alle Kombinationen von Ein- und Ausgabe getestet. Dabei ist ein Problem entdeckt worden. Wenn bei der Eingabe die Codedatei, zu welcher Feature-Tags vorgeschlagen werden sollen, keine Methoden enthält, wird für alle Feature-Tags die Hamming-Distanz gleich null errechnet. Danach haben die Aktionen des Verdoppelns oder Potenzierens auch keinen Einfluss mehr, da die Werte immer noch null bleiben. In diesem Fall wurden einfach die Feature-Tags alphabetisch angezeigt. Um dieses Problem zu lösen, wird beim Verdoppeln und Potenzieren zuerst ein Wert hinzuaddiert. Dieser muss mindestens größer als zwei sein, da sonst das Potenzieren nicht mehr gewichtet wird als das Verdoppeln. In der Umsetzung wurde der Wert der Anzahl an verschiedenen Feature-Tags gleichgesetzt.

Zuletzt wurden die nicht-funktionalen Anforderungen Benutzbarkeit und Änderbarkeit/ Erweiterbarkeit getestet. Die Benutzbarkeit wird eingehalten, indem die Feature-Tags in einer übersichtlichen Form der Reihenfolge bezüglich ihrer Relevanz nach sortiert mit dem Content Assist dargestellt werden. Durch die einfache Auswahl eines Feature-Tags wird dieses gleich in den Code geschrieben. Die Änderbarkeit und Erweiterbarkeit wurde sichergestellt, indem der Algorithmus mittels Schnittstellen implementiert wurde und dadurch einfach austauschbar und erweiterbar ist.

[4] https://junit.org/junit5/

6 Evaluation

In diesem Kapitel wird die Evaluation des implementierten Ansatzes beschrieben. Dabei wird der Ansatz als Kommandozeilenwerkzeug auf Precision und Recall bei zwei Studentenprojekten mit gegeben Goldstandards evaluiert. Der Grund für das Kommandozeilenwerkzeug ist, dass man alle Codedateien effektiv auf einmal evaluieren kann und nicht in jeder Codedatei einzeln das Plug-In ausführen muss. Zunächst werden in Abschnitt 6.1 die Studentenprojekte mit dem jeweiligen Goldstandard vorgestellt. In Abschnitt 6.2 wird das Vorgehen der Evaluation beschrieben und in Abschnitt 6.3 die resultierenden Ergebnisse vorgestellt.

6.1 Studentenprojekte

Evaluiert wird der Ansatz mittels zweier Projekte der Universität Heidelberg, welche von Studenten während zwei verschiedenen ISE-Projekten entstanden sind. Diese sind zum einen der *StudentNavigator* aus dem ISE-Projekt 2018, welcher in Java implementiert wurde und zum anderen der *MasterPatientIndex* aus den ISE-Projekt 2017, welcher in JavaScript implementiert wurde. Für diese beiden Projekte existiert ein Goldstandard mit den von den projektbeteiligten Studenten vergebenen Feature-Tags im Code. Damit werden die eigenen Ergebnisse verglichen, um damit Precision und Recall zu berechnen.

Der *MasterPatientIndex* wurde entwickelt, um Patientenakten verschiedener Organisationen wie zum Beispiel Krankenhäuser, Ärzte etc. zu einer Master-Patientenakte zu bündeln. Dabei erfolgt das Zusammenführen der Patientenakten automatisch, sofern diese mittels eines Algorithmus eine bestimmte Ähnlichkeit aufweisen. Der *MasterPatientIndex* umfasst insgesamt 91 Codedateien, wobei 62 davon mit Feature-Tags versehen wurden. Die restlichen Dateien stammen von Bibliotheken, welche die Studenten nicht selbst geschrieben haben. 61 Codedateien wurden mit nur einem Feature-Tag versehen, lediglich eine Datei implementiert zwei Features. Es wurden insgesamt fünf verschiedene Feature-Tags im Code vergeben.

