Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Studie

Entwicklung eines Plugins in IntelliJ IDEA zum Auffinden von Quellcode-Entsprechungen

Gerrit Greiert
5greiert@informatik.uni-hamburg.de
Studiengang Wirtschaftsinformatik
Matr.-Nr.: 6829930

Wintersemester 2016/17 & Sommersemester 2017

Betreuer: Tilmann Stehle

Inhaltsverzeichnis

1. Einleitung

Der Markt von Anwendungen für mobile Endgeräte lässt sich in mehrere Plattformen, also Technologien und Betriebssysteme, unterteilen. Eine Möglichkeit, Mobilanwendungen auf mehreren Plattformen anzubieten, ist die mehrfache native Umsetzung der Applikationslogik mit den Mitteln und Werkzeugen des jeweiligen Plattformherstellers. Android-Applikationen werden in Java geschrieben, wohingegen moderne iOS-Apps auf Swift basieren. Da beide Umsetzungen jedoch dieselbe Applikation realisieren, sind Quellcode-Entsprechungen zwischen Funktionalitäten auffindbar. Das grundlegende Thema dieser Studie soll die Entwicklung eines Plugins für die Entwicklungsumgebung IntelliJ IDEA zum Auffinden von solchen Quellcode-Entsprechungen in Implementierungen von mobilen Anwendungen für verschiedene Plattformen sein.

Um Gemeinsamkeiten zwischen mehreren Implementierungen zu finden soll der Quellcode analysiert werden um Tracelinks zu erzeugen und mit einem Ähnlichkeitswert zu bewerten (Traceability Link Retrieval). Diese Links können dazu dienen, Gemeinsamkeiten und Unterschiede zwischen beiden Versionen aufzudecken. Zudem sind sie vor allem bei der Wartung hilfreich, um Änderungen in beiden Versionen gleichartig durchführen zu können. Als Entwickler kann man über die erzeugten Tracelinks direkt zu verlinkten Codeelementen in einer anderen Implementierung springen, die potentiell gleiche Funktionalitäten umsetzen. Der hier aufgezeigte Ansatz soll eine bestehende Tracelink-Suche in Form eines Plugins zugänglich und nutzbar machen um von den Vorteilen von Tracelinks zwischen einer Ausgangsversion und einer Portierung profitieren zu können.

1.1. Zielsetzung

Im Rahmen der Studie soll ein Android Studio Plugin entstehen, welches dem Nutzer anhand von Klassennamen oder Methodennamen, sowie darin auftretender Bezeichner, als auch anhand von eingegebenen Suchbegriffen zu einer Java-Implementation eine Liste möglicher zugehöriger Swift-implementierungen bietet. Die vorgesehenen Use Cases sind:

- Finden von Tracelinks über eingegebene Suchbegriffe
- Finden von Tracelinks zu Klassen, Methoden und Attributen im Quellcode-Editor

Ab Seite 38 sind detaillierte Use Case-Beschreibungen zu finden. Für beide Fälle wird für die in der IDE geöffnete Java-Implementation Entsprechungen in einer verlinkten Swift-Implementation gesucht. Die Suchbegriffe, die der Nutzer beim ersten Use Case eingibt, werden mit einem Stemming-Verfahren auf ihre Wortstämme reduziert. Als Ergebnis präsentieren beide Anwendungsfälle eine nach Wahrscheinlichkeit sortierte Liste von Tracelinks, welche zudem auf Wunsch in Xcode, der offiziellen IDE für Swift-Entwicklung, geöffnet werden können.

Nach einer kurzen Einführung in die fachlichen Grundlagen beschäftigt sich diese Studie mit dem softwaretechnischen Entwurf des Plugins und der Implementierung. Anschließend soll die Usability des Plugins evaluiert werden. Darunter fallen Aspekte wie die Integration des Werkzeugs in die Entwicklungsumgebung und die Unterstützung des Arbeitsflusses. Auch soll die Nützlichkeit des Plugins in Hinsicht auf die Anwendbarkeit auf Aufgaben der parallelen Weiterentwicklung und Wartung betrachtet werden.

2. Verwandte Arbeiten

Das Thema Traceability zwischen verschiedenen Artefakten in der Softwareentwicklung und die nachträgliche Ermittlung von Links durch Information Retrieval ist in der Forschung nicht neu. In den in der Literatur zu findenden Arbeiten geht es oftmals um die Herstellung von Traceability zwischen Codeartefakten und anderen Artefakten wie Anforderungen, Dokumentationstexten, Testfällen oder Modellen [LFOT05, ZWRH06, HDS+07]. Das Ermitteln solcher Beziehungen kann bei der Weiterentwicklung helfen und das Verständnis des Codes erleichtern. So können Beziehungen zwischen Systemteilen besser erkannt werden indem auf zusätzliche Informationsdokumente zurückgegriffen werden kann. Bei Fortentwicklung von Software können diese Informationen bei der Durchführung einer Impact-Analyse hilfreich sein.

Obwohl Traceability zwischen Artefakten klare Vorteile bietet, ist die Toolunterstützung oft nicht hinreichend vorhanden. Zudem erzeugt die manuelle Pflege von Traceability Links einen großen Mehraufwand. Daher ist die automatische Erzeugung von Traceability Links besonders interessant, da sie den manuellen Aufwand im Idealfall wegfallen lässt. Ein Beispiel für ein Tool, welches automatische Traceability Link-Erzeugung über Information Retrieval realisiert, ist „Advanced Artefact Management System“ (ADAMS). Es verwendet die IR-Technik Latent Semantic Indexing (LSI), um Textdokumente auszuwerten und Ähnlichkeiten festzustellen. Daraus werden besonders wahrscheinliche Verknüpfungen zwischen Artefakten als Traceability Links ausgewählt [LFOT05].

Weitere Forschung zu diesem Thema deckt Probleme auf, die bei diesem Verfahren auftreten können. Zum einen ist der Information Retrieval-Ansatz auf eine konsistente Verwendung von Fachbegriffen aus der jeweiligen Domäne angewiesen. Inkonsistente Verwendung von Bezeichnern erhöht den Aufwand für die Ähnlichkeitsanalyse deutlich. Zum anderen muss ein Weg gefunden werden, möglichst viele korrekte Traceability Links zu erhalten und gleichzeitig die Anzahl von False Positives zu minimieren. Wenn der Anspruch verfolgt wird alle vorhandenen Links zu finden, steigt gleichzeitig auch der Aufwand die entstehenden False Positives zu erkennen und zu entfernen. Die optimale Schwelle an der die Ergebnisliste abgeschnitten wird muss jedoch von Fall zu Fall in einem inkrementellen Prozess ermittelt werden [LFOT06].

Weitere Untersuchungen betreffen die verwendete Information Retrieval-Methode. In der Literatur werden neben LSI auch die Jensen-Shannon-Methode, das Vector Space Model und Latent Dirichlet Allocation (LDA) genannt. In einem Vergleich konnte festgestellt werden, dass bis auf LDA alle Methoden äquivalente Ergebnisse bei der Traceability Link-Ermittlung erzielen konnten. LDA produzierte dagegen schlechtere Ergebnisse [OGPL10].

Diese Studie soll sich jedoch nicht mit dem Auffinden von Traceability zwischen Code und Dokumentation sondern zwischen verschiedenen Codeartefakten befassen. Dazu sind in der Literatur unter anderem Source Code Retrieval-Verfahren zu finden. Dabei wird Information Retrieval verwendet, um konzeptuell übereinstimmende Codeelemente aufzufinden. Diese Informationen können zum Beispiel bei der Aufdeckung von kopiertem Code oder mehrmals implementierter

Programmlogik helfen (Code Clone Detection) [MR04]. Auch wurde im Vorfeld dieser Studie ein Projekt durchgeführt, welches die Portierung von Android-Applikationen auf andere Mobilplattformen und die gleichzeitige Generierung von Links zwischen den Plattformen betrachtet. Dazu wurde die Erstellung der Links in die Code-Konverter Sharpen und J2Swift integriert. Im Rahmen des Projekts wurde ein in Java implementiertes Traceability-Modell entwickelt, das auch hier wieder aufgegriffen werden soll. [ABF+17] Im Gegensatz zum Projekt soll die Traceability hier jedoch zwischen nativen Applikationen nachträglich hergestellt werden. Dazu wird auf einen von Tilmann Stehle implementierten Traceability Recovery-Service zurückgegriffen, der die Erstellung der Links übernimmt.

3. Fachliche Grundlagen

Um das Verständnis der Arbeit zu erleichtern, soll dieses Kapitel in die fachlichen Grundlagen einführen. Dazu gehören die Vorstellung des oben erwähnten Traceability-Modells, ein Überblick über die Pluginentwicklung in IntelliJ und die Vorstellung von Lucene und Stemming-Verfahren.

3.1. Traceability

Cleland-Huang et al. beschreiben Traceability als das Potenzial Traces zu erzeugen und zu nutzen. Dabei geht es primär um die Rückverfolgbarkeit von zusammengehörigen Artefakten. Ein Beispiel dafür wäre die Identifikation von zu Anforderungen gehörenden Stakeholderwünschen, Designartefakten oder Testfällen. Auch die Zuordnung von Dokumentationstexten zu Programmfunktionen gehört dazu. Ein Trace besteht daher aus einem Quellartefakt, einem Zielartefakt und einem Trace Link, welcher die beiden Artefakte verbindet. Trace Artefakte können beispielsweise eine oder mehrere Anforderungen, UML-Diagramme, Quellcodedateien oder auch Personen sein. Die Verknüpfung von Artefakten über einen Trace Link kann verschiedene Bedeutungen haben. So kann ein Quellartefakt ein Zielartefakt u.a. implementieren, testen, verfeinern oder ersetzen. Traceability wird verwendet um zugeordnete Artefakte zu finden. Dieser Vorgang wird als Tracing bezeichnet. Weitere der Traceability zugeordnete Aktivitäten sind das Identifizieren von relevanten Artefakten und Verknüpfungen, deren Speicherung und eine auf den Verwendungszweck zugeschnittene Repräsentation [GCHH+12]. Im Kontext dieser Studie soll Traceability zwischen konzeptuell gleichen Codeelementen in verschiedenen Implementationen einer Mobilanwendung hergestellt werden. Über Tracing der gefundenen Trace Links können bei einer Parallelentwicklung betroffene Stellen identifiziert werden.

3.2. Traceability-Modell

Um Traceability greifbar und konkret anwendbar zu machen, muss es ein Modell geben, welches Typen von Codeelementen und Traceability Links dazwischen abbilden kann. Ein solches Modell wurde wie bereits angesprochen, für ein Projekt entwickelt, das dieser Studie vorangegangen ist [ABF+17]. Dabei ging es darum Traceability Links zwischen einer Java-Implementation und einer automatisch konvertierten C#- oder Swift-Implementation festzuhalten. Ein ähnlicher Fall ist auch für diese Studie relevant, mit dem Unterschied, dass die Implementationen für beide Plattformen manuell entwickelt und nicht konvertiert wurden. Da das Modell jedoch für eine möglichst vielseitige Anwendung konzipiert wurde und zudem Programmiersprachen-unabhängig ist, kann es hier problemlos wieder eingesetzt werden.

Die notwendige Datenstruktur wurde in Java implementiert und stellt Klassen zur Verfügung um Traceability Links zwischen Code-Artefakten Plattform-unabhängig festzuhalten. Bei Code-Elementen kann es sich um unterschiedliche Sprachkonstrukte handeln, welche unter Umständen weitere Elemente enthalten. Beispielsweise enthält eine Klasse u.a. Methoden und Attribute. Methoden enthalten wiederum Parameter und Referenzen auf Attribute mit denen sie arbeiten. Um diese Schachtelung abzubilden wurde eine rekursive Datenstruktur gewählt, in der jedes Element durch ein eigenes Traceability-Modell repräsentiert wird, welches wiederum Submodelle enthalten kann. Für die im Projekt verfolgte Konvertierung von Quellcode wurde ein Traceability-Modell für das zu übersetzende Element, in der Regel eine Klasse, vom Konverter erzeugt. Dieses enthält der Klassenstruktur entsprechend Submodelle für auftretende Subelemente. Die Verknüpfung zu den übersetzten Elementen wird durch Traceability Links realisiert, welche Traceability Pointer enthalten. Ein Traceability Pointer zeigt auf ein bestimmtes Element im Modell. Durch die Zusammenführung von einem Pointer auf Java-Seite und einem auf Swift-Seite in einem Link kann eine Verknüpfung erzeugt werden. Das Schema auf Seite 37 des Anhangs verdeutlicht dies. Verschiedene Code-Elemente werden durch verschiedene Arten von Pointern unterschieden, welche jedoch alle von der Oberklasse `TraceabilityPointer` erben. Die jeweiligen Unterklassen enthalten zusätzliche Attribute, die helfen das Element zu beschreiben. Abbildung 3.1 zeigt eine Übersicht über die gesamte Klassenstruktur und die unterschiedlichen Pointer. Das bisherige Modell wurde für Java, C# und Swift spezifiziert, jedoch könnten noch nicht abgebildete Sprachkonstrukte aus anderen Sprachen über zusätzliche Pointer abgebildet werden.

Die Implementierung des Traceability-Modells enthält zudem Import- und Export-Klassen um die Datenstruktur in ein XML-Format zu externalisieren und wieder einlesen zu können [ABF+17].

Für die Anwendung in dieser Studie sind vor allem die Traceability Pointer und deren Zusammensetzung zu Traceability Links relevant, da das Information Retrieval-Verfahren genau dies leisten soll. Im Plugin müssen die erzeugten Links anschließend korrekt auf Suchanfragen hin auffindbar gemacht und visualisiert werden.

3.3. Pluginentwicklung für die IntelliJ-Plattform

Die Aufgabe dieser Studie ist die Entwicklung eines Plugins für die Entwicklungsumgebung IntelliJ IDEA. Die IntelliJ-Plattform bietet mit der IntelliJ-Plugin-API eine Schnittstelle an, um die IDE um zusätzliche Funktionen zu erweitern. Dabei können dieselben Mechnismen verwendet werden, mit denen auch die in der IDE mitgelieferten Funktionen realisiert wurden. Um ein besseres Verständnis der Integration eines Plugins über den angebotenen Plugin-Mechanismus zu erhalten, soll nun kurz auf die grundlegende Struktur der IntelliJ-Plugin-API eingegangen werden. Grundsätzlich können Plugins in alle Produkte der IntelliJ-Reihe, wie beispielsweise Android Studio integriert werden, allerdings kann es in Spezialfällen zu Kompatibilitätsproblemen führen.

