I. Schaefer, D. Karagiannis, A. Vogelsang, D. Méndez, C. Seidl (Hrsg.): Modellierung 2018,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2018 215

Modellierung plattformübergreifender Quellcode-Entsprechungen für die koordinierte Co-Evolution portierter Software-Systeme

Tilmann Stehle Matthias Riebisch[1]

Abstract: Wird Software auf eine neue Plattform portiert, so entsteht häufig eine zusätzliche Quellcode-Basis für die neue Plattform, die gemeinsam mit dem ursprünglichen Quellcode weiterentwickelt werden muss. Doppelte Arbeit kann dabei vermieden werden, indem die portierte Implementation die Entwurfsentscheidungen sowie Terminologie und Lösungsmuster der ursprünglichen Implementation übernimmt, sodass eine vereinheitlichte Weiterentwicklung gleichartiger Codeteile ermöglicht wird. Modelle können Entsprechungen zwischen konkreten Code-Elementen wie Klassen und Methoden beider Implementationen explizit und formal erfassen, um eine solche gemeinsame Weiterentwicklung zu vereinfachen und partiell zu automatisieren. Bisherige Ansätze zur Suche nach Entsprechungen befassen sich mit der Verknüpfung von Softwareartefakten auf unterschiedlichen Ebenen, aber erlauben keinen Vergleich zwischen gleichartigen Softwareartefakten unterschiedlicher Sprachen. In diesem Paper beschreiben wir ein Verfahren zur Erhebung sprachübergreifender Entsprechungen und zeigen, wie die resultierenden Modelle zur Koordination der gemeinsamen Evolution von ursprünglicher und portierter Implementation genutzt werden können. Zur Verwirklichung des Nutzungspotentials wurden öffentlich zugängliche Erweiterungen für Entwicklungsumgebungen implementiert. Das beschriebene Verfahren zur Erhebung der Modelle wurde anhand zweier quelloffener Portierungsprojekte evaluiert.

Keywords: Modellierung; Portierung; Quellcode-Entsprechungen; Abhängigkeiten

1 Einleitung

Häufig muss eine bestehende Software auf eine neue Plattform portiert werden, um sie einem größeren Nutzerkreis zugänglich zu machen, oder neue Anwendungen zu ermöglichen. Insbesondere im Feld der Softwareentwicklung für mobile Endgeräte ist dies eine regelmäßige Herausforderung [JMK13]. Aber auch Software-Bibliotheken werden portiert, um sie auf anderen Plattformen nutzbar zu machen. Beispiele dafür sind Apache Lucene[2] oder JGit[3], deren ursprüngliche Implementationen für die Java Virtual Machine auf die .Net-Plattform übertragen wurden.
Es existieren Frameworks wie Apache Cordova [Th17a], Xamarin [Xa17] oder Unity [Un17], die es erlauben, Software für mehrere Plattformen auf einer gemeinsamen Codebasis

[1] Universität Hamburg, {stehle,riebisch}@informatik.uni-hamburg.de
[2] Website zur .Net-Implementation von Lucene: `https://lucenenet.apache.org/`
[3] Github-Seite der .Net-Implementation von JGit: https://github.com/mono/ngit

zu entwickeln. Diese können für Neuentwicklungen sinnvoll genutzt werden. Existiert aber bereits eine reife Implementation der Software für eine Plattform, so kann diese nicht einfach an die Entwurfsvorgaben eines solchen Frameworks angepasst werden. Die Übertragung hätte eine Neuentwicklung auf Basis des Frameworks zur Folge, die die ursprüngliche Implementation durch eine unreife Implementation ersetzen würde. Zudem müssen etwaige plattformspezifische Teile der Implementation über komplexe Mechanismen wie bedingtes Kompilieren realisiert und mit der gemeinsamen Codebasis verbunden werden, was die Lesbarkeit des Quelltextes vermindert. Entwickler, die vor der Aufgabe einer Portierung stehen, entscheiden sich daher regelmäßig dafür, die bestehende Implementation beizubehalten und erneuten Entwicklungsaufwand zu betreiben, um eine zusätzliche Implementation für die Zielplattform zu erstellen [JMK13]. Aus der Portierung durch erneutes Implementieren entstehen zwei parallele Entwicklungsstränge, die mit doppeltem Aufwand gepflegt werden müssen. Es erfolgt eine gemeinsame Weiterentwicklung der beiden Implementationen, die auch als Co-Evolution bezeichnet wird. Bei Änderungen in funktional gleichen Teilen der Implementationen sollten Synergieeffekte angestrebt werden, um diese doppelte Arbeit zu verhindern und die Änderungen konsistent durchzuführen.

Problemstellung: Plattformübergreifende Abhängigkeiten in der Co-Evolution portierter Software. Ursprüngliche und portierte Implementation sollen in der Regel funktional konsistent zueinander sein und gleichartig strukturiert sein. Wird eine Software beispielsweise für mobile Geräte mit unterschiedlichen Betriebssystemen bereitgestellt, so erwarten die Nutzer, dass sie auf allen Plattformen die gleichen Arbeitsabläufe mit gleichem Ergebnis durchführen können und dass ggf. ein Austausch von Daten zwischen den plattformspezifischen Implementationen möglich ist. Die Weiterentwicklung der Software muss für beide Plattformen erfolgen. Folgen die Implementationen auf beiden Plattformen einem einheitlichen Entwurf, so können Änderungen zusammengelegt werden und doppelter Aufwand kann vermieden werden. Entwickler können den Erhalt funktionaler und struktureller Konsistenzen erleichtern, indem sie für alle Plattformen dieselben Entwurfsentscheidungen und Lösungsmuster sowie eine einheitliche Terminologie einsetzen. Um die genannten Konsistenzen aufrechtzuerhalten, muss bewusst Aufwand investiert werden. Ohne diesen gezielten Aufwand würden Inkonsistenzen entstehen. Eine Ursache dafür ist, dass ursprüngliche und portierte Implementation regelmäßig von unterschiedlichen Teams weiterentwickelt werden[4]. Bei hoher Fluktuation im Entwicklungsteam sowie bei großen Systemen ist zudem die Wahrscheinlichkeit hoch, dass nicht allen Entwicklern alle Entwurfsentscheidungen bewusst sind und deswegen bei Änderungen Inkonsistenzen eingeführt werden. In der Folge verursachen die eingeführten Unterschiede zwischen den Implementationen doppelte Arbeit bei der Konzeption weiterer Änderungen für jede Plattform und die funktionale Konsistenz nimmt ab. Es ist folglich notwendig, die Gemeinsamkeiten der Implementationen explizit zu beschreiben.