Tabelle 9 stellt die Verteilung der Feature-Tags im Code dar (um Beispiel wurde das Feature *SystemAdministration* in 17 verschiedenen Codedateien implementiert):

Feature	**Anzahl Codedateien**
SystemAdministration	17
PatientDataFeedQueryRetreive	16
PatientDataManagement	15
HealthcareProviderDataManagement	8
HealthcareProviderQueryRetreive	7

Tabelle 9: Featureverteilung im MasterPatientIndex

Der *StudentNavigator* ist eine IndoorNavigation Android App für Studenten, um Räume von Vorlesungen oder Büros der Dozenten etc. zu finden.
Der *StudentNavigator* umfasst insgesamt 40 Codedateien, bei denen auch für alle mindestens ein Feature-Tag vergeben wurde. Es existieren bei diesem Projekt nur vier verschiedene Feature-Tags im Code, wobei die Verteilung der Feature-Tags eine ganz andere ist im Vergleich zu der Verteilung beim *MasterPatientIndex*. Es gibt nämlich viel mehr Codedateien, welche mit zwei oder mehreren Featurex versehen sind. Drei Codedateien wurden sogar mit allen vier Feature-Tags versehen. Insgesamt wurden 61 Feature-Tags im Code vergeben. Dabei wurde zum Beispiel das Feature RouteFinding in 30 Codedateien implementiert (vergleiche Tabelle 10):

Feature	**Anzahl Codedateien**
RouteFinding	30
Transportation	13
Filter	12
Tweets	6

Tabelle 10: Featureverteilung im Studentnavigator

6.2 Vorgehen

Die Evaluation des Plug-Ins wurde mit drei verschiedenen Szenarien auf die beiden Testprojekte durchgeführt. Die Szenarien sind das Messen von Precision und Recall beim Anzeigen der Top Drei, Top Zwei und Top Eins der Feature-Tags. Die Unterschiede dabei liegen darin, dass bei der Top Eins nur ein Tag betrachtet wird, welcher entweder richtig oder falsch sein kann und bei Top Zwei und Drei eine Menge von Tags betrachtet wird, in der entweder alle Tags richtig oder falsch oder eine Mischung aus beiden sind.

Der Grund für das Messen von Precision und Recall des Top Feature-Tags ist, dass die meisten Codedateien der Testprojekte im Goldstandard nur ein Feature implementieren. Um dabei die Genauigkeit des Top-Feature stärker zu gewichten, wird hierbei zusätzlich das $F_{0,5}$-Maß berechnet, sodass die Precision im Vergleich zum Recall viermal so hoch gewichtet wird.

Der Grund für das Messen von Precision und Recall der Top drei ist, dass es auch Codedateien gibt, die auch mehrere Features implementieren. Dafür möchte der Entwickler natürlich auch mehrere Feature-Tags zur Auswahl empfohlen bekommen. Des Weiteren ist der implementierte Ansatz ein Empfehlungssystem und wird nicht immer genau das passende Feature-Tag vorschlagen können. Damit der Entwickler dieses Feature trotzdem zur Auswahl gestellt bekommt, werden die Top drei errechneten Feature-Tags angezeigt. Um dabei die Trefferquote mehr zu gewichten, wird hierbei zusätzlich das F_2-Maß berechnet, sodass der Recall im Vergleich zur Precision viermal so hoch gewichtet wird.

Als letztes wird Precision und Recall auch für die Top zwei der angezeigten Feature-Tags gemessen. Der Grund dafür ist einfach ein Mittelmaß der beiden ersten Szenarien. Hierfür wird zusätzlich auch das F_2-Maß berechnet, sodass der Recall im Vergleich zur Precision mehr gewichtet wird, da die meisten Codedateien nur ein Feature beschreiben und somit der Wert für Precision pro weiteres Anzeigen von Feature-Tags weiter sehr schnell sinkt.

6.3 Ergebnisse

Im folgenden Abschnitt werden die Ergebnisse der Evaluation dargestellt. Die Tabellen 11 bis 13 zeigen die Ergebnisse für den *MasterPatientIndex*, dem *StudentNavigator*, sowie eine Kombination aus beiden Projekten. Dabei werden die Ergebnisse in Prozent angegeben und auf eine Nachkommastelle gerundet. (Alle Ergebnisse im Detail sind in der Tabelle A.1 für den *MasterPatientIndex* und in der Tabelle A.2 für den *StudentNavigator* im Anhang enthalten. Ist ein vorgeschlagenes Feature-Tag grün markiert, bedeutet das, dass es als das relevanteste Feature-Tag vorgeschlagen wird und gleichzeitig ein passendes Feature-Tag zu dieser Codedatei darstellt. Gelb heißt, dass es als zweites vorgeschlagen wird und ein passendes Feature-Tag ist. Orange bedeutet, dass es als drittes vorgeschlagen wird und ein passendes Feature-Tag ist. Ist es nicht markiert, wird das Feature-Tag zwar vorgeschlagen, ist aber nicht relevant für die aktuelle Codedatei):

	Top 3	**Top 2**	**Top 1**
Precision	31,7%	45,2%	72,6%
Recall	93,7%	88,9%	71,4%
F-Maß	F_2: 67,4%	F_2: 74,5%	$F_{0,5}$: 72,3%