Plugin-Projekte werden durch eine Konfigurationsdatei (plugin.xml) beschrieben, die Referenzen auf verwendete Elemente der Plugin-API enthält. Über diese Referenzen wird festgelegt, welche Klassen des Plugins welche API-Elemente umsetzen, damit sie vom Framework geladen werden können. Seite 37 des Anhangs zeigt ausschnittsweise Typen von möglichen Elementen. Die Datei plugin.xml wird für das Laden des Plugins und zum Bereitstellen der Funktionen benötigt. Das

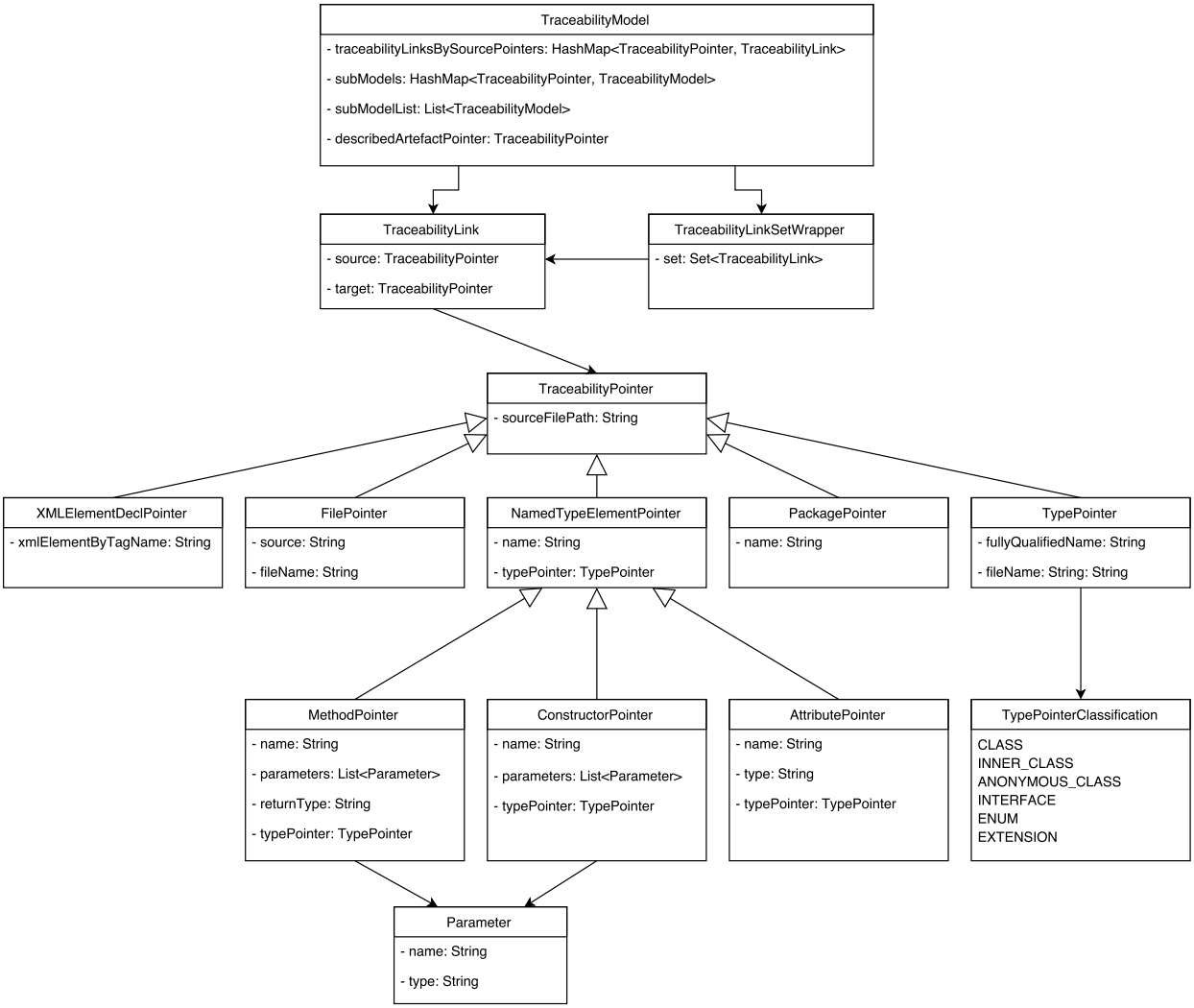


Abbildung 3.1.: Vollständiges Traceability-Modell

IntelliJ-Rahmenwerk stellt eine Vielzahl an Komponenten zur Verfügung, die zur Umsetzung eines Plugins implementiert werden können. Die wichtigsten werden im Folgenden vorgestellt:

Actions sind Klassen, die die Funktionalität von Menüeinträgen oder Buttons in Toolbars repräsentieren. Um eine Klasse als Action zu definieren, muss sie von der Klasse `AnAction` erben und die Methode `actionPerformed(AnActionEvent event)` implementieren. Diese enthält den Code, der auszuführen ist, wenn die Action angeklickt wird. In der plugin.xml wird festgelegt, an welcher Stelle die Action in das Menüsystem integriert wird. Über die Methode `update(AnActionEvent e)`, die in regelmäßigen Intervallen aufgerufen wird, kann eine Action dynamisch angepasst werden. Zum Beispiel kann sie ausgeblendet oder deaktiviert werden, wenn sie in der aktuellen Situation nicht anwendbar ist.

Components stellen Elemente dar, die Funktionalität für ein bestimmtes Level einer Projektstruktur zur Verfügung stellen. Es werden drei Arten von Komponenten unterschieden: Application-, Project- und Module-Level-Components. Applikationskomponenten werden beim Start der IDE initialisiert, Projektkomponenten beim Öffnen eines Projektes und Modulkomponenten für jedes Modul in einem Projekt. Auf initialisierte Komponenten kann über Instanzen von `Application`, `Project` oder `Module` zugegriffen werden. Im Regelfall besitzen Komponenten neben einer Implementationsklasse auch ein über die plugin.xml zugeordnetes Interface, welches bei Komponenten-Aufrufen angesprochen wird.

Eine ähnliche Funktionalität bieten *Services*, welche vom Plugin bereitgestellte Dienste darstellen, die im Gegensatz zu Komponenten auch von Plugin-fremden Teilen der Entwicklungsumgebung benutzt werden können. Services können über die Methode `getService()` des `ServiceManagers` aufgerufen werden, woraufhin eine Instanz des Service-Objektes erzeugt wird. Diese wird bei jedem weiteren Aufruf von `getService()` übergeben, sodass stets nur eine Instanz der Klasse zur Laufzeit plugin-übergreifend genutzt wird. Die von einem Plugin zur Verfügung gestellten Services werden in der plugin.xml aufgelistet. Dort wird auch auf die Service-Schnittstelle, also ein Interface verwiesen.

ToolWindows sind grafische Fenster oder Views, die in die Benutzeroberfläche eingefügt werden, um die gewünschte UI für ein Plugin darzustellen. Dabei können pro Plugin mehrere ToolWindows erzeugt bzw. hinzugefügt werden. Sie können vom zentralen Fenstermanagement der Applikation verwaltet werden, wodurch sie ohne weiteren Implementationsaufwand individuell skaliert, im Programmfenster angepinnt und minimiert oder maximiert werden können.

Configurables sind Einstellungsfenster, die in die zentralen Einstellungen der Entwicklungsumgebung eingefügt werden und der Konfiguration des Plugins dienen [Int15].

3.4. Lucene

Die genutzte IR-Methode basiert auf Apache Lucene, einem Rahmenwerk zur Implementation von Indizes und Volltext-Suchen. Auch wenn der IR-Prozess nicht direkt Bestandteil dieser Studie ist, spielt Lucene für die Implementation des Plugins dennoch eine Rolle. Bei Apache Lucene handelt es sich um eine Volltext-Suchmaschine in Form einer Java-Bibliothek. Lucene akzeptiert reinen Text-Input und bietet keine eigenen Parser für verschiedene Dateitypen. Vor der Erstellung eines durchsuchbaren Indexes für das eingelesene Dokument durchläuft der Input einen Tokenizer. Dieser unterteilt den Text in kleine Elemente, sogenannte Tokens. Die Beschaffenheit der Tokens

hat starke Auswirkung auf die Auffindbarkeit bei der Suche, weshalb die Verarbeitung zu Tokens im Tokenizer konfigurierbar und erweiterbar ist. Zu den mitgelieferten Verarbeitungsschritten gehören Natural Language Processing-Techniken wie Stemming, Stoppwortfilterung/-entfernung, Textnormalisierung und Synonym-Mapping. Über den Tokenizer hinaus wird der Input mithilfe eines Analyzers weiter verarbeitet [Fou17].

Anschließend kann der erstellte Index für Suchanfragen verwendet werden. Es werden verschiedene, variable Arten von Suchanfragen unterstützt, z.B. Phrasensuche, Wildcard-Suchen oder Ähnlichkeitsabfragen. Suchergebnisse können über Ranking-Mechanismen geordnet werden, um die besten bzw. passendsten Ergebnisse als erste zu zeigen. Hier kann unter anderem die Vector Space Model-Methode zur Ähnlichkeitsanalyse zwischen Dokumenten im Index und der Suchanfrage verwendet werden. Lucene greift dabei auf die Rankingfunktionen Okapi BM25 und das Tf-idf-Maß (inverse Dokumenthäufigkeit) zurück [Fou16]. Zur Ermittlung von ähnlichen Traceability Pointern kann die Ähnlichkeit der dazu gehörigen Dokumente betrachtet werden. Auch der in der Einleitung erwähnte Anwendungsfall einer Suchanfrage mit Keywords kann über diesen Mechanismus realisiert werden.

3.5. Stemming-Verfahren

Im Bereich des Information Retrieval bezeichnet man das Zurückführen von Worten auf ihren Wortstamm als Stemming oder Stammformreduktion. Es wird davon ausgegangen, dass Worte mit demselben Wortstamm unabhängig von der genauen Verwendung auch dieselbe Bedeutung haben. Daher können z.B. durch Konjugation oder Deklination entstandene Suffixe entfernt werden ohne die Aussage des Wortes bezüglich des Information Retrievals zu ändern. Ein Beispiel sieht wie folgt aus:

$$\text{Connect, Connected, Connecting, Connection, Connections} \Longrightarrow \text{Connect}$$

Durch das Stemming von Tokens soll das Auffinden von übereinstimmenden Bedeutungen vereinfacht werden. Sowohl die Verarbeitung von Suchanfragen mit Suchbegriffen als auch die Performance von IR-Techniken kann so verbessert werden. Zudem sinkt die Datenmenge und -Komplexität, da weniger einzelne Token betrachtet werden müssen. Zu beachten ist, dass die gestemmten Tokens keine linguistisch korrekten Worte sein müssen, da sie lediglich in den IR-Algorithmen verwendet werden.

Es gibt eine Vielzahl von Ansätzen und Algorithmen zum Durchführen von Stemming. Einer der verbreitetsten ist der von Martin Porter erarbeitete Porter-Stemmer, welcher in vielen Natural Language Processing-Frameworks als Standardstemmer integriert ist. Der Porter-Algorithmus besteht aus mehreren Sets von Verkürzungsregeln welche schrittweise auf jedes Inputtoken angewendet werden. Neben den Verkürzungsschritten beachtet der Stemmer aber auch die Anzahl von speziell definierten Sequenzen aus Konsonanten und Vokalen. [Por80]. Aufgrund der Relevanz von Stemmern für IR und der hohen Fehlerrate bei Implementierungen von Stemming-Algorithmen für andere Sprachen und Plattformen hat Porter zudem die Sprache Snowball entwickelt, mit der Stemmer formal spezifiziert werden können [Por01].

Eine Studie von 2011 führt weitere Stemming-Algorithmen auf. Sie unterscheidet zwischen Affix-entfernenden und auf statistischen Methoden basierenden Stemmern. Erstere arbeiten wie der

Porter-Stemmer mit einer Menge an Verkürzungsregeln. Letztere ziehen statistische Analysen mit hinzu. Beispielsweise arbeitet KSTEM mit einem Lexikon, welches korrekte Wortstämme enthält. Verkürzungen werden nur angewendet, wenn dadurch ein im Lexikon verzeichneter Wortstamm entsteht [J+11].

4. Anforderungen

Bevor ein Überblick über den Entwurf des IntelliJ-Plugins gegeben werden kann, sollen die Anforderungen erläutert werden, die an die Implementation des Plugins gestellt werden.

Der gewünschte Funktionsumfang und die angestrebte Qualität der Software wird in der Regel in Form von Anforderungen festgehalten. Durch das Aufstellen einer Anforderungsliste kann ermittelt werden, ob alle Anforderungen nötig sind oder es Konflikte zwischen ihnen gibt. Zudem sollten Anforderungen überprüfbar formuliert sein, sodass eindeutig ermittelt werden kann ob sie durch eine Implementation erfüllt sind oder nicht [Som11, S. 110]. Die Überprüfung kann beispielsweise durch Unit-Tests durchgeführt werden.

Eine gute Möglichkeit Anforderungen festzuhalten ist die Erstellung von gewünschten Use Cases, wie sie in Kapitel 1 erwähnt wurden. Neben den Funktionen an sich muss auch betrachtet werden, welche Voraussetzungen eventuell für deren Ausführung erfüllt sein müssen. Im Rahmen dieser Studie wurde das Thema zudem Jörg Pechau von ICNH vorgestellt, welcher einige Ideen beisteuerte.

Die funktionalen Anforderungen können grundlegend über die Use Cases definiert werden. Dazu gehören das Aufrufen eines Suchfensters zur Suche von Tracelinks über Suchbegriffe und die Suche über im Editor markierte Codeelemente. Diese Funktionen entspringen direkt der in Abschnitt 1.1 beschriebenen Aufgabenstellung der Studie. Neben dem Aufrufen der Funktionen durch den Nutzer enthalten die funktionalen Anforderungen außerdem die Suchanfrage an den angebundenen Recovery Service (Information Retrieval) und eine Darstellung der Ergebnisse für den Nutzer. Somit ergeben sich Voraussetzungen für diese Anforderungen. Zum einen muss der Recovery Service eingebunden sein. Bevor der Service benutzt werden kann, muss er alle Quellcodedateien einlesen und verarbeiten, die zu den zu betrachtenden Implementierungen gehören. Über das in der IDE geöffnete Java-Projekt können alle relevanten Java-Dateien erfasst werden. Zudem muss eine Swift-Implementierung verlinkt werden. Dazu wird der Pfad zum Ordner des Swift-Projektes angegeben. Auch diese Verlinkung von Projektdateien muss vom Nutzer vorgenommen werden können, wodurch eine weitere Anforderung für einen Einstellungsdialog entsteht. Zum anderen bekommen beide Anwendungsfälle eine Ergebnisliste, welche auf dieselbe Weise visualisiert werden kann. Für die beiden in Kapitel 1 genannten Use Cases sind in Abbildung 4.1 Use Case-Beschreibungen dargestellt. Eine Darstellung aller primären Anwendungsfälle sowohl in einem Use Case-Diagramm als auch in Use Case-Beschreibungen mit Aktivitätsdiagramm kann auf Seite 38 eingesehen werden.

Neben den funktionalen Anforderungen sind auch noch die Qualitätsmerkmale oder nicht-funktionale Anforderungen zu betrachten. Darunter fallen beispielsweise Aspekte wie die Performance oder die Usability der Software. Auch ist zu beachten, dass die Erfüllung von nicht-funktionalen Anforderungen unter Umständen große Auswirkungen auf die Architektur des Systems haben kann. Zum Beispiel wäre es bei der Ausführung längerer Aufgaben, wie des Einlesens der Dokumente in den Recovery Service, erstrebenswert, den Prozess nebenläufig auszuführen. Aus Sicht des Nutzers hat dies den Vorteil, dass die Benutzeroberfläche der Entwicklungsumgebung nicht durch

Use Case:	Tracelinks zu Suchbegriffen finden
Ziel:	Nutzer bekommt zu den Suchbegriffen passende Tracelinks gezeigt
Kategorie:	primär
Vorbedingung:	Der Recovery Service hat sowohl die Java- als auch Swift-Implementation analysiert und einen Index erstellt
Nachbedingung Erfolg:	Ermittelte Tracelinks wurden dem Nutzer gezeigt
Nachbedingung Fehlschlag:	Es wurde eine Fehlermeldung angezeigt, die dem Nutzer Hilfestellung zur Fehlerbehebung bietet
Akteure:	Nutzer
Auslösendes Ereignis:	Der Nutzer klickt auf den Menüpunkt „Search tracelinks“ oder drückt die korrespondierende Tastenkombination

Use Case:	Tracelinks zu Codeelementen finden
Ziel:	Nutzer bekommt zum ausgewählten Codeelement passende Tracelinks gezeigt
Kategorie:	primär
Vorbedingung:	Der Recovery Service hat sowohl die Java- als auch Swift-Implementation analysiert und einen Index erstellt
Nachbedingung Erfolg:	Ermittelte Tracelinks wurden dem Nutzer gezeigt
Nachbedingung Fehlschlag:	Es wurde eine Fehlermeldung angezeigt, die dem Nutzer Hilfestellung zur Fehlerbehebung bietet
Akteure:	Nutzer
Auslösendes Ereignis:	Der Nutzer klickt auf den Menüpunkt „Show possible tracelinks“ oder drückt die korrespondierende Tastenkombination

Abbildung 4.1.: Use Case-Beschreibungen

einen Pluginprozess blockiert wird. Dies trägt deutlich zum Arbeitsfluss bei und entspricht den Plugin-Konventionen. Dazu muss die Implementation den Nebenläufigkeits-Mechanismus von IntelliJ einbinden.

Weitere Anforderungen betreffen hauptsächlich die Usability. Wiederum soll das Plugin möglichst gut in den typischen Arbeitsfluss mit der IDE integriert werden. Dazu gehört der problemlose und einfache Aufruf der Funktionen über sinnvoll positionierte Menüeinträge und Tasten-Kombinationen. Natürlich kann es auch zu Problemen wie fehlenden Konfigurationen oder unerwarteten Fehlern kommen. In diesen Fällen soll der Nutzer informiert werden und soweit möglich einen Lösungsvorschlag für das Problem erhalten. Im Falle notwendiger aber fehlender Einstellungen kann der Nutzer direkt zum richtigen Configurable weitergeleitet werden.