[4] Diese Vermutung legt der Vergleich der Autoren der ursprünglichen Implementation mit den Autoren der portierten Implementation sowohl bei JGit (`https://github.com/eclipse/jgit/graphs/contributors` bzw. `https://github.com/mono/ngit/graphs/contributors`), als auch bei Lucene nahe (`https://github.com/apache/lucene-solr/graphs/contributors` bzw. `https://github.com/apache/lucenenet/graphs/contributors`)

Beitrag dieser Arbeit: Repräsentation, Erhebung und Nutzungspotentiale plattformübergreifender Abhängigkeitsmodelle. Um die Synchronisation zwischen den Implementationen zu unterstützen, ist es erforderlich, die Entsprechungen zwischen den Quellcode-Elementen beider Implementationen explizit zu modellieren. Unser Beitrag umfasst drei wesentliche Aspekte: Wir beschreiben eine Modellierungssprache zur formalen Erfassung plattformübergreifender Entsprechungen. Zweitens beschreiben wir ein Verfahren zur automatischen Erhebung dieser Entsprechungen. Drittens zeigen wir konkrete Nutzungspotentiale der Modelle für die koordinierte plattformübergreifende Co-Evolution auf und zeigen deren Realisierbarkeit durch Werkzeugimplementationen.

2 Verwandte Arbeiten

[Di11] gibt einen Überblick über formale Methoden zur Konsistenzerhaltung bei Änderungen gekoppelter Modelle. Manche dieser Techniken können auf die parallele plattformübergreifende Entwicklung übertragen werden. Deren Anwendung setzt allerdings eine formale und korrekte Beschreibung der Modellbeziehungen und Konsistenzbedingungen voraus. Dies ist insbesondere bei der Pflege manuell portierter Software nicht gegeben, sodass eine voll automatisierte Konsistenzerhaltung im Rahmen dieser Arbeit zunächst nicht angestrebt wird.

Als *Trace Link* bezeichnet man die explizite Verknüpfung zweier Softwareartefakte [CHGZ12]. In diesem Sinne sind auch die hier diskutierten expliziten Entsprechungen zwischen Code-Elementen der ursprünglichen Implementation und ihren portierten Pendants als Trace Links zu bezeichnen. Das Erstellen und Nutzen von Trace Links zwischen verschiedenartigen Softwareartefakten wie Anforderungsbeschreibungen und zugehörigen Quellcode-Dateien ist Gegenstand diverser Forschungsarbeiten [De07] [AAT10] [Ol10]. Diese setzen Techniken des *Information Retrieval* ein, die die Struktur des Quellcodes außer Acht lassen, weshalb sie nicht in unsere Arbeit einfließen. [Mi03] schlägt zur Erkennung von Code-Duplikaten einen Ansatz basierend auf formaler Konzeptanalyse vor, der sowohl Strukturen als auch Bezeichner in die Ähnlichkeitsanalyse einbezieht. Dieser ist jedoch hoch komplex und nimmt laut der Autoren bereits bei kleinen Code-Basen extrem viel Rechenzeit in Anspruch. Er ist damit nicht zum Auffinden von Entsprechungen zwischen laufend weiterentwickelten Code-Basen einsetzbar. Wertvoll ist allerdings die Idee der Gewichtung von Bezeichnern nach ihrer Hierarchie im Syntaxbaum, die wir in ähnlicher Weise zum Vergleich von Quellcode-Elementen nutzen.

Das Auffinden von Quellcode-Duplikaten auf Basis textueller oder struktureller Ähnlichkeiten ist ebenfalls gut erforscht [Ud13]. Die erforschten Techniken sind jedoch hauptsächlich auf Ähnlichkeiten zwischen Code-Elementen derselben Sprache anwendbar, weshalb wir sie zur Erhebung von Entsprechungen zwischen ursprünglichen Code-Elementen und ihren portierten Pendants nicht einsetzen. Das Problem der Entwicklung plattformübergreifender Software mit separater Quellcode-Basis je Plattform ist wissenschaftlich kaum beleuchtet. Gleichwohl gibt es Arbeiten, die sich mit sprachübergreifender und sprachunabhängiger

Softwareentwicklung befassen. [MS12] führt sogenannte *Semantic Links* ein, die Entwicklern sprachübergreifende Abhängigkeiten innerhalb einer Implementation aufzeigen. Sie setzen allerdings eine gegebene explizite Verbindung zwischen den Code-Elementen voraus, weshalb sie im Portierungs-Kontext nicht genutzt werden können.
[RCPO14] schlägt eine Technik zum Vergleich des Verhaltens von Anwendungen anhand ihrer Netzwerk-Kommunikation vor. Da es sich dabei um einen Black-Box-Ansatz handelt, ist aus dem Vergleich kein Rückschluss auf die ursächlichen Unterschiede und Gleichheiten im Quellcode zu ziehen. Der hier vorgestellte Ansatz bietet im Gegensatz dazu eine Basis für den Vergleich der Implementationen auf Ebene des Quellcodes.
[SKL06] und [Ti01] zeigen modellbasierte Ansätze auf, mit denen objektorientierte Strukturen und Refactorings sprachunabhängig beschrieben werden können, sodass ihre Anwendung auf Code in verschiedenen Sprachen eine einheitliche Strukturveränderung bewirkt. Unser Mechanismus für plattformübergreifende Refactorings setzt diese Äquivalenz für Refactorings in verschiedenen Sprachen voraus, um die strukturellen Entsprechungen zwischen synchron restrukturierten Code-Elemente zu wahren.
In [SR15] haben wir einen Ansatz vorgestellt, der während der initialen Portierung systematisch Entsprechungen im Design und in der Terminologie der ursprünglichen und portierten Implementation herstellt. Diese Entsprechungen können in den hier eingeführten Entsprechungsmodellen erfasst werden.