Tabelle 11: Evaluationsergebnisse MasterPatientIndex

	Top 3	**Top 2**	**Top 1**
Precision	44,2%	55%	87,5%
Recall	86,9%	72,1%	57,4%
F-Maß	F_2: 72,8%	F_2: 67,9%	$F_{0,5}$: 79,2%

Tabelle 12: Evaluationsergebnisse StudentNavigator

	Top 3	Top 2	Top 1
Precision	36,6%	49%	78,4%
Recall	90,3%	80,6%	64,5%
F-Maß	F_2: 69,8%	F_2: 71,4%	$F_{0,5}$: 75,2%

Tabelle 13: Evaluationsergebnisse der Kombination aus MasterPatientIndex und StudentNavigator

Ergebnisse der Top Eins-Vorschläge

Für das erste Szenario, bei dem nur das relevanteste Feature-Tag angezeigt wird, war vor allem der Wert für die Precision interessant. Denn dieser Wert sagt aus, ob das errechnete relevanteste Feature auch wirklich das passende Feature für diese Codedatei ist. Hier wird ein durchschnittlicher Wert von 78,4% erreicht, welcher schon in einem guten Bereich liegt. Dabei sticht zuerst der Wert für die Precision mit 87,5% beim *StudentNavigator* heraus. Hierzu muss aber erwähnt werden, dass beim *StudentNavigator* die Wahrscheinlichkeit das passende Feature-Tag zu treffen höher ist als beim *MasterPatientIndex*. Der Grund dafür ist, dass der StudentNavigator ein Feature weniger implementiert hat und es auch viel mehr Codedateien gibt, die mehr als nur ein Feature implementieren und dadurch die Chance größer ist, eines dieser Feature-Tags zu treffen. Durch diese Beobachtungen ist der Wert 72,6% für die Precision beim *MasterPatientIndex* nicht zu unterschätzen. Dieser liegt somit ebenfalls im guten Bereich und ist bei einer zufälligen Chance von 20% bei fünf verschiedenen Feature-Tags das Passende zu treffen, deutlich höher.
Insgesamt liegt auch das $F_{0,5}$-Maß, welches auch den Recall miteinbezieht, in einem guten Bereich mit durchschnittlich 75,2%.

Ergebnisse der Top Zwei- und Drei-Vorschläge

Für die beiden anderen Szenarien, bei denen die zwei und drei relevantesten Feature-Tags angezeigt werden, war vor allem der Wert für den Recall interessant. Hierbei wird ausgedrückt, ob alle relevanten Feature-Tags auch wirklich angezeigt werden. Die Werte sind dafür beim *MasterPatientIndex* sehr hoch. Mit Werten für die Top Drei mit 93,7% und für die Top Zwei mit 88,9% liegen diese in einem sehr guten Bereich. Bei dem *StudentNavigator* liegen die Werte mit 86,9% für die Top Drei und 72,1% für die Top Zwei nicht ganz so hoch, aber dennoch in einem guten Bereich. Dies ist jedoch damit zu begründen, dass bei manchen Codedateien im *StudentNavigator* auch alle vier Feature implementiert wurden und dadurch die Chance überhaupt nicht gegeben ist, mit zwei oder drei Feature-Tags alle vier Feature-Tags zu treffen.
Insgesamt liegen hier die F_2-Maße, welche auch die Precision mit einbeziehen, mit 69,8% für die Top Drei und 71,4% für die Top Zwei in einem guten Bereich.

6.4 Zusammenfassung der Evaluation

Die Evaluation hat gezeigt, dass der entwickelte Ansatz für automatisierte Empfehlungen für Feature-Tags im Code bereits gute Resultate für Precision und Recall erzielt hat. Mit den Ergebnissen für das jeweilige gewünschte F-Maß liegen auch diese Werte mit 69,8% bis 75,2% in einem guten Bereich.
Im Endeffekt dienen diese Szenarien lediglich zum Testen der Qualität des Ansatzes. In der Umsetzung des Ansatzes als Content Assist in Eclipse würde man eher alle Feature-Tags sortiert nach ihrer Relevanz angezeigt bekommen, sofern wie bei diesen Testobjekten nur vier oder fünf Feature implementiert werden. Es kann nämlich immer möglich sein, dass in einer Codedatei auch alle dieser vier oder fünf Features implementiert werden, und in diesem Fall möchte der Entwickler auch alle Feature-Tags zur Auswahl bekommen. In dem Testprojekt *StudentNavigator* ist dies sogar in drei der 40 Codedateien der Fall. Würde man jedoch Precision und Recall ausrechnen, wenn alle Feature-Tags angezeigt werden, wären diese Werte, egal für welche Reihenfolge, immer gleich. Aus diesem Grund ist der errechnete Wert für die Precision

des am relevantesten Feature-Tag besonders aussagekräftig und liegt demnach mit 78,4% in einem guten Bereich.