Die nichtfunktionalen Anforderungen sind in Abbildung 4.2 im Anschluss an dieses Kapitel in Form eines Utility Trees dargestellt. Dabei werden sie von übergeordneten Software-Qualitätsmerkmalen abgeleitet. Auf die normalerweise übliche Bewertung der Szenarien bzw. Einflussfaktoren nach Einfluss auf die Architektur und Beitrag zum Geschäftswert wurde verzichtet, da eine derartige Analyse nur bei einer systematischen Architekturanalyse sinnvoll ist, die in diesem Fall nicht durchgeführt wird.

Die in diesem Kapitel aufgezählten Anforderungen sollen im Folgenden durch den softwaretechnischen Entwurf und die Implementierung adressiert werden.

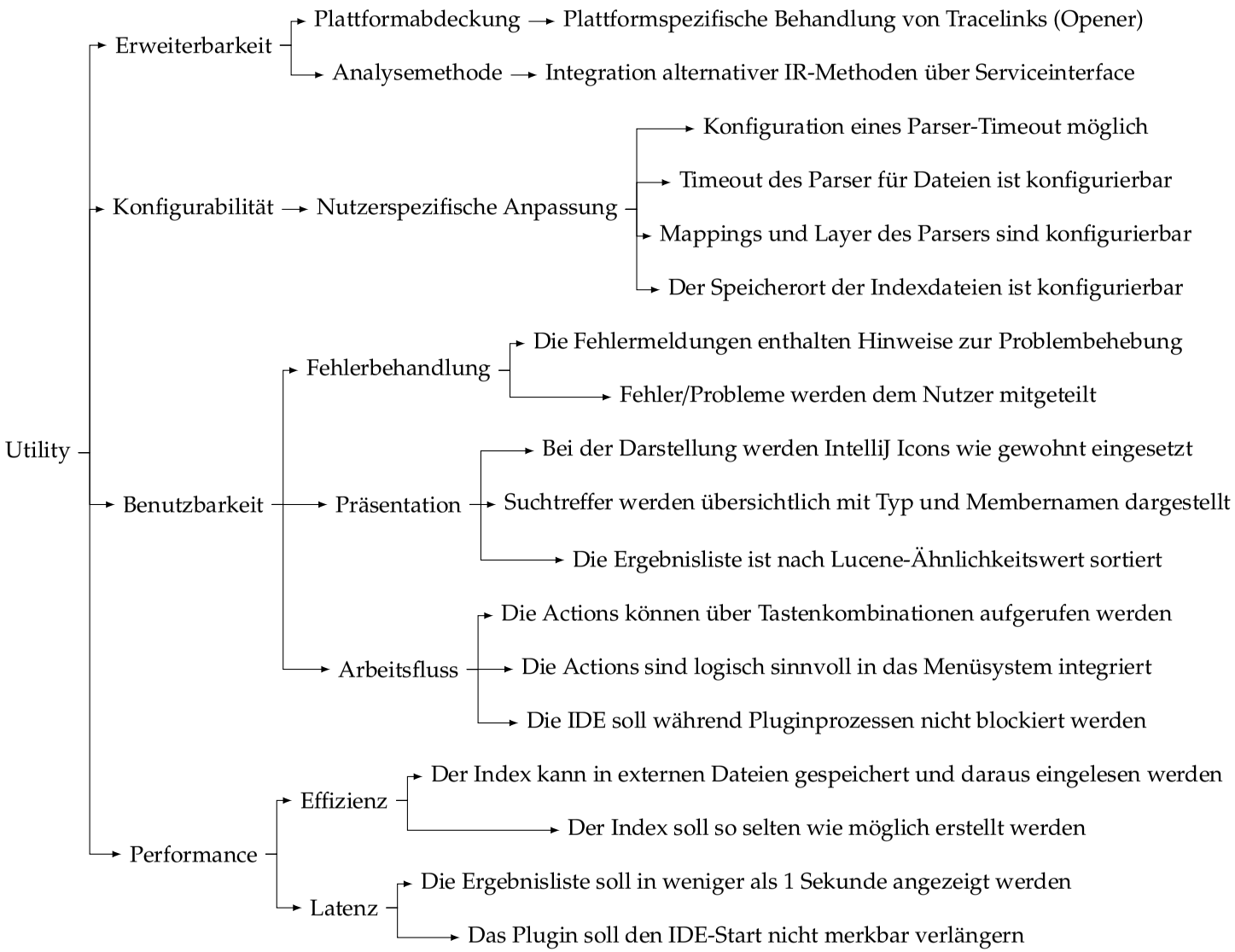


Abbildung 4.2.: Utility Tree für die nichtfunktionalen Anforderungen des Traceability Recovery Plugins

5. Entwurf

Aus den gestellten Anforderungen ergeben sich Möglichkeiten für die Struktur des Plugins. In Hinblick auf die zur Verfügung stehenden Elemente der Plugin-API muss nun ermittelt werden, wie sie umgesetzt werden können. Abbildung 5.1 zeigt eine Darstellung der hier erläuterten Architektur. Bei den beiden Anwendungsfällen, die die Suche nach korrespondierenden Traceability Links starten, ist es nötig dem Nutzer Menüpunkte bereitzustellen, über welche sie aufgerufen werden können. Daher ist es sinnvoll für beide eine Action zu implementieren, in deren `actionPerformed()`-Methode die weitere Aufruflogik implementiert wird. Die Präsentation der Ergebnisse in der UI kann nach den anderen notwendigen Schritten auch von den Actions ausgelöst werden. Aus dieser Überlegung ergeben sich die `EditorQueryAction` für die Suche anhand von Quellcodeelementen und die `SearchQueryAction` für die Suche über eingegebene Suchbegriffe. Beide benötigen Zugriff auf den Traceability Recovery Service in Form eines `ITraceabilityRecoveryService`.

Um diesen zu gewährleisten wird eine Project Component verwendet. Sowohl die Konfiguration der verlinkten Swift-Implementation zu einem Java-Projekt als auch alle weiteren Einstellungen und der Speicherort der zu einem Projekt erstellten Lucene-Index-Dateien sind projektspezifisch. Daher ist es sinnvoll, sie für jedes Projekt getrennt zu verwalten. Dafür bieten sich Projektkomponenten der IntelliJ-API besonders an, da über die plugin.xml registrierte Component-Klassen über Lebenszyklus-Methoden verfügen, die zu bestimmten Ereignissen aufgerufen werden. Klassen die das Interface `ProjectComponent` implementieren müssen die Methoden `projectOpened()` und `projectClosed()` implementieren. Dort kann das Einlesen der zu dem geöffneten Projekt gehörenden Konfiguration implementiert werden. Grundsätzlich wird für jedes geöffnete Projekt eine eigene Projektkomponente erzeugt, wodurch auch das parallele Bearbeiten mehrerer Projekte keine Probleme darstellt. Wenn die Einstellungen auf Applikationsebene vorgenommen werden würden, würde die Traceability Recovery für jedes Projekt auf dieselben Einstellungen zugreifen. Da verschiedene Java-Projekte jedoch in der Regel auch mit unterschiedlichen Swift-Projekten verlinkt sind, wäre dieser Ansatz nicht praktikabel, da man die Einstellungen bei jedem Öffnen eines anderen Projektes neu anpassen müsste.

Dementsprechend müssen natürlich auch die dem Nutzer zugänglichen Einstellungen für das Plugin und die Persistierung der Daten zwischen Beenden und Neustart der IDE projektbezogen gestaltet werden. Durch die Registrierung eines Configurable (`PluginConfigurable`) wird ein Einstellungsfenster in das zentrale Einstellungsfenster der Entwicklungsumgebung eingefügt. Die dort getätigten Einstellungen können im Anschluss von der `TraceabilityRecoveryComponent` verwendet werden.

Als weiteres Element wird das Interface `ITraceabilityRecoveryService` mit der Implementation `LuceneTraceabilityRecoveryService` als Service in der plugin.xml registriert. Theoretisch kann dessen Funktionalität so von überall aus der IDE-Instanz verwendet werden. Auch kann die Implementation ausgetauscht werden, falls Services verwendet werden sollen, die z.B. auf anderen IR-Techniken

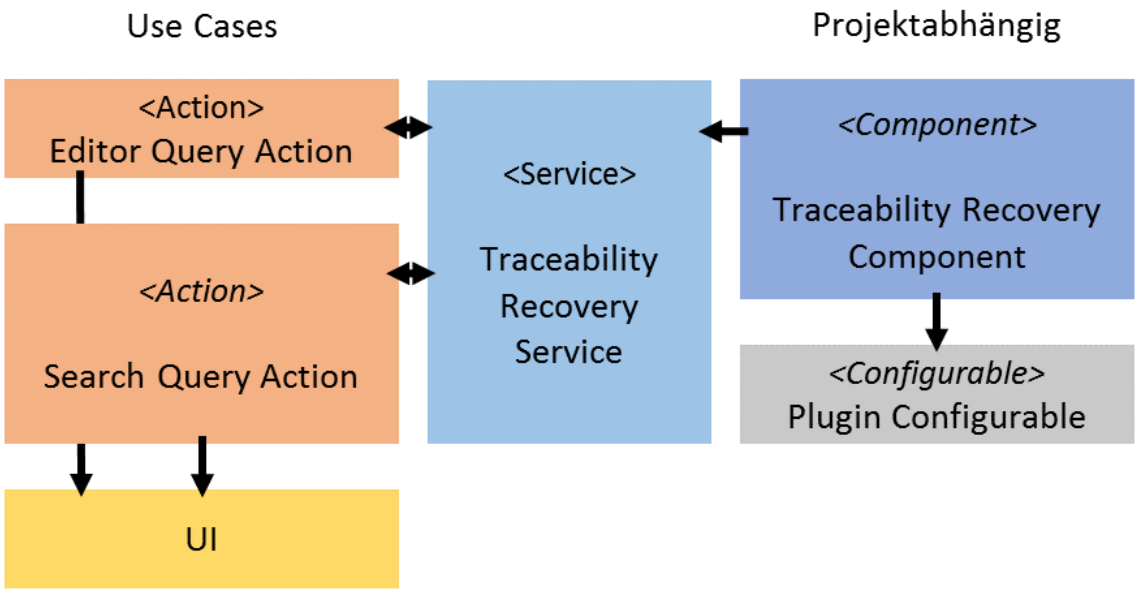


Abbildung 5.1.: Vereinfachte Darstellung der Plugin-Architektur

basieren. Vor seiner Nutzung muss der Service jedoch entweder alle relevanten Java- und Swift-Dokumente einlesen oder, im Falle des Lucene-Service, einen vorhandenen Index einlesen. Auch dies wird in den Lebenszyklus-Methoden der `TraceabilityRecoveryComponent` erledigt. Da das Einlesen von Dokumenten aufgrund des aufwändigen Parsings einige Zeit in Anspruch nehmen kann wird diese Arbeit über die Hintergrundaufgabe `RecoveryServiceRefreshTask` realisiert, welche den Nutzer bei Fertigstellung benachrichtigt. Die Klasse erbt von `Task.Backgroundable` und ist somit mit dem IntellJ-eigenen Nebenläufigkeitsmechanismus kompatibel.

Die UI für die Präsentation der Ergebnisse besteht aus einem Popup, welches eine Liste mit gefundenen Links darstellt. Sie sollen nach dem Ähnlichkeitswert sortiert sein. Zudem wird durch ein Icon dargestellt, um welche Art von Traceability Pointer es sich handelt. Wie im in Unterabschnitt C.3 dargestellten Anwendungsfall formuliert soll ein Klick auf einen Listeneintrag den korrespondierenden Swift-Pointer in der Swift-IDE Xcode öffnen. Dies wird durch Implementationen des Interfaces `TpointerOpener` realisiert. Dazu wird über die `TPointerOpenerFactory` ein `TPointerOpener` erzeugt und der Pointer über die in der Methode `openTraceabilityPointer(pointer)` implementierte Logik geöffnet. Die Factory kann je nach Plattform unterschiedliche Implementationen des `TPointerOpener` liefern. Unter Mac wird versucht, über AppleScript Xcode zu starten und die relevante Code-Stelle anzuzeigen. AppleScript ist eine in MacOS integrierte Skriptsprache, mit deren Hilfe Mac-Applikationen gestartet und gesteuert werden können [App16].

6. Implementation

In diesem Kapitel soll die Implementierung des Traceability Recovery-Plugins mithilfe der IntelliJ-Plugin-API genauer erläutert werden. Dabei wird besonders auf besondere Herausforderungen und Entscheidungen bei der Umsetzung der definierten Anforderungen eingegangen. Der Code der im Rahmen dieser Studie entstanden ist, ist auf Github verfügbar.[1]

6.1. Traceability Recovery Component

Das zentrale Element des Plugins ist die Traceability Recovery Component. Sie ist in der Klasse `TraceabilityRecoveryComponent` realisiert und implementiert das von der Plugin-API definierte Interface `ProjectComponent`. Die Schnittstelle sieht die Implementation der Lebenszyklusmethoden `projectOpened()` und `projectClosed()` vor, mit denen Aktionen bezogen auf das Öffnen und Schließen von Projekten in IntelliJ ausgeführt werden können. Bezogen auf die Arbeitsweise des Plugins ist es wichtig für bestimmte Projekte die jeweiligen verknüpften Swift-Implementationen hinterlegen zu können. Daher ist es wenig sinnvoll diese und andere Einstellungen auf Applikationsebene vorzunehmen. Auch muss der angedachte Traceability Recovery Service bezogen auf ein bestimmtes Projekt konfiguriert werden. Die Konfiguration umfasst entweder das Einlesen eines bestehenden Lucene-Index oder das Erstellen eines neuen für das geöffnete Projekt und die verknüpfte Swift-Implementation. Sowohl das Wiederherstellen der Einstellungen als auch das Konfigurieren des Service sollten daher in der `projectOpened()`-Methode der Komponente erledigt werden.

Die Komponente ist in der plugin.xml registriert. Dazu wird die Implementationsklasse `components .TraceabilityRecoveryComponent` angegeben. Es ist anzumerken, dass durch die Registrierung als Projektkomponente eine eigene Instanz der Komponente für jedes Projekt erstellt wird, welches in einer IntelliJ-Instanz geöffnet wird. Dies tritt auch ein, wenn für das geöffnete Projekt gar keine Traceability Recovery ausgeführt werden soll. Unter Umständen könnte solch ein Verhalten durch unnötige Erzeugung von Komponenteninstanzen die Performance beeinträchtigen. In diesem Fall würden aufwendigere Arbeiten der Komponente aufgrund von fehlenden Konfigurationen jedoch abbrechen. Daher wird die Erzeugung einer Instanz der Komponente auch bei Nichtverwendung in Kauf genommen.

Die Verwendung einer Applikationskomponente wurde bereits aus oben genannten Gründen ausgeschlossen. Alternativ zu einer Projektkomponente könnte jedoch auch eine Modulkomponente verwendet werden. In IntelliJ sind Module einzelne, eigenständig ausführ- und testbare Funktionalitäten eines Gesamtprojektes. Durch Verwendung einer Modulkomponente könnte unter Umständen eine feingranularere Ermittlung von Traceability auf Basis von Teilfunktionen erreicht werden. Allerdings soll im hier beschriebenen Entwurf eine solche Filterung über Suchmöglichkeiten

[1] https://github.com/TilStehle/Java-Kotlin-Swift-Trace-Link-Recovery

erreicht werden. Zudem wird eher ein holistisches Traceability-Modell zwischen verschiedenen Implementierungen angestrebt. Auch ist zu beachten, dass Module auf denen Modulkomponenten basieren ein Konstrukt der IntelliJ-IDEs sind. Für Projekte die nicht mit IntelliJ entwickelt wurden kann daher nicht garantiert werden, dass sie mit auf Modulen basierender Traceability Recovery kompatibel wären. Durch die Wahl einer Projektkomponente können solche Inkompatibilitäten vorab verhindert werden.

6.1.1. Konfiguration und Persistierung der Einstellungen

Die Konfiguration des Plugins und somit des Recovery Services ist besonders wichtig, da sie die Voraussetzung für die Erbringung der gewünschten Funktionalität ist. Beim ersten Start ist dem Plugin nur die in der IDE geöffnete Java-Implementation bekannt. Um Traceability herstellen zu können, muss der Nutzer den Pfad zur dazugehörigen Swift-Implementation angeben. Zudem soll der Nutzer auswählen können an welchem Ort die Lucene-Indexdateien abgelegt werden sollen. Als Standardpfad wird das Projektverzeichnis des geöffneten Projektes angeboten.