3 Repräsentation plattformübergreifender Quellcode-Entsprechungen

Um die benötigten plattformübergreifenden Quellcode-Entsprechungen zur Koordination der Co-Evolution zweier Implementationen nutzen zu können, müssen sie explizit in Modellen dargestellt werden, die wir im Weiteren als Entsprechungsmodelle bezeichnen. Die Modellierungstheorie [Th13] definiert wichtige Eigenschaften von Modellen für die Metamodell-Definition. *Purpose*: Der Zweck eines Entsprechungsmodells ist es, Aufwand bei der Co-Evolution der verknüpften Implementationen einzusparen. Dazu soll es folgende Aktivitäten unterstützen: (1)Die Suche nach korrespondierenden Code-Elementen, (2)die Navigation zwischen korrespondierenden Code-Elementen über die Grenzen verschiedener Entwicklungsumgebungen hinweg, (3)die Übertragung automatisierbarer Änderungen wie Refactorings, sowie (4)die Koordination händischer Änderungen sich entsprechender Code-Elemente. Diesen Zwecken trägt das Metamodell dadurch Rechnung, dass es Paare sich entsprechender Quellcode-Elemente einander zuordnet. Das Metamodell ist formal, sodass die Entsprechungsmodelle zur Reduktion von Aufwand *automatisiert* von Werkzeugen verarbeitet werden können, die den Zwecken 1 bis 4 dienen. *Impact*: Teil der Purpose-Eigenschaft ist der angestrebte Einfluss der Modelle, konkret die Reduktion des Aufwands für die parallele Weiterentwicklung durch Koordination und Zusammenlegung der Änderungen sich entsprechender Code-Elemente. Dazu gehören die Unterstützung von Verständnis und Navigation, das erleichterte Auffinden von korrespondierenden Code-Elementen sowie der Erhalt von Ähnlichkeiten bezüglich des Entwurfs und der eingesetzten Terminologie bei möglichst geringem Modellierungsaufwand. Aus der Forderung nach geringem Modellierungsaufwand

leitet sich die Forderung ab, die Modelle automatisiert zu erheben. Beim automatisierten Erheben der Entsprechungen herrscht Unsicherheit über die Korrektheit der gefundenen Entsprechungen. Diese Unsicherheit muss in den Entsprechungsmodellen erfasst werden, was im Metamodell vorgesehen ist. *Restrictions*: Aus dem Zweck der Modelle ergeben sich Einschränkungen für das Metamodell. Die Entsprechungsmodelle sind beispielsweise nicht dazu gedacht, über den Entwurf einer einzelnen Implementation zu diskutieren und enthalten folglich über die Entsprechungen hinaus keine anderen Beziehungen zwischen Code-Elementen. *Pragmatism*: Es gibt eine intendierte Nutzergruppe, die die Modelle auf eine bestimmte Weise einsetzen soll. Im konkreten Fall sollen die Modelle teil-automatisiert mit Werkzeugen für die oben genannten Zwecke während der Weiterentwicklung portierter Software genutzt werden. Dem entspricht das Metamodell durch die formale Definition erlaubter Modellelemente und Beziehungen, sodass entsprechende Modelle automatisiert mit Werkzeugen für Softwareentwickler verarbeitet werden können. *Amplification*: Ein Modell kann zusätzliche Informationen enthalten, die das Original nicht enthält. Die hier eingeführten Modelle enthalten explizite Entsprechungen zwischen Quellcode-Elementen, die im Quellcode nicht explizit sind. *Truncation*: Modelle abstrahieren vom Original und lassen dabei Informationen aus, die im Sinne des Zwecks irrelevant sind. Relevant für die Beschreibung der Entsprechungen sind lediglich Ursprung und Ziel sowie ein Konfidenzwert, der angibt, mit welcher Wahrscheinlichkeit eine automatisch erhobene Entsprechung korrekt ist. *Mapping*: Modelle beziehen sich stets auf ein Original, dessen Elemente sich im Modell wiederfinden. Die im Entsprechungsmodell verknüpften Repräsentationen von Quellcode-Elementen beziehen sich auf die konkreten Quellcode-Elemente in der ursprünglichen und portierten Implementation. *Idealisation*: Modelle nehmen im Allgemeinen eine zweckdienliche Vereinfachung vor. Im Fall der Entsprechungsmodelle werden lediglich 1:1-Beziehungen der Ist-Situation abgebildet. Es wird nicht berücksichtigt, dass eine Klasse beispielsweise auch mehr als einen Zweck haben kann und eventuell nicht vollständig in ihr Pendant in der Zielimplementation übertragen worden ist.

Um Quell- und Zielelement unabhängig von ihrer Art eindeutig zu identifizieren, gibt es im Metamodell (Abbildung 1) die abstrakte Klasse `Code-Element`, die ein Code-Element repräsentiert und Informationen zum Auffinden des verknüpften Elements hält.

Spezifische Klassen für Code-Elemente wie Methoden oder Typen erben von ihr. Die Klasse `Methode` speichert neben dem Namen der repräsentierten Methode eine Referenz auf den Typ, in dem die Methode definiert ist. `Typ` speichert als identifizierendes Attribut den vollständigen Bezeichner des Typs und erbt selbst von `Code-Element`, da Typen wie Klassen oder Interfaces ebenfalls Code-Elemente sind und Entsprechungen haben können. Die

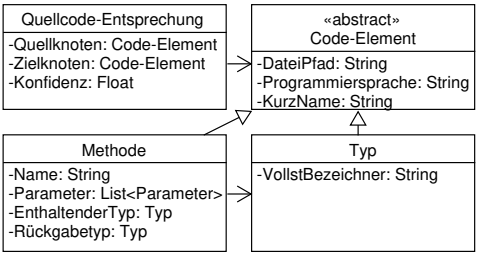

Abb. 1: Ausgewählte Klassen des Metamodells für plattformübergreifende Quellcode-Entsprechungen im UML-Klassendiagramm