Für die F-Maße wurden Werte für β mit 0,5 und 2, sodass zum einen die Precision beim Anzeigen des nur am besten passenden Feature-Tags und zum anderen der Recall beim Anzeigen der Top zwei oder drei mehr gewichtet werden. Wären die Anzahl der Feature-Tags pro Codedatei ausgeglichener, dann wäre die Wahl für β mit dem Wert 1 besser gewesen, jedoch nur genau bei der Anzahl an angezeigten Feature-Tags, welche mit dem Durchschnitt der Anzahl an implementierten Feature-Tags pro Codedatei überreinstimmt. Wenn zum Beispiel jede Codedatei genau drei Features implementieren würde, wäre die Wahl für β genau 1 beim Anzeigen der Top drei Feature-Tags, da Precision und Recall auch in einem ähnlichen Bereich liegen sollten. Ein ganz einfaches Beispiel dafür zeigt auch das Szenario 1 beim Evaluieren des *MasterPatientIndex*. Der *MasterPatientIndex* implementiert pro Codedatei, bis auf eine Ausnahme, immer nur ein Feature. Dementsprechend liegen Precision und Recall sehr nah beieinander. Der *StudentNavigator* implementiert in den meisten Codedateien zwar auch nur ein Feature, allerdings in wenigen Codedateien auch mehrere bis sogar alle Features. Dementsprechend ist der Recall beim Anzeigen des lediglich am besten passenden Feature-Tags deutlich geringer als die Precision. Der Wert für β wird dementsprechend auf 0,5 gesetzt, um die Precision mehr zu gewichten.

7 Zusammenfassung

In diesem Kapitel wird die Arbeit kurz zusammengefasst. Dafür wird zuerst in Abschnitt 7.1 ein Fazit und danach in Abschnitt 7.2 ein Ausblick gegeben.

7.1 Fazit

Ziel dieser Arbeit war die Entwicklung eines Eclipse Plug-Ins zur automatisierten Generierung von Empfehlungen für Feature-Tags im Code. Hierfür wurde zuerst eine Literaturrecherche durchgeführt, um bestehende Ansätze von verschiedenen Code-Empfehlungssystemen zu analysieren. Basierend auf gegebenen Grobanforderungen wurden die Detailanforderungen an das Plug-In und der Entwurf erhoben. Mithilfe dieser Anforderungen und den Ergebnissen der Literaturrecherche wurde schließlich die Implementierung durchgeführt. Dabei wurde der Ansatz von M. Asaduzzaman et al. [6] mit der SimHash-Technik [12] verwendet. Zuerst wurde ein Kommandozeilenwerkzeug implementiert, um den verwendeten Ansatz einfacher zu evaluieren und mit den Ergebnissen zu optimieren. Danach wurde der Ansatz als ein Plug-In in Eclipse umgesetzt, welcher Empfehlungen für Feature-Tags mittels des integrierten Content Assist in Eclipse dem Entwickler zur Verfügung stellt. Die Empfehlungen werden dabei nach ihrer Relevanz bezüglich der Ähnlichkeiten der aktuellen Codedatei absteigend sortiert.
Die Evaluation wurde anhand zweier Studentenprojekte mit gegeben Goldstandards durchgeführt. Diese zeigte, dass der entwickelte Ansatz bereits gute Werte für Precision und Recall erzielte. Das zeigen vor allem die beiden Werte zum einen für die Precision des am relevantesten Feature-Tag mit 78,4% bei gleichzeitigem Recall von 64,5% und zum anderen für den Recall beim Anzeigen der Top drei Feature-Tags mit 90,3% bei gleichzeitiger Precision von 36,6%. Dabei erreichen die Werte des $F_{0,5}$-Maß mit 75,2% beim Anzeigen des Top-Feature und des F_2-Maß mit 69,8% beim Anzeigen der Top drei Feature-Tags, welche sich in einem guten Bereich befinden. Die eigenen Überlegungen zur Optimierungen brachten dabei eine Verbesserung von 25,5% Precision bei gleichzeitiger Verbesserung des Recalls von 21% beim Anzeigen des

Top-Feature. Außerdem brachte dies eine Verbesserung des Recalls von 6,4% bei gleichzeitiger Verbesserung der Precision von 2,6% beim Anzeigen der Top drei Feature-Tags.