Die Plugineinstellungen werden als Configurable implementiert, welches in das zentrale Einstellungsfenster der Entwicklungsumgebung integriert wird. Die Klasse `PluginConfigurable` implementiert das API-Interface `Configurable`. Unter den zu implementierenden Methoden befinden sich `createComponent()`, `isModified()`, `apply()` und `reset()`. Sie sind notwendig um sich in das standardisierte Design mit Ok-, Apply- und Cancel-Knopf zu integrieren. `createComponent()` wird beim Öffnen des Einstellungsfensters aufgerufen und liefert die Benutzeroberfläche in Form einer `JComponent` von Swing. Die Klasse `PreferencesPanel` des Plugins erbt von JPanel und erzeugt die in Abbildung 6.1 dargestellte Benutzeroberfläche. Die Methode `isModified()` wird von der IDE aufgerufen und prüft in regelmäßigen Abständen ob der Nutzer Einstellungen verändert hat. Um dies zu ermitteln speichert das `PluginConfigurable` die initialen Werte für alle Einstellungen und vergleicht diese mit den aktuellen. Über `apply()` werden die aktuellen Einstellungen übernommen wohingegen sie bei `reset()` durch die initialen ersetzt werden. Die Benutzeroberfläche verwendet zum Auswählen von Pfaden das `TextFieldWithBrowseButton` und den `FileChooserDescriptor` aus der IntelliJ-API. Letzterer ist so konfiguriert, dass nur Ordner und keine Dateien gewählt werden können. Insgesamt wurde versucht möglichst nah an der Designsprache von IntelliJ zu bleiben und GUI-Komponenten aus der IntelliJ-API zu verwenden.

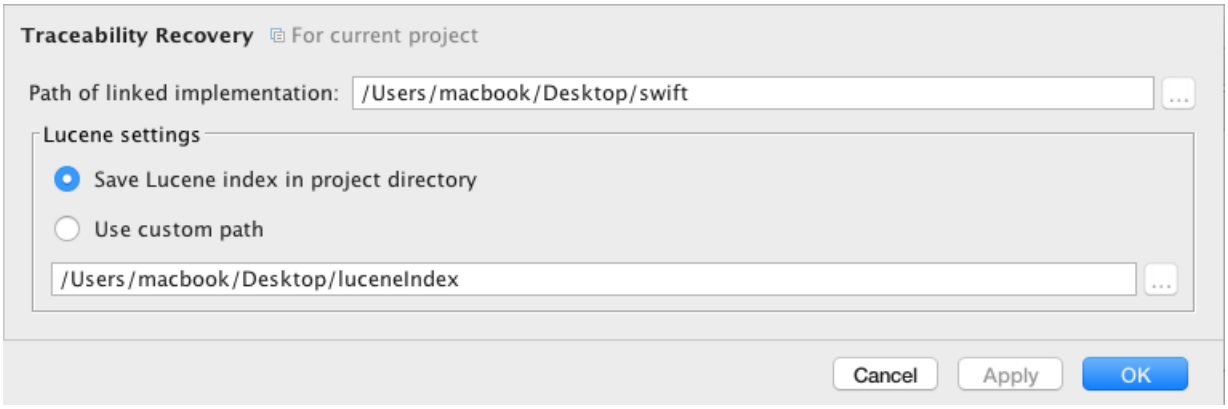


Abbildung 6.1.: Einstellungen des Traceability Recover Plugins

Natürlich müssen die vorgenommenen Einstellungen abgespeichert werden um auch nach einem Neustart der Entwicklungsumgebung noch vorhanden zu sein. Es wäre möglich eigene Logik zum Speichern der Einstellungen zu schreiben, allerdings ist die Verwendung einer der von IntelliJ bereitgestellten Mechanismen vorzuziehen. Die Plugin-API bietet zwei Möglichkeiten: Zum einen können Einstellungen als Key-Value-Paar an der `PropertiesComponent` gespeichert werden. Jedes Datenelement benötigt dafür eine einzigartige ID, über die es später wieder abgerufen werden kann. Zum Anderen besteht die Möglichkeit Plugin-Komponenten zu persistieren. Durch die Notwendigkeit einzigartiger IDs ist die erste Methode in diesem Fall eher ungeeignet. Da jedes Projekt eigene Einstellungen besitzt müssten auch für jedes Projekt neue IDs gewählt werden. Die Persistierung der bereits implementierten `TraceabilityRecoveryComponent` ist deutlich sinnvoller, weshalb diese Methode ausgewählt wurde. Bei der Persistierung von Komponenten wird eine Instanz einer Klasse in eine Datei serialisiert. Daher wird in der `TraceabilityRecoveryComponent` die Klasse `TraceabilityRecoveryComponentConfiguration` gespeichert. Diese enthält Attribute für die zu speichernden Einstellungen, welche mit `@Tag` annotiert sind. Damit die Komponente bei ihrer Erzeugung den Zustand dieser Konfiguration wiederherstellt muss sie zudem das Interface `PersistentStateComponent<>` implementieren. Diese Schnittstelle definiert die Methoden `loadState(configuration)`, welche die Einstellungsdatei lädt und die Konfigurationsinstanz füllt, und `getState()`, welche die Konfiguration der Komponente als Datei abspeichert. Die Komponente muss außerdem mit `@State()` annotiert werden. Innerhalb dieser Annotation wird der Name der Einstellungsdatei festgelegt.

```
<idea-plugin version="2">
    <extensions defaultExtensionNs="com.intellij">
     <!-- Add your extensions here -->
     <projectConfigurable
      instance="preferences.PluginConfigurable"
      id="traceabilityRecovery_settings"
      displayName="Traceability Recovery" />

     <projectService
      serviceInterface="de.unihamburg.swk.traceabilityrecovery.ITraceabilityRecoveryService"
      serviceImplementation="de.unihamburg.swk.traceabilityrecovery.lucene.LuceneTraceabilityRecoveryService" />
    </extensions>
    ...
</idea-plugin>
```

Quelltext 6.1: Extensions in der plugin.xml

Die Einstellungen die im Plugin Configurable getroffen werden, können so an der Komponente gespeichert werden. Allerdings muss das Configurable in der Lage sein auf die Komponente zugreifen zu können. Grundlegend können Projektkomponenten über den Aufruf `project.getComponent()` an der `Project`-Instanz abgerufen werden. Im Configurable ist das Projekt aber nur verfügbar, wenn es

in der plugin.xml als Projekt-Configurable registriert ist. Die Registrierung ist in Quelltext 6.1 zu sehen. Projekt-Configurables werden direkt dem geöffneten Projekt zugeordnet und bekommen dieses im Konstruktor übergeben.

6.1.2. Traceability Recovery Service

Die eigentliche Ermittlung von Traceability und der Zugriff darauf wird von einem Traceability Recovery Service übernommen. Die Entwicklung des Service war nicht Bestandteil dieser Studie, daher wurden die benötigten Klassen als externes Modul importiert. Das Modul `TraceabilityRecovery`, enthält das dazugehörige Interface `ITraceabilityRecoveryService` und eine Implementation desselben mit Lucene (`LuceneTraceabilityRecoveryService`). Quelltext 6.1 zeigt wie diese Klassen als Projekt-Service für das Plugin registriert werden. Grundlegend soll es möglich sein, an dieser Stelle die konkrete Implementation des Service auszutauschen, zum Beispiel auf Basis anderer IR-Techniken wie LSI. Dies kann durch ein gemeinsames Interface erreicht werden, das von der konkreten Implementierung abstrahiert. Klienten des Service nutzen stets das Interface als statischen Typ. Zudem definiert die Schnittstelle eine Methode über die ein Indexpfad festgelegt werden kann. Der Recovery Service kann auf diesem Pfad einen Index in Form von Dateien ablegen um seine Ergebnisse zu persistieren. Daher wäre es sinnvoll diesen beim Öffnen eines Projektes wieder einzulesen, anstatt ihn zeitaufwändig neu zu erzeugen. Aus diesem Grund wird in der `projectOpened()`-Methode der Komponente versucht, den Index vom konfigurierten Index-Pfad einzulesen. Falls dies scheitert, wird er neu generiert.

Dazu kann die Methode `refreshRecoveryService` der `TraceabilityRecoveryComponent` verwendet werden. Diese erstellt eine IntelliJ-Hintergrundaufgabe des Typs `RecoveryServiceRefreshTask` und startet diese. Dieser Mechanismus lässt das zeitaufwändige Parsen von Quellcode und Aufbauen des Index nebenläufig ablaufen, sodass die IDE nicht blockiert wird. Der Nutzer bekommt einen Hinweis im Event Log der IDE, wenn der Vorgang abgeschlossen ist und die Traceability Recovery-Funktionen genutzt werden können. Grundsätzlich stellt sich trotz nebenläufiger Aktualisierung des Services die Frage wann diese durchgeführt werden sollte. Jede Änderung am Quellcode könnte theoretisch eine Änderung der Traceability mit sich führen. Aufgrund des Rechenaufwandes und der Tatsache, das dieses Tool in den meisten Fällen auf bereits entwickeltem Code angewandt wird, wurde entschieden die Aktualisierung manuell durch den Nutzer über eine Action anzustoßen. Die `RefreshRecoveryServiceAction` ruft die Refresh-Methode der Projektkomponente auf. Eine Ausnahme ist die automatische Indexierung, falls bei Projektöffnung kein Lucene-Index gefunden wird.

6.2. Umsetzung der Anwendungsfälle

Laut der Anforderungen umfasst die Funktionalität zwei Anwendungsfälle zum Auffinden von Traceability Links zwischen Implementationen. Beide sind relativ ähnlich aufgebaut. Nach erfolgreicher Analyse der Implementationen werden sie vom Nutzer über Menüeinträge aufgerufen. In beiden Fällen wird eine Anfrage an den Recovery Service gestellt, welcher Ergebnisse liefert die visualisiert werden. Die Actions werden in den Klassen `EditorQueryAction` und `SearchQueryAction` implementiert und in der plugin.xml registriert. Dort wird auch festgelegt in welchen Knoten der

Benutzeroberfläche sie eingefügt werden. Da es sich um Quellcodeanalyse-Funktionen handelt, wurde ein neues Submenü in das „Analyze“-Menü integriert.

Die `EditorQueryAction` soll Links finden, die zum jeweiligen angeklickten PSI-Element passen. PSI-Elemente in der IntelliJ-API sind bestimmte Konstrukte der Sprache, wie z.B. Klassen-, Methoden- oder Attributsbezeichner. Die Utility-Klasse `TraceabilityPointerCreator` kann über die `getPointerForPsiElement(element)` aus einem PSI-Element einen Pointer aus dem Traceability-Konzept machen. Die Schnittstelle des Recovery Service bietet eine Methode `getSortedTraceability LinksForPointer(pointer)`, welche diesen verwendet um Entsprechungen in verlinkten Implementationen zu finden.

Bei der `SearchQueryAction` kann der User in einem Suchdialog eigene Suchbegriffe eingeben. Der Dialog ist über den API-eigenen `DialogWrapper` realisiert, von dem die GUI-Klasse `SearchQueryDialog` erbt. Der Suchdialog besteht primär aus einem Textfeld, in dem der Nutzer Suchbegriffe eingeben kann. Um die Eingabe zu vereinfachen und die Usability zu verbessern, integriert das Textfeld einen `TextFieldAutoCompletionListProvider`, welcher Vervollständigungsvorschläge auf Basis der PSI-Elemente der geöffneten Quellcodedatei macht. Ein Klick auf Ok ruft mit dem Set aus eingegebenen Suchbegriffen die Methode `getSortedTraceabilityPointersForQuery(terms)` des Service auf.

6.2.1. Verarbeitung von Suchbegriffen

Um die Suche von Tracelinks über Suchbegriffe zu verbessern, soll es möglich sein mehrere Suchbegriffe einzugeben. Durch die Angabe detaillierterer Suchparameter können passendere Ergebnisse gefunden werden. Daher ist der Suchdialog so entworfen, dass der Nutzer mehrere kommagetrennte Suchbegriffe eingeben kann. Der gesamte Suchstring wird nach Bestätigung des Dialoges innerhalb der `actionPerformed()`-Methode der `SearchQueryAction` in die einzelnen Suchbegriffe aufgetrennt. Wie im vorherigen Abschnitt beschrieben wird das Set aus Suchbegriffen dem Recovery Service übergeben.

Eine weitere Methode um die Qualität der Ergebnisse zu steigern ist das Zurückführen der eingegebenen Suchbegriffe auf ihren Wortstamm. Durch das Stemming soll eine höhere Übereinstimmung zwischen Wörtern mit gleichem Wortstamm und damit in der Regel ähnlicher Bedeutung erzielt werden. Während des in Unterabschnitt 6.1.2 beschriebenen Analyseprozesses des Quellcodes durch den Recovery Service wird dies bereits für alle Bezeichner durchgeführt. Um für die Suchbegriffe ein gleichförmiges Stemming zu erreichen ist es sinnvoll, denselben Stemmer zu verwenden. Der `LuceneTraceabilityRecoveryService` verwendet eine Instanz des `SourceCodeAnalyzer`. Wie in Abschnitt 3.4 erwähnt dienen Analyzer zur Verarbeitung eingelesener Tokens. Neben anderen Verarbeitungsschritten enthält der `SourceCodeAnalyzer` auch den in Lucene mitgelieferten Porter-Stemmer, welcher analysierte Begriffe auf ihre Wortstämme zurückführt. Innerhalb der Methode `getSortedTraceabilityLinksForQuery(terms)` wird der Analyzer und somit auch das Stemming auf die Tokens in der Suchanfrage angewendet. Durch die Verwendung desselben Analyzers sollte eine maximale Übereinstimmung zwischen Quellcodetokens und Suchanfragetokens erreicht werden.

6.2.2. Darstellen und Öffnen von Traceability Links

Auf eine Anfrage an den Recovery Service, ob über Suchbegriffe oder Codeelemente, folgt die Darstellung der Ergebnisse für den Nutzer. Da der Service auf beide Arten von Anfragen mit einer Liste von Tracelinks antwortet, können beide Anwendungsfälle auf dieselbe Logik für Visualisierung und später Öffnen der Links zugreifen. Das zentrale Element für die Darstellung der Ergebnisliste ist das `ResultsPopup`. Diese Klasse erzeugt Popup-Fenster, welches eine Liste enthält. Grundsätzlich werden Elemente von Benutzeroberflächen mithilfe der Java-eigenen GUI-Bibliothek Swing umgesetzt. Um möglichst nah am Aussehen der restlichen IDE zu bleiben, ist es sinnvoll weitestgehend die von IntelliJ angebotenen Erweiterungen von Swing-Komponenten zu verwenden. Das Popup-Fenster ist daher mit der Klasse `JBPopup` und `JBList` realisiert. Wie bei `JList` ist es möglich einen Renderer für einzelne Listeneinträge zur Verfügung zu stellen, welcher das Aussehen der Ergebnisliste anpasst. In diesem Fall sorgt der `ResultListCellRenderer` dafür, das jeder Ergebnislink mit seinem Namen, dem Namen der Swift-Datei aus der das Codeelement stammt, einem Icon das die Art des Traceability Pointers anzeigt und dem von Lucene bestimmten Übereinstimmungswert dargestellt wird. Abbildung 6.2 zeigt ausschnittsweise das Aussehen der Ergebnisliste.

Abbildung 6.2.: Darstellung der Ergebnisliste im Plugin

Eine Erweiterung der beiden Anwendungsfälle ist zudem das Öffnen der in Tracelinks hinterlegten Zielartefakte. Je nach Art des Traceability Pointers kann es sich bei den Codeelementen im Artefakt um Quellcodedateien, Klassen, Konstruktoren, Methoden oder Attribute handeln. Grundlegend sind durch die Tracing-Richtung Java zu Swift natürlich alle Arten von Zielartefakten innerhalb von Swift-Quellcodedateien enthalten. Neben dem Öffnen der richtigen Datei sollen jedoch spezifischere Codeelementen für den Nutzer hervorgehoben bzw. im Texteditor markiert werden. Zudem muss beachtet werden, dass das Plugin auf verschiedenen Betriebssystemen zum Einsatz kommen kann. Unterschiedliche Plattformen haben unterschiedliche Möglichkeiten wenn es darum geht programmatisch externe Anzeigeprogramme zu starten und zu bedienen.