Quellcode-Entsprechungen werden durch
Instanzen der Klasse `Quellcode-Entsprechung` repräsentiert. Sie halten jeweils einen Verweis auf das ursprüngliche und das portierte Code-Element sowie eine Bewertung der Ähnlichkeit zwischen den verknüpften Elementen. Diese Bewertung dient als Indikator dafür, mit welcher Wahrscheinlichkeit die identifizierte Entsprechung korrekt ist und ist dementsprechend im `Konfidenz`-Attribut einer Quellcode-Entsprechung modelliert. Die Bewertung ist notwendig, da automatisch erhobene Quellcode-Entsprechungen nicht immer mit Sicherheit zutreffen. Damit Entwickler anhand eines Entsprechungs-Modells systematisch nach dem portierten Pendant zu einem gegebenen Code-Element suchen können, werden die potentiellen Entsprechungen nach dieser Bewertung sortiert.

4 Automatische Erhebung plattformübergreifender Quellcode-Entsprechungen und Erfassung in Entsprechungsmodellen

Um den Aufwand für die Erstellung von Entsprechungsmodellen gering zu halten, schlagen wir ein Verfahren zu deren automatischen Erhebung vor. Es ermittelt Entsprechungen von Code-Elementen wie Klassen oder Methoden zwischen Quellcode der ursprünglichen und der portierten Implementation unabhängig von den eingesetzten Programmiersprachen. Die gefundenen Entsprechungen werden in Modellen gemäß des in Abschnitt 3 eingeführten Metamodells erfasst. Voraussetzung dafür ist, dass die ursprüngliche Programmiersprache und die Programmiersprache der Zielplattform demselben Paradigma folgen, sodass ein gleicher Schnitt der Quellcode-Elemente nach Zuständigkeiten möglich ist. Das wäre beispielsweise bei der Portierung einer ursprünglich funktional programmierten Software in eine objektorientierte Zeilsprache nicht der Fall.
Das Verfahren kann in zwei Teilprozesse unterteilt werden: Der erste indexiert die Code-Elemente der ursprünglichen und der portierten Implementation. Der zweite Teilprozess durchsucht den erstellten Index anhand eines gegebenen Code-Elements nach terminologisch und strukturell ähnlichen Elementen.

Der erste Teilprozess ist in Abbildung 2 dargestellt. Im ersten Schritt werden die Quellcodes sowohl der ursprünglichen als auch der portierten Implementation eingelesen und abstrakte Syntaxbäume für sie erstellt. Aus den abstrakten Syntaxbäumen wird zu jedem Quellcode-Element eine Multimenge erstellt, die die Bezeichner des Elements beinhaltet. Da diese Multimengen indexiert und später im Prozess mit einer Suchanfrage verglichen werden, bezeichnen wir sie im Sinne des Information Retrieval als *Dokument* [MRS08]. Beispielsweise werden in das Dokument zu einer Klasse neben dem Klassennamen auch sämtliche Namen von Methoden, Feldern und Variablen, eingefügt, die innerhalb der Klasse verwendet werden. So wird von der Syntax der Programmiersprache abstrahiert. Dabei soll berücksichtigt werden, dass Bezeichner auf verschiedenen Ebenen des Syntaxbaumes unterschiedlich stark zur Bedeutung eines Code-Elements beitragen. Beispielsweise ist der Klassenname für die Beschreibung einer Klasse aussagekräftiger als der Name einer lokalen Variable. Diese Strukturinformation soll erhalten bleiben. Zu diesem Zweck haben

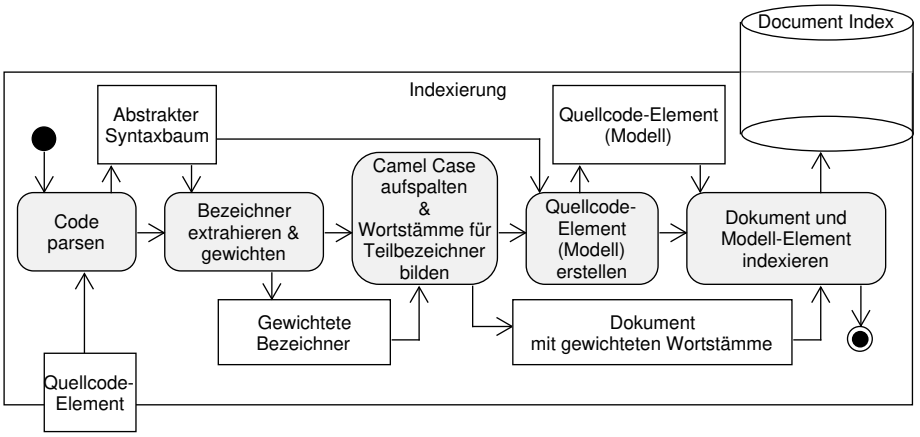

Abb. 2: Parsen und Indexieren der Quellcode-Elemente in der ursprünglichen und portierten Implementation

wir entsprechend der Grammatiken objektorientierter Sprachen wie Java, C# oder Swift eine Hierarchie abgeleitet, die festlegt, wie oft der Bezeichner eines Sprachkonstrukts in das Dokument eines Code-Elements aufgenommen wird. In das Dokument zu einer Klasse fließt beispielsweise der Bezeichner einer enthaltenen lokalen Variablen einfach ein, der Name der umschließenden Methode doppelt und der Name der Klasse vierfach. Zwischen dem Bezeichner eines enthaltenen Elements und seinem umschließenden Element liegt also der Faktor 2.

Die in das Dokument aufgenommenen gewichteten Bezeichner werden im nächsten Schritt in ihre Bestandteile zerlegt, auf ihren Wortstamm reduziert und Großbuchstaben durch Kleinbuchstaben ersetzt, sodass die spätere Suche auch Ähnlichkeiten erkennt, wenn Bezeichner sich nur in Teilen gleichen. Der Bezeichner *DateConverter* würde beispielsweise in *date* und *convert* zerlegt. Das Kompositum bleibt zusätzlich im Dokument enthalten, um zwischen Bezeichnern wie *ServiceLocation* und *LocationService* unterscheiden zu können. Das Dokument wird gemeinsam mit einer Instanz der Klasse `Quellcode-Element` (siehe Abb. 1) indexiert, die das Quellcode-Element eindeutig identifiziert.