7.2 Ausblick

Bisher werden dem Entwickler beim Anzeigen von Empfehlungen der Feature-Tags keine zusätzlichen Informationen bereitgestellt. Ein Beispiel wären hier Wahrscheinlichkeitswerte für die Feature-Tags. Dadurch könnte man den Entwickler zusätzlich unterstützen, zum einen um ihm generell eine Einschätzung zu geben, wie wahrscheinlich die vorgeschlagenen Feature-Tags zu dem Code passen und zum andern, wenn er nicht weiß, ob der Codeabschnitt nun ein oder mehrere Features implementiert. Dabei stellt es sich jedoch als technisch schwierig dar, gleich wahrscheinliche Feature-Tags in einer Form anzuzeigen, da es durch die Verwendung des Content Assist immer eine Reihenfolge geben muss, die ein Feature-Tag selbst bei gleicher Wahrscheinlichkeit über dem anderen anordnet. Wäre dies jedoch möglich, wäre es dem Entwickler aber auch unklar, wenn der Code nur ein Feature implementiert und er sich zwischen einen der beiden entscheiden müsste.
Des Weiteren ist es sicherlich interessant, welche Werte für Precision und Recall die anderen Ansätze aus der Literaturrecherche bei der Umsetzung für Feature-Tags erzielen würden. Da der Prototyp erweiterbar entworfen und implementiert wurde, kann man ohne großen Aufwand den derzeitigen Algorithmus austauschen. Das ermöglicht leicht den Vergleich verschiedener Algorithmen und somit die Auswahl des Algorithmus, der die besten Ergebnisse auf den gegebenen Studentenprojekten erzielt.

Literaturverzeichnis

[1] Sven Apel und Christian Kästner. “An overview of feature-oriented software development.”In: Journal of Object Technology 8.51 S. 49–84. (2009)

[2] Ortu M., Destefanis G., Adams B., Murgia A., Marchesi M. und Tonelli R.: The JIRA Repository Dataset: Understanding Social Aspects of Software Development. In Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering, Beijing, China (2015)

[3] Seiler M. und Paech, B.: Using Tags to Support Feature Management Across Issue Tracking Systems and Version Control Systems. In: Grünbacher P., Perini A. (eds) Requirements Engineering: Foundation for Software Quality. REFSQ 2017. Lecture Notes in Computer Science, vol 10153. Springer, Cham (2017)

[4] Hale M., Jorgenson N. und Gamble R.: Analyzing the role of tags as lightweight traceability links. In: Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering. ACM. S. 71–74. (2011)

[5] Robillard, M. P., Maalej, W., Walker, R. J. und Zimmermann, T.: Recommendation Systems in Software Engineering. Springer-Verlag Berlin Heidelberg (2014)

[6] Bruch, M., Monperrus, M. und Mezini, M.: Learning from examples to improve code completion systems. In Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE '09). ACM, New York, NY, USA, 213-222 (2009)

[7] Asaduzzaman, M., Roy, C. K., Schneider, K. A. und Hou, D.: CSCC: Simple, Efficient, Context Sensitive Code Completion, IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, 2014, pp. 71-80 (2014)

[8] Nguyen, A. T., Nguyen, T. T., Nguyen, H. A., Tamrawi, A., Nguyen, H. V., Al-Kofahi, J., und T. N. Nguyen.: Graph-based pattern-oriented, context-sensitive source code completion. In Proceedings of the 2012 International Conference on Software Engineering, pages 69-79, 2012.

[9] Levenshtein V. I.: Binary codes capable of correcting deletions, insertions, and reversals. In Soviet Physics Doklady, 10(8) S. 707–710. (1966)

[10] Proksch, S., Lerch, J. und Mezini, M.: Intelligent Code Completion with Bayesian Networks, ACM Transactions on Software Engineering and Methodology (TOSEM), v.25 n.1, p.1-31 (2015)

[11] Cover, T. und Hart, P.: Nearest neighbor pattern classification. IEEE Transactions on Information Theory (1967)

[12] Charikar, M. S.: “Similarity estimation techniques from rounding algorithms”, in Proc. STOC, pp. 380-388. (2002)

[13] Lauesen, S. und Kuhail, M.: Task descriptions versus use cases, Requirements Engineering: Foundation for Software Quality. REFSQ 2011. Lecture Notes in Computer Science, vol 6606. Springer, Berlin, Heidelberg (2011)