Ein Klick auf einen Tracelink in der Liste des `ResultsPopup` versucht dessen Zielartefakt zu öffnen. Alle Zielartefakte sind in Form von `TraceabilityPointern` hinterlegt, welche über die Methode `openTraceabilityPointer(pointer)` eines `TPointerOpener` geöffnet werden. Um die oben angesprochenen Unterschiede zwischen Betriebssystemen abzudecken, implementieren verschiedenen Unterklassen von `TPointerOpener` die plattformspezifische Logik. Über die Factory-Klasse `TPointerOpenerFactory` kann mit `createOpener` ein Opener für die Plattform erzeugt werden, auf der das Plugin läuft.

Da im Rahmen dieser Studie die Traceability zur Mac-gebundenen Sprache Swift betrachtet wird, enthält der momentane Stand des Plugins neben einer Standardimplementation einen Opener für Mac mit Xcode (`MacTPointerOpener`). Dieser delegiert Aufrufe an den `XCodeController`, welcher

versucht die übergebenen Traceability Pointer mithilfe von AppleScript in Xcode zu öffnen. Dazu werden in der Klasse definierte Strings mit AppleScript-Code ausgeführt. So wird grundsätzlich die im Pointer hinterlegte Quellcodedatei in einem Xcode-Editor geöffnet. Bei spezifischeren Codeelementen wird deren Position im Quellcode zuerst über Pattern Matching ermittelt. Dabei fließt sowohl die typische Syntax von Sprachkonstrukten in Swift als auch die im Pointer hinterlegten Bezeichner mit in das Pattern ein. Anschließend wird die Zeilensuchfunktion von Xcode verwendet um die ermittelte Position hervorzuheben.

7. Evaluation

In diesem Kapitel soll die Implementierung des Plugins evaluiert werden. Dazu sollen zwei Aspekte betrachtet werden: Zum einen die Usability der Software und zum anderen die Nützlichkeit der Toolunterstützung für die parallele Entwicklung sowie Wartung von Mobilanwendungen. Dazu werden zuerst die Evaluationsmethode erläutert und anschließend die Ergebnisse analysiert.

7.1. Usability Evaluation

Es gibt verschiedene Methoden für Usability-Evaluationen. Diese sind grundlegend unterteilt in analytische und empirische Verfahren. Erstere beziehen sich auf Aussagen von Usability-Experten wohingegen bei empirischen Methoden die Erkenntnisse vom tatsächlichen Nutzer gewonnen werden. Weil für die hier durchzuführende Evaluation keine Usability-Experten zur Verfügung stehen fällt die Wahl auf eine empirische Methode. Darunter fallen vor allem Usability-Tests und Fragebögen. Da sich besonders Fragebögen dafür eignen ein gemitteltes Meinungsbild der Nutzer über die Software und zudem Erkenntnisse zu Problemen zu erlangen, wurde eine Befragung mittels Fragebogen für diese Studie ausgewählt. Über den Rahmen dieser Arbeit hinaus wäre es natürlich auch sinnvoll eine umfangreichere und aufwändigere Evaluation über Usability-Tests durchzuführen um konkretere Erkenntnisse zur Bedienbarkeit des Plugins zu erhalten [SB11].

7.1.1. Erstellung des Fragebogens

Zu den Zielen der Softwareevaluation gehören grundsätzlich folgende Fragestellungen: Wie gut ist die Ausprägung bestimmter Systemeigenschaften und wo identifiziert der Befragte Schwachstellen und warum? Zudem könnte eine Auswahl an Design-Alternativen vorgestellt werden, von denen die geeignetste ausgewählt werden soll. Dies ist allerdings in dem hier betrachteten Fall nicht anwendbar, da eine bereits konkret implementierte Software evaluiert wird. Nichtsdestotrotz sollten konkrete Verbesserungsvorschläge des Designs durch die Evaluierenden festgehalten werden können. Sarodnick et al. stellen einige in Forschung und Praxis verwendete standardisierte Fragebögen für Softwareevaluation vor. Diese ermöglichen in der Regel bereits ohne Anpassung die Durchführung einer aussagekräftigen Evaluation. Für die hier durchzuführende Evaluation wurden nur standardisierte Fragebögen betrachtet, die kostenlos verfügbar sind. Näher zu betrachten sind daher ISONORM 9241/110-S und IsoMetrics. Beide bestehen aus Frage-Items, die direkt Bezug auf Qualitätsmerkmale der DIN EN ISO 9241-110 für Ergonomie der Mensch-System-Interaktion nehmen. ISONORM ist mit 21 Items jedoch deutlich kompakter als IsoMetrics (90 Items)[SB11]. Vor allem muss beachtet werden, dass nicht alle Items der Fragebögen auf die vorliegende Software anwendbar sind. Darunter fallen Fragen zu nicht vorkommenden Funktionalitäten, vor allem da die Vorlagen für eigenständige Softwaresysteme konzipiert sind und nicht für Plugins. Aus diesem

Grunde eignet ISONORM sich eher für eine generelle Abfrage der Usability des Plugins, da die Items besser zur hier zu evaluierenden Software passen.

Zudem wäre es sinnvoll über die allgemeinen Items hinaus einige weitere Fragen zur Nützlichkeit des Plugins im Entwicklungsprozess und des Traceability-Konzeptes einzufügen. Als Basis des entworfenen Fragebogens wurde die grundlegende Struktur und Bewertungsskala (sieben Stufen von sehr negativ bis sehr positiv) von ISONORM übernommen. Auch wurden die Items übernommen, die für die Evaluation des Traceability Recovery Plugins relevant sind. Fragen über Syntaxprüfungen für Eingaben wurden entfernt, da diese in der vorliegenden Software nicht vorkommen. Stattdessen wurden Items über die Integration des Plugins in die Entwicklungsumgebung (F4, F6, F11) und über die Nützlichkeit der Software im Entwicklungsprozess (F16, F17) eingefügt. Über die Bewertungsskalen hinaus wird dem Befragten mit Freitextfeldern die Möglichkeit gegeben Antworten zu erläutern oder konkrete Designverbesserungen vorzuschlagen. Der finale Fragebogen ist auf Seite 42 im Anhang zu finden.

7.1.2. Durchführung der Evaluation

Die Durchführung der Evaluation wurde in drei Gruppen von Evaluationsteilnehmern unterteilt, um eine Vielzahl unterschiedlicher Perspektiven und möglicher Anwendungsfälle für die Software abzudecken. Zum einen wurden Teilnehmer des Masterprojekts „SWK Softwareentwicklungsmethoden 2016“ befragt, die durch die Teilnahme an diesem Projekt bereits Vorwissen im Bereich der Software-Traceability und dem Anwendungsfeld der zu evaluierenden Software hatten. Zum anderen wurden Teilnehmer aus dem Seminar „Software-Reengineering“ im Sommersemester 2017 befragt, die das Tool im Rahmen des Seminars verwendet haben. Für diese Gruppe wurde angenommen, dass kein themenspezifisches Vorwissen vorhanden war, da dieser Aspekt nicht zuverlässig abgefragt werden konnte. Als dritte Gruppe wurden über den universitären Rahmen hinweg Entwickler der Softwareentwicklungsfirma ICNH in Hamburg befragt. Aufgrund der unterschiedlichen Umstände mit den einzelnen Gruppen unterschied sich der Ablauf der Evaluation im Detail.

Die Gruppe an Teilnehmern mit fachlichem Vorwissen umfasste vier Personen. Ihnen wurde eine vorkonfigurierte Evaluationsumgebung zu Verfügung gestellt, in welcher Android Studio mit dem Traceability Recovery Plugin sowie Xcode installiert war. Der Quellcode der Twitter-Applikation „Twidere“ lag in Java für Android und Swift für iOS vor. Im Rahmen der Evaluation sollte zusätzlich zu eigenen Erfahrungen mit dem Plugin das Szenario auf der ersten Seite des Evaluationsbogens durchgearbeitet werden um sicherzustellen, dass alle zu evaluierenden Aspekte betrachtet wurden. Im Anschluss sollte der Evaluationsbogens ausgefüllt werden.

Die Durchführung in der Gruppe ohne fachliches Vorwissen fand im Rahmen einer Seminaraktivität statt, welche sich mit dem Einarbeiten in unbekannten Code zum Zwecke eines Reengineerings beschäftigte. Dazu sollte geprüft werden, ob die Toolunterstützung das Codeverständnis unterstützen kann. Statt des im Evaluationsbogen vorgegebenen Szenarios sollten die Teilnehmer Swift-Code-Entsprechungen zu vorgegebenen Java-Klassen finden und deren Funktion beschreiben. Auch hier wurde das „Twidere“-Projekt als Codebasis verwendet. Für die Hälfte der zu findenden Verknüpfungen sollte das Traceability Recovery Plugin als Hilfestellung verwendet werden. Die

andere Hälfte sollte manuell gefunden werden. Im Anschluss an die Seminaraktivität füllten elf der Teilnehmer auch hier den Evaluationsbogen aus.

Die Evaluation mit Entwicklern von ICNH begann mit einer Präsentation der Software um den fachlichen Hintergrund, den Funktionsumfang und den Entwicklungsstand zu demonstrieren. In einer anschließenden Diskussion wurden Nützlichkeit und Benutzbarkeit des Plugins von den Entwicklern eingeschätzt und Anmerkungen zu Verbesserungen gegeben. An der Diskussion nahmen vier Mitarbeiter von ICNH teil. Zudem wurde abgesprochen, dass die Entwickler das Tool für zwei Wochen im Praxiseinsatz testen um im Anschluss die Evaluationsbögen ausfüllen zu können. Aufgrund von technischen Schwierigkeiten konnte dies jedoch leider nicht umgesetzt werden. Dennoch sollen die Diskussionsergebnisse mit in die Evaluation einfließen. Ab Seite 50 ist dazu das Gespräch in Form eines Gedächtnisprotokolls festgehalten.

7.1.3. Evaluationsergebnisse

Die Evaluation erbrachte einige sehr wertvolle Erkenntnisse über die Nützlichkeit und Benutzbarkeit des entwickelten Werkzeugs. Die Auswertung der Evaluationsbögen ist ab Seite 47 im Anhang zu finden. Darunter fallen sowohl die quantitative Auswertung der geschlossenen Fragen als auch eine Auflistung der Einträge in die Freitextfelder.

Insgesamt ist festzustellen, dass die Software sowohl in den Fragen zur Usability als auch der Nützlichkeit gegenüber dem vorliegenden Anwendungsfall eher im guten bis sehr guten Bereich bewertet wurde. Im Durchschnitt wurde kein abgefragter Aspekt schlechter als eine neutrale Bewertung eingeschätzt. Allerdings gibt es dennoch einige Auffälligkeiten auf die besonders eingegangen werden sollte. Für beide Szenarien, das in Evaluationsbogen vorgegebene und das im Seminar Durchgeführte wurde angegeben, dass die anfallenden Aufgaben durch die Toolunterstützung effizient bewältigt werden konnten. Teilnehmer der fachfremden Gruppe bewerteten die Gestaltung der Software und die Integration in die Entwicklungsumgebung generell schlechter, obwohl die hohe Standardabweichung dabei auf Uneinigkeit hindeutet. Dies könnte damit zusammenhängen, dass es ohne fachliche Kenntnis schwerer ist, die angebotenen Funktionen innerhalb der IDE in den Nutzungskontext einzuordnen. Über beide Gruppen hinweg ist jedoch festzustellen, dass der Mangel eines Handbuches, einer Dokumentation oder einer Hilfestellung im Programm die Nutzung erschwerte. Auch die Implementation von zusätzlichen Tooltips an den Menüpunkten (Actions) könnte die Software intuitiver gestalten.

Interessanterweise konnte festgestellt werden, dass die Verwendung des Plugins aus Sicht der Befragten den erwünschten Arbeitsprozess eher nicht durch Wartezeiten unterbricht. Dies ist vor allem in Hinblick auf die zeitaufwendige Quellcodeanalyse über Lucene überraschend. Natürlich muss angemerkt werden, dass die Wartezeit beim Parsen des Quellcode vom Umfang der Codebasis abhängig ist. Zum Zwecke dieser Evaluation wurde eine relativ kleine Codebasis verwendet, wodurch die Parsingzeit moderat bleibt. Im Gespräch mit den ICNH-Entwicklern wurde eine lange Parsingzeit durchaus als Problem für den produktiven Einsatz gesehen. Vor allem im Falle einer aktiven, parallelen Entwicklung würde sich der Code so oft verändern, dass eine lange Wartezeit während der Codeanalyse störend wäre. Aus den Evaluationsbögen ergibt sich als größtes Problem der momentanen Implementierung die Fehlerbehandlung. Darunter fallen auch Fehler die durch inkorrekte oder fehlende Konfiguration des Plugins auftreten. Obwohl die angezeigten

Fehlermeldungen verständlich waren, wurden dem Nutzer nicht immer konkrete Hinweise für die Behebung des Fehlers gegeben. Konnte der Nutzer die Fehlerursache dennoch ausmachen wurde der anfallende Aufwand für die Behebung als eher gering bewertet. Kritische Programmfehler die nicht durch den Nutzer zu beheben sind, sind während der Evaluation nicht aufgetreten. Einschränkend muss gesagt werden, dass das Szenario der Gruppe ohne fachlichem Vorwissen so konstruiert war, dass kaum Konfigurationsfehler auftreten konnten. Aus diesem Grund wurden die Fragen F12-14 von ca. zwei Drittel der Teilnehmern dieser Gruppe nicht beantwortet.

Die Fragen F16 und F17 beschäftigten sich nicht mit der Usability sondern der Einsetzbarkeit des Werkzeugs. Hier wurde zwischen dem Einsatz zur Unterstützung von paralleler Entwicklung sowie Wartung von nativen Implementationen von Mobilanwendungen unterschieden. Parallele Entwicklung bezieht sich auf die aktive Codeentwicklung und Umsetzung von Funktionen wohingegen Wartung die Fehlerbehebung in bestehendem Code bezeichnet. In den Bereich der Wartung fällt auch das Erreichen von Codeverständnis im Rahmen eines Reengineerings. Obwohl die Eignung des Plugins für beide Anwendungsfälle als sehr gut bewertet wurde, lässt sich aus den Evaluationsergebnissen ablesen, dass es aus Sicht der Teilnehmer in der Wartung sinnvoller einsetzbar sei. Dies wird auch durch Aussagen der ICNH-Entwickler unterstützt, die besonders die Hilfestellung bei der Einarbeitung in unbekannten, bestehenden Code als sinnvoll ansehen. Durch die Verwendung der Traceability Recovery kann auch ein Entwickler der nicht mit der Implementation für iOS vertraut ist, Swift-Codeartefakte auffinden die bekannten Java-Artefakten entsprechen. Weiterhin sahen die Entwickler bei der Nutzung für parallele Softwareentwicklung die zusätzlichen Informationen aus dem Tool als weniger hilfreich an, als eine etwaige Befragung der an anderen Implementationen arbeitenden Entwickler. Die Freitextfelder zu den oben genannten Fragen geben einige weitere Argumente. Grundsätzlich nimmt das Plugin dem Nutzer manuelle Sucharbeit ab und ermöglicht auch Entwicklern die keine Erfahrung mit der Programmiersprache Swift haben, korrespondierende Swift-Klassen und andere Artefakte zu bestimmten implementierten Konzepten zu finden. So könnte beispielsweise eine Wartung unterstützt werden, auch wenn der ursprüngliche Entwickler der Swift-Version nicht mehr verfügbar ist. Auch wurde von Befragten festgestellt, dass das Auffinden korrespondierender Code-Artefakte mit dem Werkzeug schneller ging als über manuelle Suche.

Eine weitere Anmerkung der ICNH-Entwickler betrifft die sehr linguistische Auswertung des Quellcodes. Artefaktbezeichner setzen sich oftmals aus einem sprach- bzw. plattformspezifischen Teil, der sprachspezifische Konstrukte oder Verknüpfungen zur Plattform-API identifiziert und einem fachlichen Domain-Teil, der auf Funktionalität innerhalb der Applikation hinweist, zusammen. Letzterer Teil wird in unterschiedlichen Implementationen oft gleichartig verwendet. Die sprachspezifischen Teile des Bezeichners unterscheiden sich jedoch um innerhalb der Sprachkonventionen zu bleiben. Dieses Phänomen ist beispielhaft bei den Artefaktbezeichnern `LoginActivity` und `LoginViewController` zu erkennen. Ohne weitere Konfiguration ist das Plugin nur in der Lage über den linguistisch gleichen Teil „Login“ eine Verknüpfung zu erkennen. Besser wäre eine Auswertung beider Teile des Artefaktbezeichners. Dazu wären jedoch Zusatzinformationen nötig, die dem Analyseservice helfen zu erkennen, dass die API-Konstrukte `Activity` und `ViewController` übereinstimmende Funktionen für verschiedene Sprachen bzw. Plattformen darstellen. Auf Seite des Plugins ist eine Konfigurationsoberfläche für solche Mappings vorhanden, allerdings ist die Verwendung der konfigurierten Mappings im Analyseservice nicht im Umfang dieser Ausarbeitung

enthalten und konnte so zum Zeitpunkt der Evaluation nicht mit betrachtet werden. Dieser Aspekt wird in der Studie über die Quelltextanalyse von Jakob Andersen näher untersucht [And17].