Der zweite Teilprozess ist in Abbildung 3 dargestellt. Er sucht für ein gegebenes Code-Element nach einer Entsprechung im Index. Dazu wird zu einem gewählten Code-Element die identifizierende Instanz von `Quellcode-Element` (siehe Abb. 1) erzeugt und im Index das zugehörige Dokument Q mit den Teilbezeichnern q_i nachgeschlagen. Es wird mit jedem anderen Dokument D des Index anhand der Ähnlichkeitsfunktion Okapi BM25 [RZ09] verglichen, die als eine der erfolgreichsten Text-Retrieval Algorithmen ohne Beachtung der

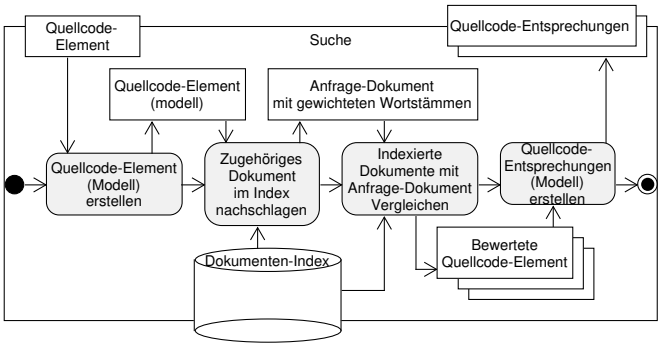

Abb. 3: Suche nach Trace Links zu einem gegebenen Quellcode-Element im erzeugten Index

Syntax gilt (ebd.). Okapi BM25 wird als numerischer Wert für die Ähnlichkeit wie folgt berechnet:

$$\sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i, D) \cdot (2.2)}{f(q_i, D) + 1.2 \cdot (0.25 + 0.75 \cdot \frac{|D|}{avgdl})}$$

$avgdl$ ist dabei die durchschnittliche Anzahl von Bezeichnern in einem Dokument und $|D|$ ist die konkrete Anzahl der Bezeichner in D. $f(q_i, D)$ ist die Häufigkeit des Auftretens von q_i in D. Dieser Wert wird für wichtige Bezeichner wie Klassennamen durch die Gewichtung der Bezeichner bei der Indexierung erhöht. Die Funktion $IDF(q_i)$ in der Gleichung ist die umgekehrte Dokumentenhäufigkeit (**I**nverse **D**ocument **F**requency) des Bezeichners q_i. Sie berechnet einen Wert dafür, wie gut das Enthaltensein von q_i ein Dokument von anderen unterscheidet. Sie wird berechnet als:

$$IDF(q_i) = log(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5})$$

wobei N die Zahl aller Dokumente im Index ist und $n(q_i)$ die Anzahl der Dokumente, die q_i enthalten. Die bei der Indexierung vorgenommene Gewichtung von Bezeichnern hat keinen Einfluss hierauf. Die rechte Seite der Formel kann für Dokumente durchschnittlicher Länge zu folgender Teilformel vereinfacht werden:

$$\frac{f(q_i, D) \cdot (2.2)}{f(q_i, D) + 2.2}$$

Sie berechnet einen Wert für die Bedeutung jedes Bezeichners q_i des Abfragedokuments für das zu vergleichende Dokument D. Die relative Dokumentlänge $\frac{|D|}{avgdl}$ im Nenner führt dazu, dass lange Dokumente niedriger bewertet werden. Dahinter verbirgt sich die Annahme, dass lange Dokumente typischerweise weniger spezifisch sind.
Okapi BM25 wird für jedes indizierte Dokument berechnet, sodass jeweils ein Wert für die Ähnlichkeit zum Anfrage-Dokument Q vorliegt. Die zum indizierten Dokument gehörige

Instanz von `Code-Element` wird jeweils mit dem `Code-Element` des Anfrage-Dokuments *Q* durch eine `Quellcode-Entsprechung` verknüpft, sofern der Ähnlichkeitswert nicht Null beträgt. Die Entsprechung erhält als Wert für das `Konfidenz`-Attribut den errechneten Ähnlichkeitswert. Die erzeugten Entsprechungen können durch Werkzeuge nach `Konfidenz` geordnet werden, sodass man systematisch nach korrekten Entsprechungen zu einem Code-Element suchen kann, indem man Entsprechungen mit hohem Konfidenzwert zuerst verfolgt.

5 Nutzung von Entsprechungsmodellen für die Co-Evolution von ursprünglicher und portierter Implementation

Die erzeugten plattformübergreifenden Entsprechungsmodelle können genutzt werden, um die koordinierte Co-Evolution mehrerer Implementationen für unterschiedliche Plattformen zu unterstützen. Konkret unterstützen wir drei Tätigkeiten: (1)Das Auffinden des implementierenden Quellcodes zu einem gegebenen Entwurfsbaustein oder einem gegebenen Verhalten, welches als *Feature Location* oder auch *Concept Location* [RW02] bezeichnet wird, (2) die Koordination von Änderungen verknüpfter Elemente und (3) die automatisierte Synchronisierung von Änderungen an äquivalenten Quellcode-Elementen.

Nutzung der Entsprechungsmodelle für Feature Location. [Si16] und [Fr14] stellen fest, dass Entwickler für das Verstehen des Codes und das Identifizieren der zu ändernden Code-Teile erheblichen Aufwand in das Navigieren entlang von Quellcode-Abhängigkeiten investieren. Sie stellen dadurch Bezüge zwischen Code-Elementen her und verstehen somit, wo im Quellcode welche Funktionalität implementiert ist.
Durch die Nutzung der eingeführten Entsprechungsmodelle kann dieses Vorgehen auf eine Implementation beschränkt werden: Ist das zu ändernde Code-Element in der ersten Implementation gefunden, kann das anzupassende korrespondierende Element in der zweiten Implementation anhand der Quellcode-Entsprechungen ohne doppelten Aufwand für Feature Location ermittelt werden.
Dazu muss die Navigation entlang der Entsprechungen in die eingesetzten Entwicklungsumgebungen integriert werden.

Koordination von Änderungen verknüpfter Elemente. Die Co-Evolution mehrerer Implementationen kann auf Basis der Entsprechungsmodelle koordiniert werden: Nach Änderung eines Code-Elements mit einer Entsprechung in der zweiten Implementation kann das Modell genutzt werden, um an dieser Entsprechung einen TODO-Kommentar zu verfassen, der die korrespondierende Angleichung fordert. TODO-Kommentare werden in der Softwareentwicklung häufig als halbformale Markierungen für ausstehende Aufgaben am Code hinterlegt. Der für das markierte Code-Element zuständige Entwickler wird so auf die Notwendigkeit hingewiesen, die beiden Implementationen durch eine entsprechende Änderung einander wieder anzugleichen.

Automatisierte Koordination von Änderungen an einander entsprechenden Quellcode-Elementen. Manche Änderungen an einander entsprechenden Quellcode-Elementen können auf Basis der Entsprechungsmodelle koordiniert und in beiden Implementationen synchronisiert durchgeführt werden. Dafür müssen zwei Voraussetzungen erfüllt sein: Erstens müssen sich die zu ändernden Quellcode-Elemente in der zu ändernden Eigenschaft (z.B. Name oder Sprachkonstrukt) gleichen; zweitens muss die Änderung in beiden Implementationen in gleicher Weise umsetzbar sein. Zum Beispiel können anstehende Refactorings gleichzeitig auf beide Implementationen angewendet werden, um strukturelle Entsprechungen zu bewahren. Das einfachste Beispiel ist die Umbenennung zweier sich entsprechender Klassen. Um die Konsistenz der Benennungen zu wahren, ist es sinnvoll, die Umbenennung gleichzeitig in beiden Implementationen durchzuführen. Dies kann auf Basis der Entsprechungsmodelle automatisiert werden.

6 Evaluierung

Um das in Abschnitt 4 beschriebene Verfahren zum Auffinden von Quellcode-Entsprechungen zu evaluieren, wurde es auf zwei Portierungsprojekte angewendet. Das erste Projekt namens Twidere ist ein quelloffener Twitter-Client, der ursprünglich für das Betriebssystem Android entwickelt wurde[5] und aktuell manuell für das Betriebssystem iOS reimplementiert wird[6]. Bei der manuellen Übertragung aus den Programmiersprachen Java und Kotlin (Android) in die Sprache Swift (iOS) sind an vielen Stellen Ähnlichkeiten in Form gleicher Entwurfsbausteine und gleicher Bezeichner für übertragene Lösungsmuster zwischen den Implementationen entstanden. Trotzdem unterscheiden sich beide Implementationen strukturell und funktional stark. Beispielsweise wurden die Interfaces `FavoritesResources`, `FriendsFollowersResources` , `HelpResources` und 3 weitere Interfaces der Android-Implementation nicht explizit nach Swift übertragen. Die Operationen dieser Interfaces sind in der Klasse `MicroblogService` vereint, die kein Interface explizit implementiert. Die iOS-Implementation enthält zudem viele Funktionalitäten der Android-Version nicht. Wir halten Twidere damit für ein Portierungsprojekt mit typischen Schwächen bei der Konsistenz zwischen ursprünglicher und portierter Implementation, wenn kein Aufwand in die Konsistenz investiert wird.
Das zweite zur Evaluation betrachtete Projekt ist Apache Lucene.Net[7]. Dabei handelt es sich um die portierte Implementation des Such-Frameworks Apache Lucene, das teilweise mit Code-Convertoren in C# übersetzt und dann manuell angepasst wird. Dabei wird besonderes Augenmerk auf Konsistenz bei der Strukturierung des Quelltextes gelegt, obgleich .Net-typische Schnittstellen angestrebt werden [Th17b]. Durch die hohe Konsistenz und dadurch, dass die ursprüngliche und die portierte Implementation von unterschiedlichen Entwicklern bearbeitet wird, stellt es einen guten Kontrast zu Twidere dar. Mit diesen beiden Projekten kann die Leistungsfähigkeit des Verfahrens bei niedriger und hoher Konsistenz

[5] Link zum Repository: `https://github.com/TwidereProject/Twidere-Android`
[6] Link zum Repository: `https://github.com/TwidereProject/Twidere-iOS`
[7] Website zur .Net-Implementation von Lucene: `https://lucenenet.apache.org/`

verglichen werden.
Wir haben 50 der 1128 Typen der Android-Implementation von Twidere ausgewählt und ihre Entsprechungen in den 132 Typen der iOS-Implementation identifiziert. Dazu haben wir systematisch die Ordnerstruktur des Projektes mittels Tiefensuche durchlaufen und Entsprechungen in der Swift-Implementation ausfindig gemacht. Konnte keine Entsprechung eindeutig identifiziert werden, wurde der Typ von der Evaluation ausgeschlossen. Daraus ergaben sich 62 Entsprechungen, die wir als Vergleichsdaten für die Evaluation nutzen[8]. Die identifizierten Links wurden von einer Person geprüft, die ansonsten nicht an dieser Arbeit beteiligt ist. Auf der Ebene von Methoden erwies es sich als schwierig, im Twidere-Projekt eindeutige Entsprechungen ausfindig zu machen, die andere Methoden als Getter und Setter mit ihren Pendants verknüpfen. Diese Methoden stellen das Verfahren angesichts ihrer kürze und banalen Funktionalität nicht auf die Probe, sodass wir keine Evaluation auf Methodenebene in Twidere vorgenommen haben.

In gleicher Weise haben wir Entsprechungen für die Kernfunktionalität von Lucene identifiziert, die im Ordner `core` der Implementation zu finden ist. Darin sind 1871 Typen definiert. Zu den 50 ersten Typen haben wir die Pendants in der C#-Implementation identifiziert[9]. Zusätzlich haben wir zu jedem dieser 50 Typen das Pendant der ersten in ihm definierten Operation identifiziert[10]. Getter und Setter haben wir dabei übersprungen. Gab es in einem Typ ausschließlich Getter und Setter, so wurde der Typ übersprungen und eine Operation eines Typs gewählt, der weiter hinten in der Sortierung nach Ordnerstruktur liegt.

Die manuell gewonnenen Entsprechungen haben wir als Sollwerte genutzt, um unser Verfahren anhand der Metrik *Mean Average Precision* (MAP) [MRS08, s.159] zu bewerten. Bei MAP handelt es sich um eine Metrik zur Bestimmung der Qualität von Suchergebnissen, die im Gegensatz zu Precision und Recall die Reihenfolge der Ergebnisse berücksichtigt. Sie ist eine der gebräuchlichsten Metriken im Bereich des Information Retrieval [MRS08, s.159]. MAP berechnet den durchschnittlichen Precision-Wert bei zunehmendem Recall-Wert wie folgt:

$$MAP(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \left(\frac{1}{m_j} \sum_{k=1}^{m_j} Precision(R_{jk})\right)$$

wobei Q die Menge aller Informationsbedarfe q_j, also aller Anfragen nach Code-Entsprechungen ist. Eine perfekte Suche würde zu einem gegebenen Code-Element $e_{j(OrigImpl)}$ der Originalimplementation die Menge $\{e_{1(ZielImpl)}, e_{2(ZielImpl)} \ldots, e_{m_j(ZielImpl)}\}$ mit allen m_j korrespondierenden Code-Elementen der Zielimplementation liefern. MAP bildet für jede der m_j korrekten Entsprechungen Teillisten R_{jk}, die nur die gelieferten Ergebnisse bis zur k-ten korrekten Entsprechung enthalten. In der Teilliste R_{j1} sind also alle Suchergebnisse bis zur ersten

[8] manche Code-Elemente in der Android-Implementation haben mehrere Entsprechungen in der iOS-Implementation. Die vollständige Liste der Entsprechungen ist unter `https://swk-www.informatik.uni-hamburg.de/~stehle/TwidereLinks.pdf` abrufbar.

[9] Diese sind unter `https://swk-www.informatik.uni-hamburg.de/~stehle/LuceneTypeLinks.pdf` abrufbar.

[10] Diese sind unter `https://swk-www.informatik.uni-hamburg.de/~stehle/LuceneMethodLinks.pdf` abrufbar.

korrekten Entsprechung enthalten; in R_{j2} alle Ergebnisse bis zur zweiten korrekten Entsprechung. Für alle Teil-Ergebnislisten wird der durchschnittliche Precision-Wert ermittelt. Dies wird für alle Suchanfragen wiederholt und abschließend der Durchschnitt über alle Suchanfragen gebildet.
Für die identifizierten Entsprechungen der Typen des Twidere-Projekts egibt sich ein MAP von 0,75; für die identifizierten Entsprechungen der Typen von Lucene bzw. Lucene.Net ein MAP von 0,86 und für die verknüpften Methoden von Lucene ebenfalls ein MAP von 0,86. Nach unserer Kenntnis gibt es kein anderes Verfahren, das sprachübergreifende Entsprechungen von Code-Elementen identifiziert, sodass ein Vergleich nicht möglich ist. Im Vergleich mit modernen Verfahren zur Verknüpfung natürlichsprachlicher Dokumente mit dem zugehörigen Quellcode erreicht unser Verfahren sehr gute MAP-Werte. Diese erreichen einen MAP zwischen 0,7 und 0,76 [ZLL13]. Insgesamt befinden wir unser Verfahren damit für effektiv. Die Grenzen des Verfahrens werden bei der Suche nach Entsprechungen für plattformspezifisch benannte Code-Elemente sichtbar. Beispielsweise wird die Methode `close()` der Klasse `org.apache.lucene.analysis.CharFilter` nicht mit ihrem Pendant **Lucene.Net.Analysis.CharFilter.dispose()** verknüpft, da eine plattformspezifische Bezeichnung gewählt wurde und wenig Gemeinsamkeiten zwischen den verwendeten Bezeichnern innerhalb der Methoden bestehen.
Unser Verfahren zur Erhebung von Entsprechungen wurde im Rahmen zweier studentischer Arbeiten in ein Plugin für die Entwicklungsumgebung IntelliJ IDEA integriert und in [Gr17] hinsichtlich Benutzbarkeit evaluiert. Der zugehörige Quellcode steht auf GitHub zur Verfügung[11]. Das Plugin bietet neben der Erkennung von Quellcode-Entsprechungen zwischen Java-, Kotlin- und Swift-Code auch die Navigation entlang der Entsprechungen und die Erzeugung von TODO-Kommentaren. Dazu wird ein Bezeichner, z.B. einer Methode oder einer Klasse markiert und über das Kontext-Menü eine Liste potentieller Entsprechungen aufgerufen, über die der Entwickler zur portierten Version navigieren, oder dort einen Kommentar hinterlegen kann. Entwickler können damit nach Änderungen an Java-Elementen einen entsprechenden Vermerk am zugehörigen Swift-Element erstellen, um die Angleichung der Swift-Implementation anzustoßen.
Ein zweites Plugin bietet für automatisch konvertierten Code das plattformübergreifende Umbenennen einer Methode und ihrer Entsprechungen in einer verknüpften Swift-Implementation[12]. Der Entwickler findet nach Installation des Plugins den zusätzlichen Eintrag *rename method and linked elements* im Kontext-Menü für Refactorings. Er kann nach Auswahl dieser Option aus der Liste potenzieller Entsprechungen eine Swift-Methode auswählen und einen neuen Namen für beide sich entsprechenden Methoden inklusive ihrer Aufrufe automatisch vergeben lassen.