Anhang

Tabelle A.1: Ergebnisse der Evaluation beim *MasterPatientIndex*

Codedateien	Implementierte Feature	Empfohlene Feature
/ise/app.js	PatientDataManagement	PatientDataFeedQueryRetreive PatientDataManagement SystemAdministration
/ise/lib/auth/oauth_model.js	SystemAdministration	PatientDataFeedQueryRetreive SystemAdministration HealthcareProviderQueryRetreive
/ise/lib/controller/acl.js	SystemAdministration	HealthcareProviderQueryRetreive SystemAdministration PatientDataFeedQueryRetreive
/ise/lib/controller/cfg.js	PatientDataFeedQueryRetreive	SystemAdministration HealthcareProviderQueryRetreive PatientDataFeedQueryRetreive
/ise/lib/controller/error.js	SystemAdministration, PatientDataManagement	HealthcareProviderQueryRetreive SystemAdministration PatientDataFeedQueryRetreive
/ise/lib/controller/organization/ createOrganization.js	HealthcareProviderQueryRetreive	HealthcareProviderQueryRetreive PatientDataFeedQueryRetreive SystemAdministration
/ise/lib/controller/organization/ deleteOrganization.js	HealthcareProviderQueryRetreive	HealthcareProviderQueryRetreive HealthcareProviderDataManagement PatientDataFeedQueryRetreive
/ise/lib/controller/organization/ searchOrganization.js	HealthcareProviderQueryRetreive	HealthcareProviderQueryRetreive PatientDataFeedQueryRetreive HealthcareProviderDataManagement
/ise/lib/controller/organization/ updateOrganization.js	HealthcareProviderQueryRetreive	HealthcareProviderQueryRetreive PatientDataFeedQueryRetreive HealthcareProviderDataManagement
/ise/lib/controller/ organziation.js	HealthcareProviderQueryRetreive	PatientDataFeedQueryRetreive HealthcareProviderQueryRetreive SystemAdministration
/ise/lib/controller/patient.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive PatientDataManagement HealthcareProviderQueryRetreive
/ise/lib/controller/patient/ assignPatient.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive PatientDataManagement HealthcareProviderQueryRetreive
/ise/lib/controller/patient/ createPatient.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive PatientDataManagement HealthcareProviderQueryRetreive
/ise/lib/controller/patient/ fetchAll.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive SystemAdministration HealthcareProviderQueryRetreive
/ise/lib/controller/patient/ getPatientByKeySearch.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive PatientDataManagement HealthcareProviderQueryRetreive
/ise/lib/controller/patient/ getPatientByPatientID.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive PatientDataManagement HealthcareProviderQueryRetreive

/ise/lib/controller/patient/getPatientByPersonID.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive
		PatientDataManagement
		HealthcareProviderQueryRetreive
/ise/lib/controller/patient/getPersonByExternalPatientIdentifier.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive
		PatientDataManagement
		SystemAdministration
/ise/lib/controller/patient/getPersonByPatientID.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive
		PatientDataManagement
		HealthcareProviderQueryRetreive
/ise/lib/controller/patient/updatePatient.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive
		PatientDataManagement
		SystemAdministration
/ise/lib/controller/practitioner.js	HealthcareProviderQueryRetreive	SystemAdministration
		HealthcareProviderDataManagement
		PatientDataFeedQueryRetreive
/ise/lib/controller/user.js	SystemAdministration	HealthcareProviderQueryRetreive
		SystemAdministration
		PatientDataFeedQueryRetreive
/ise/lib/logger/logger.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive
		PatientDataManagement
		HealthcareProviderQueryRetreive
/ise/lib/middleware/acl_middleware.js	SystemAdministration	SystemAdministration
		PatientDataFeedQueryRetreive
		HealthcareProviderQueryRetreive
/ise/lib/mongodb.js	PatientDataFeedQueryRetreive	SystemAdministration
		PatientDataFeedQueryRetreive
		HealthcareProviderQueryRetreive
/ise/lib/routes.js	SystemAdministration	SystemAdministration
		PatientDataFeedQueryRetreive
		HealthcareProviderQueryRetreive
/ise/lib/schema/acl.js	SystemAdministration	SystemAdministration
		PatientDataFeedQueryRetreive
		HealthcareProviderQueryRetreive
/ise/lib/schema/acl_presets.js	SystemAdministration	SystemAdministration
		PatientDataFeedQueryRetreive
		HealthcareProviderQueryRetreive
/ise/lib/schema/datatypes_hl7.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive
		HealthcareProviderQueryRetreive
		SystemAdministration
/ise/lib/schema/organization_hl7.js	HealthcareProviderQueryRetreive	SystemAdministration
		PatientDataFeedQueryRetreive
		HealthcareProviderQueryRetreive
/ise/lib/schema/patient_hl7.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive
		PatientDataManagement
		HealthcareProviderQueryRetreive
/ise/lib/schema/person.js	PatientDataFeedQueryRetreive	PatientDataFeedQueryRetreive
		HealthcareProviderQueryRetreive
		SystemAdministration
/ise/lib/schema/practitioner_hl7.js	SystemAdministration	HealthcareProviderQueryRetreive
		PatientDataFeedQueryRetreive
		HealthcareProviderDataManagement
/ise/lib/services/acl_service.js	SystemAdministration	SystemAdministration
		PatientDataFeedQueryRetreive
		HealthcareProviderQueryRetreive
/ise/public/controller/PatientForm.js	PatientDataManagement	PatientDataFeedQueryRetreive
		PatientDataManagement
		HealthcareProviderDataManagement