Über die Auswertung der geschlossenen Fragen hinaus konnten einige allgemeine Anmerkungen und vor allem Verbesserungsvorschläge gesammelt werden. Mehrfach trat der Wunsch nach einer Filterfunktion innerhalb der Ergebnisliste auf. In den meisten Fällen wird eine Java-Klasse auch in Swift als Klasse realisiert sein, insofern wäre es hilfreich Ergebnisartefakte wie Methoden und Attribute herauszufiltern. Ein selbst wählbarer Filter nach Artefakttyp würde die Übersichtlichkeit der Ergebnisanzeige verbessern.

Als störend wurde der Zwang zur Nutzung eines Komma zur Trennung von Suchbegriffen im Suchfenster angesehen. Technisch ist es natürlich notwendig eine klare Regel für die Trennung der Begriffe zu definieren. Alternativ wäre die Trennung nach jeglicher Form von Leerraum denkbar, allerdings würde dies unter Umständen Probleme verursachen, falls der Nutzer nach natürlichsprachlichen Suchbegriffen sucht. Als Beispiel wäre hier der Suchbegriff „Bluetooth connection“ zu sehen der in zwei Begriffe getrennt an Kontext verlieren würde.

Die momentane Implementation der Öffnung von ausgewählten Codeartefakten in Xcode öffnet diese in neuen Xcode-Fenstern. Es wurde vorgeschlagen die Artefakte stattdessen im passenden Kontext des verlinkten Swift-Projektes zu öffnen. Weitere Verbesserungsvorschläge umfassten die bessere Dokumentation der Pluginfunktionalität über ein Hilfefenster, eine einfacher zu verstehende Darstellung der Ergebnisliste mit Spaltenbezeichnern und treffender benannte Menüeinträge. Die vollständige Liste der Verbesserungsvorschläge ist auf Seite 48 und 49 zu finden.

Insgesamt kann festgehalten werden, dass das Traceability Recovery Plugin als durchaus nützliche Toolunterstützung für die parallele Entwicklung und Wartung von nativen Implementationen einer Mobilanwendung dient. Der Hauptvorteil wird jedoch im Bereich der Wartung durch die gesteigerte Nachvollziehbarkeit von Parallelen zwischen Implementierungen gesehen. Die Usability wird grundsätzlich als gut angesehen, wobei besonders die Erlernbarkeit des Plugins unterstützt und die Fehlerbehandlung verbessert werden müsste. Insgesamt ist die vorliegende Implementierung als Proof of Concept zu sehen, an welche durch umfangreichere Tests in der Praxis weitergehende Anforderungen gestellt werden könnten. Zum Zeitpunkt der Abgabe wurden einige der Verbesserungsvorschläge wie die Filterfunktion und eine hilfreichere Fehlerbehandlung in einer aktualisierten Version umgesetzt.

8. Fazit

Die grundlegende Zielsetzung dieser Studie ist die Entwicklung eines Android Studio-Plugins mit dessen Hilfe Traceability zwischen nativen Implementationen einer Mobilanwendung für verschiedene Plattformen erzeugt werden kann. Die erzeugten Tracelinks verknüpfen Codeartefakte in verschiedenen Implementationen die dieselben Funktionalitäten umsetzen. Entwicklern wird dadurch die Möglichkeit gegeben ohne eigenen Suchaufwand verknüpfte Codeartefakte zu finden. Zum einen können so Änderungen bei der Wartung von Software in allen verknüpften Implementationen gleichartig durchgeführt werden. Zum anderen wird eine möglichst gleichartige Implementierung bei der parallelen Entwicklung einer Mobilanwendung für mehrere Plattformen unterstützt. Anzumerken ist, dass die Traceability nicht bei der Erzeugung des Quellcodes, sondern nachträglich aus bestehendem Code erstellt wird.

Diese Studie definiert zur Entwicklung dieser Toolunterstützung zuerst die notwendigen Anforderungen. Die theoretische Zielsetzung wurde beispielhaft mit der Erzeugung der Nachverfolgbarkeit von Java-Codeartefakten für Android-Projekte zu Swift-Codeartefakten für iOS-Projekte mittels des Android Studio-Plugins umgesetzt. Der Entwurf des Plugins sowie dessen Implementation sind dargestellt und dokumentiert.

Durch die Evaluation der Software konnte festgestellt werden, dass die vorliegende Implementation die oben genannten Anwendungsfälle sinnvoll unterstützt. Sowohl für die parallele Entwicklung als auch die Wartung von nativen Implementationen von Mobilanwendungen wurde das Werkzeug als hilfreiche Unterstützung angesehen. Allerdings kann festgehalten werden, dass sich das Werkzeug besonders für die Wartung von bestehendem Code eignet, da es primär das Codeverständnis fördert, welches bei der parallelen Entwicklung oftmals bereits präsent ist. Weiterhin konnten aus der Evaluation wertvolle Verbesserungsvorschläge gewonnen werden, die detailliert im vorherigen Kapitel dargelegt sind.

Im Ausblick sollten diese Verbesserungsvorschläge geprüft und bei Eignung umgesetzt werden, um die Benutzbarkeit und Nützlichkeit des Plugins zu steigern. Des Weiteren wäre eine umfangreichere Evaluation im Rahmen von realen Entwicklungs- und Wartungsprojekten hilfreich um weitere, bisher nicht betrachtete Anforderungen aufzudecken. Als denkbare Erweiterungen wäre beispielsweise eine Verlagerung des Codeanalyseprozesses vom Entwicklerrechner auf ein verwendetes Versionierungssystem, um die momentan auftretenden Wartezeiten zu minimieren. Auf einen Commit der Codebasis könnte automatisch die Traceability-Analyse folgen, damit die Tracelink-Informationen zur Verfügung stehen sobald sie vom Entwickler benötigt werden. Eine weitere Erweiterung wäre die Möglichkeit gefundene Tracelinks als korrekt oder falsch zu kennzeichnen und so während der Nutzung ein zusammenhängendes, persistentes Traceability-Modell zwischen verschiedenen Implementationen aufbauen zu können. Zudem sollte geprüft werden ob das verwendete Stemming-Verfahren die Qualität der gefundenen Tracelinks bei Suchanfragen tatsächlich verbessert oder ob andere Methoden der Inputverarbeitung zu besseren Ergebnissen

führen würden. Insgesamt kann festgehalten werden, dass die hier vorgestellte Toolunterstützung durchaus sinnvolles Anwendungs- und Erweiterungspotential besitzt.

Literaturverzeichnis

[ABF+17] Andersen, Jakob ; Berger, Nils-Hendrik ; Fischer, Evelyn ; Greiert, Gerrit ; Jährling, Claas ; Khanji, Fares ; Schulz, Maike: *Mobile Anwendungen für mehrere Plattformen – Portierung, Architektur- und Tool-Entwicklung*. April 2017. – Universität Hamburg SWK Masterprojekt 2016/17

[And17] Andersen, Jakob: *Zerlegen von Java- und Swift-Quelltexten zwecks Ähnlichkeitsanalyse*, Universität Hamburg, Masterstudie, 2017

[App16] Apple: *Introduction to AppleScript Language Guide*. `https://developer.apple.com/library/content/documentation/AppleScript/Conceptual/AppleScriptLangGuide/introduction/ASLR_intro.html`, 2016. – zuletzt abgerufen am 22.04.2017

[Fou16] Foundation, The Apache S.: *Apache Lucene Core*. `https://lucene.apache.org/core/`, 2016. – zuletzt abgerufen am 23.03.2017

[Fou17] Foundation, The Apache S.: *Apache Lucene Package Description org.apache.lucene.analysis*. `https://lucene.apache.org/core/6_4_2/core/org/apache/lucene/analysis/package-summary.html#`, 2017. – zuletzt abgerufen am 23.03.2017

[GCHH+12] Gotel, Orlena ; Cleland-Huang, Jane ; Hayes, Jane H. ; Zisman, Andrea ; Egyed, Alexander ; Grünbacher, Paul ; Dekhtyar, Alex ; Antoniol, Giuliano ; Maletic, Jonathan ; Mäder, Patrick: Traceability fundamentals. In: *Software and Systems Traceability*. Springer, 2012, S. 3–22

[HDS+07] Hayes, Jane H. ; Dekhtyar, Alex ; Sundaram, Senthil K. ; Holbrook, E. A. ; Vadlamudi, Sravanthi ; April, Alain: *REquirements TRacing On target (RETRO): improving software maintenance through traceability recovery*. 2007

[Int15] IntelliJ: *IntelliJ Platform SDK DevGuide: Plugin Structure*. `http://www.jetbrains.org/intellij/sdk/docs/basics/plugin_structure.html`, November 2015. – zuletzt abgerufen am 22.03.2017

[J+11] Jivani, Anjali G. u. a.: A comparative study of stemming algorithms. In: *Int. J. Comp. Tech. Appl* 2 (2011), Nr. 6, S. 1930–1938

[LFOT05] Lucia, Andrea D. ; Fasano, Fausto ; Oliveto, Rocco ; Tortora, Genoveffa: *ADAMS Re-Trace: a Traceability Recovery Tool*. 2005

[LFOT06] Lucia, Andrea D. ; Fasano, Fausto ; Oliveto, Rocco ; Tortora, Genoveffa: *Can Information Retrieval Techniques Effectively Support Traceability Link Recovery?* 2006

[MR04] Mishne, Gilad ; Rijke, Maarten de: *Source Code Retrieval using Conceptual Similarity*. 2004

[OGPL10] Oliveto, Rocco ; Gethers, Malcom ; Poshyvanyk, Denys ; Lucia, Andrea D.: *On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery*. 2010

[Por80] Porter, Martin F.: An algorithm for suffix stripping. In: *Program* 14 (1980), Nr. 3, S. 130–137

[Por01] Porter, Martin F.: *Snowball: A language for stemming algorithms*. `http://snowball.tartarus.org/texts/introduction.html`, 2001. – zuletzt abgerufen am 23.03.2017

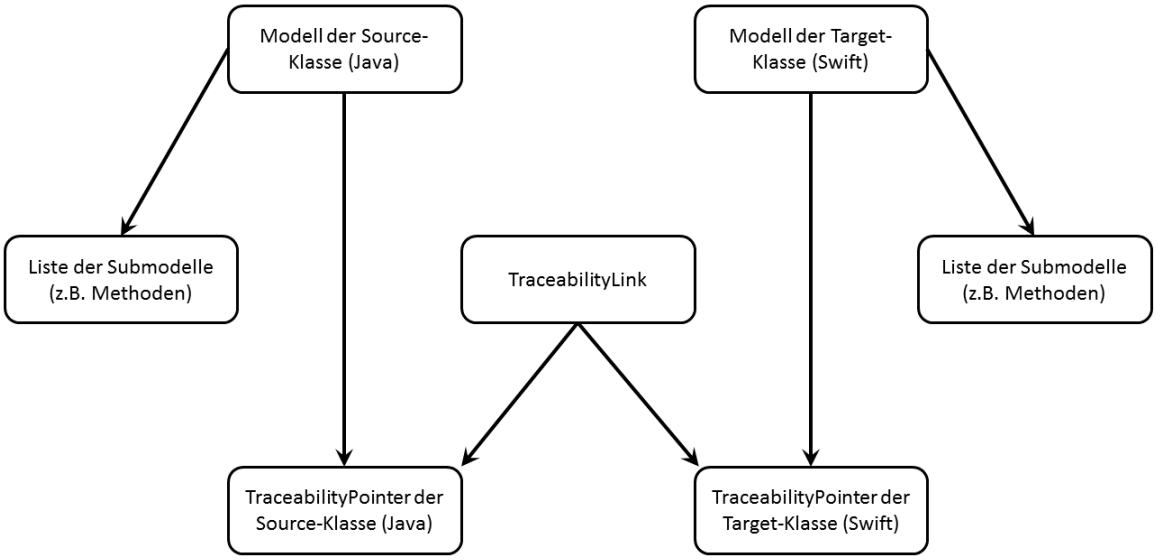
[SB11] Sarodnick, Florian ; Brau, Henning: *Methoden der Usability Evaluation: wissenschaftliche Grundlagen und praktische Anwendung*. Huber, 2011 (Aus dem Programm Huber: Wirtschaftspsychologie in Anwendung). – ISBN 9783456848839

[Som11] Sommerville, I.: *Software Engineering*. Pearson, 2011 (International Computer Science Series). – ISBN 9780137053469

[ZWRH06] Zhang, Yonggang ; Witte, René ; Rilling, Juergen ; Haarslev, Volker: *An Ontology-based Approach for Traceability Recovery*. 2006

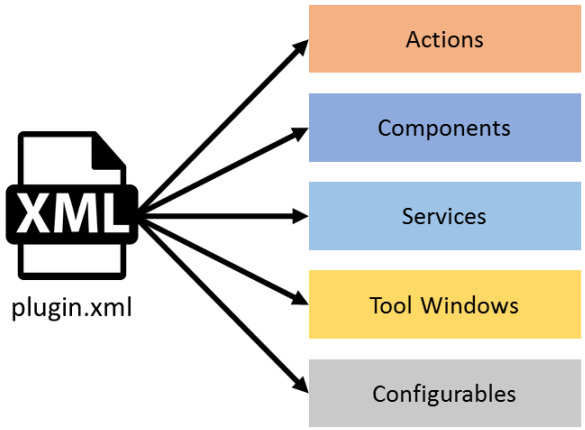
Anhang

A. Traceability Links im Traceability-Modell



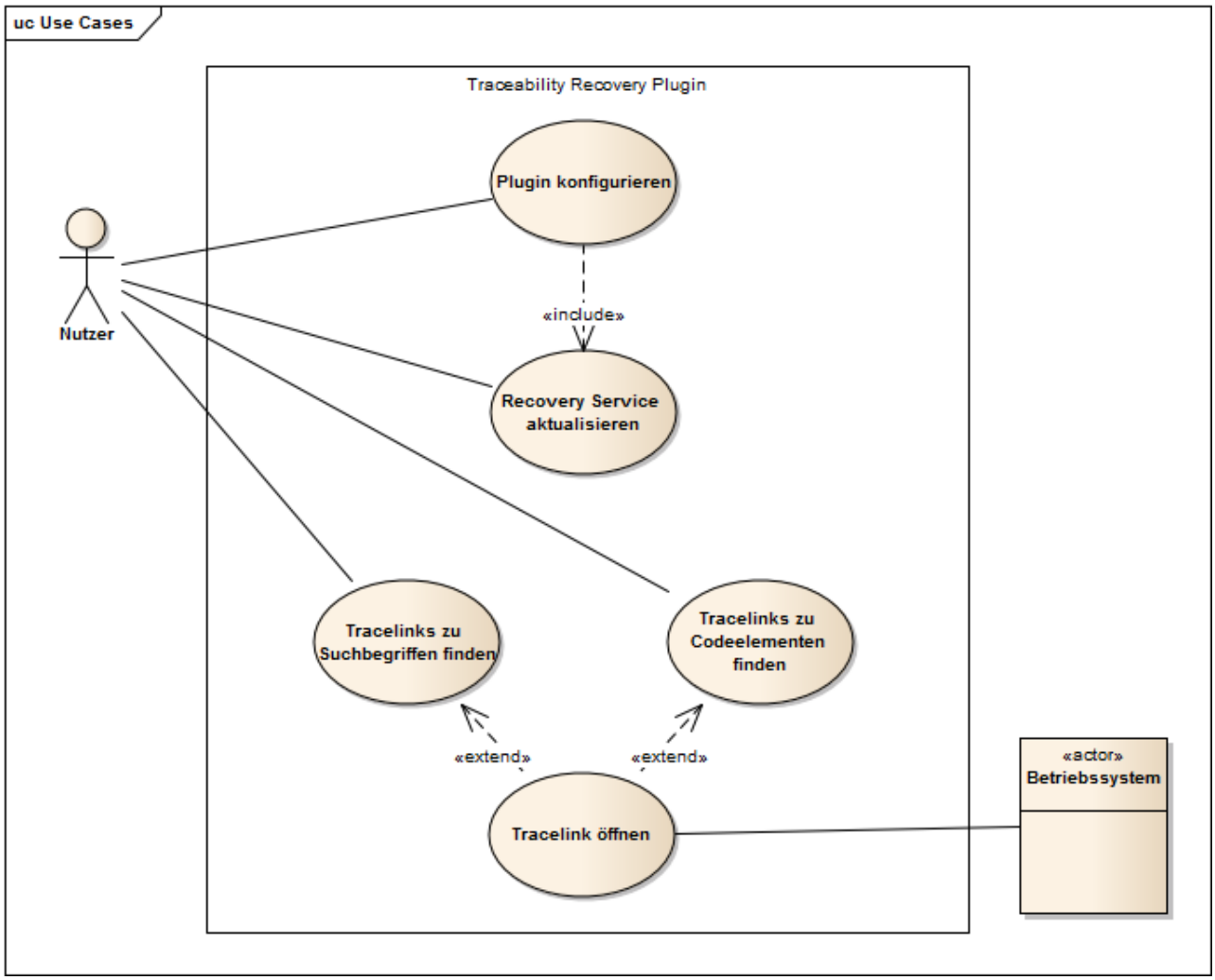
Verbindung eines Java- und Swift-Modells durch einen Traceability Link