[11] Der Quellcode des Plugins sowie eine Anleitung zum Kompilieren und Nutzen des Plugins finden sich unter: `https://github.com/TilStehle/Java-Kotlin-Swift-Trace-Link-Recovery`
[12] Der Quellcode des Plugins sowie eine Anleitung zum Kompilieren und Nutzen des Plugins finden sich unter: `https://github.com/TilStehle/Cross-Platform-Traceability`

7 Fazit und Ausblick

In diesem Paper haben wir einen Ansatz zur Modellierung von Quellcodeentsprechungen in der plattformübergreifenden Entwicklung und ein sprachunabhängiges Verfahren zu deren Erhebung entwickelt. Es führt zur Vermeidung doppelter Arbeit bei der plattformübergreifenden Co-Evolution, weil Feature Location sowie automatisierbare Änderungen nur einmal durchgeführt werden müssen und nicht automatisierbare Änderungen koordiniert werden. Um das Verfahren zur Erhebung der Quellcode-Entsprechungen zu evaluieren, haben wir es erfolgreich auf zwei quelloffene Portierungsprojekte angewendet. Als Proof of Concept haben wir Werkzeuge entwickelt, die dieses Verfahren implementieren und die koordinierte Co-Evolution unterstützen.
Unsere zukünftigen Arbeiten werden die Nutzung der beschriebenen Entsprechungsmodelle für automatisierte Konsistenzprüfungen zwischen den plattformspezifischen Implementationen untersuchen. Darüber hinaus lohnt es sich, Bibliotheken von Entsprechungen auf der Ebene plattformtypischer Strukturen aufzubauen, um konkrete Quellcode-Entsprechungen trotz plattformspezifischer Strukturen aufzufinden. Zudem könnte die Qualität der Entsprechungsmodelle durch das Wissen und Nutzungsverhalten der Entwickler verbessert werden, was weitere Informationen im Modell erfordert.

Literaturverzeichnis

[AAT10] Asuncion, Hazeline U.; Asuncion, Arthur U.; Taylor, Richard N.: Software Traceability with Topic Modeling. In: ICSE2010, Vol. 1. ACM, S. 95–104, 2010.

[CHGZ12] Cleland-Huang, Jane; Gotel, Orlena; Zisman, Andrea: Software and Systems Traceability. Springer, 2012.

[De07] De Lucia, Andrea; Fasano, Fausto; Oliveto, Rocco; Tortora, Genoveffa: Recovering Traceability Links in Software Artifact Management Systems Using Information Retrieval Methods. ACM TOSEM, 16(4), September 2007.

[Di11] Diskin, Zinovy: Model Synchronization: Mappings, Tiles, and Categories. In: GTTSE 2009. Springer, S. 92–165, 2011.

[Fr14] Fritz, Thomas; Shepherd, David C.; Kevic, Katja; Snipes, Will; Bräunlich, Christoph: Developers' code context models for change tasks. In: FSE2014. ACM, S. 7–18, 2014.

[Gr17] Greiert, Gerrit: , Entwicklung eines Plugins in IntelliJ IDEA zum Auffinden von Quellcode-Entsprechungen. `https://github.com/TilStehle/Java-Kotlin-Swift-Trace-Link-Recovery/blob/master/Entwicklung%20eines%20Plugins%20in%20IntelliJIDEA%20zum%20Auffinden%20von%20Quellcode-Entsprechungen.pdf`, 2017. Studie.

[JMK13] Joorabchi, M. E.; Mesbah, A.; Kruchten, P.: Real Challenges in Mobile App Development. In: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. S. 15–24, Oct 2013.

[Mi03] Mishne, Gilad: Source Code Retrieval using Conceptual Graphs. Masterarbeit, University of Amsterdam, 2003.

[MRS08] Manning, Christopher D.; Raghavan, Prabhakar; Schütze, Heinrich: Introduction to Information Retrieval. Cambridge University Press, 2008.

[MS12] Mayer, P.; Schroeder, A.: Cross-Language Code Analysis and Refactoring. In: SCAM2012. IEEE, S. 94–103, Sept 2012.

[Ol10] Oliveto, R.; Gethers, M.; Poshyvanyk, D.; De Lucia, A.: On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery. In: ICPC2019. IEEE, S. 68–71, June 2010.

[RCPO14] Roy Choudhary, Shauvik; Prasad, Mukul R.; Orso, Alessandro: Cross-platform Feature Matching for Web Applications. In: ISSTA2014. ACM, S. 82–92, 2014.

[RW02] Rajlich, Václav; Wilde, Norman: The Role of Concepts in Program Comprehension. In: IWPC '02. IEEE, S. 271–278, 2002.

[RZ09] Robertson, Stephen; Zaragoza, Hugo: The Probabilistic Relevance Framework: BM25 and Beyond. Found. Trends Inf. Retr., 3(4):333–389, April 2009.

[Si16] Singh, Alka; Henley, Austin Z.; Flemming, Scott D.; Luong, Maria V.: An Empirical Evaluation of Models of Programmer Navigation. In: ICSME2016. IEEE, S. 9–19, 2016.

[SKL06] Strein, D.; Kratz, H.; Lowe, W.: Cross-Language Program Analysis and Refactoring. In: SCAM'06. IEEE, S. 207–216, Sept 2006.

[SR15] Stehle, Tilmann; Riebisch, Matthias: Establishing Common Architectures for Porting Mobile Applications to new Platforms. In: WSRE2015. GI-FG SRE, S. 26–27, 2015.

[Th13] Thalheim, Bernhard: The Conception of the Model. In: BIS2013. S. 113–124, 2013.

[Th17a] The Apache Software Foundation: , Apache Lucene Core, March 2017.

[Th17b] The Apache Software Foundation: , Lucene.net, 2017.

[Ti01] Tichelaar, Sander: Modeling object-oriented software for reverse engineering and refactoring. Dissertation, University of Berne, 2001.

[Ud13] Udagawa, Yoshihisa: Source Code Retrieval Using Sequence Based Similarity. Journ. IJDKP, 3(4):57–74, July 2013.

[Un17] Unity Technologies: , Unity - Game Engine. https://unity3d.com/de/, 2017.

[Xa17] Xamarin Inc.: , Xamarin. https://www.xamarin.com/, 2017.

[ZLL13] Zhou, J.; Lu, Y.; Lundqvist, K.: A Context-based Information Retrieval Technique for Recovering Use-Case-to-Source-Code Trace Links in Embedded Software Systems. In: Euromicro Conference on Software Engineering and Advanced Applications. S. 252–259, Sept 2013.