/ise/public/controller/Routes.js	SystemAdministration	SystemAdministration
		PatientDataManagement
		HealthcareProviderQueryRetreive
/ise/public/functions.js	PatientDataManagement	PatientDataManagement
		PatientDataFeedQueryRetreive
		HealthcareProviderQueryRetreive
/ise/public/views/Attribute.js	PatientDataManagement	PatientDataManagement
		HealthcareProviderDataManagement
		SystemAdministration
/ise/public/views/ CommonFormSchema.js	PatientDataManagement	PatientDataManagement
		HealthcareProviderDataManagement
		HealthcareProviderQueryRetreive
/ise/public/views/ EditThresholds.js	PatientDataManagement	SystemAdministration
		HealthcareProviderDataManagement
		PatientDataManagement
/ise/public/views/FormBuilder.js	PatientDataManagement	PatientDataManagement
		HealthcareProviderDataManagement
		PatientDataFeedQueryRetreive
/ise/public/views/GeoMap.js	HealthcareProviderDataManagement	PatientDataManagement
		HealthcareProviderDataManagement
		PatientDataFeedQueryRetreive
/ise/public/views/Organization/ OrganizationEdit.js	HealthcareProviderDataManagement	PatientDataManagement
		SystemAdministration
		PatientDataFeedQueryRetreive
/ise/public/views/organization/ OrganizationListAgent.js	HealthcareProviderDataManagement	HealthcareProviderDataManagement
		PatientDataManagement
		SystemAdministration
/ise/public/views/organization/ OrganizationSearchGeo.js	HealthcareProviderDataManagement	HealthcareProviderDataManagement
		SystemAdministration
		PatientDataManagement
/ise/public/views/Paginator.js	HealthcareProviderDataManagement	PatientDataManagement
		HealthcareProviderDataManagement
		SystemAdministration
/ise/public/views/patient/ PatientEdit.js	PatientDataManagement	PatientDataManagement
		SystemAdministration
		PatientDataFeedQueryRetreive
/ise/public/views/patient/ PatientList.js	PatientDataManagement	PatientDataManagement
		PatientDataFeedQueryRetreive
		SystemAdministration
/ise/public/views/patient/ PatientListAgent.js	PatientDataManagement	PatientDataManagement
		PatientDataFeedQueryRetreive
		SystemAdministration
/ise/public/views/patient/ PatientListBox.js	PatientDataManagement	PatientDataManagement
		PatientDataFeedQueryRetreive
		SystemAdministration
/ise/public/views/patient/ PatientListing.js	PatientDataManagement	PatientDataManagement
		PatientDataFeedQueryRetreive
		SystemAdministration
/ise/public/views/patient/ PatientOrigin.js	PatientDataManagement	PatientDataManagement
		PatientDataFeedQueryRetreive
		SystemAdministration
/ise/public/views/patient/ PatientSearch.js	PatientDataManagement	PatientDataManagement
		PatientDataFeedQueryRetreive
		HealthcareProviderDataManagement
/ise/public/views/practitioner/ EditPractitioner.js	HealthcareProviderDataManagement	SystemAdministration
		HealthcareProviderDataManagement
		PatientDataManagement

/ise/public/views/practitioner/ ListPractitioner.js	HealthcareProviderDataManagement	HealthcareProviderDataManagement SystemAdministration PatientDataManagement
/ise/public/views/practitioner/ SearchPractitioner.js	HealthcareProviderDataManagement	HealthcareProviderDataManagement PatientDataManagement SystemAdministration
/ise/public/views/user/ CreateACL.js	SystemAdministration	SystemAdministration PatientDataFeedQueryRetreive HealthcareProviderQueryRetreive
/ise/public/views/user/ EditUser.js	SystemAdministration	SystemAdministration HealthcareProviderDataManagement PatientDataManagement
/ise/public/views/user/ ListUser.js	SystemAdministration	SystemAdministration HealthcareProviderDataManagement PatientDataManagement
/ise/public/views/user/ ListUserAgent.js	SystemAdministration	SystemAdministration PatientDataManagement HealthcareProviderDataManagement
/ise/public/views/user/ ManageACL.js	SystemAdministration	SystemAdministration HealthcareProviderDataManagement PatientDataManagement
/ise/public/views/user/ SearchUser.js	SystemAdministration	SystemAdministration PatientDataManagement HealthcareProviderDataManagement

Tabelle A.2: Ergebnisse der Evaluation beim *StudentNavigator*

Code File	**Implemented Features**	**Empfohlene Feature**
ElevatorManager.java	RouteFinding, Filter	RouteFinding Filter Tweets
extern/AsyncRetrieve TweetsTask.java	Tweets	Tweets RouteFinding Transportation
extern/IAsyncRetrieve TweetsResponse.java	Tweets	Tweets Transportation RouteFinding
extern/IOpnvManager.java	Transportation	Transportation Tweets RouteFinding
extern/OpnvManager.java	Transportation	RouteFinding Filter Transportation
extern/QueryCallback.java	Transportation	Transportation RouteFinding Tweets
extern/RnvDeparture.java	Transportation	Transportation Tweets Filter
extern/RnvJourney.java	Transportation	Transportation RouteFinding Tweets
FilterManager.java	Filter	Filter RouteFinding Tweets

InfoBoxManager.java	RouteFinding, Filter, Transportation	Filter RouteFinding Tweets
InfoBoxPersonManager.java	RouteFinding	RouteFinding Filter Tweets
LocationHandler.java	RouteFinding	RouteFinding Filter Tweets
MainActivity.java	RouteFinding, Filter, Tweets, Transportation	RouteFinding Filter Transportation
model/api/DataManager.java	RouteFinding, Tweets, Transportation	RouteFinding Filter Transportation
model/api/IDataManager.java	RouteFinding, Transportation	Transportation RouteFinding Filter
model//api/ISearchable.java	RouteFinding, Filter	Transportation Tweets RouteFinding
model/Building.java	RouteFinding	RouteFinding Transportation Filter
model/DatabaseConfiguration Util.java	RouteFinding	RouteFinding Transportation Filter
model/DatabaseHelper.java	RouteFinding, Filter, Tweets, Transportation	RouteFinding Tweets Transportation
model/DataGenerator.java	RouteFinding	RouteFinding Transportation Filter
model/DataToFeatureIdMapper.java	RouteFinding	RouteFinding Tweets Transportation
model/Event.java	RouteFinding	RouteFinding Transportation Filter
model/EventLocation.java	RouteFinding	RouteFinding Transportation Filter
model/EventOrganization.java	RouteFinding	RouteFinding Transportation Filter
model/MapObject.java	RouteFinding, Filter	RouteFinding Transportation Filter
model/Office.java	RouteFinding	RouteFinding Transportation Filter
model/Organization.java	RouteFinding	RouteFinding Transportation Filter
model/Organizer.java	RouteFinding	RouteFinding Transportation Filter

model/Period.java	RouteFinding	RouteFinding Transportation Filter
model/Person.java	RouteFinding	RouteFinding Transportation Filter
model/Room.java	RouteFinding	RouteFinding Transportation Filter
Model/SearchResult.java	Filter	RouteFinding Transportation Filter
Model/Station.java	RouteFinding, Transportation	RouteFinding Filter Tweets
Model/Toilet.java	RouteFinding	RouteFinding Transportation Filter
NavigationInfoManger.java	RouteFinding	RouteFinding Filter Tweets
RnvDepartureArrayAdapter.java	Transportation	RouteFinding Filter Tweets
SearchBarManager.java	RouteFinding, Filter	RouteFinding Filter Tweets
SearchManager.java	RouteFinding, Filter	RouteFinding Filter Tweets
SearchResultsArrayAdapter.java	RouteFinding, Filter	Transportation RouteFinding Filter
StringConstants.java	RouteFinding, Filter, Tweets, Transportation	RouteFinding Tweets Transportation