B. IntelliJ-Plugin-API



Elemente einer plugin.xml-Datei

C. Use Case-Beschreibungen

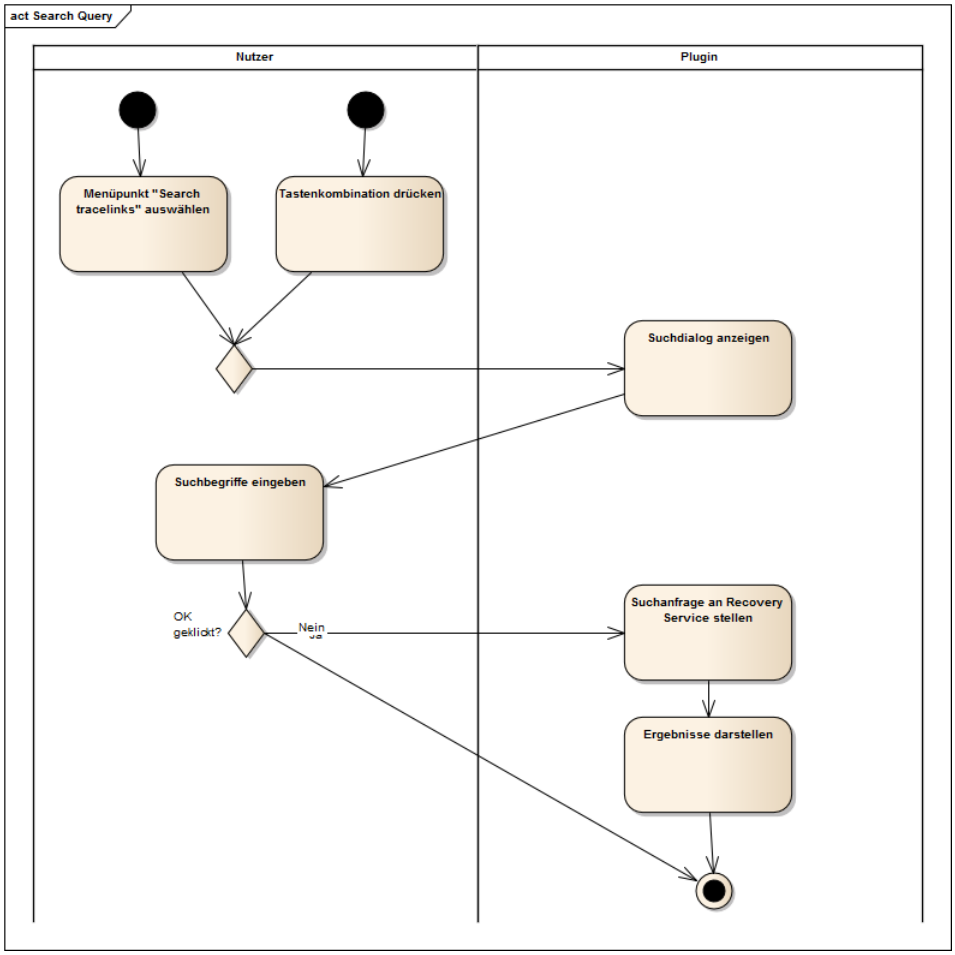


Use Cases des Traceability Recovery Plugins

C.1. Finden von Tracelinks über eingegebene Suchbegriffe

Use Case:	Tracelinks zu Suchbegriffen finden
Ziel:	Nutzer bekommt zu den Suchbegriffen passende Tracelinks gezeigt
Kategorie:	primär
Vorbedingung:	Der Recovery Service hat sowohl die Java- als auch Swift-Implementation analysiert und einen Index erstellt
Nachbedingung Erfolg:	Ermittelte Tracelinks wurden dem Nutzer gezeigt
Nachbedingung Fehlschlag:	Es wurde eine Fehlermeldung angezeigt, die dem Nutzer Hilfestellung zur Fehlerbehebung bietet
Akteure:	Nutzer
Auslösendes Ereignis:	Der Nutzer klickt auf den Menüpunkt „Search tracelinks“ oder drückt die korrespondierende Tastenkombination

Ablauf:

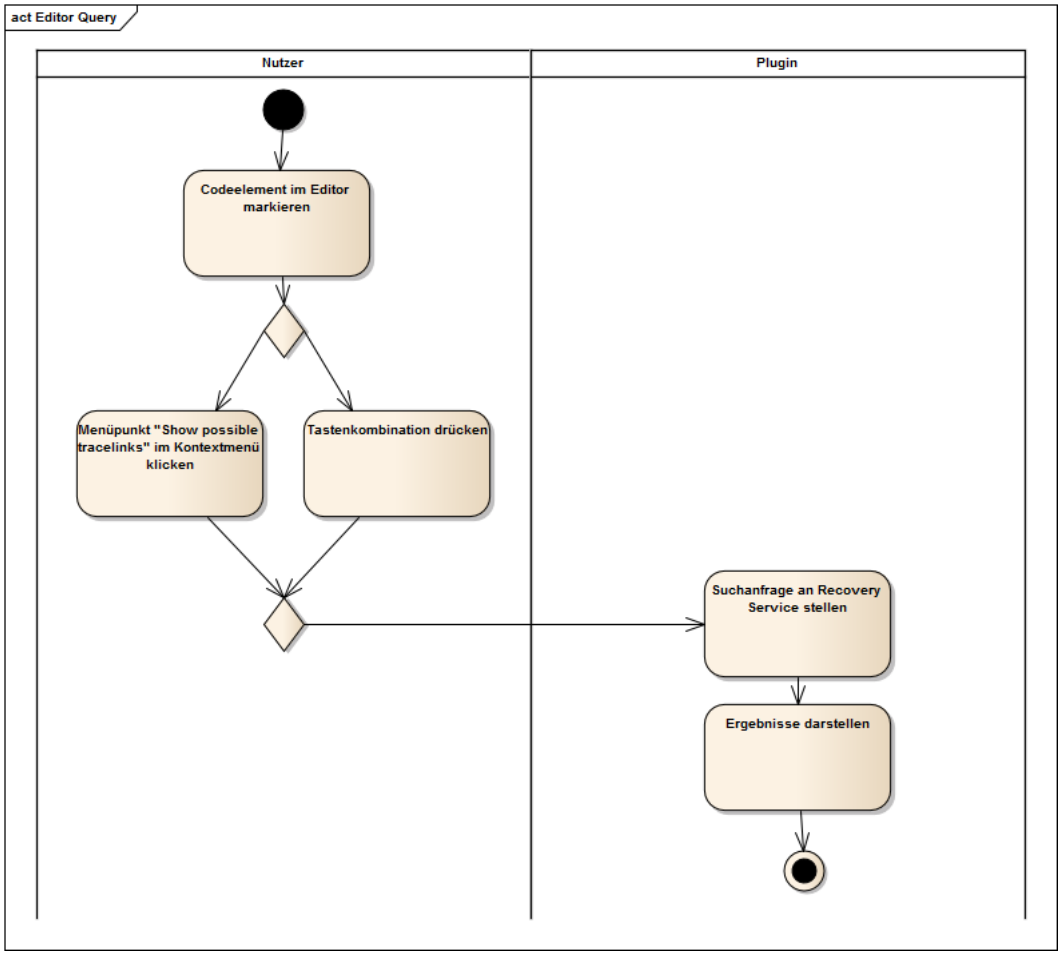


Aktivitätsdiagramm zum Use Case "Tracelinks zu Suchbegriffen finden"

C.2. Finden von Tracelinks zu Codeelementen im Quellcode-Editor

Use Case:	Tracelinks zu Codeelementen finden
Ziel:	Nutzer bekommt zum ausgewählten Codeelement passende Tracelinks gezeigt
Kategorie:	primär
Vorbedingung:	Der Recovery Service hat sowohl die Java- als auch Swift-Implementation analysiert und einen Index erstellt
Nachbedingung Erfolg:	Ermittelte Tracelinks wurden dem Nutzer gezeigt
Nachbedingung Fehlschlag:	Es wurde eine Fehlermeldung angezeigt, die dem Nutzer Hilfestellung zur Fehlerbehebung bietet
Akteure:	Nutzer
Auslösendes Ereignis:	Der Nutzer klickt auf den Menüpunkt „Show possible tracelinks“ oder drückt die korrespondierende Tastenkombination

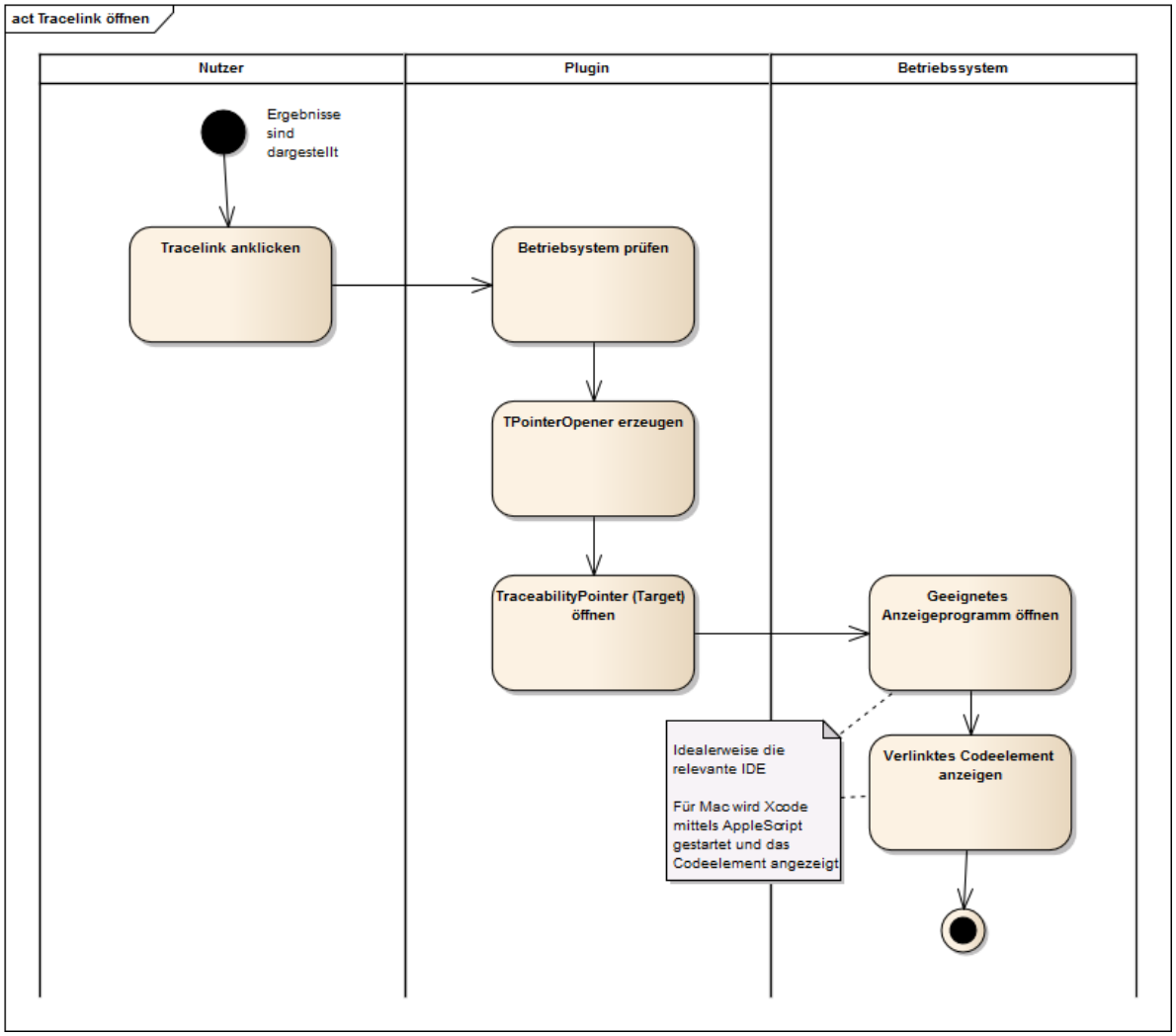
Ablauf:



Aktivitätsdiagramm zum Use Case "Tracelinks zu Codeelementen finden"

C.3. Öffnen von Zielartefakten von Tracelinks

Use Case:	Öffnen von Zielartefakten
Ziel:	Das Zielartefakt eines ausgewählten Tracelinks wird in einem geeigneten Programm geöffnet und dem Nutzer angezeigt
Kategorie:	primär
Vorbedingung:	Die Ergebnisse eines Tracing-Prozessen wurden angezeigt
Nachbedingung Erfolg:	Das Zielartefakt wurde geöffnet und dem Nutzer angemessen präsentiert
Nachbedingung Fehlschlag:	Es wurde eine Fehlermeldung angezeigt, die dem Nutzer Hilfestellung zur Fehlerbehebung bietet
Akteure:	Nutzer, Betriebssystem
Auslösendes Ereignis:	Der Nutzer klickt auf einen Tracelink in der Ergebnisanzeige
Ablauf:	



Aktivitätsdiagramm zum Use Case "Öffnen von Zielartefakten"

D. Evaluationsbogen

Evaluation des Traceability Recovery Plugins für IntelliJ

entwickelt im Rahmen einer Studie im Arbeitsbereich SWK der Universität Hamburg

Bitte beachten Sie:

- Das Ziel dieser Beurteilung ist es, die Qualität der Usability der Software zu ermitteln und konkrete Verbesserungsvorschläge zu entwickeln
- Dabei geht es nicht um eine Beurteilung Ihrer Person, sondern um Ihre persönliche Bewertung der Software
- Bitte bearbeiten Sie zusätzlich zu bereits vorhandenen Erfahrungen mit der Software folgendes Szenario:

Szenario:

Machen Sie sich innerhalb von 5-10 Minuten mit der Funktionsweise des Traceability Recovery Plugins vertraut. Folgen Sie dazu zunächst dem folgenden Ablauf:

1. Öffnen Sie das Android Studio-Projekt „Twidere_Domain_Android“ in Android Studio.
2. Konfigurieren Sie das Plugin über die Android Studio-Einstellungen. (Cmd + Komma)
   - Der Ordner „Desktop/Studie_Evaluation/Twidere_Domain_iOS“ enthält die Implementation in Swift.

Die Funktionen des Plugins befinden sich in der Menügruppe „Analyze“. Zum Parsen des Codes muss der Traceability-Service aktualisiert werden („Refresh recovery service“).

3. Rufen Sie Trace-Links zu Klassen-, Methode- und Attributsbezeichnern auf („Show possible traceability links“)
   - Finden Sie Codeartefakte in Swift, die laut Recovery-Analyse die Java-Klasse „Credentials“ implementieren
   - Die Funktion ist auch als Intention verfügbar (Glühlampen-Icon oder Alt + Enter auf markierten Bezeichnern)
4. Suchen Sie Codeartefakte in Swift mithilfe von Suchbegriffen („Search traceability links“)
   - Finden Sie Codeartefakte in Swift, die laut Recovery-Analyse das Konzept eines „Account“ implementieren
   - Versuchen Sie die Suchergebnisse durch mehrere Suchbegriffe zu verbessern
5. Klicken Sie auf einen der Tracelinks um das verlinkte Codeartefakt in Xcode zu öffnen.

Fragen:

	Die Software…	--- -- - -/+ + ++ +++	Die Software…
F1	bietet nicht alle Funktionen um die anfallenden Aufgaben des Szenarios effizient zu bewältigen	○ ○ ○ ○ ○ ○ ○	bietet alle Funktionen um die anfallenden Aufgaben des Szenarios effizient zu bewältigen
F2	erfordert überflüssige Eingaben	○ ○ ○ ○ ○ ○ ○	erfordert keine überflüssigen Eingaben
F3	ist schlecht auf die Anforderungen der Arbeit zugeschnitten	○ ○ ○ ○ ○ ○ ○	ist gut auf die Anforderungen der Arbeit zugeschnitten
F4	ist schlecht in den Arbeitsfluss mit IntelliJ integriert	○ ○ ○ ○ ○ ○ ○	ist gut in den Arbeitsfluss mit IntelliJ integriert
F5	erschwert die Orientierung durch uneinheitliche Gestaltung	○ ○ ○ ○ ○ ○ ○	erleichtert die Orientierung durch einheitlich Gestaltung
F6	ist durch ihre Gestaltung schlecht in das Designkonzept von IntelliJ integriert	○ ○ ○ ○ ○ ○ ○	ist durch ihre Gestaltung gut in das Designkonzept von IntelliJ integriert
F7	informiert in unzureichendem Maße über das, was sie gerade macht	○ ○ ○ ○ ○ ○ ○	informiert in ausreichendem Maße über das, was sie gerade macht
F8	erfordert, dass man sich viele Details merken muss	○ ○ ○ ○ ○ ○ ○	erfordert nicht, dass man sich viele Details merken muss
F9	ist schlecht ohne fremde Hilfe oder Handbuch erlernbar	○ ○ ○ ○ ○ ○ ○	ist gut ohne fremde Hilfe oder Handbuch erlernbar
F10	erzwingt unnötige Unterbrechungen der Arbeit	○ ○ ○ ○ ○ ○ ○	erzwingt keine unnötigen Unterbrechungen der Arbeit
F11	unterbricht durch Wartezeiten den erwünschten Arbeitsprozess in IntelliJ	○ ○ ○ ○ ○ ○ ○	unterbricht nicht durch Wartezeiten den erwünschten Arbeitsprozess in IntelliJ

	Die Software…	--- -- - -/+ + ++ +++	Die Software…
F12	liefert schlecht verständliche Fehlermeldungen	○ ○ ○ ○ ○ ○ ○	liefert gut verständliche Fehlermeldungen
F13	gibt keine konkreten Hinweise zur Fehlerbehebung	○ ○ ○ ○ ○ ○ ○	gibt konkrete Hinweise zur Fehlerbehebung
F14	erfordert bei Fehlern einen hohen Korrekturaufwand	○ ○ ○ ○ ○ ○ ○	erfordert bei Fehlern einen geringen Korrekturaufwand
F15	lässt sich schlecht an meine persönliche Art der Arbeitserledigung anpassen	○ ○ ○ ○ ○ ○ ○	lässt sich gut an meine persönliche Art der Arbeitserledigung anpassen
F16	erscheint mir bei der *parallelen Entwicklung* von nativen Implementationen von Mobilapplikationen nicht sinnvoll einsetzbar	○ ○ ○ ○ ○ ○ ○	erscheint mir bei der *parallelen Entwicklung* von nativen Implementationen von Mobilapplikationen sinnvoll einsetzbar
Gründe?			
F17	erscheint mir bei der *parallelen Wartung* von nativen Implementationen von Mobilapplikationen nicht sinnvoll einsetzbar	○ ○ ○ ○ ○ ○ ○	erscheint mir bei der *parallelen Wartung* von nativen Implementationen von Mobilapplikationen sinnvoll einsetzbar
Gründe?			

Raum für **Anmerkungen** und **Verbesserungsvorschläge** (Weiter auf Rückseite):

E. Auswertung der Evaluationsbögen

E.1. Quantitative Auswertung der Fragen

Gruppe A: Evaluationsteilnehmer mit fachlichem Vorwissen — 1 (sehr schlecht) - 7 (sehr gut)

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
P1	7	7	7	7	7	7	7	7	5	7	7	7	3	7	7	7	7
P2	7	6	7	7	7	7	7	5	3	7	6	7	3	7	7	4	7
P3	7	7	6	7	7	7	5	7	7	7	7	3		4	7	7	7
P4	6	6	6	5	7	7	6	7	6	7	7	7	7	6		6	7
Mittelwert	6,75	6,5	6,5	6,5	7	7	6,25	6,5	5,25	7	6,75	6	4,333333	6	7	6	7
Standardabw.	0,433013	0,5	0,5	0,866025	0	0	0,829156	0,866025	1,47902	0	0,433013	1,732051	1,885618	1,224745	0	1,224745	0

Gruppe B: Evaluationsteilnehmer ohne fachliches Vorwissen — 1 (sehr schlecht) - 7 (sehr gut)

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
P1	7	1	3	5	5	5	4	4	5	6	6					4	2
P2	7	7	6	6	7	4	7	6	6	6	7	7	7	7	7	7	6
P3	6	4	3	6	7	6	6	7	7	5	7	4	4	4	7	7	7
P4	6	7	7	7	4	6	5	7	7	7	7	4	4	4	4	6	
P5	7	7	6	7	7	6	3	4	4	5	7				6	6	6
P6	7	6	6	7	7	6	6	7	7	7	7				6	7	7
P7	6	7	4	6	7	7	7	7	6	7	7	4	4	4	4	6	7
P8	6	6	6		6		5	6	6	6	7				6	7	7
P9	7	2	6	7	1	6		1	1	2	1				2	7	7
P10	6	7	7	4	5	7	5	7	6	7	7				4	6	7
P11	7	7	7	6	7	6	6	6	7	6	6				6	6	6
Mittelwert	6,545455	5,545455	5,545455	6,1	5,727273	5,9	5,4	5,636364	5,636364	5,818182	6,272727	4,75	4,75	4,75	5,2	6,272727	6,2
Standardabw.	0,49793	2,104698	1,437399	0,943398	1,813631	0,830662	1,2	1,822722	1,720081	1,402477	1,710444	1,299038	1,299038	1,299038	1,536229	0,862439	1,469694

Gesamt

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
Mittelwert	6,647727	6,022727	6,022727	6,3	6,363636	6,45	5,825	6,068182	5,443182	6,409091	6,511364	5,375	4,541667	5,375	6,1	6,136364	6,6
Standardabw.	0,489898	1,869046	1,32665	0,939496	1,651935	0,860114	1,171516	1,667999	1,667999	1,309792	1,496663	1,653595	1,59079	1,408678	1,546135	0,979796	1,293626

E.2. Auswertung der Freitextfelder (Anwender mit Vorwissen)

Die folgenden Kommentare stammen wortwörtlich aus den Freitextfeldern der Evaluationsbögen entnommen. Anmerkungen des Verfassers sind durch eckige Klammern gekennzeICNHet. Abschnitte in runden Klammern können für besseren Lesefluss ausgelassen werden.

Freitext zu F16: Eignung für parallele Entwicklung

- Für parallele Entwicklung nur begrenzt einsetzbar. Eventuell zur Unterstützung der Wartung sinnvoll
- Ermöglicht das Auffinden von Swiftcode ohne Verständnis über Swift oder deren Ordnerstrukturen
- Nimmt dem Programmierer manuelles Suchen ab
- Hilft nur wenn das Programm schon in einer Version vorliegt, für eine initiale Entwicklung nicht geeignet

Freitext zu F17: Eignung für Wartung

- Traceability-Links automatisiert zu finden ist ungemein hilfreich bei der Einarbeitung in ein neues portiertes Projekt sowie bei der Bewertung der Ähnlichkeit zweier Implementationen. Auch das Finden von konzeptionell gleichen Code-Artefakten wird erleichtert
- Für mich besser geeignet für Wartung als für Entwicklung
- Featurepunkte können schnell identifiziert werden

Anmerkungen, Verbesserungsvorschläge und Bugs

- Es wäre wünschenswert die verlinkten Swift-Dateien im zugehörigen Projekt-Kontext anzeigen zu lassen
- Filtern nach syntaktischem Konstrukt im Pop-Up wäre eine schöne Ergänzung (3x)
- Komma-Zwang [als Trennzeichen] bei mehreren Suchbegriffen (2x)
- Java-Fehler werden weitergegeben [Nicht alle Exceptions gefangen]
- Namengebung [der Actions] kann an „Apple Style“ angepasst werden
- Anwender versteht die Bewertung [der Tracelinks] nicht -> evtl. in Prozent?
- Menünamen [der Actions] teilweise ungenau
- Result-Pop-Up verschwindet nicht durch Menüklick
- Feature: Traceability-Explorer-Window wäre toll zur Übersicht. Nutzer könnte Link-Favoriten auswählen/korrekte Links speichern
- Optionen verbergen beim Berechnen [Actions deaktivieren während geparsed wird]

- Suchfenster verschwindet nach Eingabe sofort, evtl. vorhalten
- Weitere Änderungen der Suche ermöglichen, Ergebnisse der veränderten Anfrage anpassen
- Spaltennamen im Pop-Up einfügen oder über Tooltip erläutern
- Vorausgewählte Suchbegriffe im Suchfenster anbieten z.B. basierend auf oft verwendeten Bezeichnern
- Bei der Suche über Suchbegriffe Permutationen der Begriffe verwenden
- Tutorial beim ersten Start des Plugins
- Fully Qualified Name eines Codeartefakts im Pop-Up anzeigen

E.3. Auswertung der Freitextfelder (Fachfremde Anwender)

Freitext zu F16: Eignung für parallele Entwicklung

- Schnelleres Finden von Pendants zwischen Codebases
- Vielleicht nicht immer, da nicht immer eine direkte Entsprechung vorhanden ist
- Gute Nutzerführung zum Finden von inhaltlich gleichem Code
- Spart Zeit
- Findet schnell Teile wo Funktion [Implementation] vielleicht sinnvoll integrierbar [ist]
- Findet fast immer die gesuchte Klasse/Interface im anderen Code

Freitext zu F17: Eignung für Wartung

- Schnelleres Finden von Pendants zwischen Codebases
- Vielleicht nicht immer, da nicht immer eine direkte Entsprechung vorhanden ist, allerdings hilft es sonst (über) keine Anpassung zu vergessen
- Paralleler Code (somit) schnell zu finden
- Spart Zeit
- Gerade das Finden von Klassen die in verschiedenen Sprachen [sind] erscheint mit kritisch
- Findet Features und sollte daher gut für die Wartung einsetzbar sein

Anmerkungen, Verbesserungsvorschläge und Bugs

- Beispiele zeigen keine Probleme und Schwächen [Anwendungsfall evtl. ungeeignet]

F. Gesprächsprotokoll: Präsentation bei ICNH

Präsentation des Traceability Recovery-Plugins und anschließende Diskussion über die Anwendbarkeit, den Nutzen und die Usability.

Zeit: 10:40 - 11:20, 03.07.17

Teilnehmer: Jörg Pechau und 3 ICNH-Entwickler (ICNH), Verfasser (GG)

GG Erläuterung der Funktionalität und Demonstration anhand des ICNH-Projektes „Mellow“.

GG Das Parsing des Projektes dauert relativ lange, da es recht umfangreich ist. Man profitiert natürlich von schnelleren Rechnern. [*Anmerkung: Der Parsingvorgang wurde aufgrund des Zeitaufwandes nicht während der Präsentation gezeigt sondern vorher ausgeführt.*]

ICNH Die Wartezeit ist natürlich ein Problem, vor allem für parallele Entwicklung wenn sich der Code oft ändert und neu geparsed werden müsste.

ICNH Eventuell sollte man schnellere Parser (z.B. IntelliJ PSI-Parser) in Erwägung ziehen oder die Analyse nicht auf dem Entwicklerrechner laufen lassen.

ICNH Frage: Welche Bezeichner werden in eine Suchanfrage einbezogen?

GG Editor Query: Der angeklickte Bezeichner und alle darin enthaltenen Klassen-, Methoden- und Attributsbezeichner.

GG Search Query: Alle eingegebenen Suchbegriffe um Whitespace am Anfang und Ende bereinigt.

ICNH Komma als erzwungenes Trennzeichen etwas unschön.

ICNH Allerdings kann nicht nach Whitespace getrennt werden, falls der Anwender in natürlicher Sprache nach Domainkonzepten sucht (z.B. „Bluetooth connection“).

- Möglicher Kontextverlust bei unklarer Trennung der Suchbegriffe.

ICNH Frage: Kennt die Software unterschiedliche Bezeichner für gleiche Konzepte in den beiden Sprachen Java und Swift? (z.B. Activity in Java und ViewController in Swift), Werden solche Unterschiede automatisch gemappt?

GG Nein, nicht automatisch. Allerdings ist auf Plugin-Seite die UI vorhanden gewünschte Mappings dieser Art anzulegen. *[Anmerkung: Die Beachtung der konfigurierten Mappings ist beim Parsing des Codes umzusetzen und nicht im Umfang dieser Studie enthalten.]*

ICNH Es wäre sinnvoll oft auftretende Mappings zu identifizieren und im Plugin mitzuliefern.

ICNH Im jetzigen Zustand findet die Software nur Tracelinks zwischen Codeartefakten die mit übereinstimmenden Domain-Bezeichnern benannt sind, sprachspezifische können oft nicht sinnvoll ausgewertet werden. (int zu Int klappt aufgrund von übereinstimmender Zeichenkette, bei Activity zu ViewController besteht keine Chance eine Übereinstimmung zu finden.)

- Die sprachspezifische Ebene sollte auf irgendeine Weise mehr gefördert werden.

GG Einwurf: Das Plugin unterstützt die Zuordnung von Bezeichnern zu Layers wodurch Artefakte auf gleichen Layers als ähnlicher bewertet werden als auf verschiedenen. Man könnte also Activity und ViewController zum Layer UI hinzufügen. Selbst wenn eine Artefaktverknüpfung nur über einen davor auftretenden Domain-Bezeichner (z.B. „LoginActivity“) gefunden wird, kann der Verknüpfung LoginActivity zu LoginViewController mehr Ähnlichkeit zugewiesen werden.

- *[Anmerkung: Auch diese Funktion ist als UI im Plugin enthalten. Die eigentliche Umsetzung findet im Parsing des Codes statt und ist nicht im Umfang dieser Studie enthalten.]*

ICNH Das könnte helfen aber auch hier ist viel kleinteilige Eigenarbeit nötig um die Zuordnung der Layer zu konfigurieren. Durch die ganze Konfiguration geht Zeit verloren die man durch die Nutzung des Tools eigentlich einsparen wollte.

GG Einwurf: Allerdings bleibt die Konfiguration in der Regel konstant und kann nach einmaliger Einrichtung wiederverwendet werden. Eine Import/Export-Funktion wäre denkbar.

GG Frage: Haben Sie Ideen wie man die Benutzbarkeit verbessern könnte? (z.B. weitere Zugriffsmöglichkeiten, Features?)

ICNH Nein, die gezeigten Actions und Intentions erscheinen sinnvoll und decken den Funktionsumfang ab. Sie sind schnell und unproblematisch verwendbar.

GG Frage: Wie würden sie die Anwendbarkeit des Tools für die parallele Entwicklung sowie die Wartung von Mobilanwendungen einschätzen?

ICNH Vermutlich besonders hilfreich um Überblick über unbekannten, bestehenden Code zu erlangen.

ICNH Unterstützung von paralleler Entwicklung nicht unbedingt realistisch, da Entwickler meist einen besseren Überblick über die Code-Basis haben und selten auf nachträgliche Analyse durch das Tool angewiesen sind.

ICNH Als Proof-of-Concept macht das Tool auf jeden Fall einen guten Eindruck, müsste aber sicherlich noch an den erwähnten Stellen verbessert werden um produktiv eingesetzt werden zu können.

ICNH Problematisch ist vor allem die sehr auf natürlicher Sprache basierende Analyse da technische Details der unterschiedlichen Plattformen nicht/wenig beachtet werden. Allerdings ist gerade das Erkennen über technische Unterschiede hinweg relevant.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Datum: Hamburg, den

Unterschrift: