Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Strukturangleichende Portierung von Software mit Traceability für die koordinierte plattformübergreifende Co-Evolution

An der Universität Hamburg eingereichte

Dissertation

zur Erlangung des Doktorgrades (Dr. rer. nat.)

An der
Fakultät für Mathematik, Informatik und Naturwissenschaften
der Universität Hamburg

vorgelegt von

Tilmann Stehle

Hamburg, 2019

Datum der Disputation: 4. März 2020

Gutachter: Prof. Dr.-Ing. Matthias Riebisch

Prof. Dr.-Ing. Patrick Mäder (JP)

Kurzfassung

Häufig wird Software für eine spezifische Plattform entwickelt, ehe der Bedarf erkannt wird, sie auch für Nutzer anderer Plattformen zur Verfügung zu stellen. Bestehende Ansätze wie die modellbasierte Softwareentwicklung und Cross-Platform-Frameworks sind für diese Situation nicht angemessen, da sie nur auf Projekte anwendbar sind, die die Entwicklung von Grund auf neu beginnen. Um eine bestehende, reife Software nachträglich auf einer zusätzlichen Plattform zur Verfügung zu stellen, müssen Entwickler diese Software portieren.

Durch die Portierung entsteht eine zusätzliche Codebasis für die neue Plattform, die zusätzlich zum ursprünglichen Quellcode weiterentwickelt werden muss. Bei Änderungen müssen Aufgaben wie Concept Location, Impact Analyse, Refactoring und Umsetzung der Änderung sowohl für die ursprüngliche Implementation als auch für die portierte Implementation der Software durchgeführt werden. Die Entwickler müssen diese Aufgaben somit doppelt verrichten und laufen Gefahr, dabei Inkonsistenzen zwischen den Implementationen zu verursachen.

Um die doppelt zu verrichtenden Aufgaben zu vereinfachen und zu koordinieren, entwickelt diese Arbeit eine Portierungsmethode, die die Entwürfe und Codestrukturen beider Implementationen bereits bei der Portierung vereinheitlicht. Darüber hinaus entwickelt sie Mechanismen, die plattformübergreifende Entsprechungen zwischen konkreten Code-Elementen wie Klassen und Methoden explizit in Trace Links erfassen. Auf dieser Basis schlägt die Arbeit Ansätze zur plattformübergreifenden Koordination und Vereinfachung der gemeinsamen Weiterentwicklung beider Implementationen vor.

Zur Evaluation wurde die Portierungsmethode in drei Fallstudien angewendet. Hohe Anteile der portierten Klassen entsprechen ihren Vorbildern in Bezug auf Zuständigkeit und Schnittstelle oder gar Anweisung für Anweisung. Die entwickelten Mechanismen zur Ermittlung plattformübergreifender Trace Links verknüpfen diese Entsprechungen mit hoher Präzision und Korrektheit. Die Ansätze zur plattformübergreifend koordinierten Weiterentwicklung wurden in prototypischen Werkzeugen implementiert. Sie werden anhand einer der Fallstudien in dieser Arbeit demonstriert.

Abstract

Software is often developed for a specific platform before developers notice the demand to also provide users of other platforms with its features. Exisiting approaches such as model-based development or cross-platform frameworks are not appropriate in this situation as they are only applicable to projects that develop a new software from scratch. To provide users of other platforms with the features of an existing, mature software, developers have to port it, thereby creating an additional codebase.

Evolutionary tasks such as concept location, impact analysis, refactoring and implementing change must be conducted on both the original and ported implementation, which leads to double work.

To simplify and coordinate these tasks, this thesis develops a porting method that unifies the design and code structures of both codebases. Furthermore, it develops mechanisms that capture explicit cross-platform trace links between equivalent code-elements such as classes and methods. Based on the unified implementations and trace links between corresponding elements, this thesis proposes approaches for coordinating and simplifying evolutionary tasks.

To evaluate the proposed porting method, it has been applied to three case studies, each of which is porting a software to a different platform. As a result, high portions of the ported type definitions are equivalent to their original pendants regarding their responsibilities and interfaces, or even match their original counterpart statement by statement. The developed mechanisms for establishing traceability models link these equivalent elements with high precision and recall. The approaches for coordinating the evolution of both codebases have been implemented in prototypical tools. They are exemplarily applied to one of the conducted case studies.

Danksagung

Allen voran danke ich Prof. Dr.-Ing. Matthias Riebisch. Er hat mir nicht nur den geistigen und zeitlichen Freiraum gegeben, diese Arbeit anzufertigen und ein ausgezeichnetes Arbeitsumfeld geschaffen, sondern sich stets mit seiner Erfahrung und mit organisatorischer Unterstützung für den Erfolg dieser Arbeit eingesetzt. Vielen Dank dafür, Matthias!

Prof. Dr.-Ing. Patrick Mäder danke ich ganz herzlich für die Bereitschaft, sich trotz seines vollen Terminkalenders mit dieser Arbeit zu befassen und mir wertvolle Hinweise für ihre Anfertigung zu geben.

Ich danke meinen Eltern, die mir beigebracht haben, im Vertrauen auf ihren Rückhalt auch schwierige Herausforderungen anzunehmen. Meiner Mutter, Doris Stehle, möchte ich auch für den Enthusiasmus danken, mit dem sie sich auf die für sie fremde Thematik dieser Arbeit eingelassen hat. Sie hat mir dadurch nicht nur als verständnisvolle Zuhörerin, sondern als wichtige Diskussionspartnerin zur Seite stehen können.

Ich möchte auch meiner Freundin Janina Wurpts danken. Sie hat in der Endphase dieser Arbeit häufig auf mich verzichten müssen, hat mich moralisch unterstützt und mir nicht nur ein Büro, sondern auch ein Zuhause in Oldenburg gegeben.

Auch Manuel Hendrix und Janina Beckermann danke ich für die Herberge, den Rückhalt und einen Arbeitsplatz in einer intensiven Phase dieser Arbeit. Manuel steht mir stets mit einem offenen Ohr, wenn angebracht mit einem guten Rat und wenn nötig mit einem Motivationsschub zur Seite. Danke dafür!

Auch meinen Kollegen an der Uni möchte ich für die großartige Arbeitsatmosphäre und die fruchtbaren Diskussionen danken. Mein besonderer Dank gilt Paula Rachow, die diese Arbeit intensiv gelesen und in der abschließenden Phase mit mir diskutiert hat. Paula ist ausgesprochen präzise und hat die Gabe, ehrliche Kritik und fröhliches Beisammensein wie selbstverständlich miteinander zu vereinen.

Jörg Pechau danke ich für den Einblick in die konkreten praktischen Herausforderungen der Portierung. Die Gespräche mit ihm über Portierungsprojekte, Architekturen und Konsistenz waren der Ausgangspunkt für diese Arbeit. Er hat mir darüber hinaus seinen Quellcode zur Durchführung einer Fallstudie gegeben und sich stets die Zeit für wertvolles Feedback genommen. Dafür danke ich ihm sehr!

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Um den Nutzerkreis einer Software erweitern zu können, muss diese in vielen Fällen für zusätzliche Endgeräte und Betriebssysteme verfügbar gemacht werden. Insbesondere im Bereich der Softwareentwicklung für mobile Endgeräte birgt dies nach wie vor große Herausforderungen und Forschungspotentiale [Dehlinger u. Dixon 2011], [Nagappan u. Shihab 2016], [Joorabchi 2016]. Wer neue Software von Grund auf entwickelt, kann Rahmenwerke wie Xamarin[1], Apache Cordova[2] oder Unity[3] einsetzen, mit denen ein gemeinsamer Quellcode zur Softwareentwicklung für mehrere Plattformen genutzt wird. Solche Rahmenwerke werden hier als Cross-Platform-Frameworks bezeichnet. Sie bieten insbesondere für die Evolution der Software den Vorteil, dass Änderungen nur einmal durchgeführt werden, sich aber auf alle adressierten Plattformen auswirken. Bei Neuentwicklungen ist der Einsatz solcher Frameworks eine gute Option. Nichtsdestotrotz gibt es auch gute Gründe dafür, sich gegen den Einsatz dieser Frameworks zu entscheiden. Abschnitt 3.1.2 beleuchtet die Gründe für diese Entscheidung näher.

Soll bestehende plattform-spezifisch entwickelte Software auf weiteren Plattformen genutzt werden, muss sie portiert werden. Entwickler entscheiden sich häufig dagegen, die bestehende Implementation durch eine zunächst unreife Re-Implementation auf Basis eines Cross-Platform-Frameworks zu ersetzen. Stattdessen fertigen sie eine zusätzliche plattform-spezifische Implementation für die Zielplattform an [Joorabchi u. a. 2013]. Abbildung 1.1 veranschaulicht dieses Vorgehen.

[1] `https://www.xamarin.com/`
[2] `https://cordova.apache.org/`
[3] `https://unity3d.com/de/unity/features/multiplatform`

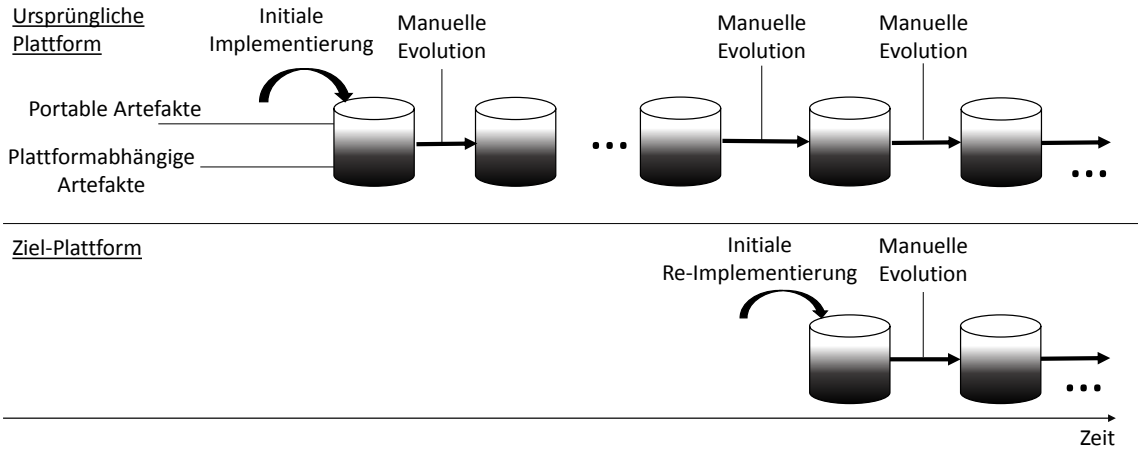

Abbildung 1.1: Gewöhnlicher Ansatz: Vollständige Re-Implementation und redundante Weiterentwicklung einer bestehenden Software

Der Quellcode der ursprünglichen Implementation besteht aus portablen und plattformspezifischen Anteilen, die vermischt in den Quelltextdokumenten vorkommen. Nach der initialen Entwicklung und einer Phase der manuellen Weiterentwicklung für den ursprünglichen Quellcode findet die initiale Entwicklung für die Zielplattform statt. Bei dieser Re-Implementierung besteht die Gefahr, Fehler und Mängel in die Zielimplementation einzuführen, sodass der Reifegrad der Ausgangsimplementation nicht übertragen wird.

Im Anschluss an die Portierung müssen Ausgangs- und Zielimplementation parallel weiterentwickelt werden. Dabei müssen die Entwickler die Konsistenz beider Implementationen bezüglich ihres Funktionsumfangs und ihres Verhaltens wahren. Somit verursachen Änderungen der Ausgangsimplementation entsprechende Änderungen der Zielimplementation und umgekehrt. Dieses Phänomen wird hier als plattform-übergreifende Co-Evolution bezeichnet.

Unterschiede im Entwurf und der Quelltext-Struktur der Implementationen führen dazu, dass Entwickler bei der Co-Evolution Aufgaben doppelt durchführen müssen. Insbesondere müssen sie für jede Implementation einzeln die Quelltextstellen identifizieren, die für die Änderung relevant sind. Sie müssen diese Quelltextstellen verstehen, die notwendigen Änderungen konzipieren und umsetzen. Die Notwendigkeit, diese Arbeiten mehrfach durchzuführen, behindert die effiziente und konsistente Wartung und Weiterentwicklung beider Implementationen.

Um diese Mehrfacharbeiten zu mindern, sollte der Entwurf und die Quelltextstruktur der Implementationen für die ursprüngliche Plattform und die Zielplattform bereits bei der Portierung aneinander angeglichen werden. So können die Entwickler ihr Verständnis der ursprünglichen Implementation auf die portierte Implementation übertragen [Mooney 1990]. Darüber hinaus fehlen in aktuellen Portierungsprojekten explizite

Verknüpfungen zwischen einander entsprechenden Elementen im Quellcode. Solche Verknüpfungen zeigen einem Entwickler auf, welche Elemente der ursprünglichen und portierten Implementation gleiche Funktionalitäten umsetzen und ggf. denselben Konzepten folgen. Dies können Entwickler nutzen, um Änderungen plattform-übergreifend zusammenzuführen, zu koordinieren und durch Wiederverwendung von Änderungskonzepten zu vereinfachen.

1.2 Grundlegende Vision der Arbeit

Diese Arbeit soll dazu beitragen, die bestehende Qualität einer Software bei ihrer Portierung auf die Implementation für die Zielplattform zu übertragen und die anschließende Co-Evolution der ursprünglichen und der portierten Implementation zu vereinfachen. In Abbildung 1.2 ist ein Ablauf der Portierung und plattform-übergreifender Co-Evolution zu sehen, der dieser Arbeit als Vision zukünftiger Portierungsprojekte dient.

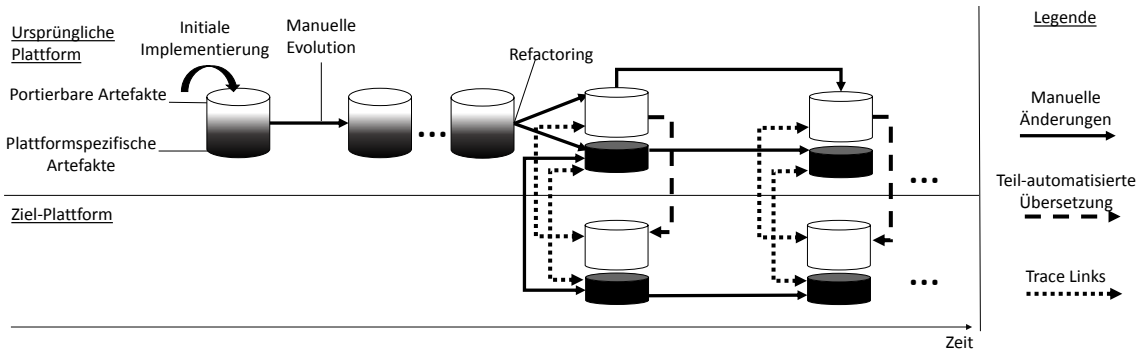

Abbildung 1.2: Vision: Synchronisierung der plattform-übergreifenden Co-Evolution durch Angleichung von Entwürfen und automatisierte Übertragung

Zunächst wird die Software initial implementiert und manuell evolviert. Nachdem entschieden wurde, dass die Software portiert werden soll, wird die ursprüngliche Implementation einer Refactoring-Phase unterzogen, die die Portabilität des Quelltextes systematisch erhöht. Sie trennt dazu portierbare Quellcode-Elemente von möglichst kleinen plattform-spezifischen Quellcode-Elementen. Portierbare Quellcode-Elemente sind in der Abbildung zu weißen Zylindern zusammengefasst, plattform-spezifische Elemente zu schwarzen Zylindern.

Im Anschluss an die Aufteilung des Quellcodes werden die portierbaren Elemente möglichst automatisiert in die Zielsprache übersetzt. Abbildung 1.2 stellt dies in Form gestrichelter Kanten dar. Ist kein passender Quellcode-Konverter verfügbar, so werden portierbare Elemente manuell in Entsprechungen übersetzt, die die Strukturen und Konzepte ihrer Vorbilder übernehmen. Ursprüngliche Code-Elemente und ihre Übersetzungen werden durch plattform-übergreifende Trace Links miteinander verknüpft.

In Abbildung 1.2 sind diese Trace Links als gepunktete Kanten eingezeichnet. Entwickler können anhand dieser Trace Links zwischen einander entsprechenden Elementen navigieren, plattform-übergreifende Analysen von Änderungsauswirkungen durchführen und Änderungen koordinieren oder gar automatisiert auf die Zielplattform übertragen.

Im Anschluss an die Portierung beginnt die koordinierte plattform-übergreifende Co-Evolution. In der dargestellten Vision müssen Änderungen nur noch für die ursprüngliche Implementation vollständig geplant und manuell durchgeführt werden. Durch die systematisch herbeigeführten konzeptuellen Ähnlichkeiten und durch die plattform-übergreifenden Trace Links können die Änderungen mit wenig Aufwand auf den Code für die Zielplattform übertragen werden. Änderungen an automatisch konvertierbaren Quellcode-Elementen können sogar vollautomatisiert in die Zielimplementation übertragen werden, indem der Quellcode-Konverter erneut auf sie angewendet wird. Durch die automatisierte Übertragung von Änderungen müssen weniger Aufgaben doppelt ausgeführt werden. Zudem wird dadurch die Konsistenz zwischen Ausgangs- und Zielimplementation automatisch gewahrt. Diese Konsistenz wird zusätzlich dadurch unterstützt, dass die plattform-übergreifenden Trace Links die ursprünglichen Code-Elemente mit ihren Übersetzungen verknüpfen. Entwickler können die Trace Links werkzeuggestützt nutzen, um zwischen den Entsprechungen zu navigieren und um konsistente Änderungen plattform-übergreifend zu planen und zu koordinieren.

1.3 Herausforderungen bei der strukturangleichenden Portierung und plattform-übergreifenden Co-Evolution

Die oben beschriebene Vision basiert darauf, dass die Entwickler die Strukturen der Ausgangsimplementation weitgehend auf die Zielimplementation übertragen. Beispielsweise sollten beide Implementationen denselben Entwurf auf Typebene aufweisen. Die definierten Typen sollten sich, soweit möglich, in Bezug auf ihre Schnittstellen, implementierte Algorithmen und Abhängigkeiten zu den APIs der Plattform entsprechen. Allerdings wird diese Angleichung der Strukturen von Ausgangs- und Zielimplementation durch verschiedene Faktoren erschwert.

Abhängigkeiten zu plattform-spezifischen APIs
Eine der größten Hürden bei der strukturangleichenden Portierung ist, dass sich die APIs beider Plattformen in Bezug auf die angebotenen Typen und deren Schnittstellen unterscheiden [Tanaka u. a. 1995] [Alves u. a. 2005]. Das betrifft zum einen die grundlegenden Basisdatentypen und Sammlungen, die von der Laufzeitumgebung der Plattform bereitgestellt werden. Zum anderen unterscheiden sich die Typen, die die

Funktionalitäten des Betriebssystems exponieren sowie Typen, die von Programmbibliotheken von Drittanbietern bereitgestellt werden. Wird Code portiert, der von solchen plattform-spezifischen Typen abhängt, so unterscheiden sich die Abhängigkeiten des ursprünglichen Codes von den Abhängigkeiten des portierten Codes. Diese Unterschiede erschweren die Überführung der ursprünglichen Strukturen in die Zielimplementation.

Auch die Anwendung automatischer Quellcode-Konvertoren wird durch diese Unterschiede erschwert, da zusätzliche Abbildungsvorschriften definiert werden müssen, die die ursprünglichen Abhängigkeiten automatisch auf ihre Entsprechung in der Zielplattform abbilden. Die Anpassung des Codes an die Schnittstellen der Zielplattform ist daher ein wichtiger Kostentreiber in Portierungsprojekten [Broeksema 2010, S. 3].

Unterschiede zwischen den eingesetzten Programmiersprachen
Eine weitere Hürde für die Übertragung von Strukturen in die portierte Implementation sind Unterschiede zwischen der ursprünglich eingesetzten Programmiersprache und der Zielsprache. Nicht alle Sprachkonstrukte, die in der ursprünglichen Implementation genutzt werden, sind auch Teil der Zielsprache. Diese müssen dann entweder in der ursprünglichen Implementation ersetzt oder in der Zielimplementation durch andere Konstrukte simuliert werden [Terekhov u. Verhoef 2000]. Beispielsweise bietet Java das Konstrukt der anonymen inneren Klassen. Diese ermöglichen, ein Interface in einer unbenannten Klasse zu implementieren und den Konstruktor dieser Klasse im selben Ausdruck aufzurufen[4]. Dieses Konstrukt ist zum Beispiel in der Programmiersprache C# nicht enthalten. Entwickler, die eine anonyme innere Klasse nach C# übersetzen wollen, müssen das Konstrukt durch eine benannte Klasse simulieren, die gesondert vom Konstruktoraufruf implementiert wird. Alternativ kann diese Auflösung von Klasse und Konstruktoraufruf bereits im ursprünglichen Java-Quelltext vorgenommen werden, sodass der zu übersetzende Code ausschließlich Sprachkonstrukte einsetzt, die auch in C# verfügbar sind.

Einschränkungen von Quellcode-Konvertoren
Für viele Kombinationen von ursprünglicher Programmiersprache und Zielsprache sind automatische Quellcode-Konvertoren verfügbar. Durch ihren Einsatz werden die Strukturen des ursprünglichen Codes in die Zielimplementation übertragen. Sie können allerdings häufig nur auf Teile des Quellcodes angewendet werden, weil sie nicht alle Sprachkonstrukte der ursprünglichen Programmiersprache korrekt übersetzen. Ist der zu übersetzende Code abhängig von Schnittstellen zur Ausgangsplattform, so müssen außerdem formale Abbildungsregeln konfiguriert werden, die die ursprünglichen Abhängigkeiten auf Abhängigkeiten zur Zielplattform abbilden. Nicht alle Quellcode-Konvertoren erlauben die Konfiguration solcher Abbildungsregeln.

[4] `https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html`

Durch diese Einschränkungen muss der konvertierte Quellcode selbst beim Einsatz ausgereifter Konvertoren durch manuelle Anpassungen vervollständigt werden[5][6] [Terekhov 2001]. Im Kontext von Portierungsprojekten sind diese manuellen Änderungen problematisch. Wird der ursprüngliche Code weiterentwickelt und erneut konvertiert, so werden sie durch die erneute Konversion überschrieben. Gerade bei einer langfristigen gemeinsamen Weiterentwicklung des ursprünglichen und portierten Codes ist der Einsatz von Konvertoren zur Übertragung von Änderungen aber wünschenswert.

Beschränkung von Änderungen am ursprünglichen Quellcode
Um den ursprünglichen Code für die automatische Übersetzung durch einen Konverter vorzubereiten, kann es erforderlich sein, ihn zu ändern. Beispielsweise kann es sinnvoll sein, die Schnittstellen der Ausgangsplattform an die Schnittstellen der Zielplattform anzupassen. Dies ist gelegentlich einfacher, als umgekehrt die Schnittstellen der Zielplattform anzupassen. Verschiedene Faktoren verhindern Änderungen am ursprünglichen Code allerdings. In manchen Fällen haben die portierenden Entwickler gar keinen schreibenden Zugriff auf die ursprüngliche Codebasis. Damit sind Änderungen ausgeschlossen. Die Entwickler der ursprünglichen Implementation sind zudem an die bestehenden Strukturen gewöhnt. Sie zu ändern verursacht zusätzlichen Aufwand für die erneute Einarbeitung in die veränderten Strukturen. Änderungen der ursprünglichen Codebasis sind außerdem mit dem Risiko verbunden, neue Fehler einzuführen. Hook [2005, S. 42] empfiehlt dementsprechend, die ursprüngliche Codebasis in Portierungsprojekten möglichst unverändert zu lassen. Soll eine Programmbibliothek portiert werden, die außerhalb des Portierungsprojekts bereits eingesetzt wird, so müssen die portierenden Entwickler sicherstellen, dass sie die öffentliche Schnittstelle der Bibliothek nicht ändern. Solche Änderungen können Fehler in extern entwickeltem Code verursachen, der die ursprüngliche Schnittstelle der Bibliothek nutzt.

Bedarf plattform-typischer Schnittstellen für externen Code
Ein weiteres Hemmnis für die Angleichung der Strukturen in der ursprünglichen und portierten Implementation besteht vor allem bei der Portierung von Programmbibliotheken. Diese werden portiert, um die Funktionalität der Bibliothek auf der Zielplattform für Entwickler bereitzustellen, die nicht Teil des Portierungsprojekts sind. Ein Beispiel dafür ist das Projekt Lucene.Net, das das Suchframework Lucene[7] auf die .Net-Plattform portiert. Die intendierten Nutzer der portierten Bibliothek sind an die Zielplattform mit ihren Typen, Schnittstellen und den Besonderheiten ihrer Programmiersprache gewöhnt. Dementsprechend sollte der portierte Quellcode möglichst *plattform-typische* Schnittstellen anbieten, mit denen externer Code die Funktionalitäten der Bibliothek nutzen kann. Plattform-typische Schnittstellen zeichnen sich dadurch aus, dass sie die nativen Typen der Zielplattform als Parameter- und Rückgabetypen verwenden und dass die typischen

[5] `https://github.com/apache/lucenenet/blob/master/CONTRIBUTING.md`
[6] `https://github.com/mono/ngit/issues/72`
[7] `http://lucenenet.apache.org/`

Idiome der Zielsprache eingesetzt werden können. Beispiele dafür sind `out`-Parameter in C#, oder Lambda-Ausdrücke als Parameter in Java. Die Forderung nach plattformtypischen Schnittstellen steht offensichtlich in Konflikt mit dem Ziel, den portierten Code möglichst ähnlich zum ursprünglichen Code zu entwickeln.

Fehlen von Traceability zwischen ursprünglichem und portiertem Code
Es fehlen explizite Verknüpfungen von Elementen des ursprünglichen Codes mit ihren Übersetzungen in der portierten Implementation. Solche Verknüpfungen sind notwendig, um die Evolution beider Implementationen teilautomatisiert zu vereinheitlichen und zu koordinieren. Es gibt bislang allerdings weder Werkzeuge für die Erkennung von Übersetzungsbeziehungen zwischen Code-Elementen, noch für ihre Nutzung während der Co-Evolution.

Mangel an Konzepten und Werkzeugen für plattform-übergreifende Co-Evolution
Um evolutionäre Änderungen vorzunehmen, müssen die Entwickler den zu ändernden Code identifizieren, die Auswirkung der Änderungen abschätzen, die Änderung planen, den Code ggf. restrukturieren, und die Änderung umsetzen. Bei portierter Software fallen diese Aufgaben sowohl für die ursprüngliche als auch für die portierte Implementation an. Es fehlen Konzepte und Werkzeuge, um diese Aufgaben plattformübergreifend zu koordinieren und durchzuführen. In der Folge werden Änderungen manuell koordiniert, doppelt geplant und doppelt durchgeführt. Refactorings werden mitunter sogar unterlassen, um die Konsistenz beider Implementationen nicht zu gefährden. Ein Beispiel dafür findet sich im Portierungsprojekt *Google Libphonenumber*[8].

1.4 Zielstellung der Arbeit

Diese Arbeit soll portierende Entwickler dabei anleiten, die oben genannten Hürden zu überwinden. Dazu wird eine Portierungsmethode entwickelt, nach der die Entwickler eine bestehende Software unter Einsatz unvollständiger Quellcode-Konvertoren schrittweise auf die Zielplattform übertragen können. Sie führt dabei systematisch Entsprechungen zwischen den Konzepten und Strukturen der Ausgangsimplementation und der Zielimplementation ein. Dazu wird eine Richtlinie aufgestellt, die angemessene Entwurfsmuster identifiziert, um die Schnittstellen der Zielplattform an die ursprünglich genutzten Schnittstellen anzugleichen.

Die Arbeit definiert Mechanismen zur Einführung plattform-übergreifender Trace Links, die die Quellcode-Elemente der Ausgangsimplementation mit ihren Entsprechungen in der Zielimplementation verknüpfen.

[8] `https://groups.google.com/forum/#!searchin/libphonenumber-discuss/port/libphonenumber-discuss/WoG8ojGSggk/IGNzC86v8yQJ`

Darauf aufbauend entwickelt sie Konzepte, die diese plattform-übergreifenden Ähnlichkeiten und Trace Links nutzen, um Aufgaben plattform-übergreifend zu koordinieren und zusammenzuführen. Dazu werden teil-automatische Abläufe entworfen, die die Arbeitsergebnisse übertragen oder die Durchführung der Aufgaben synchronisieren. So sollen doppelt auszuführende Arbeiten reduziert und das Risiko gemindert werden, dass Entwickler während der Evolution Inkonsistenzen zwischen ursprünglicher und portierter Implementation verursachen.

1.5 Annahmen und Einsatzbereich

Grundsätzlich kann der Bedarf bestehen, Software jeglicher Art zu portieren. Um die Richtlinien der Portierungsmethode möglichst konkret und anwendbar formulieren zu können, wird ihr Einsatzkontext wie folgt beschränkt:

Die Portierungsmethode adressiert Portierungsvorhaben, die objektorientierten Quellcode in eine ebenfalls objektorientierte Zielsprache überführen. Nur dann können die diskutierten Entwurfsmuster eingesetzt werden. Gibt es Unterschiede bezüglich des Programmierparadigmas der ursprünglichen Programmiersprache und der Zielsprache, so sollten Entwickler dem Paradigma der Programmiersprache folgen, um wartbaren Code zu produzieren. Die hier entwickelte Portierungsmethode liefert dazu keine Anleitung.

Die vorgestellte Portierungsmethode überträgt Strukturen der ursprünglichen Implementation auf die portierte Implementation. Ihr Einsatz ist somit vor allem dann angebracht, wenn diese Strukturen eine hohe Wartbarkeit aufweisen und es somit wünschenswert ist, sie zu übertragen. Ist dies nicht der Fall, so ist eine Restrukturierung der ursprünglichen Implementation vor der Portierung zu erwägen. Alternativ können die Entwickler insbesondere bei unreifer Software in Erwägung ziehen, die bestehende Implementation durch eine Neuentwicklung mittels Cross-Platform-Framework zu ersetzen. Gründe, die für eine Portierung anstelle einer solchen Neuentwicklung sprechen, werden in Abschnitt 3.1.2 dargelegt.

1.6 Methodik der Arbeit

Die vorliegende Arbeit ist in das Gebiet des Design Science einzuordnen. Darunter werden Forschungsarbeiten gezählt, „die IT-Artefakte entwickeln und evaluieren, welche ein organisatorisches Problem lösen sollen“[Hevner u. a. 2008, S. 77 (Übersetzung des Autors)]. Ein *Artefakt* kann in diesem Kontext jede Art von Vorschrift sein, die IT-Wissenschaftlern oder -Praktikern ermöglicht, ein Problem bei der Entwicklung und Realisierung von Informationssystemen in Organisationen zu verstehen und zu lösen.

Die Portierung und plattform-übergreifenden Co-Evolution objektorientierter Software stellt ein solches Problem dar. Bei der Entwicklung der Portierungsmethode, die dieses Problem adressiert, folgt die vorliegende Arbeit der Design Science Research Methodology (DSRM) nach Peffers u. a. [2007]. Die DSRM gibt den iterativen Forschungsprozess für Design Science vor, der in Abbildung 1.3 dargestellt ist.

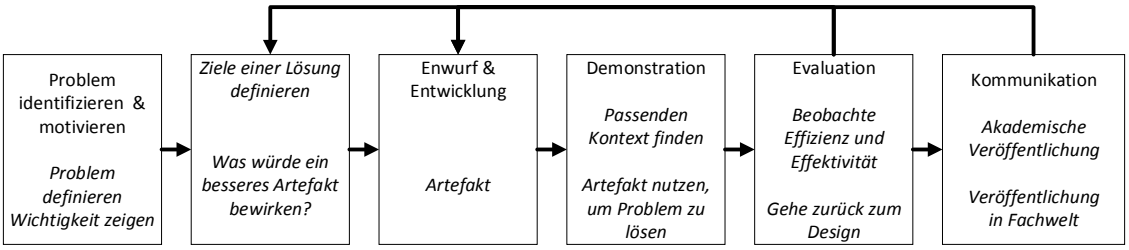

Abbildung 1.3: DSRM Process Model, übersetzt aus [Peffers u. a. 2007]

Im ersten Schritt ist das zu behandelnde Problem zu definieren und seine Relevanz ist herauszustellen. Dazu wurde in Abschnitt 1.1 dieser Arbeit bereits beleuchtet, vor welchen Problemen Entwickler stehen, die Software auf eine neue Plattform portieren. Insbesondere ist hier die Herausforderung zu nennen, dass die ursprüngliche und die portierte Implementation im Anschluss an die Portierung langfristig konsistent zueinander weiterentwickelt werden sollen. In Kapitel 3 werden bestehende Ansätze bezüglich ihres Beitrags zu den genannten Herausforderungen diskutiert und offene Forschungslücken werden herausgestellt.

Im zweiten Schritt ist das Ziel zu definieren, welches das zu entwerfende Artefakt konkret erfüllen soll. Eine solche Zielstellung findet sich in Kapitel 1.4 dieser Arbeit. Ziel der Portierungsmethode ist die Angleichung der Konzepte und Strukturen in den Implementationen für beide Plattformen sowie das Einführen expliziter plattform-übergreifender Verknüpfungen zwischen einander entsprechenden Quellcode-Elementen wie Klassen und Methoden. Auf Basis dieser Verknüpfungen sollen Konzepte definiert werden, die die plattform-übergreifende Co-Evolution koordinieren und vereinfachen.

Im dritten Schritt der Design Science Research Methodology wird das Artefakt entworfen und entwickelt. Im Fall der vorliegenden Arbeit wurde eine Portierungsmethode entwickelt, die die portierenden Entwickler bei der systematischen Angleichung der Strukturen von Ausgangs- und Zielimplementation anleitet. Darüber hinaus wurden Mechanismen entworfen und implementiert, die explizite Verknüpfungen zwischen den ursprünglichen Code-Elementen und ihren portierten Entsprechungen erzeugen. Basierend auf den plattformübergreifenden Entsprechungen und den expliziten Verknüpfungen wurden Konzepte zur Zusammenführung und Koordinierung der Evolution auf beiden Plattformen entwickelt.

Viertens ist der Einsatz des entwickelten Artefakts darzustellen. Dabei ist fünftens die Effizienz und Effektivität des Artefakts zu messen. Dazu werden der Einsatz der entwickelten Portierungsmethode in drei Fallstudien sowie die Anwendung der entwickelten Werkzeuge zur Ermittlung und Nutzung plattform-übergreifender Trace Links in Kapitel 8 beschrieben und ausgewertet.

Abschließend ist die durchgeführte Forschung inklusive des behandelten Problems, des Artefakts und seiner Evaluation in wissenschaftlichen Kreisen und in der Fachwelt zu veröffentlichen. Neben dem vorliegenden Dokument wurden die Bestandteile der Portierungsmethode in akademischen Workshops[Stehle u. Riebisch 2015] [Stehle u. Riebisch 2017], Konferenzen [Stehle u. Riebisch 2018a] [Stehle u. Riebisch 2018b] und in einem Journal [Stehle u. Riebisch 2019] veröffentlicht. Darüber hinaus wurden die Portierungsmethode und die entwickelten Werkzeuge mit der Firma ICNH[9] diskutiert, die regelmäßig Portierungen vornimmt. Die Ergebnisse wurden außerdem in der Fachgruppe *Mobile* des Wirtschaftsnetzwerks Digitale Wirtschaft Schleswig-Holstein (DiWiSH) vorgestellt[10].

Der Prozess der Design Science Research Methodology definiert Zyklen, die mehrfach durchlaufen werden können. So können im Anschluss an die Evaluation oder Kommunikation sowohl die Ziele als auch das entwickelte Artefakt verändert werden. Auch die vorliegende Arbeit hat diese Schleifen durchlaufen. Eine wesentliche Entwicklung, die nicht von vorn herein Teil der Portierungsmethode war, ist ein Mechanismus zur nachträglichen Ermittlung von Trace Links. Im ersten Durchlauf des Forschungsprozesses war geplant, sämtliche Trace Links ausschließlich bei der automatischen Übersetzung von Quellcode mit Quellcode-Konvertoren zu erheben. Im Rahmen eines studentischen Projekts wurde allerdings deutlich, dass nicht immer ein passender, erweiterbarer Konverter verfügbar ist und dass manuelle Anpassungen an konvertiertem Code nicht immer vermeidbar sind. Für manuell übersetzte oder angepasste Code-Elemente ist der ursprünglich vorgesehene Mechanismus zur Aufzeichnung von Trace Links nicht anwendbar. Dementsprechend wurde die Methode um den Mechanismus zur nachträglichen Ermittlung von Trace Links erweitert, der in Kapitel 6 vorgestellt wird.

1.7 Aufbau der Arbeit

Der Rest dieser Arbeit ist in acht Kapitel unterteilt. Das unmittelbar folgende Kapitel 2 führt grundlegende Konzepte und Begriffe ein, die für das weitere Verständnis der Arbeit notwendig sind. Daran anschließend werden in Kapitel 3 verwandte Arbeiten diskutiert und ihr Beitrag zu den oben gesetzten Zielen wird bewertet. In Kapitel 4 werden zunächst

[9] http://www.icnh.de/

[10] https://www.diwish.de/fachgruppen-termin/diwish-fachgruppe-mobile-the-never-ending-story-cross-platform-development-part-1.html

grundlegende Entscheidungen beleuchtet, die sich auf das Vorgehen bei der Portierung auswirken, ehe der Ablauf der Portierungsmethode geschildert wird. Teil dieses Ablaufs ist der Einsatz von Entwurfsmustern, die die Schnittstellen der ursprünglich genutzten APIs in der Zielplattform nachempfinden und plattform-spezifischen Code isolieren. Diese Entwurfsmuster werden in Kapitel 5 bezüglich ihrer Auswirkungen auf Portierungsprojekte verglichen. Aus diesem Vergleich wird eine Richtlinie abgeleitet, die portierende Entwickler bei der Wahl eines angemessenen Entwurfsmusters anleitet. Kapitel 6 führt einen Mechanismus ein, der plattform-übergreifende Trace Links zwischen den Code-Elementen der Ausgangsimplementation und ihren Entsprechungen in der Zielimplementation ermittelt. Auf Basis der erzielten Entsprechungen und der eingeführten Trace Links entwickelt Kapitel 7 Ansätze zur Vereinfachung und Koordinierung der plattform-übergreifenden Co-Evolution. Die entwickelte Portierungsmethode inklusive der genannten Richtlinien, Mechanismen und Konzepte zur Co-Evolution wird in Kapitel 8 evaluiert. Die Arbeit schließt in Kapitel 9 mit einem zusammenfassendem Fazit und einem Ausblick auf weiterführende Arbeiten ab.

2 Grundlegende Begriffe und Konzepte

2.1 Plattform

Der Begriff *Plattform* wird in verschiedenen Kontexten sehr unterschiedlich gebraucht. Im Kontext von Software-Produktlinien ist die Plattform Teil der eigenen Entwicklung. Der Begriff bezeichnet dort die „gemeinsame Basis aller individuellen Produkte einer Produktfamilie“[Halman u. a. 2006, S. 29 (Übersetzung des Autors)]. Eine ganz andere Definition gilt bei der Konstruktion von Rechnerarchitekturen. Dort ist eine Plattform eine Abstraktion, die einen Satz von Regeln in Bezug auf die Hardware-Architektur definiert [Sangiovanni-Vincentelli u. Martin 2001, S. 26]. Eine für diese Arbeit angemessenere Definition findet sich im Kontext der modellgetriebenen Softwareentwicklung, deren Modelle häufig von Plattformen abstrahieren. In diesem Kontext ist eine Plattform eine *Umgebung zur Ausführung von Softwaresystemen wie die Java Plattform* [Lodderstedt u. a. 2002, S. 6 (Übersetzung des Autors)]. Ali u. a. [2001] definieren konkreter, was die Bestandteile einer Plattform sind: „A platform is a combination of device, operating system and toolkit. An example of a platform is a PC running Windows 2000 on which applications use the Java Swing toolkit.“

In dieser Arbeit wird der Plattform-Begriff zusammenfassend wie folgt definiert:

Definition 2.1.1: Plattform

Eine Plattform ist eine Umgebung zur Ausführung von Software. Sie definiert eine Gruppe von Endgeräten sowie ein Betriebssystem und Laufzeitumgebungen, in denen Software ausgeführt werden kann.

Durch die verfügbaren Laufzeitumgebungen grenzt die Plattform auch die Programmiersprachen ein, die zur Entwicklung von Software für die jeweilige Plattform eingesetzt werden können. Die Plattform definiert Schnittstellen zur Programmierung von Anwendungen, sogenannte Application Programming Interfaces oder auch APIs. Diese APIs sind in drei Kategorien einzuordnen. Erstens stehen APIs für den Zugriff auf die Funktionalitäten des Betriebssystems zur Verfügung. Zweitens bietet jede Laufzeitumgebung eine API, die Basisdatentypen, Sammlungen und weitere grundlegende Funktionalitäten

bereitstellt und gegebenenfalls vom direkten Zugriff auf die Betriebssystem-API abstrahiert. Darüber hinaus werden Programmbibliotheken von Drittanbietern für die Plattform bereitgestellt, die der Plattform spezifische Funktionalitäten hinzufügen.

Quellcode, der unmittelbar auf den oben genannten APIs der Plattform basiert, wird in dieser Arbeit als *nativ* bezeichnet. Das grenzt ihn insbesondere von Quellcode ab, der Plattform-Abstraktionen nutzt, um auf verschiedenen Plattformen ausführbar zu sein. Beispielsweise wird eine auf dem Android SDK basierende Implementation einer App als nativ bezeichnet, nicht aber eine entsprechende Implementation auf Basis des Cross-Platform-Frameworks Xamarin[1].

2.2 Portieren und Portabilität

Für die Tätigkeit des *Portierens* wird folgende Definition nach Mooney [1997] zugrunde gelegt:

Definition 2.2.1: Portieren

„Portieren ist die Tätigkeit, eine lauffähige Version einer Software-Einheit oder eines Software-Systems für eine neue Umgebung auf Basis einer bestehenden Version zu erschaffen“[Mooney 1997, S. 2 (Übersetzung des Autors)].

Im Kontext dieser Arbeit ist stets die Portierung einer Software von der ursprünglichen Plattform auf eine zusätzliche Plattform gemeint. Diese ursprüngliche Plattform wird hier auch als *Ausgangsplattform* bezeichnet und die zusätzliche Plattform als *Zielplattform*. Analog werden die Begriffe *Ausgangssprache* und *Zielsprache* für die zugehörigen Programmiersprachen verwendet. Die ursprüngliche und die portierte Implementation werden entsprechend *Ausgangs-* und *Zielimplementation* genannt.

Der allgemeine Begriff der *Portabilität* bezeichnet den Grad der Effektivität und Effizienz, mit der eine Software von einer Plattform in eine andere überführt werden kann [in Anlehnung an International Organization for Standardization 2011].

Mooney [1990] definiert darüber hinaus den Begriff der *Erfahrungsportabilität* (englisch: *Experience Portability*), die bei der Portierung anzustreben ist: „If the experience gained by [...] users and programmers in one environment can be reused in a new environment, then experience portability has been achieved“[Mooney 1990, S. 61]. Im Kontext dieser Arbeit steht die Perspektive der portierenden Entwickler im Vordergrund, sodass folgende Definition für den Begriff der Erfahrungsportabilität aufgestellt wird:

[1] `https://docs.microsoft.com/de-de/xamarin/`

Definition 2.2.2: Erfahrungsportabilität

Erfahrungsportabilität ist der Grad, zu dem Entwickler ihre Erfahrungen aus der Entwicklung der ursprünglichen Implementation bei der (Weiter-)Entwicklung der Zielimplementation wiederverwenden können.

Diese Erfahrungsportabilität wird vor allem dadurch erzielt, dass die Zielimplementation die Konzepte und Strukturen des ursprünglichen Quellcodes übernimmt. Je nachdem, ob diese Konzepte und Strukturen in der Zielimplementation geschaffen werden können, wird ein Quellcode-Element hier als *portierbar* bezeichnet.

Definition 2.2.3: *Portierbarer* Quellcode und *plattform-spezifischer* Quellcode

In dieser Arbeit wird ein Code-Element als *portierbar* bezeichnet, wenn es durch Konversion oder Re-Implementierung in eine Übersetzung überführt werden kann, die ihrem Vorbild strukturell und konzeptuell entspricht.

Im Gegensatz dazu wird in dieser Arbeit ein Code-Element als *plattform-spezifisch* bezeichnet, wenn keine solche Entsprechung erstellt werden kann.

Quellcode-Elemente können in Bezug auf verschiedene Aspekte portierbar sein. Beispielsweise kann eine Klasse Anweisung für Anweisung in entsprechende Übersetzungen überführt werden. Alternativ kann sich die Portierbarkeit auf ihre Schnittstelle oder ihren Zuständigkeit beschränken.

2.3 Quellcode-Elemente und Ähnlichkeiten zwischen ihnen

Definition 2.3.1: Quellcode-Element (kurz: Code-Element)

Der Begriff *Quellcode-Element* bezeichnet in dieser Arbeit ein Exemplar eines programmiersprachlichen Konstrukts im Quelltext. Beispiele sind eine konkrete Methodendefinition, eine Klassendefinition, eine Variablendeklaration, ein Methodenaufruf oder ein Kommentar.

Ziel dieser Arbeit ist unter anderem, dass die Quellcode-Elemente der Zielimplementation ihren Vorbildern in der Ausgangsimplementation konzeptuell möglichst ähnlich sind. In diesem Kontext sind die Begriffe *strukturelle Ähnlichkeit* und *konzeptuelle Ähnlichkeit* zu konkretisieren.

Im Kontext dieser Arbeit wird vor allem der Fall betrachtet, dass Ausgangs- und Zielimplementation unterschiedliche Programmiersprachen einsetzen. Die strukturelle Ähnlichkeit kann somit nicht an einem unmittelbaren Vergleich der abstrakten Syntaxbäume ausgemacht werden. Stattdessen können sogenannte Quellcode-Konzeptgraphen miteinander vergleichen werden, die von der Syntax der zugrundeliegenden Programmiersprache abstrahieren [Mishne u. De Rijke 2004]. Sie erfassen die programmiersprachlichen Konstrukte wie Klassendefinitionen, Methoden, Attribute und lokale Variablen im Quellcode und setzen sie zueinander in Beziehung. Codebeispiel 2.1 zeigt den Code des simplen Interfaces `Fahrzeug`, welches lediglich zwei Operationen definiert. Die erste ruft die Farbe des Fahrzeugs ab, die zweite setzt die Farbe des Fahrzeugs.

```
public interface Fahrzeug
{
        public String gibFarbe();
        public void setzeFarbe(String neueFarbe);
}
```

Codebeispiel 2.1: Codebeispiel zur Konstruktion eines Quellcode-Konzeptgraphen

Der zugehörige Quellcode-Konzeptgraph ist in Abbildung 2.1 zu sehen. Er erfasst das definierte Interface und die darin enthaltenen Operationen sowie deren Parameter und Rückgabetypen.

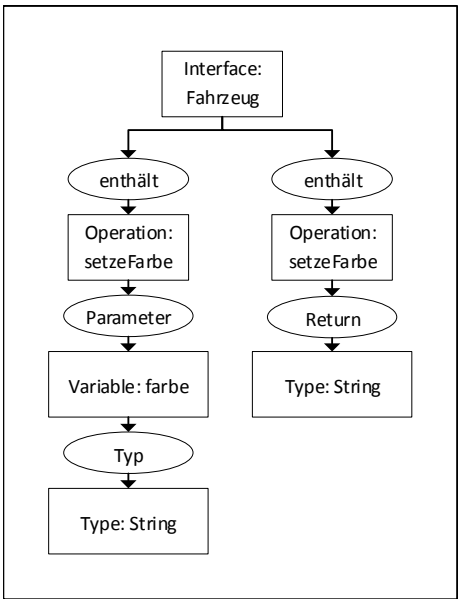

Abbildung 2.1: Quellcode-Konzeptgraph zu Codebeispiel 2.1

Wird in dieser Arbeit von struktureller Ähnlichkeit gesprochen, so ist stets die Ähnlichkeit der Quellcode-Konzeptgraphen zueinander gemeint. Um diese Ähnlichkeit zu quantifizieren, können diverse Ähnlichkeitsmetriken aus der Graphentheorie genutzt werden [Zager u. Verghese 2008].

Im Gegensatz dazu gelten zwei Code-Elemente in dieser Arbeit als konzeptuell ähnlich, wenn sie denselben *Lösungsentwurf* implementieren. Dazu zählt, dass sie zu einem gewissen Grad dieselben Algorithmen und Datenstrukturen verwenden, ihre Verantwortlichkeiten in gleicher Weise auf untergeordnete Code-Elemente aufteilen und dieselbe Terminologie bei der Wahl von Bezeichnern verwenden. Die konzeptuelle Ähnlichkeit manifestiert sich damit auch in einer strukturellen Ähnlichkeit, geht jedoch darüber hinaus. Sie umfasst auch Ähnlichkeiten zwischen den konzeptionellen Ideen, die die Entwickler entwickelt haben. Im Kontext dieser Arbeit wird vor allem die konzeptuelle Ähnlichkeit zwischen Typdefinitionen wie Klassen und Interfaces betrachtet, die die Basis eines objektorientierten Entwurfs bilden und damit grundlegend für dessen Verständnis sind [Moreno u. a. 2013]. Die konzeptuelle Ähnlichkeit wird dementsprechend auf Basis dreier grundlegender Aspekte objektorientierter Typdefinitionen betrachtet. Erstens können zwei Typdefinitionen sich in Bezug auf ihre *Zuständigkeit* entsprechen, also dieselbe Dienstleistung anbieten. Zweitens können sich ihre *Schnittstellen* ähneln, über die sie diese Dienstleistung anbieten. Dazu können sie Operationen mit einander entsprechenden Signaturen anbieten. Drittens können sich die *Implementationen* dieser Dienstleistung ähneln, indem sie durch einander entsprechende Anweisungen und Kontrollstrukturen realisiert werden.

2.4 Evolution, Wartung und Co-Evolution

Der Begriff *Software-Evolution* wird häufig als Ersatz für *Software-Wartung* verwendet [Parets u. Torres 1996], weshalb hier eine klare Abgrenzung vollzogen wird. Software-Wartung (englisch: *Software Maintenance*) bezeichnet die „Änderung eines Software-Produkts nach seiner Auslieferung, um Fehler zu beheben, ihre Performanz oder andere Attribute zu verbessern, oder es an eine veränderte Umgebung anzupassen“[Institute of Electrical and Electronics Engineers 1998, Übersetzung des Autors]. Dies umfasst nicht das Hinzufügen neuer Funktionalitäten zur Weiterentwicklung der Software. Parets u. Torres [1996] kritisieren zudem, der Begriff impliziere, dass Softwareentwicklungs-Projekte zu einem bestimmten Zeitpunkt an einen Nutzer ausgeliefert würden und sich die Entwicklungstätigkeit dadurch qualitativ verändere. Sneed merkt dazu an, „dass komplexe Softwareprodukte nie vollendet sind“[Sneed 2015, S. 387].

Dem trägt der Begriff der Software-Evolution Rechnung. Im Gegensatz zu *Wartung* meint *Software-Evolution*, oder einfach *Evolution* über Wartungstätigkeiten hinaus auch die wertsteigernde Weiterentwicklung und Verbesserung der Software [Sneed 2015, S. 388]. Im Sinne des Stufenmodells der Softwareentwicklung nach Rajlich u. Bennett [2000] meint Evolution aber auch eine Phase, in der eben diese Tätigkeiten vollzogen werden. In dieser Arbeit wird der Begriff wie folgt verwendet:

Definition 2.4.1: (Software-)Evolution

(Software-)Evolution bezeichnet die Phase der Weiterentwicklung eines Softwareprodukts im Anschluss an die initiale Entwicklung. Während dieser Phase erweitern die Entwickler die Funktionalität der Software, verbessern ihre Qualität, passen sie an geänderte oder neue Anforderungen an und beheben Fehler.

Änderungen an einer Software vollziehen sich nach Rajlich [2012, S. 74] im Allgemeinen in einer Kombination der Phasen, die in Abbildung 2.2 dargestellt sind.

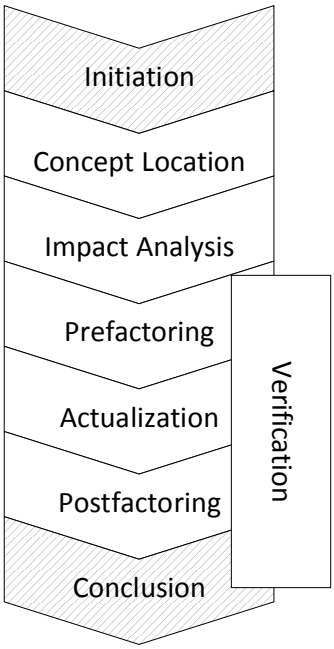

Abbildung 2.2: Phasen einer Software-Änderung nach [Rajlich 2012, S. 74]

In der Phase *Initiation* beschließen die Entwickler, eine Änderung durchzuführen. In der darauf folgenden Phase *Concept Location* identifizieren sie die Code-Elemente, die die zu ändernde Funktionalität umsetzen und dementsprechend angepasst werden müssen. Im Anschluss daran ermitteln die Entwickler im Rahmen einer *Impact-Analyse* alle Elemente im Code, auf die sich die initiale Änderung fortpflanzt. Um die geplante Änderung zu vereinfachen, restrukturieren die Entwickler den Quellcode gegebenenfalls vorab. Dies bezeichnet Rajlich als *Prefactoring*-Phase. Auf Basis des so vorbereiteten Quellcodes setzen die Entwickler die Änderung in der Phase *Actualization* um. Im Anschluss daran werden in der Phase des *Postfactoring* gegebenenfalls erneut Refactorings durchgeführt, um die Struktur des geänderten Codes zu bereinigen. In der Phase *Conclusion* wird die Änderung finalisiert. Dabei wird beispielsweise die zugehörige Dokumentation aktualisiert, die Änderung wird per Commit in einem Versionierungssystem gespeichert oder in ein bevorstehendes Release integriert. Parallel zu den Tätigkeiten der Phasen Prefactoring, Actualization, Postfactoring und Conclusion führen die Entwickler qualitätssichernde Maßnahmen durch. Dies ordnet Rajlich [2012, S. 75] in eine eigene Phase namens *Verification* ein. Die Bezeichnung Verification meint hier nicht nur formale Korrektheitsbeweise im Sinne der Programmverifikation nach Apt u. Olderog [1994]. Sie umfasst auch alle anderen qualitätssichernden Maßnahmen wie Tests und Code Reviews.

Ist eine Software portiert worden, so muss nicht nur die Ausgangsimplementation evolviert werden sondern auch die Zielimplementation. Wird im Zuge der Evolution eine Änderung an der Ausgangsimplementation vorgenommen, so soll diese häufig auch auf die Zielimplementation übertragen werden, um die Konsistenz zwischen beiden Implementationen zu wahren. Der gleiche Effekt kann auch umgekehrt eintreten, wenn die Zielimplementation gemäß einer neuen Anforderung weiterentwickelt wird. So beeinflussen sich die Änderungen der Ausgangs- und Zielimplementation gegenseitig. In der Biologie spricht man von *Co-Evolution*, wenn sich zwei Populationen gegenseitig in ihrer evolutionären Entwicklung beeinflussen [Janzen 1980]. Dieser Begriff wurde bereits in verschiedene Forschungsfelder der Softwareentwicklung übernommen und meint dort die sich gegenseitig beeinflussende Evolution von Softwareartefakten. Beispielhaft seien die Co-Evolution von Entwurfsmodellen und der zugehörigen Implementation genannt [D'Hondt u. a. 2002] oder die Co-Evolution von Tests mit dem Code, den sie prüfen [Zaidman u. a. 2008]. In dieser Arbeit ist mit Co-Evolution stets die plattform-übergreifende Co-Evolution der Ausgangs- und Zielimplementation gemeint.

Definition 2.4.2: (plattform-übergreifende) Co-Evolution

Als (plattform-übergreifende) Co-Evolution wird die synchronisierte Evolution zweier Implementationen derselben Software für unterschiedliche Plattformen bezeichnet.

2.5 Traceability

Diese Arbeit zielt unter anderem darauf ab, explizite Verknüpfungen zwischen Quellcode-Elementen der Ausgangsimplementation und ihren Entsprechungen in der Zielimplementation zu erzeugen. In diesem Kontext sind die Begriffe *Trace Link*, *Traceability*, *Trace Capture* und *Trace Recovery* zu definieren, die im weiteren Verlauf der Arbeit genutzt werden. Die in dieser Arbeit angewendete Terminologie der Traceability basiert auf dem Standardwerk von Cleland-Huang u. a. [2012].

Definition 2.5.1: Trace Link

Ein Trace Link ist eine explizite Verknüpfung eines Quellartefaktes mit einem Zielartefakt [Cleland-Huang u. a. 2012, S. 6].

Im Kontext dieser Arbeit handelt es sich bei Quell- und Zielartefakt um Quellcode-Elemente, sodass auch von Quell- und Zielelement eines Trace Links gesprochen wird. Eines der beiden Code-Elemente stammt stets aus der Ausgangsimplementation, das

andere aus der Zielimplementation. Da sie zwei Code-Elemente miteinander assoziieren, werden sie in den UML-Diagrammen dieser Arbeit als Assoziationen dargestellt. Ein *korrekter* Trace Link drückt in dieser Arbeit stets aus, dass Quell- und Zielelement sich in einem wesentlichen Zweck entsprechen. Beispielsweise verbindet ein korrekter Trace Link eine Methode der Ausgangsimplementation mit ihrer konvertierten Übersetzung in der Zielimplementation. Diese Entsprechungsbeziehungen sind naturgemäß bidirektional.

Definition 2.5.2: Traceability

Das Potential, Quell- und Zielelement miteinander zu Verbinden und diese explizite Verbindung zu nutzen, wird als Traceability bezeichnet [Cleland-Huang u. a. 2012, S. 9].

Trace Links können grundsätzlich durch zwei Strategien erhoben werden [Cleland-Huang u. a. 2012, S. 15]:

Im Falle der Portierung können die plattform-übergreifenden Trace Links zum einen unmittelbar bei der Übersetzung oder Re-Implementierung der ursprünglichen Code-Elemente erzeugt werden. Dieses Vorgehen wird als *Trace Capture* bezeichnet.

Definition 2.5.3: Trace Capture

Trace Capture bezeichnet das Erzeugen von Trace Links unmittelbar bei der Erstellung der verbundenen Elemente [Cleland-Huang u. a. 2012, S. 15].

Zum anderen können die plattform-übergreifenden Entsprechungsbeziehungen zwischen Code-Elementen erst nach der Übersetzung oder Re-Implementierung ermittelt und durch Trace Links dokumentiert werden. Hierfür wird der Begriff *Trace Recovery* verwendet.

Definition 2.5.4: Trace Recovery

Trace Recovery bezeichnet das nachträgliche Erheben von Trace Links nach der Erzeugung und Bearbeitung der verbundenen Elemente [Cleland-Huang u. a. 2012, S. 15].

3 Bewertung des aktuellen Stands der Technik

In den folgenden Abschnitten werden Forschungsarbeiten und Technologien bewertet, die das Thema dieser Arbeit berühren. Es werden vor allem solche Arbeiten beleuchtet, die einen Bezug zur Portierung, zur Vereinheitlichung von Software-Strukturen, zur parallelen Weiterentwicklung oder zur Gewinnung, Wartung und Nutzung von Trace Links haben. Dabei wird jeweils betrachtet, welchen Bezug sie zu dieser Arbeit haben und inwiefern sie zur Erreichung der in Abschnitt 1.4 definierten Ziele verwertbar sind.

3.1 Plattform-übergreifende Entwicklung auf Basis von Modellen und Cross-Platform-Frameworks

Verschiedene Ansätze dienen dazu, eine Software auf Basis vereinheitlichter Artefakte von vornherein für mehr als eine Plattform zu entwickeln. Dazu zählen zum einen Ansätze der modellgetriebenen Softwareentwicklung, die eine Software auf Basis plattformunabhängiger Modelle definieren, welche schrittweise in plattform-spezifischen Quellcode übersetzt werden. Zum anderen gibt es Cross-Platform-Frameworks, die eine vereinheitlichte API bereitstellen, mit der verschiedene Plattformen adressiert werden. Entwickler können Quellcode auf Basis dieser API entwickeln, der dann auf verschiedenen Plattformen ausführbar ist. Diese Ansätze stehen in Konkurrenz zur plattformspezifischen Entwicklung mit späterer Portierung, weshalb sie hier zu diskutieren sind.

3.1.1 Modellgetriebene Softwareentwicklung

Die modellgetriebene Softwareentwicklung zielt darauf ab, lauffähige Software aus abstrakten Modellen zu generieren. In der plattform-übergreifenden Entwicklung wird dabei zunächst ein plattform-unabhängiges Modell (Platform Independent Model, kurz: PIM) entworfen, das von plattform-spezifischen Aspekten abstrahiert. Aus diesem plattform-unabhängigen Modell wird - gelegentlich über den Umweg eines plattformspezifischen Modells (PSM) - der plattform-spezifische Quellcode generiert.

Zur Entwicklung plattform-unabhängiger Modelle werden sowohl grafische als auch textuelle Modellierungssprachen eingesetzt. Beispielsweise stellen Botturi u. a. [2013] ein UML2-Profil vor, mit dem Entwickler auf Basis von Klassen- und Zustandsdiagrammen mobile Apps für die Plattformen Android und Windows Phone entwickeln können. Heitkötter u. a. [2013] führen im Gegensatz dazu die textuelle domänenspezifische Sprache MD^2 ein, mit der datengetriebene Geschäftsanwendungen für die mobilen Plattformen Android und iOS entwickelt werden können. Solche modellgetriebenen Ansätze senken den Aufwand für die Entwicklung plattform-spezifischer Implementationen, indem gemeinsame Konzepte im plattform-unabhängigen Modell erfasst werden. Darüber hinaus wahren sie automatisch die Konsistenz zwischen den generierten Implementationen für die verschiedenen Plattformen, indem diese aus demselben Model erzeugt werden.

Allerdings sind modellgetriebene Technologien wie MD^2 auf die Klasse von Anwendungen beschränkt, die mit der Modellierungssprache beschrieben und mit den zugehörigen Code-Generatoren nur für wenige Plattformen erzeugt werden können. Für die in dieser Arbeit betrachtete Portierung bestehender, ausgereifter Implementationen ist die modellgetriebene Entwicklung zudem nicht hilfreich, da die ursprüngliche ausgereifte Implementation durch eine unausgereifte modellgetriebene Re-Implementation ersetzt würde. Da die Nutzer sich an den Reifegrad der ursprünglichen Implementation gewöhnt haben, besteht ein hohes Risiko, dass dadurch die Zufriedenheit bestehender Nutzer sinkt [Demeyer u. a. 2009, S. 191]. Selbst bei mangelhafter Qualität der Ausgangsimplementation ist nicht davon auszugehen, dass eine bessere Implementation durch eine Neuentwicklung entsteht [Spolsky 2000]. Sinnvoller kann es sein, die bestehende Implementation zu restrukturieren und dann zu portieren. Zudem ist bei weitem nicht für alle Arten von Anwendungen und für beliebige Kombinationen von Ausgangs- und Zielimplementation überhaupt eine modellgetriebene Technologie verfügbar. Hinzu kommt, dass sich die Entwickler mit dem Einsatz modellgetriebener Technologie von der Weiterentwicklung dieser Technologie abhängig machen. Nur wenn die Quellcode-Generatoren kontinuierlich an die Änderungen der APIs auf den Zielplattformen angepasst werden, bleibt der Quellcode funktionsfähig und kann die neuesten Versionen der Plattform-APIs nutzen.

Trotz dieser Einschränkung können einige Konzepte der modellgetriebenen Softwareentwicklung auf den Kontext dieser Arbeit übertragen werden. Insbesondere spielt auch bei der Portierung die automatische Erzeugung von Quellcode eine Rolle. In der Portierung entsteht generierter Quellcode beim Einsatz von Quellcode-Konvertoren, die den Code der Ausgangsimplementation in die Zielsprache übersetzen. Ähnlich wie bei der modellgetriebenen Entwicklung muss dieser automatisch erzeugte Quellcode gegebenenfalls plattform-spezifisch ergänzt werden. In beiden Fällen müssen generierter und manuelle entwickelter Code integriert werden. Gleichzeitig sollten manueller und generierter Code nicht im selben Quellcode-Element vermischt werden [Petrasch u. Meimberg 2006,

S. 14]. Eine solche Vermischung birgt die Gefahr, dass manuelle Änderungen beim wiederholten Generieren überschrieben werden. Grundsätzlich gibt es drei Ansätze, um dies zu vermeiden.

Erstens können Entwurfsmuster eingesetzt werden, die manuell implementierte, plattform-spezifische Code-Elemente von generierten Code-Elementen trennen [Pietrek u. Trompeter 2007, S. 162 f.] [Greifenberg u. a. 2015]. Diese Entwurfsmuster werden in der hier entwickelten Portierungsmethode aufgegriffen und in den Kontext der Portierung übertragen. Sie isolieren plattform-spezifische Code-Elemente und verbergen sie hinter portierbaren Schnittstellen, die in Ausgangs- und Zielplattform äquivalent genutzt werden. Abschnitt 3.2 geht näher auf verwandte Arbeiten ein, die solche Entwurfsmuster behandeln.

Zweitens können die plattform-spezifischen Anpassungen in die Regeln des Code-Generators integriert werden [Rumpe 2012]. Dieser Ansatz ist auch in vielen Quellcode-Konvertoren implementiert, die es ihren Nutzern erlauben, Abbildungsregeln zu konfigurieren, die die Schnittstellen der ursprünglich genutzten APIs auf ihre Entsprechungen in der Zielplattform abbilden. Diese Möglichkeit nutzen auch portierende Entwickler im Rahmen der hier entwickelten Portierungsmethode, sofern der eingesetzte Quellcode-Konverter es zulässt.

Drittens können die Übersetzungswerkzeuge Mechanismen wie *Protected Regions* oder *PartMerger* implementieren, die generierten und händisch geschriebenen Code in der Ausgabe zusammenführen [Greifenberg u. a. 2015]. Nur wenige Quellcode-Konvertoren implementieren solche Mechanismen, weshalb sie für die Anwendung der hier entwickelten Methode nicht vorausgesetzt werden.

3.1.2 Cross-Platform-Frameworks

Cross-Platform-Frameworks bieten Entwicklern die Möglichkeit, ihre Software von vornherein für mehrere Plattformen in einer gemeinsamen Codebasis zu entwickeln. Zur Nutzung von Plattformfunktionalitäten definieren sie einheitliche Schnittstellen, die von den Unterschieden der plattform-spezifischen APIs abstrahieren. Solche Frameworks stehen unter anderem zur Verfügung, um Software für verschiedene Plattformen mobiler Endgeräte [Rieger u. Majchrzak 2016], für verschiedene Cloud-Dienstleiser [Knipp u. a. 2016] oder auch für verschiedene Spielkonsolen [Petridis u. a. 2010] zu entwickeln. Grundlegend können dabei drei Arten der Cross-Platform-Entwicklung unterschieden werden:

Erstens gibt es den Ansatz der webbasierten Entwicklung. Entwickler programmieren dabei eine Anwendung auf Basis von Webtechnologien wie HTML und JavaScript. Dadurch, dass auf vielen Plattformen Browser zur Verfügung stehen, sind diese Anwendungen plattform-übergreifend nutzbar. Auch sogenannte Progressive Web Apps, die offline-

fähig sind und in einem Browser ohne Adressleiste angezeigt werden, fallen in diesen Ansatz [Majchrzak u. a. 2018]. Die Entwicklung grafischer Oberflächen ist mit diesem Ansatz allerdings auf die vom Browser darstellbaren Elemente beschränkt. Es ist außerdem aufwendig, die entwickelten Oberflächen an die Richtlinien anzupassen, die die einzelnen Plattformhersteller für Benutzerschnittstellen festlegen. Diese Anpassungen sind allerdings notwendig, da die Nutzer der Plattformen eine plattform-typische Oberfläche erwarten, die den spezifischen Richtlinien folgt [Wasserman 2010]. Ein weiterer Nachteil webbasierter Entwicklung besteht darin, dass die Entwickler die Funktionalitäten der adressierten Plattformen nur in begrenztem Umfang nutzen können [Heitkötter u. a. 2012] [Rieger u. Majchrzak 2016].

Zweitens gibt es Frameworks, mit denen sogenannte hybride Anwendungen entwickelt werden können. Sie werden ähnlich wie die webbasierten Anwendungen in einem Browser ausgeführt. Dieser ist jedoch in eine native Basisanwendung eingebettet, die den Zugriff auf einige Funktionalitäten der Plattform erlaubt. Die Entwickler nutzen eine JavaScript-API, um auf die Funktionalitäten der Plattform zuzugreifen. Innerhalb dieser API werden die Aufrufe an die spezifischen APIs der Plattform delegiert. Ein Beispiel für diesen Ansatz ist das Framework Adobe PhoneGap[1] für die Entwicklung von Anwendungen für mobile Endgeräte. Zwar können diese Ansätze einen höheren Anteil der Plattformfunktionalitäten nutzen, die Einschränkungen in Bezug auf die Benutzeroberflächen gleichen allerdings denen der webbasierten Entwicklung [Majchrzak u. a. 2018].

Drittens gibt es Frameworks wie Unity[2] oder Xamarin[3], die auf einer eigenen vollständigen Laufzeitumgebung wie Mono[4] mit eigenen Basisdatentypen und Schnittstellen zum Betriebssystem basieren. Die Hersteller solcher Cross-Platform-Frameworks erweitern diese Laufzeitumgebung um den Zugriff auf die nativen plattform-spezifischen APIs oder um andere Basisfunktionalitäten, die in einer Klasse von Anwendungen häufig benötigt werden. Anwendungen, die auf Basis dieser Frameworks entwickelt wurden, werden gemeinsam mit der erweiterten Laufzeitumgebung auf dem Gerät des Nutzers installiert.

Detailliertere Vergleiche dieser drei Ansätze und entsprechender Technologien wurden insbesondere für den Bereich der Entwicklung für mobile Endgeräte publiziert [Heitkötter u. a. 2012] [Rieger u. Majchrzak 2016]. Der Einsatz dieser Frameworks bringt unabhängig vom konkreten Ansatz allerdings diverse Nachteile mit sich, sodass Entwickler nach wie vor häufig native Entwicklung ohne solche Frameworks durchführen [Joorabchi u. a. 2013]. Einige dieser Nachteile entsprechen denen der modellgetriebenen Entwicklung. Beispielsweise machen sich die portierenden Entwickler von den Herstellern

[1] `https://phonegap.com/`
[2] `https://unity3d.com/de`
[3] `https://docs.microsoft.com/de-de/xamarin/`
[4] `https://www.mono-project.com/`

des Cross-Platform-Frameworks ebenso abhängig wie beim Einsatz von modellgetriebenen Technologien. Neuerungen der plattform-spezifischen APIs sind bei Einsatz solcher Frameworks erst dann wirksam, wenn das Framework an diese Neuerungen angepasst wurde. Dies ist insbesondere im Bereich der mobilen Anwendungen ein Problem, da die Evolution der mobilen Plattformen sehr schnelllebig ist[5] [McDonnell u. a. 2013]. Im schlimmsten Fall wird die Entwicklung des gewählten Frameworks wie im Falle von RoboVM eingestellt[6], sodass Entwickler den darauf aufbauenden Quellcode reimplementieren müssen. Ein grundsätzliches Problem liegt darin, dass es nicht für jede Anwendungsdomäne und für jede Kombination von Plattformen ein passendes Cross-Platform-Framework gibt.

Für die Portierung bestehender, ausgereifter Software sind Cross-Platform-Frameworks zudem ungeeignet. Die ursprüngliche Implementation wird beim Einsatz eines solchen Frameworks durch eine Re-Implementation auf Basis des Frameworks ersetzt. Das radikale Ersetzen eines gewohnten Systems durch ein weniger ausgereiftes stößt, wie bereits erwähnt, bei den Nutzern häufig nicht auf Akzeptanz [Demeyer u. a. 2009, S. 191].

3.2 Techniken und Entwurfsmuster für die Vereinfachung des Austauschs von APIs durch Einführung plattform-übergreifend gleicher Schnittstellen

In verschiedenen Forschungsbereichen der Softwaretechnik werden Entwurfsmuster beschrieben, die den Austausch der ursprünglich eingesetzten APIs durch die APIs der Zielplattform vereinfachen. Insbesondere können die portierenden Entwickler auf Entwurfsmuster zurückgreifen, die die ursprünglich genutzten Schnittstellen der Ausgangsplattform in der Zielplattform nachbilden und sie auf Basis der Ziel-APIs implementieren. Dadurch wird es vereinfacht, die ursprünglichen Abhängigkeiten zur Ausgangsplattform auf äquivalente Abhängigkeiten zur Zielplattform abzubilden und den ursprünglichen Quellcode teilautomatisiert zu übersetzen. Durch sie Angleichung der Schnittstellen wird vermieden, dass automatisch übersetzter Code händisch an die Schnittstellen der Ziel-API angeglichen werden muss. Gleichzeitig führt die Einführung solcher plattform-übergreifend gleicher Schnittstellen dazu, dass plattform-unabhängige Code-Elemente von plattform-spezifischen Abhängigkeiten befreit werden. Somit wird die Konsistenz zwischen Ausgangs- und Zielimplementation erhöht.

Ein Forschungsfeld, in dem solche Entwurfsmuster genutzt werden, sind Migrationsprojekte, in denen eine alte API durch eine neue API ersetzt wird. Diese Aufgabe wird

[5] http://codergears.com/Blog/?p=1451

[6] https://www.heise.de/developer/meldung/Opfer-der-Xamarin-Uebernahme-Microsoft-stellt-Entwicklung-von-RoboVM-ein-3176539.html

auch als *API-Migration* bezeichnet. Ähnlich wie bei der Portierung ist es auch bei der API-Migration erstrebenswert, die Schnittstellen der Ziel-API an die ursprünglich genutzte API anzugleichen. So wird vermieden, dass der Code, der von der ursprünglichen API abhängt, händisch an die neuen Schnittstellen angepasst werden muss. Dazu können unter anderem die Entwurfsmuster *Object Adapter* und *Class Adapter* eingesetzt werden. Bartolomei u. a. [2010] untersuchen deren Einsatz beim Austausch einer API für die Entwicklung grafischer Benutzerschnittstellen und identifizieren Herausforderungen bei ihrer Umsetzung. Eine dieser Herausforderungen besteht darin, dass die ursprünglich genutzten Typen nicht immer auf exakt einen korrespondierenden Typ der Ziel-API abgebildet werden können. Stattdessen ist die Funktionalität eines ursprünglich eingesetzten Typs gelegentlich auf mehrere Typen der Ziel-API verteilt. Nicht alle Entwurfsmuster zur Anpassung von Schnittstellen können solche 1-zu-n-Abbildungen behandeln. Diese Herausforderung besteht auch in Portierungsprojekten. Dementsprechend werden die Entwurfsmuster zur Anpassung der Ziel-API in Kapitel 5 dieser Arbeit unter anderem dahingehend verglichen, ob sie mehrere Typen der Ziel-API gemeinsam hinter einer Schnittstelle verbergen können.

Pehl [2012] stellt ebenfalls Entwurfsmuster im Kontext der API-Migration vor. Die Arbeit richtet sich allerdings nicht an Entwickler, die den Austausch einer API selbst vornehmen, sondern an Entwickler von Werkzeugen zur Automatisierung solcher API-Migrationen. Sie stellt ausschließlich Muster zum Aufbau und zur Funktionsweise solcher Werkzeuge vor. Die Entwurfsmuster, die diese Werkzeuge bei ihrer Anwendung einführen, um Abhängigkeiten zur ursprünglichen API zu behandeln, werden nur am Rande erwähnt. Eines der vorgestellten Werkzeuge führt beispielsweise das Entwurfsmuster *Oject-Adapter* ein, um die Schnittstellen der Ziel-API anzupassen. Ein bewertender Vergleich verschiedener Entwurfsmuster für diesen Zweck wie in Kapitel 5 der vorliegenden Arbeit erfolgt dort allerdings nicht.

Im Kontext der cloudbasierten Softwareentwicklung erkennen Knipp u. a. [2016] das Problem, dass Entwickler sich an die Cloud-Anbieter binden, wenn sie deren spezifische Cloud-APIs einsetzen. Sie schlagen vor, direkte Abhängigkeiten zur API einer Cloud-Plattform zu vermeiden, um eine etwaige spätere API-Migration zu erleichtern. Dazu empfehlen sie, eine API-Abstraktionsschicht zu erstellen, die vereinheitlichte Schnittstellen zur Nutzung verschiedener Cloud-APIs definiert. Die Abstraktion von spezifischen APIs ist auch im Kontext von Portierungsprojekten angebracht, um die Strukturen der Ausgangs- und Zielimplementation einander anzugleichen. Knipp u. a. [2016] geben jedoch keine Hinweise, *wie* diese entkoppelnde Abstraktionsschicht zu implementieren ist. In Kapitel 5 dieser Arbeit wird eine Richtlinie erarbeitet, die Entwickler bei der Umsetzung einer solchen Abstraktionsschicht anleitet, indem sie Regeln für den Einsatz entsprechender Entwurfsmuster definiert.

Auch in der modellgetriebenen Softwareentwicklung werden Entwurfsmuster zur Isola-

tion plattformspezifischen Codes eingesetzt. Dort besteht die Notwendigkeit, plattformspezifischen, manuell verfassten Code genau wie bei der Portierung hinter plattformunabhängigen Schnittstellen zu verbergen, um händische Änderungen an automatisch generiertem Code zu vermeiden [Pietrek u. Trompeter 2007, S. 162 f.] [Greifenberg u. a. 2015] [Vlissides 1996]. Greifenberg u. a. [2015] vergleichen in diesem Kontext den Einsatz verschiedener Techniken und Entwurfsmuster. Es fehlt allerdings eine Untersuchung, die die Auswirkungen der verschiedenen Entwurfsmuster auf Portierungsprojekte vergleicht. Außerdem fehlt eine Richtlinie, die angibt, wann welches Entwurfsmuster in einer konkreten Situation bei der Portierung einzusetzen ist. Diese Lücken werden in Kapitel 5 dieser Arbeit geschlossen.

Zellagui u. a. [2017] beschreiben eine Methode zur Auflösung von Aufruf- und Instanzierungs-Beziehungen zur Modularisierung einer Codebasis. Sie setzen dazu die Entwurfsmuster *Adapter* bzw. *Dependency Injection* ein. Beide können auch bei der Entkopplung zu portierenden Codes von plattform-spezifischen APIs verwendet werden. Eine Abwägung von Alternativen zum Entwurfsmuster *Adapter* fehlt bei Zellagui u. a. [2017] allerdings. In Kapitel 5 werden solche Alternativen hinsichtlich ihrer Auswirkungen auf Portierungsprojekte verglichen. Die hier entwickelte Richtlinie unterstützt die Entwickler bei deren Auswahl.

Cerny u. Donahoo [2015] präsentieren einen Ansatz zur Unterteilung von Benutzeroberflächen in plattform-spezifische und plattform-übergreifende Aspekte. Dieser kann die hier beschriebene Methode ergänzen, indem er portierenden Entwicklern zumindest für die Benutzeroberfläche vorgibt, welche Teile als plattform-spezifisch zu isolieren sind. Wie diese Isolation auf Code-Ebene umgesetzt werden sollte, ist nicht Teil der Arbeit von Cerny u. Donahoo [2015]. Die in Kapitel 5 diskutierten Entwurfsmuster bewirken genau diese Isolation. Sie führen plattformübergreifend gleiche Schnittstellen ein und isolieren plattformspezifischen Code. Der Austausch der eingesetzten plattform-spezifischen APIs muss dann nur im isolierten plattform-spezifischen Teil des Codes durchgeführt werden.

„Eine Software-Produktlinie ist eine Gruppe von Software-Systemen, die auf Basis einer gemeinsamen Menge von Kern-Artefakten entwickelt wurden, um eine gemeinsame Menge von Features zu implementieren, die die spezifischen Bedarfe einer Aufgabe oder eines Marktsegments erfüllen“[Clements u. Northrop 2001, S. 5 (Übersetzung des Autors)]. Auch im Kontext solcher Software-Produktlinien besteht der Bedarf zur Trennung von wiederverwendbaren Code-Elementen, die in allen Produkten einer Produktlinie benötigt werden, von speziellen Elementen, die spezifisch für einzelne Produkte sind. Um ein einzelnes Softwareprodukt zu einer Produktlinie weiterzuentwickeln, müssen solche produktspezifischen Code-Elemente isoliert werden. Eine ähnliche Anforderung existiert in Portierungsprojekten. Hier müssen portierbare Code-Elemente, die automatisiert Anweisung für Anweisung in eine Entsprechung übersetzt werden können,

von plattform-spezifischen Elementen isoliert werden, die Abhängigkeiten zu plattformspezifischen APIs aufweisen. Verschiedene Ansätze beschreiben die Zerlegung eines Softwareprodukts in einzelne, wiederverwendbare Features. Durch die Kombination dieser Features können dann Varianten des ursprünglichen Systems erstellt werden, die gemeinsam die Produktlinie bilden [Liu u. a. 2006] [Lopez-Herrejon u. a. 2011] [Valente u. a. 2012]. Diese Ansätze könnten auf den Kontext der Portierung übertragen werden und zur Isolation von plattformspezifischen Code-Elementen dienen. Allerdings zerlegen diese Ansätze den ursprünglichen Code zeilenweise, um einzelne Zeilen zu Features zuzuordnen, die dann in verschiedenen Varianten des Produkts wiederverwendet werden. Dieses Vorgehen verändert das Programmierparadigma des ursprünglichen Codes von einer objektorientierten Sichtweise hin zu einer feature-orientierten. Im Kontext der Portierung sollte dieser Paradigmenwechsel nicht durchgeführt werden, da er erheblichen Aufwand für das Erlernen der geänderten Strukturen verursacht.

3.3 Richtlinien und Werkzeugunterstützung für die Portierung

Hook [2005], Mooney [2004] und Wilson[7] definieren Richtlinien, nach denen eine Software von vornherein portabel entwickelt werden kann, sodass die Portierung vereinfacht wird. Diese Regeln sind bei der nachträglichen Portierung nicht unmittelbar anwendbar, da sie sich auf die initiale Entwicklung der Ausgangsimplementation beziehen. Sie können aber auf die Portierung bereits existierender Software übertragen werden.

Hook [2005] empfiehlt beispielsweise, die Nutzung verschiedener APIs durch die Einführung von Abstraktionen zu unterstützen. Kapitel 5 überträgt diesen Ansatz auf die Portierung und konkretisiert sie durch eine Richtlinie zum Einsatz von Entwurfsmustern, die einheitliche Schnittstellen zur Nutzung plattform-spezifischer APIs einführen. Hook nennt außerdem Richtlinien zum Portieren selbst, die die hier entwickelte Portierungsmethode übernimmt [Hook 2005, S. 43]:

Erstens rät er, Quellcode nicht vor der Portierung für portabel zu erklären, da die konkreten Hindernisse sich erst bei der Portierung selbst zeigen. Dem trägt diese Arbeit unter anderem dadurch Rechnung, dass die gewählte Portierungsstrategie iterativ an die konkret aufgetretenen Hürden anpasst wird.

Zweitens empfiehlt Hook, die bestehende Codebasis, soweit möglich, unangetastet zu lassen, um ihre Lauffähigkeit und ihren Reifegrad nicht zu gefährden. Diese Arbeit empfiehlt dementsprechend, die Schnittstellen zur Zielplattform möglichst an die Schnittstellen der Ausgangsplattform anzupassen und nicht umgekehrt. Die dazu eingesetzten Entwurfsmuster werden dahingehend verglichen, welche Änderungen am ursprünglichen Quelltext sie erfordern.

[7] `https://www.backblaze.com/blog/10-rules-for-how-to-write-cross-platform-code`

Drittens rät Hook, zunächst die kritischen Stellen zu identifizieren, die die Portierung behindern. Genau diesen Ansatz verfolgt die Portierungsmethode dadurch, dass die Plattformabhängigkeiten der Ausgangsimplementation unmittelbar nach Festlegung der zu portierenden Funktionalitäten identifiziert werden.

Im Gegensatz zu dieser Arbeit definiert Hook [2005] keine konkreten Schritte, die portierende Entwickler anleiten. Er geht zudem davon aus, dass Ausgangs- und Zielimplementation in derselben Programmiersprache geschrieben sind, was die Portierung im Vergleich zu den hier adressierten Portierungsprojekten deutlich vereinfacht. Traceability und Konzepte zur Co-Evolution werden in diesen Fällen nicht benötigt und dementsprechend von Hook nicht beleuchtet.

Mooney [2004] definiert ebenfalls Regeln, durch die Entwickler bereits bei der initialen Entwicklung eine hohe Portabilität ihres Codes erreichen.

Entwickler sollten **erstens** ein Programmierparadigma wählen, dass der Portabilität zuträglich ist. Mooney nennt hier die Objektorientierung als passendes Paradigma, das in dieser Arbeit vorausgesetzt wird (siehe Abschnitt 1.5).

Zweitens sollten Entwickler die benötigten Schnittstellen zur Plattform identifizieren, um sich der Abhängigkeiten ihrer Software zur Plattform bewusst zu werden. Dieser Ansatz wird auch in der hier entwickelten Methode verfolgt, deren zweiter Schritt die Abhängigkeiten der Ausgangsimplementation zur ursprünglichen Plattform analysiert (vgl. Abschnitt 4.4.2).

Drittens sollen Entwickler laut Mooney [2004] plattform-unabhängige Standards identifizieren, an denen die Nutzung der identifizierten Plattform-Schnittstellen von vornherein ausgerichtet werden muss. Diese Regel ist bei der Portierung nur bedingt anwendbar, da die Nutzung der ursprünglichen Plattform bereits implementiert ist. Bei Unterschieden zwischen den Schnittstellen der Ausgangs- und Zielplattform empfiehlt diese Arbeit die Nachbildung der ursprünglich genutzten Schnittstellen in der Zielimplementation, um den Code der Ausgangsimplementation möglichst unangetastet zu lassen.

Viertens empfiehlt Mooney [2004], Plattformabhängigkeiten zu isolieren. Mooney gibt allerdings keine Hinweise darauf, *wie* diese Isolation umgesetzt werden kann. Wie oben erwähnt, schließt die vorliegende Arbeit diese Lücke, indem sie in Kapitel 5 passende Entwurfsmuster für diese Isolation identifiziert und vergleicht und eine Richtlinie für ihren Einsatz formuliert.

Brian Wilson definiert zehn Regeln zur Entwicklung plattform-übergreifend nutzbaren Codes in der Programmiersprache C[8]. Die meisten davon werden in dieser Arbeit nicht

[8] `https://www.backblaze.com/blog/10-rules-for-how-to-write-cross-platform-code`

übernommen, da sie spezifisch für die Programmiersprache C sind und bereits bei der initialen Entwicklung der Ausgangsimplementation befolgt werden müssen. Eine Ausnahme bildet die Empfehlung, plattform-spezifischen Code beispielsweise für die Umsetzung einer Benutzeroberfläche in eigene Code-Elemente zu extrahieren. Schritt sechs der hier entwickelten Portierungsmethode führt diese Extraktion von plattform-spezifischem Code durch.

Neben diesen allgemeinen Vorschriften sind mehrere Fallstudien durchgeführt und entsprechende Richtlinien aus ihnen abgeleitet worden [Alves u. a. 2005] [Schmitz 2014] [Tisdall u. a. 2018]. Schmitz [2014] beschreibt die Portierung einer .Net-Anwendung auf die iOS-Plattform und leitet einen allgemeinen Portierungsprozess daraus ab. Ähnlich wie in dieser Arbeit wird die Trennung von plattform-spezifischen und plattformunabhängigen Elementen forciert. Im Gegensatz zu dieser Arbeit wird allerdings vorausgesetzt, dass Ausgangs- und Zielimplementation in derselben Programmiersprache entwickelt wurden. Der beschriebene Prozess ist somit nur begrenzt einsetzbar. Er betrachtet den Einsatz von Quellcode-Konvertoren nicht und bietet keine Lösungen für sprachübergreifende Traceability. Außerdem fehlen konkrete Richtlinien für die Isolation plattform-spezifischer Elemente durch Entwurfsmuster wie sie in Kapitel 5 dieser Arbeit diskutiert werden.

Tisdall u. a. [2018] beschreiben die organisatorischen Herausforderungen, auf die sie bei der Portierung einer Desktop-Anwendung auf mobile Endgeräte gestoßen sind. Die dazu genannten Lösungen sind allerdings sehr kurz und allgemein formuliert. Technische Herausforderungen bezüglich der Übertragung von Konzepten und Strukturen werden nicht konkret behandelt. Zur Co-Evolution der Ausgangs- und Zielimplementation werden ebenfalls keine Aussagen gemacht.

Alves u. a. [2005] präsentieren drei Fallstudien zur Portierung von Spielen auf Basis der gemeinsamen Plattform J2ME für zusätzliche Gerätefamilien. Bei diesen Fallstudien handelt es sich um Portierungen, bei denen die Programmiersprache der Ausgangsimplementation auch für die Zielimplementation genutzt werden konnte. Die Aufgabe umfasste damit keine Quellcode-Übersetzung, sondern lediglich den Austausch gerätespezifischer APIs und die Anpassung des Quellcodes an die Hardware-Einschränkungen der Zielgeräte. Die Arbeit leistet allerdings Beiträge zum Austausch von APIs. Alves u. a. [2005] schlagen dazu unter anderem eine plattform-übergreifende Abstraktion von verschiedenen APIs vor und empfehlen, plattform-spezifischen Code zu isolieren. Die in Kapitel 5 diskutierten Entwurfsmuster konkretisieren diese Empfehlung.

Eine wichtige Klasse von Werkzeugen für die Portierung sind Quellcode-Konvertoren, die den Quelltext der ursprünglichen Codebasis in eine Programmiersprache übersetzen, die auf der Zielplattform eingesetzt werden kann. Verschiedene Forschungsarbeiten schlagen Ansätze zur Entwicklung und zum Einsatz solcher Quellcode-Konvertoren

vor. Arango u. a. [1985] definieren beispielsweise eine Methode namens *Port by Abstraction* für die automatisierte Übersetzung der ursprünglichen Codebasis. Sie überführt den Quellcode in ein Modell, das von der konkreten Syntax der ursprünglich eingesetzten Programmiersprache abstrahiert. Die Anwender der Methode müssen Transformationsregeln definieren, mit denen die Zielimplementation aus diesem Modell generiert wird. Ein Vorteil dieses Ansatzes ist, dass das extrahierte Modell die Gemeinsamkeiten zwischen Ausgangs- und Zielimplementation explizit festhält. Allerdings entwickeln die portierenden Entwickler bei dem Vorgehen einen vollständigen, projektspezifischen Quellcode-Konverter, was extrem hohen Aufwand verursacht. Eine vollständig automatisierte Übersetzung macht es erforderlich, Transformationsregeln für *alle* eingesetzten Sprachkonstrukte und API-Abhängigkeiten zu definieren. Dieser Aufwand ist insbesondere für solche Regeln nicht gerechtfertigt, die nur zur Übersetzung weniger Codezeilen angewendet werden.

Um den Aufwand für die Entwicklung von Quellcode-Transformatoren zu mindern, stellen Arrighi u. a. [2014] die abstrakte objektorientierte Programmiersprache *GOOL* vor. Sie beinhaltet eine gemeinsame Teilmenge mehrerer objektorientierter Programmiersprachen. Sie entwickeln für diese Programmiersprache einen Übersetzer, der Quellcode in den Sprachen Java und C++ in GOOL übersetzt und GOOL wiederum in Java und C++ übersetzen kann. Der Vorteil dieses Ansatzes ist, dass Sprachen automatisch in alle Zielsprachen des Konverters übersetzt werden können, sobald die Übersetzung nach GOOL implementiert wurde. Die Konversion ist allerdings nicht vollständig, da GOOL nur die kleinste gemeinsame Teilmenge der objektorientierten Programmiersprachen ausdrücken kann. Dementsprechend müssen zusätzliche Übersetzungsregeln definiert werden, um die übrigen Sprachkonzepte zu übersetzen. Zur Übersetzung von API-Aufrufen müssen die APIs zudem in Konfigurationsdateien für GOOL definiert werden. Außerdem müssen formale API-Mappings definiert werden, die die ursprünglichen API-Aufrufe auf diese Definitionen abbilden. Um die in GOOL definierten API-Aufrufe in eine konkrete Zielsprache übersetzen zu können, sind erneut entsprechende Abbildungsvorschriften erforderlich. Der Einsatz von Konvertoren auf Basis von GOOL ist problemlos im Einklang mit der hier entwickelten Portierungsmethode möglich. Die vorliegende Arbeit liefert zudem Ansätze, um einheitliche Schnittstellen auf Ausgangs- und Zielplattform bereitzustellen, sodass die Komplexität der notwendigen API-Mappings sinkt.

Quellcode-Konvertoren sind häufig unvollständig, sodass sowohl wissenschaftliche [Terekhov 2001] als auch praktische[9] Anleitungen zu ihrer Anwendung empfehlen, Quellcode automatisch zu übersetzen und die unvollständige Übersetzung manuell zu vervollständigen. Dies birgt den Nachteil, dass die manuellen Anpassungen verloren gehen, wenn der Konverter erneut angewendet wird, um Änderungen am Ausgangsco-

[9] `https://github.com/apache/lucenenet/blob/master/CONTRIBUTING.md`

de in die Zielimplementation zu übertragen. Dadurch wird der Einsatz automatischer Konvertoren in der Co-Evolution behindert. Um diese Nachteile zu umgehen, werden in dieser Arbeit konvertierbare und händisch zu übersetzende Code-Elemente voneinander getrennt, ehe Quellcode-Konvertoren eingesetzt werden.

3.4 Richtlinien und Werkzeugunterstützung für Migrationsprojekte

Die Ziele und Herausforderungen von Migrationsprojekten ähneln denen der Portierung stark. Dementsprechend können auch viele Strategien der Migration wie Re-Implementierung, Konversion und Wrapping [Sneed u. a. 2010, S. 10ff] für die Portierung genutzt werden. Auch allgemeine Vorgehensmuster, die im Kontext der Migration formuliert werden, sind in diese Arbeit eingeflossen. Insbesondere die Vorgehensmuster *Present the Right Interface* und *Migrate Systems Incrementally* [Demeyer u. a. 2009] werden für die Isolation von Plattformabhängigkeiten und die schrittweise Portierung angewendet.

Auch bei Migrationsprojekten wird empfohlen, die Konzepte der ursprünglichen Implementation in die Zielimplementation zu überführen [Terekhov 2003]. Dadurch wird es den ursprünglichen Entwicklern erleichtert, ihre Erfahrungen für die Evolution des migrierten Codes wiederzuverwenden. Dies gilt für Portierungsprojekte umso mehr, da die ursprüngliche Version im Gegensatz zu Migrationsprojekten erhalten bleibt. Sie muss parallel zur Zielimplementation weiterentwickelt werden. Die entsprechende Empfehlung zur Übertragung der ursprünglichen Konzepte in die Zielimplementation findet sich bereits in der Zielstellung dieser Arbeit wieder.

Bei Migrationsprojekten spielt die Co-Evolution der Ausgangs und der Zielimplementation in der Regel keine Rolle, da der ursprüngliche Code durch den migrierten Code ersetzt und nicht weiterentwickelt wird. Eine Ausnahme bilden langfristige Migrationsprojekte, in denen noch aktive Teile der Ausgangsimplementation und ihre bereits migrierten Entsprechungen gemeinsam weiterentwickelt werden müssen. Um einen solchen Fall handelt es sich bei der Migration der Kollaborationsplattform *CommSy* von PHP nach Java [Dittberner 2007]. In diesem Migrationsprojekt führte Dittberner [2007] händisch Trace Links zwischen den ursprünglichen und migrierten Elementen beider Implementationen ein, um Anpassungen an den noch genutzten Elementen der ursprünglichen Implementation auf ihre migrierten Entsprechungen zu übertragen. Auf Basis dieser Trace Links wurde auch ein Eclipse-Plugin entwickelt, das die Navigation zwischen verknüpften Quellcode-Elementen ermöglicht. Der Bedarf zur Navigation zwischen einander entsprechenden Quellcode-Elementen besteht in Portierungsprojekten umso mehr,

da Ausgangs- und Zielimplementation langfristig co-evolviert werden. Die von Dittberner [2007] durchgeführte manuelle Zuordnung von Übersetzungen verursacht hohen Aufwand für die Entwickler, weshalb Trace Links in dieser Arbeit ausschließlich automatisch erzeugt werden. Abschnitt 4.5.1 und Kapitel 6 führen dementsprechend einen Trace-Capture- bzw. Trace-Recovery-Mechanismus ein.

Eine gemeinsame Kernaufgabe von Migrations- und Portierungsprojekten ist die Übersetzung von Quellcode in neue Programmiersprachen. Terekhov u. Verhoef [2000] schildern Erfahrungen beim Einsatz von Quellcode-Konvertoren in Migrationsprojekten, die Cobol- bzw. PL/1-Programme in objektorientierte Programmiersprachen übersetzt haben. Insbesondere die von Terekhov u. Verhoef [2000] geschilderten Wechselwirkungen zwischen dem Automatisierungsgrad der Sprachkonversion, den Fähigkeiten der Entwickler und der späteren Evolution des konvertierten Codes sind auf die Portierung übertragbar. Sie werden in Abschnitt 4.1.1 aufgegriffen, um über angemessene Abstraktionen bei der Übersetzung von Quellcode zu entscheiden.

Eine weitere Kernaufgabe in Migrationsprojekten ist der Austausch einer ursprünglich genutzten API gegen eine Ziel-API [Sneed u. a. 2010, S. 44]. Dies wird auch als API-Migration bezeichnet. Ganz ähnlich müssen auch bei der Portierung die Abhängigkeiten zu APIs der Ausgangsplattform in entsprechende Abhängigkeiten zu APIs der Zielplattform überführt werden. Das betrifft erstens die Schnittstellen zur Programmiersprache mit ihren Basisdatentypen und anderen grundlegenden Datenstrukturen wie Sammlungen. Zweitens müssen Abhängigkeiten zur API des ursprünglichen Betriebssystems durch die entsprechenden APIs der Zielplattform ersetzt werden. Drittens müssen etwaige Abhängigkeiten zu externen Programmbibliotheken auf der Zielplattform durch ihre Entsprechungen oder eigene Re-Implementationen ersetzt werden. Arbeiten, die Entwurfsmuster für diesem Zweck beschreiben, wurden bereits in Abschnitt 3.2 diskutiert.

Soll die API-Migration automatisiert werden, so müssen formale Abbildungsregeln, sogenannte *Mappings* definiert werden, die die Typen, Attribute und Operationen der ursprünglich eingesetzten API ihren Entsprechungen in der Ziel-API zuordnen. Auf Basis solcher formalen Mappings beschreiben Nita u. Notkin [2010] einen teil-automatisierten Ansatz namens *Deep Adaptation*, der das Entwurfsmuster Object Adapter einführt, um Ausgangs- und Ziel-API aneinander anzugleichen. Einen ähnlichen Ansatz beschreiben Balaban u. a. [2005]. Dieser tauscht eine API auf Basis von Ersetzungsregeln gegen eine Entsprechung aus, ohne Adapter einzuführen. Beide oben genannten Ansätze vollziehen den Austausch einer API nur innerhalb einer Programmiersprache, weshalb sie nicht ohne weiteres in Portierungsprojekten eingesetzt werden können, in denen sich Ausgangs- und Zielsprache unterscheiden. Sie könnten allerdings erweitert werden, um sie auch für den Austausch von APIs in übersetztem Quellcode zu nutzen.

Ausgereifte Quellcode-Konvertoren wie Sharpen[10] bieten die Möglichkeit, Mappings zu konfigurieren, die die ursprünglichen API-Aufrufe bei der Übersetzung korrekt abbilden. Die Ausdrucksmächtigkeit dieser Mappings ist allerdings beschränkt. So können sie zwar Entsprechungen zwischen unterschiedlich benannten Operationen und Typen ausdrücken, Operationen mit unterschiedlichen Parameter-Reihenfolgen können jedoch nicht aufeinander abgebildet werden. Um solche Unterschiede zwischen den plattform-spezifischen APIs zu überwinden und die Übersetzung auch mit einfachen Mappings zu ermöglichen, können die angleichenden Entwurfsmuster eingesetzt werden, die in Kapitel 5 diskutiert werden.

Di Martino u. Cretella [2013] stellen einen Mechanismus vor, der solche Mappings zwischen APIs automatisch ermittelt. Dieser erfordert, dass Ausgangs- und Ziel-API manuell mit Annotationen versehen werden, die ihre Funktion beschreiben. Basierend auf diesen Annotationen werden dann gleichartige Schnittstellen in verschiedenen APIs einander zugeordnet. Der Nachteil dieses Ansatzes liegt darin, dass erst dann eine Zuordnung stattfinden kann, wenn mehrere APIs mit solchen Annotationen vorliegen, was aktuell nicht gegeben ist.

Nguyen u. a. [2014] und Zhong u. a. [2010] stellen im Kontext der Portierung Ansätze vor, die ohne solche manuellen Vorarbeiten auskommen. Sie nehmen Paare von ursprünglichem und portiertem Code entgegen und ordnen anhand statischer Quellcode-Analyse sich entsprechende API-Aufrufe einander zu. Diese Ansätze sind sehr wertvoll, sofern bereits portierter Quellcode und entsprechende Werkzeuge vorliegen, die diese Analyse durchführen. Da die Ansätze bislang nur prototypisch für die Programmiersprachen Java und C# implementiert wurden, werden sie nicht für die allgemeine Anwendung der Portierungsmethode vorausgesetzt, die hier entwickelt wird. Stattdessen wird angenommen, dass Entwickler die plattform-übergreifenden Mappings zwischen gleichartigen APIs in der Regel händisch konfigurieren.

3.5 Automatisierte Gewinnung von Trace Links

Um die Co-Evolution einander entsprechender Quellcode-Elemente zu vereinfachen, zielt die hier entwickelte Portierungsmethode darauf ab, plattform-übergreifende Trace Links einzuführen. Diese dokumentieren zum einen die Entsprechungsbeziehungen und zum anderen erlauben sie den Einsatz von Werkzeugen für plattform-übergreifende Aufgaben bei der Co-Evolution. In verschiedenen Forschungsgebieten wurden Techniken entwickelt, um Trace Links entweder bei der Erstellung der zu verknüpfenden

[10] Quellcode-Konverter zur Übersetzung von Java-Quellcode nach C#: `https://github.com/mono/sharpen`

Artefakte zu erzeugen oder nachträglich zu ermitteln [Grammel u. a. 2012] [Antoniol u. a. 2001] [Hübner u. Paech 2017].

Eines dieser Forschungsgebiete ist die modellgetriebene Softwareentwicklung. Dort besteht beim Einsatz von Modelltransformatoren der Bedarf, die Modellelemente der Eingabe mit den zugehörigen Ausgabeelementen durch Trace Links zu verknüpfen. Diese werden zum einen für Aufgaben wie Impact-Analyse, Modellsynchronisation, modellbasiertes Debugging oder zur Prüfung eines Modelltransformators genutzt [Czarnecki u. Helsen 2006, S. 634]. Zum anderen dienen sie als Input für weitere Modelltransformationen [Vanhooff u. a. 2007].

In der modellgetriebenen Entwicklung können diese Trace Links **erstens** automatisch vom Modelltransformator erzeugt werden. Die vorliegende Arbeit greift diesen Ansatz zur Erzeugung von Trace Links auf, da er keinen zusätzlichen Aufwand für portierende Entwickler verursacht. Dementsprechend werden Quellcode-Konvertoren hier um die Erzeugung von Trace Links erweitert, die die ursprünglichen Quellcode-Elementen mit ihren Übersetzungen verknüpfen.

Zweitens können die Entwickler zusätzliche Transformationsregeln definieren, die Trace Links generieren [Czarnecki u. Helsen 2006]. Dieser Ansatz wird hier nicht eingesetzt, um den zusätzlichen Aufwand für Entwickler zu vermeiden.

Drittens können Entwickler formale Zuordnungsregeln definieren, die zusammengehörige Elemente der Ein- und Ausgabe nach Abschluss einer Transformation identifizieren [Grammel u. a. 2012]. Auch dieser Ansatz fließt nicht in die vorliegende Arbeit ein, da er zusätzlichen Aufwand für Entwickler verursacht. Darüber hinaus ist er ungeeignet, weil bei der händischen Übersetzung von Quellcode keine formalen Beziehungen zwischen den ursprünglichen und übersetzten Code-Elementen gelten, die die eindeutige automatische Zuordnung mittels formaler Regeln zulassen.

In anderen Forschungsbereichen werden Trace Links zwischen Software-Artefakten erzeugt, die nicht aus einer automatischen Transformation entstanden sind. In diesen Fällen liegt zwischen den Artefakten genau wie bei der händischen Übersetzung von Quellcode keine formale Übersetzungsbeziehung vor, die eine eindeutige Zuordnung zulässt. Joorabchi u. a. [2015] schlagen beispielsweise ein Verfahren für den Vergleich von Versionen einer Anwendung für verschiedene Plattformen vor. Dabei beschreiben die Entwickler ein Nutzungsszenario, das in beiden zu vergleichenden Versionen automatisiert ausgeführt wird. Der auszuführende Quellcode wird instrumentiert und seine Ausführung wird aufgezeichnet. Aus den erzeugten Aufzeichnungen wird für jede der Implementationen ein Zustandsautomat abgeleitet, der ihr Verhalten dokumentiert. Inkonsistenzen zwischen den Implementationen werden dann durch den Vergleich dieser Automaten ermittelt. Die Zuordnung zwischen den Implementationen geschieht nicht in Bezug auf

einzelne Code-Elemente sondern in Bezug auf die Zustände der Automaten. Der Ansatz ist somit nicht unmittelbar für die Ermittlung plattformübergreifender Trace Links zwischen Code-Elementen geeignet. Er ist nur für ausführbaren Code anwendbar und erfordert entsprechende Werkzeuge für die Instrumentierung und automatisierte Ausführung. Auch die zusätzliche Arbeit der Entwickler für die formale Definition der Nutzungsszenarien soll hier vermieden werden.

Antoniol u. a. [2001] entwickeln einen Trace-Recovery-Mechanismus, der verschiedene Versionen einer Typdefinition einander zuordnet. Dazu wird der Code in die *Abstract Object Language* überführt, die von der konkreten Syntax der Programmiersprache abstrahiert. Darauf aufbauend setzen die Autoren einen Graph-Matching-Algorithmus ein, der die Eigenschaften der Typdefinitionen miteinander vergleicht und Ähnlichkeiten erkennt. Dieser Ansatz hat den Vorteil, dass er nicht auf eine Programmiersprache beschränkt ist, sodass er auch sprachübergreifend eingesetzt werden könnte. Er basiert auf dem Vergleich der Klassennamen und Methodensignaturen als Zeichenketten. Bei portiertem Code gibt es Unterschiede zwischen den Typen der zugrundeliegenden APIs und zwischen den Namenskonventionen der Ausgangs- und Zielimplementation. Es ist daher davon auszugehen, dass die Suche nach exakt gleichen Signaturen für portierten Code nicht anwendbar ist. Zudem ist der eingesetzte Vergleichsalgorithmus sehr rechenintensiv, sodass fraglich ist, ob er zum spontanen Auffinden plattform-übergreifender Entsprechungen geeignet ist.

Andere Trace-Recovery-Verfahren wurden entwickelt, um Quellcode-Elemente den zugehörigen Anforderungsdokumenten zuzuordnen. Hübner u. Paech [2017] zeichnen dazu die Tätigkeiten der Entwickler auf, die die dokumentierten Anforderungen umsetzen und dabei den Code und die zu verknüpfenden Dokumente nutzen und editieren. Grundsätzlich wäre es möglich, auch übersetzte Code-Elemente ihren Originalen zuzuordnen, indem man die zeitlich zusammenhängende Bearbeitung der Elemente beobachtet und analysiert. Da Entwickler zwei Code-Elemente aus verschiedenen Gründen gemeinsam bearbeiten, ist jedoch davon auszugehen, dass dieser Ansatz für die Erkennung plattform-übergreifender Trace Links nur bedingt effektiv ist. Zudem sind die Trace Links mit diesem Verfahren erst dann verfügbar, wenn bereits eine gemeinsame Bearbeitung von Original und Übersetzung stattgefunden hat. Es könnte allerdings zur Verbesserung von Traceability-Modellen mit anderen Verfahren kombiniert werden.

Des Weiteren werden Techniken des Information Retrieval für Trace Recovery eingesetzt, um natürlichsprachliche Dokumente wie Kapitel eines Handbuchs, oder Use-Case-Beschreibungen mit den implementierenden Quellcode-Elementen durch Trace Links zu verknüpfen [Antoniol u. a. 2002], [Hayes u. a. 2006], [De Lucia u. a. 2007],[Eyal-Salman u. a. 2013], [Gethers u. a. 2011], [Marcus u. Maletic 2001], [Asuncion u. a. 2010], [Panichella u. a. 2013]. Dazu werden natürlichsprachliche Dokument und Code-Elemente gleichermaßen als Vektoren dargestellt. Diese Vektoren erfassen zu jedem im Text bzw.

Code-Element auftretenden Wort, wie oft es im jeweiligen Dokument bzw. Code-Element enthalten ist. Sie werden gegebenenfalls mit Gewichtsvektoren multipliziert, um Wörter höher zu gewichten, die nur in wenigen Dokumenten enthalten sind und somit repräsentativer für ein Dokument sind als andere. Die Vektoren werden auf Basis eines Vergleichsalgorithmus miteinander verglichen, beispielsweise, indem der Kosinus des Winkels zwischen den Vektoren berechnet wird. Durch dieses Vorgehen werden Texte unterschiedlicher Sprachen vergleichbar, was in dieser Arbeit für die Ermittlung plattformübergreifender Trace Links zwischen Code-Elementen notwendig ist. Diese Arbeit übernimmt daher den Ansatz, Code-Elemente anhand ihrer Bezeichner zu repräsentieren und auf Basis der Häufigkeiten von Bezeichnern miteinander zu vergleichen.

Verschiedene sogenannter Topic-Modeling-Techniken wurden entwickelt, um die Dimensionalität der oben beschriebenen Wortvektoren zu verringern. Dazu werden die Wörter gemeinsamen Themen zugeordnet. Anstatt der Wortvektoren findet der Vergleich der Dokumente dann anhand von Themenvektoren statt, die erfassen, welche Themen im jeweiligen Dokument wie stark vertreten sind. Eine dieser Techniken ist das *Latent Semantic Indexing* (LSI). Es fasst alle Wortvektoren zunächst in einer gemeinsamen Matrix zusammen und führt eine Singulärwertdekomposition durch. Wörter, die häufig gemeinsam in Dokumenten auftreten, werden dabei derselben Dimension zugeordnet, die ein Thema repräsentiert. Dem Verfahren liegt die Annahme zugrunde, dass die Ähnlichkeit von Wörtern sich anhand ihres gemeinsamen Auftretens in Dokumenten zeigt. Die identifizierten Themen werden durch das Vorgehen nicht benannt. Es kann lediglich bestimmt werden, welche Wörter den größten Beitrag zu einem Thema haben. Diejenigen Themen, die die beste Unterscheidung der Vektoren zulassen, werden genutzt, um die ursprünglichen Wortvektoren der Dokumente in Themenvektoren zu überführen. Diese Vektoren sagen nun nicht mehr aus, wie repräsentativ die jeweiligen Wörter für das Dokument sind, sondern wie repräsentativ die jeweiligen *Themen* für das Dokument sind. Auf dieser Basis werden Dokumente nicht mehr anhand ähnlicher Worthäufigkeiten, sondern anhand gemeinsamer Themenschwerpunkte für ähnlich befunden. Der entscheidende Vorteil gegenüber dem Vergleich von Wortvektoren ist, dass Dokumente bei einer Suche auch dann als relevant für eine Suchanfrage erkannt werden, wenn sie keines der gesuchten Wörter enthalten, aber Wörter, die denselben Themen zugeordnet sind. Latent Semantic Indexing wurde in der Forschung genutzt, um verschiedene natürlichsprachliche Dokumente zu Quellcode-Ausschnitten zuzuordnen [Hayes u. a. 2006], [De Lucia u. a. 2007],[Eyal-Salman u. a. 2013], [Gethers u. a. 2011], [Marcus u. Maletic 2001].

Eine andere Topic-Modeling-Technik ist die sogenannte *Latent Dirichlet Allocation* (LDA), die ebenfalls für Trace Recovery eingesetzt wird [Asuncion u. a. 2010] [Panichella u. a. 2013]. Sie repräsentiert die Themen im Gegensatz zu LSI als Wahrscheinlichkeitsverte-

lungen[11]. Bei der Zuordnung von Use Case Beschreibungen zu Quellcode hat dieses Verfahren zu niedrigeren Precision- und Recall-Werten geführt als LSI [Oliveto u. a. 2010].

Lucia u. a. [2012] stellen fest, dass Topic-Modeling-Techniken wie LSI und LDA im Allgemeinen die Kernsemantik von Quellcode schlecht erfassen, während sie auf natürlichsprachlichen Texten sinnvoll anwendbar sind. Da in dieser Arbeit ausschließlich Verknüpfungen zwischen Quellcode-Elementen zu ermitteln sind, werden diese Techniken nicht angewendet. Stattdessen nutzt diese Arbeit die Erkenntnis von Lucia u. a. [2012] und gewichtet die Bezeichner eines Code-Elements entsprechend des programmiersprachlichen Konstrukts, das sie repräsentieren. So wird beispielsweise der Name einer Klasse im zugehörigen Wortvektor höher gewichtet als die Bezeichner der lokalen Variablen innerhalb der Klasse (vgl. Abschnitt 6).

Neben diesen Ansätzen für Trace Recovery werden auch Techniken des maschinellen Lernens eingesetzt, um das Auffinden zusammengehöriger Softwareartefakte zu automatisieren [Guo u. a. 2017] [Cleland-Huang u. a. 2010]. Dazu werden beispielsweise neuronale Netze anhand eines Datensatzes korrekter Trace Links trainiert, sodass sie zusammenhängende Artefakte erkennen können. Die entstehenden trainierten Mechanismen sind allerdings nur innerhalb derjenigen Domäne effektiv, aus der die Trainingsdaten stammen. Um den portierenden Entwicklern den zusätzlichen Aufwand für das Erheben domänenspezifischer Trainingsdaten zu ersparen, wird hier auf den Einsatz dieser Techniken verzichtet. Es ist allerdings denkbar, diese Techniken mit dem hier entwickelten Trace-Recovery-Mechanismus zu kombinieren, um die Qualität der vorgeschlagenen Trace Links für einzelne Domänen mit vorhandenen Trainingsdaten zu verbessern.

Eine Aufgabe, die dem Auffinden plattform-übergreifender Quellcode-Entsprechungen sehr ähnelt, ist das Erkennen von Code-Duplikaten. Udagawa [2013] unterteilt dieses Forschungsgebiet treffend in textbasierte, metrikbasierte, token-basierte und strukturbasierte Ansätze.

Textbasierte Ansätze erkennen Code-Duplikate, indem sie Quellcode-Ausschnitte als Zeichenketten miteinander vergleichen [Baker 1993] [Ducasse u. a. 1999]. Sie werden unter anderem zur Erkennung von unveränderten Code-Duplikaten eingesetzt. Beispielsweise vergleicht der Ansatz von Ducasse u. a. [1999] ganze Codezeilen miteinander und identifiziert exakte Übereinstimmungen. Bei der Portierung von Quellcode für eine andere Plattform wird der ursprüngliche Quellcode als Zeichenkette allerdings stark verändert. Es findet gegebenenfalls eine Übersetzung in eine andere Programmiersprache statt, es werden andere Basisdatentypen und APIs verwendet und andere Quelltextkonventionen angewendet. Vergleiche ganzer Quelltextzeilen sind daher für das Auffinden

[11] Eine umfassende Beschreibung würde den Rahmen dieses Abschnitts sprengen, sodass an dieser Stelle auf eine verständliche Erklärung unter folgender URL verwiesen wird: `https://medium.com/@lettier/how-does-lda-work-ill-explain-using-emoji-108abf40fa7d`

plattform-übergreifender Entsprechungsbeziehungen zwischen Code-Elementen gänzlich ungeeignet.

Metrikbasierte Ansätze zur Erkennung von Code-Duplikaten vergleichen im Unterschied dazu nicht unmittelbar den Quellcode als Text sondern berechnen Metriken, die den Quelltext charakterisieren. Duplikate werden dann anhand ähnlicher Werte für diese Metriken erkannt [Jiang u. a. 2007] [Kodhai u. a. 2010] [Raheja u. Tekchandani 2013] [Abd-El-Hafiz 2012]. Die eingesetzten Metriken messen in der Regel strukturelle Eigenschaften wie die Anzahl von Schleifen, Codezeilen oder Variablen. Insbesondere für die Erkennung von re-implementiertem Code, dessen Struktur sich gegebenenfalls vom ursprünglichen Vorbild unterscheidet, sind diese Methoden daher nicht geeignet.

Token-basierte Ansätze vergleichen Quellcode-Elemente anhand der in ihnen enthaltenen Wörter, insbesondere der verwendeten Bezeichner für Typen, Methoden und Variablen [Kamiya u. a. 2002] [Flores u. a. 2012] [Byalik u. a. 2015] oder anhand natürlichsprachlicher Beschreibungen des Quelltextes, etwa auf Plattformen wie Stack Overflow[12] [Sinai u. Yahav 2014]. Um die extrahierten Bezeichner miteinander zu vergleichen, werden genau wie bei den diskutierten Trace-Recovery-Ansätzen Vektoren miteinander verglichen, die die Häufigkeiten der auftretenden Bezeichner erfassen. So abstrahieren token-basierte Ansätze zur Duplikaterkennung von der zugrundeliegenden Sprachsyntax. Sie identifizieren nicht nur exakte Kopien eines Codeausschnitts als Duplikate, sondern entdecken auch semantische Code-Klone, die die gleiche Funktionalität erfüllen wie die ursprüngliche Version und dabei die gleichen Bezeichner nutzen. Sie setzen keine strukturellen Ähnlichkeiten voraus, um Code-Klone zu erkennen. Damit sind sie gut geeignet, um auch händische Übersetzungen eines Quelltextes in eine andere Programmiersprache zu erkennen. Nachteil dieser Techniken ist, dass strukturelle Informationen ignoriert werden, obwohl sie wichtige Hinweise auf den Zweck eines Code-Elements geben. In Abschnitt 6 dieser Arbeit wird daher ein token-basierter Mechanismus entwickelt, der die Bezeichner gemäß ihrer Position im abstrakten Syntaxbaum des Quelltextes gewichtet. So wird etwa berücksichtigt, dass ein Klassenname typischerweise die Funktionalität einer Klasse besser beschreibt als der Name einer beliebigen lokalen Variable innerhalb der Klasse. Zumindest ein Teil der strukturellen Information des Quelltextes wird so in die Bezeichner-Vektoren übernommen.

Strukturbasierte Ansätze zur Duplikaterkennung vergleichen Code-Elemente anhand ihrer Strukturen. Dazu bilden sie Graphen, die die Struktur eines Code-Elements beispielsweise durch seinen abstrakten Syntaxbaum repräsentieren [Baxter u. a. 1998]. Auf Basis dieser Graphen werden Duplikate durch den Einsatz von Vergleichsalgorithmen erkannt, die Metriken für die Ähnlichkeit dieser Graphen oder ihrer Teil-Graphen berechnen. Strukturbasierte Ansätze, die unmittelbar auf abstrakten Syntaxbäumen arbeiten,

[12] `https://stackoverflow.com/`

sind zum Auffinden plattform-übergreifender Entsprechungen nicht geeignet, da sich die Syntaxen der Ausgangs- und Zielsprache erheblich unterscheiden können.

Allerdings gibt es auch strukturbasierte Ansätze, die zu einem gewissen Grad sprachunabhängig arbeiten. Ein Beispiel dafür ist der Ansatz von Vislavski u. a. [2018]. Dieser arbeitet sprachunabhängig, indem die abstrakten Syntaxbäume zunächst in sprachunabhängige Bäume überführt werden, die lediglich die Typen der Knoten, wie z.B. *Variablenbezeichner* oder *ternärer Operator* erfassen, nicht aber die verwendeten Bezeichner. Vislavski u. a. [2018] finden so mit hoher Genauigkeit Codeduplikate mit veränderten Bezeichnern. Der Ansatz wurde bislang nur für die Klonerkennung innerhalb einer Programmiersprache evaluiert. Die erzeugten Strukturgraphen sind zwar sprachunabhängig, erfassen aber lediglich die Struktur des Quelltextes. Diese kann sich insbesondere bei Code-Elementen stark von der Struktur ihres Vorbilds unterscheiden, die zur Portierung re-implementiert wurden. Die re-implementierenden Entwickler können beliebige Strukturen und APIs mit unterschiedlichen Schnittstellen einsetzen. Es ist daher nicht anzunehmen, dass der Ansatz von Vislavski u. a. [2018] Entsprechungsbeziehungen zwischen händisch re-implementierten Code-Elementen und ihren Vorbildern in der Ausgangsimplementation zuverlässig erkennt.

Kraft u. a. [2008], Al-Omari u. a. [2012] und Avetisyan u. a. [2015] stellen strukturbasierte Ansätze zur Klonerkennung vor, die sprachübergreifend arbeiten, indem sie Kompilate für dieselbe Ausführungsumgebung miteinander vergleichen. Sie sind damit auf die Klasse der Sprachen eingeschränkt, die für dieselbe Ausführungsumgebung kompiliert werden können. Für die Zwecke dieser Arbeit sind sie somit ungeeignet.

Mishne u. De Rijke [2004] präsentieren einen Ansatz, der sowohl textuelle als auch strukturelle Ähnlichkeiten erfasst. Dazu werden zunächst Quellcode-Konzeptgraphen für die zu vergleichenden Code-Elemente erstellt. Diese erfassen zu einer Klasse beispielsweise die implementierten Methoden, Variablen und Rückgabe-Anweisungen im Code und stellen ihre strukturellen Beziehungen zueinander dar. Mishne u. De Rijke [2004] vergleichen im Gegensatz zu Vislavski u. a. [2018] nicht nur die Struktur der Quellcode-Elemente sondern auch die Ähnlichkeiten der verwendeten Bezeichner. Auch dieser Ansatz wurde ausschließlich in Bezug auf Code-Klone innerhalb einer Programmiersprache evaluiert. Grundsätzlich ist er aber vielversprechend für die plattform-übergreifende Suche nach semantisch ähnlichen Code-Elementen. Er birgt allerdings zwei entscheidende Nachteile. Erstens geben die Autoren nicht an, wie die vielen Parameter der Methode zu wählen sind. Diesbezügliche Nachfragen blieben unbeantwortet. Zweitens ist die entwickelte Ähnlichkeitsberechnung extrem komplex und benötigt laut der Autoren bereits für kleine Mengen zu vergleichender Elemente sehr viel Rechenzeit [Mishne 2003, S. 50]. Für potenziell große Portierungsprojekte mit vielen zu vergleichenden Code-Elementen ist sie damit nicht geeignet. Ihr Einsatz zur spontanen Suche nach Trace Links, auch für kürzlich geänderte Code-Elemente, ist dadurch ebenfalls ausgeschlossen.

3.6 Aktualisierung von Traceability-Modellen

Um ein Traceability-Modell langfristig zu nutzen, muss es aktualisiert werden, wenn die Softwareartefakte, die es verknüpft, geändert wurden. Durch das Hinzufügen, Löschen, Ersetzen, Aufspalten oder Zusammenführen von Elementen, die durch Trace Links verknüpft sind, werden Änderungen am gespeicherten Traceability-Modell notwendig [Mäder u. a. 2009]. Die manuelle Aktualisierung der Trace Links ist aufwendig und fehleranfällig, sodass eine automatisierte Aktualisierung angestrebt werden sollte [Maro u. a. 2016] [Seibel u. a. 2010]. Grundlegend sind drei Strategien für die automatisierte Aktualisierung von Traceability-Modellen zu unterscheiden [Maro u. a. 2016].

Erstens können die Änderungen an verknüpften Elementen zustandsbasiert erkannt und Änderungsbedarfe für das Traceability-Modell erkannt werden. Dazu werden die verknüpften Artefakte nach einer Änderung mit ihren Versionen vor der Änderung verglichen und die durchgeführten Änderungsoperationen werden aus diesem Vergleich abgeleitet [Rahimi u. a. 2016][Murta u. a. 2008]. Mäder u. Gotel [2012] identifizieren dabei den Nachteil, dass strukturelle Änderungen wie Ersetzungen, Aufteilung oder Zusammenführung verknüpfter Elemente auf Basis des Versionsvergleichs nicht korrekt erkannt werden können. Zudem werden die notwendigen Aktualisierungen nicht so konkret erkannt, dass sie vollständig automatisiert umgesetzt werden können.

Die **zweite** Strategie beobachtet die Änderungen an den verknüpften Artefakten selbst, sodass die tatsächlichen Änderungsoperationen festgehalten werden [Mäder u. Gotel 2012]. Aus den aufgezeichneten Änderungsoperationen werden dann die notwendigen Änderungen des Traceability-Modells abgeleitet. Die identifizierten Änderungsbedarfe müssen, wie auch im ersten Ansatz vom Entwickler geprüft und konkretisiert werden, sodass dieser Ansatz hier nicht verfolgt wird.

Die **dritte** Strategie verwirft das ursprüngliche Traceability-Modell vollständig und erzeugt ein neues, wenn die verknüpften Elemente geändert wurden [Seibel u. a. 2010]. Dieser Ansatz hat den Vorteil, dass die Änderungen an den verknüpften Elementen nicht konkret beobachtet werden müssen und dass er vollständig automatisiert werden kann. Er kann allerdings nur dann eingesetzt werden, wenn die Erzeugung der Trace Links automatisiert abläuft und ein Traceability-Modell in ausreichender Qualität erstellt wird. Manuelle Ergänzungen oder Korrekturen gehen bei diesem Wartungsansatz verloren, sobald die Trace Links neu erzeugt werden. Diese Arbeit verfolgt eine vollautomatisierte Erzeugung von Trace Links, teils durch einen Trace-Capture-Mechanismus, teils durch Trace Recovery. Manuelle Änderungen am Traceability-Modell sind nicht vorgesehen, sodass diese Strategie anwendbar ist.

3.7 Sprachübergreifende Konsistenz und gekoppelte Änderungen

Ziel dieser Arbeit ist es, die konsistente Co-Evolution mehrerer Implementationen einer Software durch konzeptuelle Ähnlichkeiten zwischen der ursprünglichen und der portierten Implementation zu vereinfachen. Lösungsansätze zur Erhaltung von Konsistenz zwischen Softwareartefakten in unterschiedlichen Sprachen sind außerhalb der plattform-übergreifenden Entwicklung durchaus bekannt. Im Kontext der modellbasierten Softwareentwicklung besteht beispielsweise die Herausforderung, die Konsistenz zwischen verschiedenen Modellen bzw. zwischen Modellen und anderen Softwareartefakten zu bewahren. Lämmel [2004] beschreibt in diesem Kontext automatisierte *gekoppelte Transformationen* zweier Softwareartefakte A und B, deren Konsistenz zueinander gewahrt werden soll. Er nennt Optionen, um gekoppelte Transformationen unter Wahrung der Konsistenz durchzuführen. Zwei dieser Optionen werden in der vorliegenden Arbeit angewendet. Die erste Option setzt voraus, dass B aus A generiert wurde. Um die Konsistenz zwischen A und B nach einer Änderung wiederherzustellen, wird B nach der Änderung gelöscht und erneut aus A generiert. Diesen Ansatz greift die vorliegende Arbeit auf und strebt bei der automatischen Übersetzung von Quellcode für die Zielplattform an, konvertierbare Elemente vollständig von nicht konvertierbaren Elementen zu Trennen. Dadurch können Entwickler bei der Evolution von händisch portiertem Code gemäß des zweiten von Lämmel [2004] beschriebenen Szenarios vorgehen. Dabei werden A und B synchron transformiert. Dieser Ansatz wird im Kontext der vorliegenden Arbeit durch das Konzept der plattform-übergreifenden Refactorings verfolgt. Ein Refactoring wird dabei in gleicher Weise auf einem Element der Ausgangsimplementation und auf seiner Entsprechung in der Zielimplementation durchgeführt (vgl. Abschnitt 7.3).

Eine weitere Möglichkeit, Ausgangs- und Zielimplementation konsistent zu halten, besteht in sogenannten bidirektionalen Transformationen. Konvertoren für Softwareartefakte werden dazu so beschrieben, dass sie nicht nur eine Konversion von Artefakten der Sprache S_1 in die Sprache S_2 durchführen können, sondern auch eine rückwärtige Konversion von S_2 nach S_1 [Abou-Saleh u. a. 2018]. Der Ansatz wird neben der bidirektionalen Übersetzung von Modellen unter anderem auch zur Synchronisierung von Benutzerschnittstellen für unterschiedliche Plattformen und zur Konversion von Daten in unterschiedliche Formate genutzt [Czarnecki u. a. 2009]. Anwendungen zur Definition von Quellcode-Konvertoren sind allerdings nicht bekannt. Die meisten verfügbaren Quellcode-Konvertoren arbeiten unidirektional, sodass die hier entwickelte Portierungsmethode keine bidirektionalen Quellcode-Konvertoren voraussetzt.

Auch außerhalb der modellbasierten Softwareentwicklung existieren Ansätze für die Co-Evolution über die Grenzen einer Programmiersprache hinweg. Bereits innerhalb einer Implementation bestehen häufig Abhängigkeiten zwischen Quellcode-Elementen,

die in unterschiedlichen Programmiersprachen definiert sind. Ein Beispiel sind XML-Beschreibungen grafischer Benutzeroberflächen, die formal an Daten haltende Klassen in Java gebunden sind. Mayer u. Schroeder [2012] beschreiben in diesem Kontext die Erhebung und Nutzungspotentiale sprachübergreifender Trace Links zwischen Code-Elementen innerhalb einer Implementation. Die Erhebung basiert auf formalen, benutzerdefinierten Regeln. Solche Verbindungen sind inzwischen Teil vieler IDEs, wie zum Beispiel Android Studio. Sie werden erfolgreich genutzt, um die konsistente Änderung voneinander abhängiger Code-Elemente in verschiedenen Sprachen zu gewährleisten und zwischen ihnen zu navigieren. Die von Mayer u. Schroeder [2012] aufgezeigten Nutzungspotentiale für Navigation und Refactoring sind auf den Kontext von Portierungsprojekten übertragbar. Kapitel 7 behandelt die Nutzung plattform-übergreifender Trace Links in der Co-Evolution von Ausgangs- und Zielimplementation.

Heutige Entwicklungsumgebungen bieten automatisierte Refactorings jeweils für mehrere Sprachen an. Jemerov [2008] sieht den Bedarf, die abstrakten Gemeinsamkeiten dieser Refactorings so zu implementieren, dass alle sprachspezifischen Konkretisierungen ein gemeinsames Kernverhalten definieren. Dadurch arbeiten die sprachspezifischen Ausprägungen eines Refactorings zu einem gewissen Grad gleichartig. Entwickler können dadurch ihre Erfahrungen bezüglich der Funktionsweise eines Refactoring-Werkzeugs leichter auf dessen Einsatz in einer anderen Programmiersprache übertragen. Mens u. a. [2003] identifizieren entsprechend den Trend, Refactorings auf sprachunabhängiger Ebene abstrakt zu beschreiben. Tichelaar [2001] führt dazu eine Modellierungssprache für objektorientierte Programmiersprachen ein, auf deren Basis Reengineering- und Refactoring-Aufgaben sprachübergreifend partiell definiert werden können. Sprachspezifische Teildefinitionen bleiben allerdings notwendig. Strein u. a. [2006] präsentieren einen neuen Ansatz zur Definition sprachübergreifender Refactorings innerhalb einer Anwendung. Auch dieser basiert auf einem sprachübergreifenden Architekturmodell, auf dem die strukturellen Veränderungen beschrieben werden. Ansätze wie diese sind sehr vielversprechend und können Anwendung bei plattform-übergreifenden Restrukturierungen finden, wie sie in dieser Arbeit unterstützt werden. Dazu können plattformübergreifende Trace Links genutzt werden, um plattform-übergreifend äquivalente Refactorings auf einander entsprechende Quellcode-Elementen anzuwenden und gleichzeitig die strukturellen Entsprechungen zwischen den Implementationen zu bewahren.

Um Inkonsistenzen zwischen zwei Implementationen einer Software zu entdecken, verfolgen manche Arbeiten einen Black-Box-Ansatz. Sie beobachten das Verhalten beider Implementationen an den Schnittstellen zur Umgebung und vergleichen die Ausgaben beider Implementationen bei äquivalenten Eingaben. Mesbah u. Prasad [2011] stellen zum Beispiel ein Werkzeug zur automatisierten Analyse von Verhaltensunterschieden zwischen Browsern bei der Ausführung einer Anwendung vor. Es nutzt die untersuch-

te Anwendung automatisiert in verschiedenen Browsern, extrahiert jeweils einen Zustandsautomaten und vergleicht diese miteinander. Unterschiedliche Werkzeugkästen zum Testen von browser-übergreifend eingesetzten Anwendungen sind bereits im Einsatz [Chapman 2011]. Roy Choudhary [2014] stellt fest, dass diese Werkzeuge noch fehlerhaft sind und weitere Entwicklung notwendig bleibt. Solche Werkzeuge sind naturgemäß auf den Einsatz in browserbasierten Anwendungen beschränkt, die eine gemeinsame Codebasis zur Ausführung in verschiedenen Browsern nutzen. Zudem erkennen sie ausschließlich die Unterschiede aus Sicht eines Beobachters bei der Ausführung. Sie geben keine Hinweise auf die Code-Elemente, die diese Inkonsistenzen verursachen.

Roy Choudhary u. a. [2014] führen einen Ansatz zur Erkennung gleicher Funktionalitäten in Softwaresystemen für unterschiedliche Plattformen ein. Dazu werden HTTP-Requests verglichen, die diese Systeme bei Anfragen an zugehörige Webservices ausführen. Dieser Ansatz ist wertvoll in Projekten, bei denen Unklarheit darüber herrscht, welche Funktionen für verschiedene Plattformen bereits implementiert sind. Dies kann in Portierungsprojekten dazu genutzt werden, noch zu portierende Funktionen zu erkennen.

4 Ablauf der Portierungsmethode

In den folgenden Abschnitten wird eine Methode entwickelt, die einen Ablauf für Portierungsprojekte vorgibt. Sie leitet die portierenden Entwickler durch einen iterativen Prozess, der ein reflektiertes, risikoarmes und teil-automatisiertes Vorgehen beschreibt, welches die Implementationen für die ursprüngliche Plattform und für die Zielplattform systematisch aneinander angleicht. Dazu werden in Abschnitt **??**zunächst grundlegende Entscheidungen diskutiert, die den Ablauf der Portierung beeinflussen, ehe dieser Ablauf in Abschnitt 4.2 im Überblick dargestellt wird. Abschnitt 4.3 führt ein Fallbeispiel zur Illustration dieses Ablaufs ein. Die Abschnitte 4.4 und 4.5 legen die durchzuführenden Schritte innerhalb der zwei Phasen des Ablaufs dar und konkretisieren diese Schritte anhand des Fallbeispiels.

4.1 Strategische Entscheidungen mit Auswirkung auf den Ablauf von Portierung und Co-Evolution

Sowohl der Portierungsprozess als auch die Co-Evolution werden maßgeblich von zwei Entscheidungen beeinflusst. Die erste Entscheidung betrifft die angestrebten plattformübergreifenden Entsprechungen. Beispielsweise können die Entwickler anstreben, dass die portierten Klassen ihren Vorbildern Anweisung für Anweisung entsprechen. Änderungen dieser Anweisungen können während der Co-Evolution dann in beiden Implementationen gleichartig umgesetzt oder gar automatisch übertragen werden. Alternativ können die Entwickler entscheiden, dass die portierten Klassen zwar Operationen mit gleichen Signaturen definieren, diese aber unterschiedlich implementieren. Die Entwickler können als dritte Option lediglich die Zuständigkeiten der Typdefinitionen bei der Portierung übernehmen, also den Entwurf auf Typebene. In Abschnitt 4.1.1 wird die Anwendung dieser unterschiedlichen Abstraktionsgrade diskutiert.
Zweitens ist zu entscheiden, wie Trace Links zwischen den ursprünglichen Elementen und ihren portierten Entsprechungen erhoben werden und wie die erhobenen Traceability-Modelle gewartet werden. Diese Entscheidung wird in Abschnitt 4.1.2 getroffen.

4.1.1 Wahl des Abstraktionsgrades der angestrebten Entsprechungen

Wenn die Quellcode-Elemente der ursprünglichen Implementation in Entsprechungen für die Zielplattform überführt werden, können dabei verschiedene Abstraktionsgrade angewendet werden. In dieser Arbeit werden drei Abstraktionsebenen unterschieden:

Erstens können die portierenden Entwickler nur von der Syntax der jeweiligen Programmiersprache abstrahieren. In der Zielimplementation entsteht dann Quellcode, der dem ursprünglichen Code Anweisung für Anweisung entspricht. **Zweitens** können die Entwickler von einzelnen Anweisungen abstrahieren, aber die Schnittstelle eines Typs übernehmen. Die Typdefinitionen der Zielimplementation entsprechen ihren Vorbildern dann in Bezug auf öffentlich zugreifbare Operationen und Attribute. **Drittens** können sie zusätzlich von diesen Schnittstellen abstrahieren, und lediglich die Zuständigkeiten der Typdefinitionen übernehmen.

Grundsätzlich haben sowohl die Portierung, die Anweisung für Anweisung vorgeht, als auch die abstrakten Varianten Vor- und Nachteile. Ein offensichtlicher Vorteil der konkreten Portierung ist, dass die Entwickler die ursprünglichen Anweisungen lediglich in die Syntax der Zielsprache übersetzen müssen. Das kann gegebenenfalls sogar automatisiert mit einem Quellcode-Konverter geschehen. Auch während der Co-Evolution können die Entwickler Änderungen an den Elementen der Ausgangsimplementation in die Zielimplementation übertragen, ohne neue Änderungskonzepte entwickeln und umzusetzen zu müssen. Wird hingegen nur die Schnittstelle einer Klasse in die Zielplattform überführt, so müssen die zugehörigen Methodenrümpfe neu implementiert und während der Co-Evolution separat weiterentwickelt werden.

Ein Vorteil der Abstraktionen ist allerdings, dass die konkreten Schnittstellen der eingesetzten APIs in der Zielimplementation nicht nachgebildet werden müssen. Zudem haben die Entwickler mehr Freiheiten bei der Entwicklung der Zielimplementation, wenn sie von den Details der ursprünglichen Implementation abstrahieren. Sie können die für die Zielplattform angemessenen APIs und Sprachkonstrukte verwenden, auch wenn diese sich von den ursprünglich eingesetzten unterscheiden. Sie müssen weniger Aufwand betreiben, um die Schnittstellen der Zielplattform an die ursprünglich genutzten Schnittstellen anzupassen. Maßgeblich für die Wahl eines angemessenen Abstraktionsgrades ist also unter anderem die Dichte der Abhängigkeiten zu plattformspezifischen APIs. Um dies zu konkretisieren, werden hier zwei Extrema genannt:

Code, der Konzepte der Anwendungsdomäne implementiert, hat meistens kaum kritische Plattformabhängigkeiten und kann automatisch übertragen werden, sofern ein ausgereifter Konverter verfügbar ist. Somit entstehen Entsprechungen zwischen konkreten Anweisungen. Steht kein Quellcode-Konverter zur Verfügung, können einzelne

Anweisungen abweichend implementiert werden, um aufwendige Anpassungen der genutzten APIs zu vermeiden.

Code-Elemente mit technischen Aufgaben, etwa zur Realisierung der Benutzerschnittstelle und Infrastruktur sind das andere Extrem. Sie haben in der Regel viele Plattformabhängigkeiten und können nicht automatisiert übersetzt werden [Terekhov 2001]. Stattdessen sollten sie ihrer Re-Implementation auf der Zielplattform in Bezug auf die Zuständigkeiten der Typen entsprechen. Darüber hinaus sollten Entsprechungen bzgl. der Schnittstellen angestrebt werden, sofern die Schnittstellen nicht durch Vererbungsbeziehungen zu plattform-spezifischen Typen vorgegeben sind.

Abgesehen von diesen zwei Extrema hängt der angemessene Abstraktionsgrad stark vom verfügbaren Konversionswerkzeug ab. Steht ein ausgereifter Quellcode-Konverter zur Verfügung, der Abhängigkeiten problemlos anhand von API-Mappings übersetzt, so sollte dieser genutzt werden, um Entsprechungen auf der Ebene von Anweisungen zu erzielen.

Muss der Code allerdings händisch portiert und folglich auch während der Co-Evolution händisch bearbeitet werden, so sollte immer dann eine Abstraktion angewendet werden, wenn sie in der Zielimplementation eine besser wartbare Version ermöglicht. Entsprechungen zwischen einzelnen Anweisungen und Kontrollstrukturen sollten dann verworfen werden. Die Schnittstellen der ursprünglichen Typen sollten allerdings erhalten bleiben. Eine Ausnahme bilden öffentliche Schnittstellen, die die portierte Implementation für externen Quellcode bereitstellt. Diese können sich von ihren Vorbildern unterscheiden, wenn dies der einfacheren Nutzung durch externen Code dient.

Vom Entwurf der Typdefinitionen sollten portierende Entwickler nicht abstrahieren. Typdefinitionen wie Klassen, Interfaces und Enumerations sind der Kern eines objektorientierten Entwurfs und bilden somit die Basis zum Verständnis des Codes [Moreno u. a. 2013]. Um die erwünschte Erfahrungsportabilität zwischen ursprünglicher und portierter Implementation zu erzielen, sollte der Entwurf auf Typ-Ebene also erhalten bleiben.

4.1.2 Wahl der Strategien zur Erhebung und Aktualisierung plattform-übergreifender Trace Links

Die hier entwickelte Portierungsmethode soll die Co-Evolution der ursprünglichen und portierten Implementation im Anschluss an die Portierung vereinfachen. Dazu sollen Trace Links erhoben werden, die einander entsprechende Code-Elemente plattformübergreifend verknüpfen. Wird beispielsweise die Klasse `Fahrzeug` aus der ursprünglichen Implementation in eine gleichnamige Entsprechung in die Zielimplementation portiert, so sollten diese beiden Klassen durch einen Trace Link verknüpft werden, der

ihre Entsprechungsbeziehung dokumentiert. Es ist festzulegen, wie solche Trace Links im Rahmen der Portierung erhoben werden sollen.

Wie bereits in Abschnitt 2.5 einführend erläutert, gibt es grundsätzlich zwei Ansätze zum Erheben von Trace Links. Der erste Ansatz heißt *Trace Capture*. Er erzeugt einen Trace Link unmittelbar, wenn das abhängige Artefakt, in diesem Fall die portierte Klasse `Fahrzeug`, in der Zielimplementation erzeugt wird. Der zweite, als *Trace Recovery* bezeichnete Ansatz erhebt die Trace Links nachträglich. Der Trace Link wird also erst erzeugt, wenn die portierte Klasse bereits existiert.

Die Entscheidung, welcher Ansatz im Kontext der Portierung anzuwenden ist, muss sich erstens daran orientieren, welcher der beiden Ansätze Traceability-Modelle mit höherer Qualität liefert. Zweitens soll grundsätzlich vermieden werden, dass die erhobenen Traceability-Modelle manuell gewartet werden müssen. Die manuelle Wartung von Traceability-Modellen ist aufwendig und naturgemäß fehleranfällig. Entwickler behandeln zudem bei der manuellen Wartung der Trace Links nicht alle entstandenen Inkonsistenzen zwischen Traceability-Modell und den verknüpften Artefakten [Mäder u. a. 2009] [Maro u. a. 2016]. Der Effekt zunehmender Inkonsistenzen zwischen Code und manuell gewarteten Modellen ist ein seit langem beobachtetes Phänomen, das hier vermieden werden soll [Fiutem u. Antoniol 1998] [Forward u. Lethbridge 2008] [Gorschek u. a. 2014]. Die Traceability-Modelle sollen daher im Kontext dieser Arbeit automatisiert erhoben und gewartet werden.

Werden Quellcode-Elemente automatisch mithilfe eines Konverters übersetzt, so kann ebenfalls automatisch ein vollständiges Traceability-Modell generiert werden. Dazu kann der Konverter so erweitert werden, dass er neben der Übersetzung auch ein Traceability-Modell erzeugt, das sämtliche Entsprechungsbeziehungen zwischen den Eingabeelementen und ihren Übersetzungen erfasst. Die Wartung der so gewonnenen Traceability-Modelle ist unproblematisch, da Änderungen am ursprünglichen Code durch erneute Anwendung des Konverters übertragen werden. Bei dieser erneuten Konversion kann das alte Traceability-Modell durch ein neu generiertes ersetzt werden. Dies ist allerdings nur möglich, wenn tatsächlich vollautomatisiert übersetzt wurde. Manuelle Änderungen an der Übersetzung können dazu führen, dass die erfassten Entsprechungsbeziehungen ungültig werden. Trace Capture wird im Kontext dieser Arbeit daher nur auf vollständig konvertierbaren Quellcode angewendet. Abschnitt 4.5.1 entwickelt ein entsprechendes Konzept zur Erweiterung von Quellcode-Konvertoren um einen Trace-Capture-Mechanismus.

Quellcode-Elemente, die nicht von Quellcode-Konvertoren mit entsprechender Erweiterung übersetzt werden können, müssen nachträglich per Trace Recovery mit ihren portierten Entsprechungen verknüpft werden. Die so erhobenen Trace Links werden nicht

gespeichert und nicht manuell gewartet. Um manuellen Aufwand und zunehmende Inkonsistenzen zu vermeiden, werden sie stattdessen stets neu erhoben, wenn sie genutzt werden sollen. Dabei ist damit zu rechnen, dass das automatisch erhobene Traceability-Modell in gewissem Maße inkorrekt und unvollständig ist. Auf Basis dieser Anforderungen wurde ein Trace-Recovery-Mechanismus entwickelt, der als Suchverfahren fungiert, das zu einem gewählten ursprünglichen Code-Element potenzielle Entsprechungen als Suchergebnisse ermittelt. Der Mechanismus sollte korrekte Trace Links möglichst weit vorne in die Ergebnisliste einordnen, sodass ein Entwickler die gesuchte Entsprechung schnell identifizieren kann. Abschnitt 6 beschreibt die Entwicklung eines solchen Mechanismus.

4.2 Überblick über den Ablauf der Portierungsmethode

Abbildung 4.1 bietet einen Überblick über den Ablauf der Portierung.

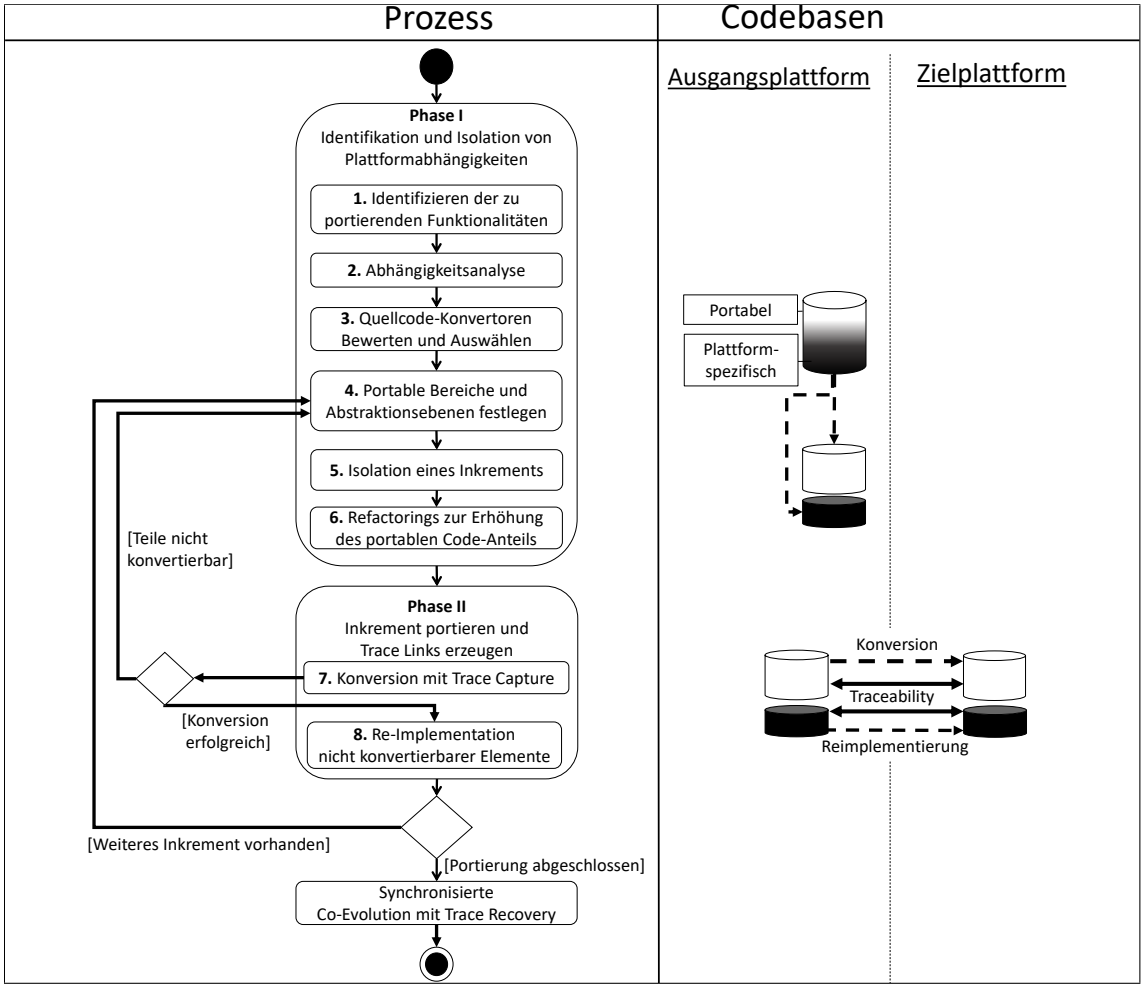

Abbildung 4.1: Übersicht über den Portierungsprozess und die Veränderung der Codebasen

Die linke Seite stellt den Prozess der Portierung als Aktivitätsdiagramm dar. Auf der rechten Seite ist die Portierung anhand der Codebasen illustriert. Der Ablauf Portierungsmethode lässt sich grob in zwei Phasen aufteilen, die in den folgenden Abschnitten beschrieben werden:

Phase I

Zunächst identifizieren die Entwickler die zu portierenden Funktionalitäten der ursprünglichen Implementation. Im Anschluss werden die Abhängigkeiten zur ursprünglichen Plattform erhoben. Im dritten Schritt werden die zur Verfügung stehenden Quellcode-Konvertoren bewertet und es wird entschieden, welcher Konverter eingesetzt werden soll. Darauf aufbauend legen die Entwickler im vierten Schritt fest, welche Abstraktionen bei der Portierung der Code-Elemente anzuwenden sind. Dies kann für

verschiedene Bereiche der ursprünglichen Codebasis unterschiedlich entschieden werden.

Um diese Entscheidung frühzeitig reflektieren zu können, folgt die Portierungsmethode einem iterativen Vorgehen. Dazu isolieren die Entwickler im fünften Schritt ein zu portierendes Inkrement von der ursprünglichen Codebasis. Ein solches Inkrement realisiert jeweils eine abgeschlossene, testbare Funktionalität. Das aktuelle Inkrement wird im sechsten Schritt so restrukturiert, dass die angestrebten Entsprechungsbeziehungen ermöglicht werden. Dazu ist es gegebenenfalls nötig, Code-Elemente aufzuspalten, um Elemente voneinander zu trennen, die unter Anwendung verschiedener Abstraktionsgrade portiert oder re-implementiert werden sollen.

Phase II
In der zweiten Phase werden die gewählten Konversionswerkzeuge auf etwaige konvertierbare Quellcode-Elemente angewendet. Dabei erzeugen sie idealerweise bereits Trace Links zwischen den ursprünglichen Quellcode-Elementen und ihren Übersetzungen. Im Anschluss werden die nicht konvertierbaren Code-Elemente händisch portiert. Dabei sind die festgelegten Abstraktionen für Entsprechungsbeziehungen zu berücksichtigen, die in der ersten Phase festgelegt wurden. Im weiteren Verlauf der Portierung wiederholt sich der Prozess, bis die ursprüngliche Implementation vollständig portiert ist.

Synchronisierte Co-Evolution
Schließlich geht die Entwicklung in die Phase der synchronisierten Co-Evolution über. Dabei nutzen die Entwickler die erzielten Entsprechungsbeziehungen und die erhobenen Trace Links werkzeuggestützt. Bislang nicht durch plattformübergreifende Trace Links verknüpfte Code-Elemente werden ad-hoc per Trace Recovery mit ihren Entsprechungen verbunden.

4.3 Fallbeispiel zur Illustration der Portierungsmethode

Um die Anwendung der hier entwickelten Portierungsmethode zu veranschaulichen, wird das Beispiel einer in Java implementierten Android-App eingeführt, die für das Betriebssystem iOS portiert werden soll. Die Zielimplementation soll dabei die Programmiersprache Swift einsetzen. Die Anwendung trägt den Namen *Bulky Waste Companion*. Mit ihr können Nutzer die Entsorgung ihres Sperrmülls durch einen Entsorgungsdienstleister planen. Dieser stellt seinen Kunden Sperrmüll-Container bereit und transportiert diese zu einem vereinbarten Zeitpunkt wieder ab.

Der Funktionsumfang der Anwendung umfasst die folgenden fünf Hauptfunktionalitäten:

1. **Berechnung von Containermaßen:** Nutzer können die Maße ihrer zu entsorgenden Gegenstände eingeben, woraufhin die Anwendung eine passende Containergröße und den zugehörigen Preis anzeigt.

2. **Terminvereinbarung:** Sie können anschließend einen Termin für die Bereitstellung des Containers wählen. Die verfügbaren Termine werden dazu vom Webservice des Entsorgungsdienstleisters verwaltet.

3. **Buchung:** Ferner können sie den benötigten Container über die Anwendung buchen.

4. **Freigabe ungenutzter Kapazitäten:** Etwaiges ungenutztes Containervolumen können die Nutzer freigeben, um es benachbarten Nutzern zur Verfügung zu stellen.

5. **Containerkarte:** Die so veröffentlichten freien Kapazitäten werden auf einer Straßenkarte verzeichnet, sodass andere Nutzer sich zur Mitnutzung eines Containers anmelden können.

Abbildung 4.2 stellt die Architektur dar, mit der diese Funktionalitäten umgesetzt wurden.

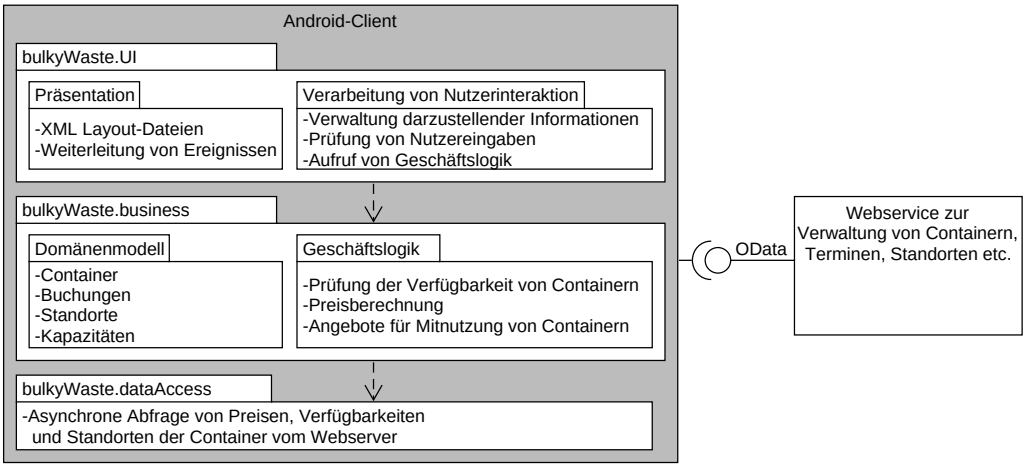

Abbildung 4.2: Architektur des Bulky Waste Companion

Die zu portierende Android-Anwendung ist auf der linken Seite der Abbildung zu sehen, der Webserver des Entsorgungsunternehmens auf der rechten. Die Android-Anwendung ist in drei Schichten unterteilt, die je ein eigenes Paket definieren. Die Nutzerinteraktionsschicht im Paket `bulkyWaste.UI` ist in zwei Unterpakete unterteilt. Das eine beinhaltet Layout-Dateien, die die Anordnung der eingesetzten Oberflächenelemente auf dem Display definieren, sowie Klassen, die diese Layouts instanziieren und Interaktionsereignisse weiterleiten. Das zweite Unterpaket enthält Klassen, die die anzuzeigenden Informationen aktualisieren, Nutzereingaben validieren und das entsprechende Verhalten der Geschäftslogik anstoßen.

Die Geschäftslogik-Schicht ist im Paket `bulkyWaste.business` implementiert. Es enthält wiederum zwei Unterpakete. Das eine umfasst die Klassen des Domänenmodells, die beispielsweise Container, Buchungen und Standorte repräsentieren. Das zweite beinhaltet Dienstklassen, die beispielsweise passende Containergrößen und -preise ermitteln, Verfügbarkeiten prüfen und den Zugriff auf Standorte freigegebener Container erlauben.

Klassen der Datenzugriffsschicht sind im Paket `bulkyWaste.dataAccess` enthalten. Sie ermöglichen den Zugriff auf den Webservice des Entsorgungsdienstleisters. Dazu nutzen sie unter anderem die Klasse `AsyncTask` der Android-API sowie eine zusätzliche Programmbibliothek, die das Protokoll OData[1] umsetzt.

4.4 Phase 1: Analyse, Planung und Refactoring

4.4.1 Schritt 1: Identifizieren der zu portierenden Funktionalitäten

Zunächst legen die portierenden Entwickler fest, welche Funktionalitäten der zu portierenden Software in die Zielplattform übertragen werden sollen. Dies sind gegebenenfalls nicht alle Funktionalitäten der ursprünglichen Implementation. Beispielsweise kann die ursprüngliche Plattform Sensoren und Aktuatoren haben, die auf der Zielplattform nicht zur Verfügung stehen. Funktionalitäten der Software, die diese Hardware voraussetzen, können nicht portiert werden.

Das Ergebnis dieses ersten Schritts ist eine Aufstellung der zu portierenden Funktionalitäten. Im Beispiel des Bulky Waste Companion beschließen die Entwickler, sämtliche der fünf Hauptfunktionalitäten zu portieren.

4.4.2 Schritt 2: Abhängigkeitsanalyse

Abhängigkeiten zur ursprünglichen Plattform behindern die Übertragung der bestehenden Code-Strukturen in die Zielplattform. Wie in Abschnitt 4.1.1 diskutiert, sind diese Abhängigkeiten maßgeblich für die Entscheidung, ob eine Typdefinition Anweisung für Anweisung in die Zielimplementation überführt wird, oder ob nur ihre Schnittstelle oder Zuständigkeit nachgebildet wird. Die Entwickler müssen daher im zweiten Schritt den Code identifizieren, der die zu portierenden Funktionalitäten umsetzt und seine Abhängigkeiten zur Plattform ermitteln. Dazu zählen unter anderem Aufruf-, Vererbungs- und Instanziierungsbeziehungen zu Typen in plattform-spezifischen APIs. Relevant sind vor

[1] `http://www.odata.org/`

allem die Abhängigkeiten zur API der Programmiersprache und des Betriebssystems sowie zu externen Programmbibliotheken von Drittanbietern. Darüber hinaus sind Abhängigkeiten zu Code-Elementen zu erheben, die nicht portiert werden sollen.

Für diese Analyse können die portierenden Entwickler zum einen Konfigurationsdateien nutzen, in denen die Abhängigkeiten des Entwicklungsprojekts verwaltet werden. Zum anderen können sie Werkzeuge einsetzen, um kritische Import-Anweisungen zu identifizieren. Die für den Bulky Waste Companion genutzte Entwicklungsumgebung Android Studio enthält ein solches Analysewerkzeug, mit dem sämtliche Abhängigkeiten aufgelistet werden. So identifizieren die Entwickler im gesamten Code der App Abhängigkeiten zu den von Java definierten primitiven Datentypen, Sammlungen, und Hilfsklassen der Pakete `java.lang` und `java.util`.

Die Datenzugriffsschicht nutzt darüber hinaus eine API zur Kommunikation mit dem Webservice des Entsorgungsunternehmens. Sie nutzt außerdem die Android-API, um die Anfragen an den Webservice asynchron auszuführen. Zur Deserialisierung der erhaltenen Daten aus der JavaScript Object Notation (JSON) kommt die Programmbibliothek Gson[2] zum Einsatz.

Die Klassen des Domänenmodells, der Geschäftslogik und der Verarbeitung von Nutzerinteraktionen nutzen die primitiven Datentypen und Sammlungen, die die Java Laufzeitumgebung bereitstellt. Darüber hinaus haben sie keine Abhängigkeiten zum Betriebssystem oder zu externen Programmbibliotheken.

Die Klassen und XML-Layouts zur Definition der Nutzeroberfläche sind hingegen stark von der Android-API abhängig. Sie nutzen die nativen Oberflächenelemente und erben von den Typen der Android-API, um ihren Lebenszyklus vom Betriebssystem steuern zu lassen.

4.4.3 Schritt 3: Bewertung und Auswahl von Quellcode-Konvertoren

Um die mögliche Automatisierung der Übersetzung zu planen, müssen die portierenden Entwickler recherchieren, welche Quellcode-Konvertoren für die benötigte Übersetzung infrage kommen. Bei der Auswahl eines Konverters sind unter anderem folgende Kriterien zu berücksichtigen:

Unterstützter Sprachumfang der Eingabe
Um möglichst viel Code automatisch übersetzen zu können, muss der gewählte Konverter einen möglichst großen Teil der Syntax der Ausgangssprache verarbeiten können.

Vollständigkeit und Korrektheit der Ausgabe
Ein optimaler Konverter erzeugt eine vollständig kompilierbare Übersetzung, die das

[2] `https://github.com/google/gson`

gleiche Verhalten aufweist wie der ursprüngliche Code. Sie muss vor allem nicht durch manuelle Änderungen vervollständigt werden. Solche manuellen Anpassungen werden überschrieben, wenn der Konverter später erneut angewendet wird. Dadurch wird der Einsatz des Konverters während der Co-Evolution erheblich behindert. Außerdem können manuelle Änderungen dazu führen, dass per Trace Capture erhobene Trace Links ungültig werden.

Verständlichkeit der generierten Übersetzungen
In vielen Fällen hängt der manuell portierte Code der Zielimplementation von automatisch übersetztem Code ab, etwa dadurch, dass manuell portierter Code Methoden des automatisch portierten Codes aufruft. Die Entwickler des manuellen Codes müssen daher die Schnittstellen und Funktionen der automatischen Übersetzung leicht verstehen und nutzen können. Langfristig besteht immer die Gefahr, dass das Generat händisch gewartet werden muss, beispielsweise, weil der Konverter nicht mehr an die Entwicklung der Ausgangssprache angepasst wird. Entsprechend wichtig ist es, dass die Ausgabe des Konverters für die Entwickler verständlich ist. Die Bezeichner und Strukturen des ursprünglichen Codes werden im Idealfall in verständliche und einfach zuzuordnende Entsprechungen konvertiert, sodass die Entwickler der Zielimplementation die ursprünglichen Konzepte wiedererkennen. Gute Quellcode-Konvertoren formatieren die Ausgabe außerdem übersichtlich. Wenn sie Bezeichner generiert, sind diese lesbar und aussagekräftig.

Konfigurierbarkeit von Mappings
Entwickler sollten idealerweise die Möglichkeit haben, die Abbildung der ursprünglichen Plattformabhängigkeiten auf ihre Entsprechungen in der Zielplattform zu konfigurieren. Dazu können für einige Konverter API-Mappings definiert werden. Dabei handelt es sich um formale Abbildungsregeln, nach denen der Konverter die Abhängigkeiten zu plattformspezifischen APIs übersetzt. Je nach Art der Unterschiede zwischen ursprünglich eingesetzten APIs und Ziel-APIs kann auch plattformabhängiger Code auf Basis solcher Mappings vollständig konvertiert werden.

Benutzbarkeit und Support
Bei der Wahl eines Konverters muss berücksichtigt werden, wie einfach er zu konfigurieren und anzuwenden ist. Außerdem ist zu prüfen, ob es angemessene Dokumentation und Support vom Hersteller oder von einer hinreichend großen Nutzergemeinde gibt. Eine große, aktive Nutzergemeinde verspricht nicht nur schnelle Hilfe bei Anwendungsproblemen. Sie spricht auch dafür, dass der Konverter in Zukunft weiterentwickelt wird, sodass er auch während der Co-Evolution genutzt werden kann.

Lizenzkosten
Um das Budget der Portierung nicht zu überziehen, müssen auch die Lizenzkosten bei der Wahl des Konverters berücksichtigt werden.

Auf Basis dieser Kriterien entscheiden sich die portierenden Entwickler für einen angemessenen Konverter und führen Probekonversionen durch. Dabei übersetzen sie einzelne Typdefinitionen mit unterschiedlichen Plattformabhängigkeiten. Anhand der Ergebnisse dieser Probekonversionen stellen sie fest, welche der identifizierten Plattformabhängigkeiten korrekt verarbeitet werden und welche Qualität die erzeugten Übersetzungen haben.

Im Beispiel des Bulky Waste Companion fällt die Wahl auf eine erweiterte Version des quelloffenen Konverters *J2Swift*[3]. Die erweiterte Version steht als Plugin für die Entwicklungsumgebung Android Studio zur Verfügung[4]. Sie übersetzt den genutzten Sprachumfang korrekt und erlaubt die Konfiguration von Mappings.

4.4.4 Schritt 4: Festlegung portabler Bereiche und Abstraktionsebenen für die Portierung

Auf Basis der Abhängigkeitsanalyse und der Entscheidung für einen Konverter legen die Entwickler im vierten Schritt fest, welche Abstraktionen bei der Portierung welcher Teile des Codes anzuwenden sind.

In Abschnitt 4.1.1 wurde ein Vorgehen definiert, nach dem diese Entscheidung getroffen wird. Es wird hier verkürzt wiederholt: Zunächst wird angestrebt, eine Typdefinition Anweisung für Anweisung zu portieren, sodass lediglich von der Syntax der ursprünglichen Programmiersprache abstrahiert wird. Dies ist typischerweise bei Klassen möglich, die Konzepte der Anwendungsdomäne repräsentieren oder die Geschäftslogik der Software implementieren. Solche Klassen sind meist einfach strukturiert und haben nur wenige Abhängigkeiten zur Ausgangsplattform. Technischer Code, der die Einbindung in die Infrastruktur der Plattform oder die Benutzerschnittstelle realisiert, hat dagegen sehr viele Abhängigkeiten und kann in der Regel nicht konvertiert werden [Terekhov 2001]. Dieser sollte so portiert werden, dass zumindest der Entwurf auf Typebene übertragen wird, also Entsprechungen zu Typen mit gleichem Namen und gleicher Zuständigkeit in der Zielimplementation entstehen. Für alle anderen Klassen wird dann eine Portierung auf der Ebene von Anweisungen angestrebt, wenn davon auszugehen ist, dass der gewählte Konverter sie vollständig übersetzen kann. Ansonsten können einzelne Anweisungen abweichend vom Original implementiert werden, wenn es die händische Weiterentwicklung vereinfacht. Entsprechungen in Bezug auf die Schnittstellen einer Klasse sollten dabei erhalten bleiben, insbesondere, wenn weiterer zu portierender Code diese Schnittstellen nutzt.

[3] `https://github.com/patniemeyer/j2swift`
[4] `https://github.com/TilStehle/Cross-Platform-Traceability`

Um möglichst große Teile des Codes in konkrete Entsprechungen zu überführen, ist es sinnvoll, die bestehende Architektur zu prüfen und gegebenenfalls zu überarbeiten. Insbesondere sind logische Klassen mit wenigen Abhängigkeiten zur Plattform von technischen Klassen zu trennen, die meist sehr viele Abhängigkeiten zu plattform-spezifischen Schnittstellen haben. Die Entwickler prüfen, ob einzelne Typdefinitionen portierbaren und plattform-spezifischen Code vermischen. Für diese Typdefinitionen planen sie eine Aufspaltung, die portierbaren Code von plattform-spezifischem Code isoliert.

Viele gängige Architektur- und Entwurfsmuster unterstützen diese Trennung ohnehin, sodass eine grundlegende Änderung der Architektur nicht notwendig ist. Ein gutes Beispiel sind klassische Schichten-Architekturen, die auch in der zunehmend eingesetzten Entwurfsmethode *Domain-Driven Design* verwendet werden [Vernon 2013, S. 119ff.]. Sie definieren in der Regel technische Schichten wie eine Präsentations- oder Infrastrukturschicht, die streng von logischen Schichten getrennt sind, welche die Konzepte der Anwendungsdomäne oder die Geschäftslogik eines Systems realisieren.

Durch diese Revision ergibt sich eine Soll-Architektur, die portierbare Typen isoliert. Sie legt zu jedem dieser Typen fest, ob er Anweisung für Anweisung in eine exakte Entsprechung portiert wird, oder ob nur seine Schnittstelle oder seine Zuständigkeit übertragen wird. Abbildung 4.3 auf der nächsten Seite stellt diese Soll-Architektur für den Bulky Waste Companion dar.

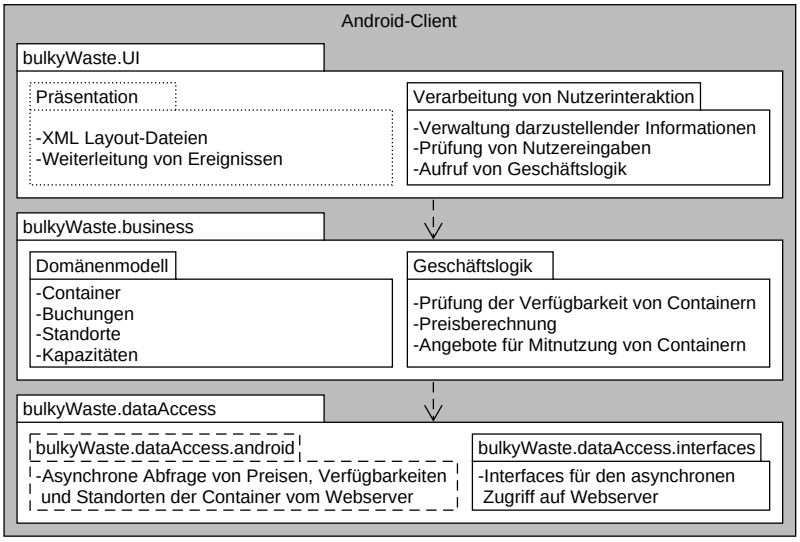

Abbildung 4.3: Soll-Architektur der App *Bulky Waste*

Pakete mit durchgehender Umrandung enthalten Typdefinitionen, die vollständig konvertiert werden sollen. Pakete mit gestrichelter Umrandung enthalten Typdefinitionen, deren Zuständigkeiten und Schnittstellen in die Zielimplementation überführt werden sollen. Pakete mit gepunkteter Umrandung beinhalten Typdefinitionen, die lediglich eine Entsprechung mit gleicher Zuständigkeit in der Zielimplementation erhalten sollen.

Strukturell besteht der einzige Unterschied zur Ist-Architektur darin, dass die Datenzugriffsschicht in zwei Pakete unterteilt wird. Das Paket `bulkyWaste.dataAccess.interfaces` enthält lediglich Interfaces, die die Operationen zur asynchronen Ausführung von Server-Anfragen definieren. Diese sind vollständig konvertierbar. Die implementierenden Klassen im Paket `bulkyWaste.dataAccess.android` implementieren diese Interfaces plattform-spezifisch und werden nur anhand ihrer Schnittstellen portiert.

Die Klassen des Domänenmodells, der Geschäftslogik und der Verarbeitung von Nutzerinteraktionen sollen ebenfalls vollständig konvertiert werden.

Die Klassen zur Definition der Benutzeroberfläche haben hingegen sehr viele Abhängigkeiten zur Plattform-API. Sie sollen in der iOS-Implementation in zweckgleichen Klassen mit plattform-spezifischen Schnittstellen re-implementiert werden.

4.4.5 Schritt 5: Isolation eines Inkrements

Die tatsächlichen Hürden für die Anwendung eines Konverters und für das Erzielen von konkreten Entsprechungsbeziehungen zeigen sich häufig erst, wenn die Portierung durchgeführt wird [Hook 2005, S. 43]. Fehlentscheidungen, die vor Bekanntwerden

dieser Hürden getroffen wurden, sollten allerdings möglichst schnell erkannt und revidiert werden. Dazu ist es notwendig, möglichst früh Feedback darüber zu bekommen, ob der Aufwand für Refactorings der Ausgangsimplementation und für händische Re-Implementierungen in der Zielimplementation in einem angemessenen Verhältnis zur Qualität des portierten Codes stehen. Auf Basis dieses Feedbacks sollten die portierenden Entwickler ihr Vorgehen regelmäßig prüfen und ändern. Das Reengineering Pattern *Migrate Systems Incrementally* wird im Kontext klassischer Migrationsprojekte exakt zu diesem Zweck formuliert. Bei seiner Anwendung wird die Migration schrittweise vollzogen, wobei jeder Schritt einen kleinen, in sich abgeschlossenen Teil des ursprünglichen Systems migriert. Nach einem solchen Schritt erheben die Entwickler Feedback über das Ergebnis und passen ihr Vorgehen gegebenenfalls an [Demeyer u. a. 2009, S. 191].

Diese Herangehensweise übernimmt die hier entwickelte Portierungsmethode. Dazu teilen die portierenden Entwickler die ursprüngliche Codebasis in Inkremente ein, die nacheinander portiert werden. Ein Inkrement implementiert dabei eine abgeschlossene Funktionalität. Es ist ausführbar und liefert einen Mehrwert für den Kunden [Graham 1992].

Insbesondere beim Einsatz von Quellcode-Konvertoren ist es dazu gelegentlich notwendig, das aktuelle Inkrement vom Code anderer Inkremente zu isolieren. Dazu können die Entwickler die benötigten Implementationen temporär durch Dummies ersetzen, die die benötigten Schnittstellen implementieren aber lediglich Testdaten liefern.

Soll die bestehende Implementation gleichzeitig weiterentwickelt werden, so werden die Abspaltung und die benötigten Dummy-Implementierungen in einem gesonderten Entwicklungsstrang durchgeführt. Dieser Entwicklungsstrang wird hier als *Porting Branch* bezeichnet, während der ursprüngliche Entwicklungsstrang *Development Branch* genannt wird.

Abbildung 4.4 stellt Development Branch und Porting Branch zu zwei aufeinander folgenden Zeitpunkten dar.

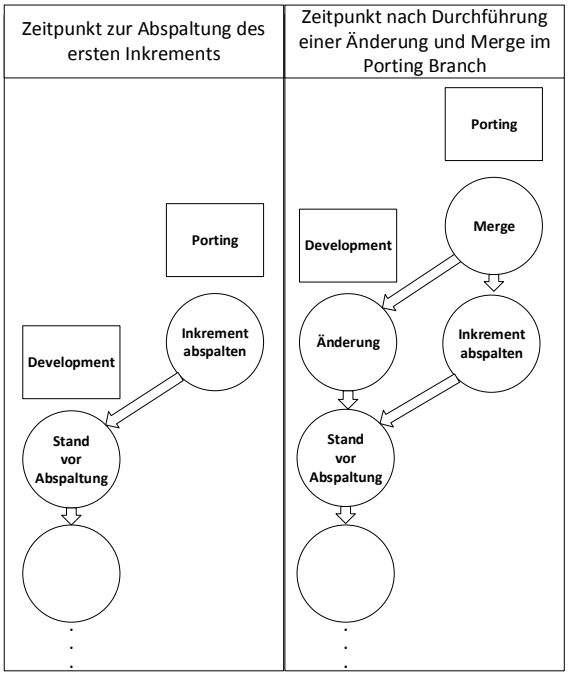

Abbildung 4.4: Parallele Entwicklung in Development Branch und Porting Branch

Links ist die Situation unmittelbar nach der Abspaltung des ersten Inkrements im Porting Branch zu sehen. Die letzte Änderung mit der Bezeichnung *Inkrement abspalten* entfernt alle nicht zum Inkrement gehörenden Code-Elemente und ersetzt einzelne von ihnen durch Dummy-Implementationen, sodass das Inkrement ausführbar und testbar ist. Im rechten Teil der Abbildung ist zu sehen, dass eine Änderung im Development Branch vorgenommen wurde. Diese wird durch eine *Merge*-Operation in den Porting Branch übernommen. Nach der erfolgreichen Portierung des ersten Inkrements wird dem Porting Branch schrittweise jeweils ein weiteres Inkrement hinzugefügt.

Im Fallbeispiel des Bulky Waste Companion wird jede der fünf Hauptfunktionalitäten als eigenes Inkrement portiert, beginnend mit der Buchung von Containern. Dazu gehören Klassen, die das entsprechende Domänenmodell der Container und der Buchungen definieren, Klassen, die die Kommunikation mit dem Webserver umsetzen und solche, die Containerbuchungen an der Benutzeroberfläche darstellen und entsprechende Eingaben verarbeiten. Auch die zugehörigen Unit Tests werden als Teil des ersten Inkrements portiert.

4.4.6 Schritt 6: Refactorings zur Erhöhung des portablen Code-Anteils

Im sechsten Schritt setzen die portierenden Entwickler im aktuellen Inkrement die Zielarchitektur um, die im vierten Schritt definiert wurde. Dazu spalten sie einzelne Typdefinitionen auf, deren Code mit unterschiedlichen Abstraktionen portiert werden soll und verschieben deplatzierten Code.

Im Beispiel des Bulky Waste Companion spalten die Entwickler diejenigen Klassen auf, die gegen die Trennung von Präsentation und Verarbeitung von Nutzereingaben verstoßen. Dadurch wird der Anteil des konvertierbaren Codes in den Klassen erhöht, die die Eingaben des Benutzers prüfen und verarbeiten.

Darüber hinaus erstellen sie das Paket `bulkyWaste.dataAccess.core`, das in Abschnitt 4.4.4 definiert wurde. Die Interfaces zum Zugriff auf den Webservice werden aus den bestehenden Klassen der Datenzugriffsschicht in dieses Paket extrahiert.

Wurde für die Portierung ein Porting Branch erstellt, so werden die Refactorings erst im Development Branch umgesetzt und erst dann in den Porting Branch übertragen, wie Abbildung 4.5 analog zu Abbildung 4.4 darstellt.

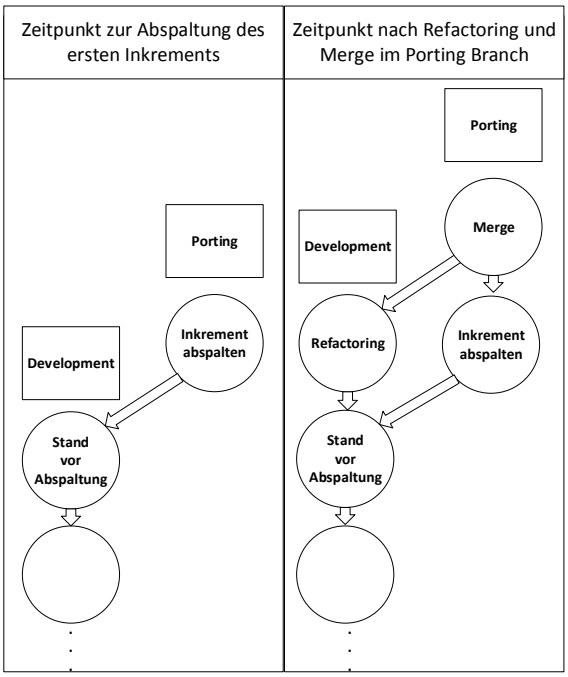

Abbildung 4.5: Umsetzung von Refactorings im Development Branch und Porting Branch

Spätere Weiterentwicklungen im Development Branch werden durch dieses Vorgehen immer auf Basis der letzten für die Portierung notwendigen Restrukturierung durchgeführt. Das erleichtert es, solche Weiterentwicklungen in den Porting Branch zu übertragen, da die Restrukturierung auf beide Entwicklungsstränge angewendet wurde.

4.5 Phase 2: Portierung eines Inkrements und Erzeugung plattform-übergreifender Trace Links

4.5.1 Schritt 7: Nutzung von Quellcode-Konvertoren und Erhebung von Traceability-Modellen

Konfiguration des Konverters
Ehe der gewählte Quellcode-Konverter eingesetzt werden kann, müssen die Entwickler ihn konfigurieren. Die vorzunehmenden Einstellungen unterscheiden sich dabei stark von Konverter zu Konverter. Mögliche Parameter einer Konfiguration sind Namens- und Formatierungskonventionen, die in der Übersetzung angewendet werden sollen sowie die Pfade zu Programmbibliotheken, die der Konverter benötigt, um den ursprünglichen Quellcode zu kompilieren.

Primitive Datentypen werden von einem Konverter in der Regel ohne explizite Konfiguration korrekt übersetzt. Abhängigkeiten zu anderen Typen der ursprünglichen Plattform müssen explizit auf ihre Entsprechungen in der Zielplattform abgebildet werden. Diese expliziten Abbildungsvorschriften oder auch *Mappings* sind ebenfalls Teil der Konfiguration, die die Entwickler zur Anwendung eines Konverters vornehmen.

Im Beispiel des Bulky Waste Companion werden etwa der Typ `java.util.HashMap` und seine Operationen genutzt. Sie werden durch API-Mappings auf den entsprechenden Swift-Typ `Dictionary` und seine Operationen abgebildet.

Nachbildung ursprünglich genutzter Schnittstellen in der Zielplattform
Nicht alle ursprünglich genutzten Typen haben eine exakte Entsprechung in der Zielplattform. Oftmals bietet die Zielplattform zwar einen Typ an, der die gleiche Funktionalität bereitstellt, der allerdings eine andere Schnittstelle hat als der ursprünglich genutzte Typ.

Um auch Code automatisch Übersetzen zu können, der Abhängigkeiten zu solchen plattform-spezifischen Typen hat, müssen die Schnittstellen in Ausgangs- und Zieplattform aneinander angeglichen werden. Kapitel 5 identifiziert Entwurfsmuster, die für eben diesen Zweck eingesetzt werden können. Abschnitt 5.3 entwickelt dort eine Richtlinie für die Auswahl eines angemessenen Entwurfsmusters. Hook [2005, S. 42] empfiehlt, die ursprüngliche Codebasis möglichst unangetastet zu lassen, um keine Fehler in den Code einzuführen und um die Struktur des Codes beizubehalten, an die die Entwickler gewöhnt sind. Dementsprechend sollten die portierenden Entwickler zunächst versuchen, die ursprünglich genutzten Schnittstellen in der Zielplattform nachzubilden. Nur wenn dies scheitert und wenn der ursprüngliche Quellcode verändert werden kann, wenden sie die Entwurfsmuster in der ursprünglichen Codebasis an, um dort

die Schnittstellen zur Ausgangsplattform an eine Entsprechung in der Zielplattform zu adaptieren.

Im Code des Bulky Waste Companion findet sich beispielsweise in der Klasse `AvailabilityViewModel` eine Abhängigkeit zum plattformspezifischen Typ `java.text.SimpleDateFormat` wie in Abbildung 4.6 dargestellt.

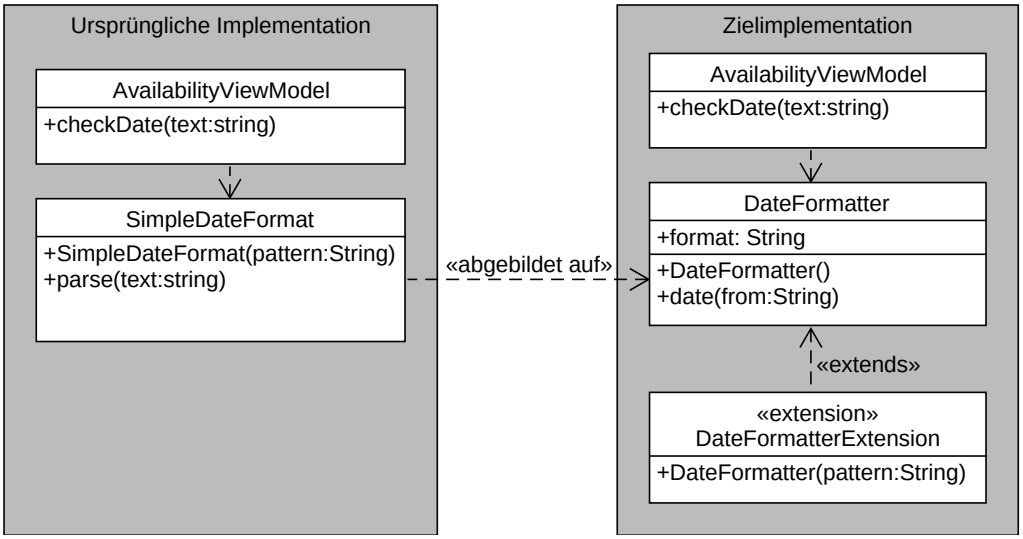

Abbildung 4.6: Angleichung der Schnittstelle von DateFormatter an SimpleDateFormat

Die Klasse `AvailabilityViewModel` prüft die Eingaben des Nutzers, bei der Abfrage verfügbarer Container. Sie verarbeitet unter anderem ein vom Nutzer eingegebenes Datum und prüft, ob dieses Datum in der Zukunft liegt. Um aus der Eingabe des Nutzers ein Datumsobjekt zu erzeugten, nutzt sie ein Exemplar von `java.text.SimpleDateFormat`. Dessen Entsprechung in der iOS-Plattform ist die Klasse `DateFormatter`. `DateFormatter` definiert allerdings den ursprünglich genutzten Konstruktor nicht, der das Datumsformat entgegennimmt. Um diesen Konstruktor auch in der Zielplattform nutzen zu können, folgen die Entwickler der in Abschnitt 5.3 definierten Richtlinie und setzen das Entwurfsmuster *Extension Method* ein. Dazu implementieren sie die Extension `DateFormatterExtension`, die unten rechts in der Abbildung zu sehen ist. Sie ergänzt die Klasse `DateFormatter` der Zielplattform um den fehlenden Konstruktor.

Nicht immer ist die Angleichung plattform-spezifischer Schnittstellen effizient möglich. Die portierenden Entwickler müssen abwägen, ob die Anwendung der Richtlinien aus Abschnitt 5.3 mit angemessenem Aufwand zu einer Erhöhung des konvertierbaren Codes führen. Ist dies nicht der Fall, so können sie entscheiden, die zu konvertierende Typdefinition doch von der Konversion auszuschließen oder sie zu restrukturieren, um einen nicht konvertierbaren Anteil abzuspalten. Somit kehren sie zu Schritt vier der Methode zurück.

Anwendung des Transformators und Prüfung der Ausgabe
Sind die Mappings konfiguriert und die benötigten Schnittstellen in der Zielplattform nachgebildet, so wenden die Entwickler den Konverter an. Sie prüfen anschließend die Ausgabe, indem sie den konvertierten Code kompilieren und Unit-Tests ausführen. Dabei stellen sie gegebenenfalls fest, dass die konfigurierten Mappings nicht korrekt oder unvollständig sind, sodass einige der konvertierten Abhängigkeiten zu Compiler-Fehlern führen. Auf dieser Basis verbessern sie die Konfiguration des Konverters und die nachgebildeten Schnittstellen in der Zielplattform, bis der konvertierte Code kompilierbar ist und die Tests besteht.

Erhebung von Traceability-Modellen für vollständig konvertierbare Code-Elemente
Bei der automatisierten Übersetzung bildet der Konverter die ursprünglichen Quellcode-Elemente auf entsprechende Elemente der Zielsprache ab. Für vollständig konvertierbaren Code bietet dies die Gelegenheit, Traceability-Modelle zu generieren, die die ursprünglichen Code-Elemente mit ihren Übersetzungen verknüpfen.

Dazu müssen die Übersetzungsregeln des Konverters so erweitert werden, dass sie zu jeder Übersetzung auch den zugehörigen Trace Link generieren. Sämtliche konvertierten Typdefinitionen, ihre Operationen und Attribute und sogar lokale Variablen und Anweisungen werden so mit ihren Entsprechungen verknüpft. Wie in Abschnitt 4.1.2 entschieden, wird diese Strategie nur auf vollständig konvertierbaren Code angewendet, da manuelle Änderungen am Generat die erzeugten Links ungültig machen können.

Um die geschilderte Erweiterung umsetzen zu können, sind Kenntnisse über den Quellcode des Konverters notwendig und dieser Quellcode muss zur Verfügung stehen. Die portierenden Entwickler sollten den eingesetzten Konverter nur dann selbst um Trace Capture erweitern, wenn sie ihn selbst entwickelt oder bereits erweitert haben. Ansonsten nutzen sie auch für konvertierten Quellcode das Trace-Recovery-Verfahren, das in Abschnitt 6 vorgestellt wird.

Repräsentation plattform-übergreifender Quellcode-Entsprechungen durch Traceability-Modelle

Um Trace Links zwischen einander entsprechenden Code-Elementen automatisiert nutzen zu können, müssen sie eine formale Repräsentation haben. Zu diesem Zweck wurde ein Metamodell aufgestellt, das dem konkreten Einsatz von Trace Links im Kontext der plattform-übergreifenden Co-Evolution dient [Stehle u. Riebisch 2018b]. Ein repräsentativer Ausschnitt des Metamodells ist in Abbildung 4.7 dargestellt.

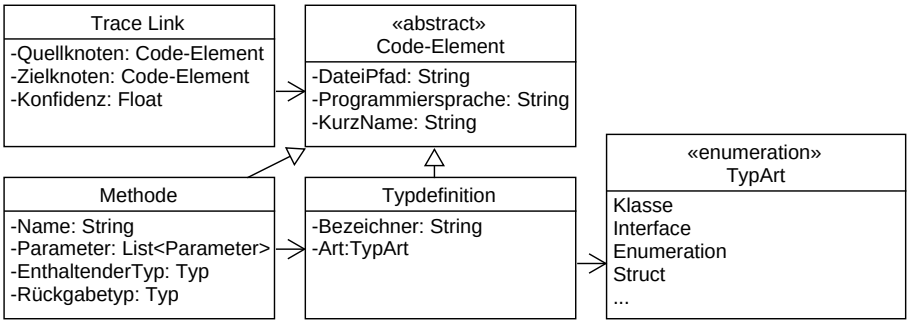

Abbildung 4.7: Ausgewählte Klassen des Metamodells für plattform-übergreifende Quellcode-Entsprechungen im UML-Klassendiagramm

Die abstrakte Klasse `Code-Element` repräsentiert Quellcode-Elemente. Spezifische Klassen für Code-Elemente wie Methoden oder Typdefinitionen erben von ihr. Sie enthalten alle notwendigen Informationen, um ein Code-Element zu identifizieren und aufzufinden. Die Klasse `Methode` erfasst dazu beispielsweise den Namen der repräsentierten Methode sowie eine Referenz auf die Typdefinition, in der die Methode definiert ist. Eine `Typdefinition` speichert als identifizierendes Attribut ihren vollständigen Bezeichner. Darüber hinaus wird erfasst, um welche Art von Typdefinition es sich handelt. Dazu referenziert jede Typdefinition ein Element der Aufzählung `TypArt`, die zwischen Klassen, Interfaces, Aufzählungen und anderen Arten von Typdefinitionen unterscheidet. `Typdefinition` erbt selbst von `Code-Element`, da Typdefinitionen ebenfalls Code-Elemente sind, die Entsprechungen in einer zweiten Codebasis haben können.

Die Entsprechungsbeziehungen werden durch Instanzen der Klasse `Trace Link` repräsentiert. Sie halten jeweils einen Verweis auf das ursprüngliche und das portierte Code-Element sowie ein `Konfidenz`-Attribut, das ausdrückt, mit welcher Wahrscheinlichkeit eine erfasste Entsprechung korrekt ist. Im Kontext des Trace Capture Verfahrens ist dieser Konfidenzwert stets mit dem Maximalwert belegt, da der Konverter die Entsprechungsbeziehung selbst verursacht und speichert. Es besteht somit kein Zweifel daran, dass es sich um einen korrekten Trace Link handelt. Der Konfidenzwert ist vor allem im Kontext des Trace Recovery relevant und wird dementsprechend in Abschnitt 6 näher behandelt.

Thalheim [2013] definiert wichtige Modelleigenschaften, die bei der Definition eines

Metamodells zu berücksichtigen sind:

Purpose: Das Metamodell muss auf den Einsatzzweck der Modelle zugeschnitten sein. Zweck der darzustellenden Traceability-Modelle ist es, die Co-Evolution einander entsprechender Code-Elemente zu vereinfachen. Dazu muss das Metamodell folgende Aktivitäten unterstützen:

1. Suche nach korrespondierenden Code-Elementen
2. Navigation zwischen korrespondierenden Code-Elementen über die Grenzen verschiedener Entwicklungsumgebungen hinweg
3. Plattform-übergreifende Impact-Analyse; Dies ist notwendig, um bei der Erhebung notwendiger Änderungen auch Änderungen portierter Elemente zu erkennen, die für den Erhalt der plattform-übergreifenden Konsistenz nötig sind.
4. Übertragung automatisierbarer Änderungen wie Refactorings
5. Koordination händischer Änderungen sich entsprechender Code-Elemente.

Diesen Zwecken trägt das Metamodell dadurch Rechnung, dass es zweckgleiche Quellcode-Elemente paarweise einander zuordnet. Das Metamodell ist formal, sodass die Entsprechungsmodelle zur Reduktion von Aufwand *automatisiert* von Werkzeugen verarbeitet werden können, die den Aktivitäten eins bis fünf dienen.

Restrictions: Aus dem Zweck der Modelle ergeben sich Einschränkungen für das Metamodell. Die Entsprechungsmodelle sind beispielsweise nicht dazu gedacht, über den Entwurf einer einzelnen Implementation zu diskutieren und enthalten folglich über die Entsprechungen hinaus keine anderen Beziehungen zwischen Code-Elementen.

Pragmatism: Es gibt eine intendierte Nutzergruppe, die die Modelle auf eine bestimmte Weise einsetzen soll. Im konkreten Fall sollen die Modelle teil-automatisiert mit Werkzeugen für die oben genannten Zwecke während der Weiterentwicklung portierter Software genutzt werden. Dem entspricht das Metamodell durch die formale Definition erlaubter Modellelemente und Beziehungen, sodass entsprechende Modelle automatisiert mit Werkzeugen für Softwareentwickler verarbeitet werden können.

Amplification: Ein Modell kann zusätzliche Informationen enthalten, die das Original nicht enthält. Die hier eingeführten Modelle enthalten explizite Entsprechungen zwischen Quellcode-Elementen, die im Quellcode nicht dokumentiert sind.

Truncation: Modelle abstrahieren vom Original und lassen dabei Informationen aus, die im Sinne des Zwecks irrelevant sind. Relevant für die Beschreibung der Entsprechungen sind lediglich Ursprung und Ziel sowie ein Konfidenzwert, der angibt, mit welcher Wahrscheinlichkeit eine automatisch erhobene Entsprechung korrekt ist.

Mapping: Modelle beziehen sich stets auf ein Original, dessen Elemente sich im Modell wiederfinden. Die im Entsprechungsmodell verknüpften Repräsentationen von Quellcode-Elementen beziehen sich auf die konkreten Quellcode-Elemente in der ursprünglichen und portierten Implementation.

Idealisation: Modelle nehmen im Allgemeinen eine zweckdienliche Vereinfachung vor. Im Fall der Entsprechungsmodelle werden lediglich 1:1-Beziehungen zwischen Code-Elementen erfasst. Beispielsweise wird nicht modelliert, dass eine Klasse ggf. nicht vollständig in ihre Übersetzung in der Zielimplementation übertragen worden ist, sondern nur Teile ihrer ursprünglichen Aufgabe.

4.5.2 Schritt 8: Re-Implementieren nicht konvertierbarer Quellcode-Elemente

Im Anschluss an die Konversion re-implementieren die Entwickler sämtliche Code-Elemente, die gar nicht oder nicht vollständig konvertiert werden konnten. Grundsätzlich orientieren sie sich dabei an den ursprünglichen Code-Elementen. Sie übernehmen deren Konzepte, Strukturen und die Terminologie, die in den Bezeichnern des ursprünglichen Codes angewendet wurde. Dadurch vereinfachen die Entwickler nicht nur die Portierung selbst, sondern auch die spätere Co-Evolution, die dann für die re-implementierten Code-Elemente und ihre Vorbilder einheitlich durchgeführt werden kann.

Um die angestrebten Entsprechungen zwischen den ursprünglichen Elementen und ihren Re-Implementationen zu ermöglichen, kann es erneut notwendig sein, die Schnittstellen der Zielplattform anzugleichen. Auch hier wird die in Abschnitt 5.3 definierte Richtlinie eingesetzt wie bereits in Schritt sieben. Auch hier gilt, dass die Entwickler zunächst versuchen sollten, die ursprünglich genutzten Schnittstellen in der Zielimplementation nachzubilden. Nur wenn dies nicht effizient möglich ist, passen sie stattdessen den ursprünglichen Code an.

4.5.3 Ergebnis einer Iteration und Retrospektive

Nach der Re-Implementierung enthält die portierte Implementation die Funktionalität des aktuellen Inkrements. Sie ist testbar und ausführbar. Im Beispiel des Bulky Waste Companion können die Entwickler nach der ersten Iteration in der iOS-Version der App einen Container buchen. Die portierten Typdefinitionen entsprechen ihren Vorbildern in Bezug auf ihre Anweisungen, ihre Schnittstellen oder zumindest in Bezug auf ihre Zuständigkeiten. Darüber hinaus liegt ein Traceability-Modell vor, das die vollständig konvertierten Typdefinitionen mit ihren Vorbildern verknüpft. Es enthält auch Trace Links,

die die darin enthaltenen Code-Elemente wie Attribute und Methoden mit ihren ursprünglichen Pendants verbinden. Solche Trace Links liegen allerdings nicht für Code-Elemente vor, die nur teilweise konvertiert wurden oder manuell re-implementiert werden mussten. Sie werden erst während der Co-Evolution mit Hilfe des Trace-Recovery-Mechanismus erhoben, der in Kapitel 6 entwickelt wird.

Eine Iteration des Portierungsprozesses schließt jeweils damit ab, dass die Entwickler über das Vorgehen reflektieren. Sie bewerten, ob der betriebene Aufwand für die Angleichung von Schnittstellen und für die Re-Implementierung angemessen ist. Ist dies nicht der Fall, so können sie für das nächste Inkrement entscheiden, den Abstraktionsgrad für die Portierung bestimmter Typdefinitionen zu erhöhen, oder zu senken. Außerdem können sie beschließen, den gewählten Konverter auf einen größeren oder kleineren Anteil des Codes anzuwenden. Damit kehren sie zu Schritt vier des Portierungsprozesses zurück, der die Portierung des nächsten Inkrements plant.

Darüber hinaus legen die Entwickler nach der Portierung eines Inkrements dem Kunden die aktuelle Version der portierten Implementation vor und holen Feedback ein. Dabei kann sich herausstellen, dass sich die Anforderungen des Kunden für die Zielplattform unerwartet von den Anforderungen an die ursprüngliche Implementation unterscheiden. Dementsprechend sind Code-Elemente im nächsten Inkrement von der Portierung auszuschließen, die plattform-spezifische Funktionen realisieren.

4.6 Einbettung in den Entwicklungsprozess

Die oben beschriebene Portierungsmethode folgt einem inkrementellen Vorgehen, das den ursprünglichen Quellcode schrittweise portiert und nach jedem Schritt die gewählte Portierungsstrategie reflektiert. Mit diesem Vorgehen steht die Methode im Einklang mit agilen Vorgehensweisen. Die am häufigsten genutzte agile Vorgehensweise ist *Scrum* [Stellman u. Greene 2019, S. 71]. Im folgenden wird die Portierungsmethode exemplarisch in den Projektablauf von Scrum eingeordnet. Abbildung 4.8 stellt den Ablauf eines nach Scrum organisierten Projekts dar.

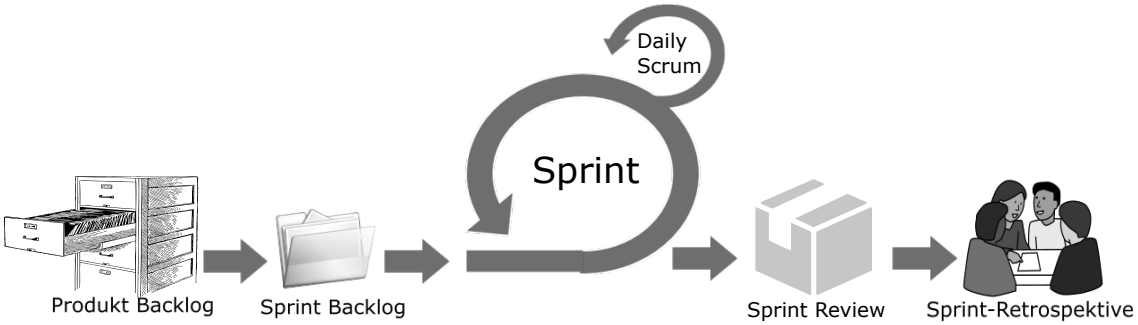

Abbildung 4.8: Scrum-Projektablauf, in Anlehnung an [Wolf u. Bleek 2011, S. 164]

Das *Product Backlog* erfasst alle Anforderungen eines Projekts. Diese können grundsätzlich jederzeit ergänzt werden. Im Rahmen der Portierungsmethode wird das Product Backlog hauptsächlich im ersten Schritt gefüllt. Es definiert die zu portierenden Funktionalitäten und die zur Portierung notwendigen Aufgaben.

In einem sogenannten Sprint-Planning-Meeting wird beschlossen, welche Funktionalitäten das nächste zu portierende Inkrement enthalten soll. An diesem Meeting sind sowohl die portierenden Entwickler als auch der Product Owner beteiligt, der für die Auswahl und Priorisierung der Anforderungen verantwortlich ist und somit den Kunden im Projekt vertritt [Wolf u. Bleek 2011, S. 161]. Das Ergebnis dieses Meetings ist das Sprint Backlog, in dem jeweils die Funktionalitäten des Inkrements dokumentiert sind, die in der nächsten Iteration in den Schritten vier bis acht portiert werden. Eine solche Iteration wird im Kontext von Scrum als Sprint bezeichnet. Sprints haben gleichbleibende Dauern, die typischerweise zwei Wochen lang sind [Stellman u. Greene 2019, S. 78]. Sie schließen jeweils mit einem Sprint Review ab, bei dem die Entwickler dem Product Owner die Inkrementell erweiterte Version der portierten Implementation präsentieren. Dabei können sich die im Produkt Backlog erfassten Anforderungen ändern. Im Kontext der Portierung kann sich dabei etwa herausstellen, dass der Kunde für die Zielimplementation andere Anforderungen hat als für die ursprüngliche Implementation (vgl. Abschnitt 4.5.3).

Neben dieser Reflektion aus Kunden- und Anwendersicht gibt es auch eine sogenannte

Sprint-Retrospektive, in der die Entwickler ihr Vorgehen reflektieren. Dabei stellen sie sich folgende Fragen

- Wie hoch war der Aufwand für die Angleichung von Schnittstellen?
- Steht dieser Aufwand in einem angemessenen Verhältnis zu den erzielten Entsprechungsbeziehungen und zur Aufwandsersparnis durch die Anwendung eines etwaigen Konverters?
- Wurde der gegebenenfalls eingesetzte Konverter optimal genutzt?
- Konnte der Konverter auf die Teile des Codes angewendet werden, die Anweisung für Anweisung in exakte Entsprechungen übersetzt werden sollten (Entscheidung aus Schritt vier)? Welche Probleme gab es dabei?
- Könnte der Konverter effizient auf weitere Teile angewendet werden?

Auf dieser Basis können die Entwickler in der nächsten Iteration in Schritt vier andere Abstraktionen für die angestrebten Entsprechungen definieren und den Einsatzbereich des Quellcode-Konverters neu definieren. Auch die bei der Portierung anzuwendenden Abstraktionen können sie neu festlegen.

5 Entwicklung einer Richtlinie zur Anwendung strukturangleichender Entwurfsmuster

Die Strukturen und Konzepte der Ausgangs- und Zielimplementation sollen möglichst ähnlich zueinander sein, um die plattform-übergreifende Co-Evolution zu erleichtern. Dem stehen die in Abschnitt 1.3 beschriebenen Hürden entgegen. Insbesondere Abhängigkeiten zu plattform-spezifischen APIs und Quellcode-Konverter, die nur begrenzt mit Mappings zwischen diesen APIs konfiguriert werden können, verhindern solche Ähnlichkeiten.

Um diese Hürde zu senken, können verschiedene Entwurfsmuster eingesetzt werden, um die Schnittstellen plattform-spezifischer APIs aneinander anzugleichen [Shah u. a. 2013; Gamma u. a. 1995; Fowler 2010; Freeman u. a. 2004; Martin 2003]. In diesem Abschnitt wird die Anwendung solcher Entwurfsmuster in Open-Source-Portierungsprojekten und Erweiterungen von Cross-Platform-Frameworks untersucht. Sie werden anhand ihrer unterschiedlichen Auswirkungen auf die spezifischen Herausforderungen von Portierungsprojekten miteinander verglichen. Die dabei gewonnenen Erkenntnisse werden in einer Richtlinie zusammengeführt. Diese trägt als Teil der hier entwickelten Portierungsmethode systematisch zur Strukturangleichung zwischen ursprünglicher und portierter Implementation bei.

Im folgenden Abschnitt werden zunächst die untersuchten Portierungsprojekte vorgestellt. In Abschnitt 5.2 werden anschließend Kriterien aufgestellt, anhand derer der Einsatz dieser Entwurfsmuster in den Portierungsprojekten bewertet wird. Abschnitt 5.3 fasst die Ergebnisse dieser Bewertung zusammen. Es wird eine Richtlinie abgeleitet, die Entwicklern dabei hilft, ein für die konkrete Situation angemessenes strukturangleichendes Entwurfsmuster zu wählen. Abschnitt 5.4 ergänzt die Betrachtung um Erzeugungsmuster, die in Kombination mit den untersuchten Strukturmustern einzusetzen sind, um plattform-spezifische Instanziierungen hinter einheitlichen Schnittstellen zu verbergen.

5.1 Untersuchte Open-Source-Projekte

Um den erfolgreichen Einsatz strukturangleichender Entwurfsmuster zu untersuchen, waren zunächst Projekte zu identifizieren, die eine plattform-übergreifende Strukturangleichung anstreben. Dazu wurde die Plattform GitHub nach Portierungsprojekten durchsucht, die bewusst darauf abzielen, plattform-übergreifende Ähnlichkeiten zwischen den Implementationen zu erreichen. Darüber hinaus wurden zwei quelloffene Erweiterungen des Cross-Plattform-Frameworks Xamarin identifiziert, die eine plattformübergreifend nutzbare Schnittstelle für Near Field Communication[1] bzw. für die Wiedergabe von Mediendateien[2] anbieten.

Tabelle 5.1 bietet einen Überblick über diese Projekte und die darin eingesetzten plattformspezifischen APIs. Die ersten beiden Spalten erfassen den Namen und eine Beschreibung des jeweiligen Projekts. In der dritten Spalte sind die ursprünglich eingesetzte Programmiersprache und die Zielsprache der Portierung notiert. Die vierte Spalte listet die in der ursprünglichen Plattform eingesetzten APIs auf. Die Abhängigkeiten zu diesen APIs wurden dahingehend untersucht, wie sie auf eine Entsprechung in der Zielplattform abgebildet wurden und welche strukturangleichenden Entwurfsmuster dabei angewendet wurden.

[1] `https://github.com/smstuebe/xamarin-nfc`
[2] `https://github.com/martijn00/XamarinMediaManager`

Name des Projekts	**Beschreibung**	**Ausgangs-sprache → Zielsprache**	**Eingesetzte APIs**
Libphonenumber	API für die Verarbeitung von Telefonnummern	Java → JavaScript	-Java Runtime Environment -Protocol Buffers
Libphonenumber	API für die Verarbeitung von Telefonnummern	Java→ C++	-Java Runtime Environment -Protocol Buffers
docopt	DSL zur Erstellung von Kommandozeilen-Schnittstellen	Python→ Swift	-Python Standard Library
docopt	DSL zur Erstellung von Kommandozeilen-Schnittstellen	Python→ Java	-Python Standard Library
JGit	API für Git-Clients	Java → C#	-Java Runtime Environment -JSch
Lucene	API zur Konstruktion effizient durchsuchbarer Indizes	Java→ C#	-Java Runtime Environment -ICU4J -javax.xml -Apache Commons Codec -Spatial4J -javax.servlet -Apache HttpClient -HttpCore -Carrotsearch Randomized Testing -Eclipse Jetty -SLF4J
Near Field Communication (Xamarin Plugin)	Near Field Communication-API für Xamarin-basierte Projekte	C#	-Android Plattform-API
Media Player (Xamarin Plugin)	API zur Wiedergabe von Medien in Xamarin-basierten Projekten	C#	-Android Plattform-API

Tabelle 5.1: Übersicht über die analysierten Open Source Portierungsprojekte

5.1.1 Portierungen von Google Libphonenumber

Libphonenumber ist eine Programmbibliothek von Google, mit der Telefonnummern geparst, formatiert und validiert werden können. Sie steht in den Sprachen Java, C++ und JavaScript zur Verfügung.[3] Änderungen werden stets zuerst in der Java-Version umgesetzt und dann in zwei Portierungsprojekten nach C++ und JavaScript portiert. Teile des Quellcodes werden aus Modellen in der textuellen DSL *Protocol Buffers*[4] generiert. Das betrifft beispielsweise die Klassen `PhoneNumber` und `Phonemetadata`, die Telefonnummern bzw. Metadaten zu diesen Telefonnummern erfassen. Über die modellierten Klassen hinaus werden jeweils Klassen zum Serialisieren und Deserialisieren ihrer Instanzen aus der DSL generiert.

Plattform-spezifisch sind in diesen Projekten die Schnittstellen zu Protocol Buffers, mit denen beispielsweise Metadaten serialisiert werden. Hinzu kommen Abhängigkeiten zur *Java Runtime Environment* (JRE), die unter anderem primitive Datentypen und Sammlungen zur Verfügung stellt.

Über alle drei Implementationen von Libphonenumber hinweg gleichen sich die definierten Typen in Bezug auf ihre Namen, Zuständigkeiten und Schnittstellen. In den meisten Fällen setzen sie die angebotene Funktionalität auch mit den gleichen Algorithmen um und definieren Variablen, die sich in Typ und Namen plattform-übergreifend entsprechen.

Die Implementationen unterscheiden sich allerdings in Bezug auf einzelne Anweisungen, die plattform-spezifische APIs wie *Protocol Buffers* nutzen.

5.1.2 Portierungen von docopt

Bei *docopt* handelt es sich um eine Programmbibliothek zur Erstellung von Kommandozeilen-Schnittstellen. Mit docopt können Entwickler in einer DSL Hilfetexte für ihre Nutzer sowie Kommandozeilenparameter deklarieren, die aus der Konsole eingelesen werden. Die Bibliothek wurde ursprünglich in der Sprache Python entwickelt[5]. Abhängigkeiten zu externen APIs wurden dabei bewusst vermieden, sodass die ursprüngliche Implementation ausschließlich von der Python Standard Library abhängt.

Die Ausgangsimplementation wurde in verschiedene Programmiersprachen portiert. Hier werden ausschließlich die Portierungen nach Swift[6] und Java[7] betrachtet, da diese besonderes Augenmerk auf konzeptuelle Ähnlichkeit zwischen den Implementationen

[3] `https://github.com/googlei18n/libphonenumber`
[4] `https://github.com/google/protobuf`
[5] `https://github.com/docopt/docopt/`
[6] `https://github.com/docopt/docopt.swift`
[7] `https://github.com/docopt/docopt.java`

legen. Viele Anweisungen des ursprünglichen Quellcodes sind in diesen Projekten unmittelbar in eine entsprechende Anweisung überführt worden. Die Implementationen unterscheiden sich allerdings in Bezug auf die Nutzung von Basisdatentypen und auf die APIs zum Umgang mit regulären Ausdrücken.

5.1.3 Portierung von JGit

Bei *JGit* handelt es sich um eine Programmbibliothek zur Ausführung von Befehlen auf Git-Repositories. Die ursprünglich in Java implementierte Bibliothek wurde im Rahmen des Projekts *NGit* nach C# übersetzt, um sie auf der .Net-Plattform anzubieten.

Das Projekt wurde teilautomatisiert mit Hilfe des Konverters *Sharpen*[8] durchgeführt. Fehlerhafte Übersetzungen wurden nachträglich händisch korrigiert. Durch den Einsatz des automatischen Quellcode-Konverters entspricht die Zielimplementation dem Original mit wenigen Ausnahmen Anweisung für Anweisung. Teile der Schnittstellen zur Zielplattform wurden auf Basis verschiedener Entwurfsmuster an die ursprünglich eingesetzte Java Runtime Environment angepasst.

Die im ursprünglichen Code eingesetzte API *Java Secure Channel* (*JSch*) wurde im Portierungsprojekt *NSch* ebenfalls für die .Net-Plattform portiert. Abhängigkeiten zu *JSch* konnten damit direkt auf ihre Entsprechungen in *NSch* abgebildet werden.

5.1.4 Portierung von Lucene

Auf Basis der Java-Programmbibliothek *Lucene* können Entwickler verschiedenste Dokumente in einem Index ablegen, und diesen effizient durchsuchen[9]. Das Projekt Lucene.Net[10] hat diese Bibliothek für die .Net-Plattform portiert. Dazu wurde Lucene teilautomatisiert in die Programmiersprache C# übersetzt. Ähnlich wie im Projekt JGit wurden dazu zunächst automatische Quellcode-Konvertoren eingesetzt. Fehlerhafte Übersetzungen korrigierten die portierenden Entwickler nachträglich manuell[11]. Durch dieses Vorgehen ähnelt der portierte Quellcode dem ursprünglichen größtenteils Anweisung für Anweisung. Auch hier wurden die Schnittstellen zur Zielplattform an die Java Runtime Environment angeglichen. Teile der APIs *Apache Commons Codec* und *Carrotsearch Randomized Testing* wurden zudem in der Zielplattform re-implementiert.

Bezüglich anderer APIs wurden Unterschiede zwischen ursprünglicher und portierter Implementation in Kauf genommen. Beispielsweise wurde die in Java eingesetzte API

[8] `https://github.com/mono/sharpen`
[9] `https://github.com/apache/lucene-solr`
[10] `https://github.com/apache/lucenenet`
[11] `https://github.com/apache/lucenenet/blob/master/CONTRIBUTING.md`

Apache HttpCore in der .Net-Implementation durch *System.Net.Http* ersetzt. Die Anweisungen in den abhängigen Klassen unterscheiden sich dementsprechend.

5.1.5 Open-Source-Erweiterungen des Cross-Platform-Frameworks Xamarin

Neben den oben skizzierten Portierungsprojekten sind zwei Erweiterungen des Cross-Platform-Frameworks Xamarin Gegenstand der Untersuchung. Xamarin bietet Entwicklern mobiler Apps eine einheitliche Schnittstelle zur Nutzung von Funktionalitäten der Plattformen Android und iOS. Die erste untersuchte Erweiterung ergänzt das Framework um eine plattform-übergreifend vereinheitlichte Schnittstelle für Near Field Communication[12]. Die zweite Erweiterung bietet eine vereinheitlichte Schnittstelle für die Wiedergabe von Mediendateien an[13]. Diese Erweiterungen wurden dahingehend untersucht, welche Entwurfsmuster im Code implementiert sind, um von den Unterschieden der nativen Plattform-APIs zu abstrahieren und vereinheitlichte Schnittstellen bereitzustellen.

5.2 Beschreibung und Bewertung der eingesetzten Entwurfsmuster

Zu Angleichung der Schnittstellen von Ausgangs- und Zielplattform setzen die untersuchten Projekte vor allem die Strukturmuster *Object Adapter*, *Class Adapter*, *Extension Method* und *Generation Gap* ein. Sie führen einheitliche Schnittstellen ein und verbergen plattform-spezifischen Quellcode in isolierten Code-Elementen, die diese Schnittstellen implementieren. Dadurch können die Nutzungs- und Vererbungsbeziehungen der ursprünglichen Code-Elemente durch einfache API-Mappings auf ihre Entsprechungen in der Zielplattform abgebildet werden.

Um eine Richtlinie für die Wahl eines dieser Entwurfsmuster aufzustellen, werden sie im Folgenden zunächst miteinander verglichen. Dazu wird die Problemsituation skizziert, auf die die identifizierten Entwurfsmuster angewendet werden. Im Anschluss werden Kriterien aufgestellt, die die Anwendbarkeit und den Einfluss eines strukturangleichenden Entwurfsmusters auf ein Portierungsprojekt erfassen. Der Einsatz der Entwurfsmuster in den untersuchten Projekten wird dargestellt und anhand der aufgestellten Kriterien bewertet.

Problemsituation bei plattform-spezifischen APIs
Abbildung 5.1 zeigt die Problemsituation, die von allen identifizierten Entwurfsmustern adressiert wird.

[12] `https://github.com/smstuebe/xamarin-nfc`
[13] `https://github.com/martijn00/XamarinMediaManager`

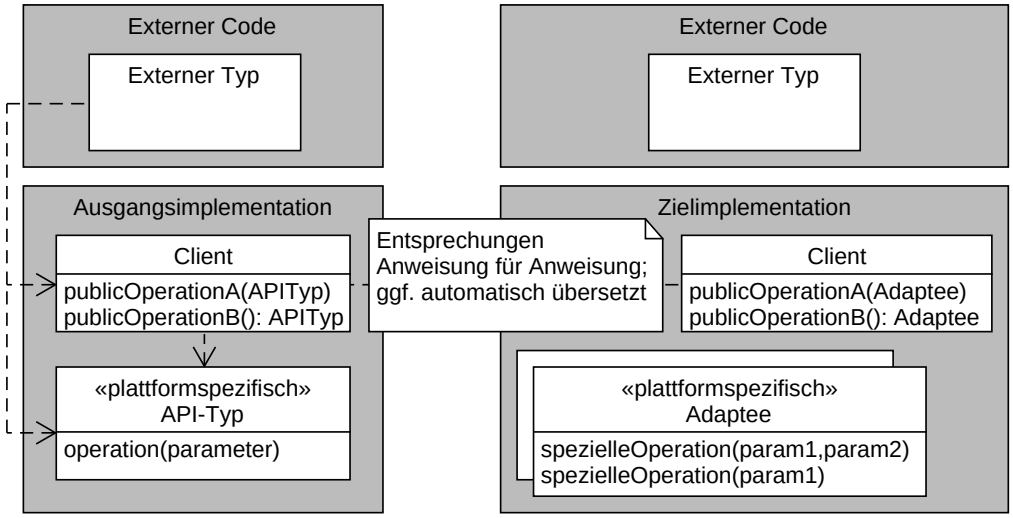

Abbildung 5.1: Problemsituation: Die Klasse `Client` hängt von einem plattformspezifischen Typ ab, der keine Entsprechung mit exakt gleicher Schnittstelle in der Zielplattform hat.

Die Ausgangsimplementation enthält eine Klasse `Client`, die möglichst Anweisung für Anweisung in eine exakte Entsprechung in der Zielimplementation übertragen werden soll. Dazu wird gegebenenfalls ein automatischer Quellcode-Konverter eingesetzt. Der zu portierende Client hängt in der ursprünglichen Implementation allerdings von einem Typ ab, der in einer plattform-spezifischen API definiert ist.

In der Zielplattform gibt es zwar eine entsprechende API, die dieselbe Funktionalität anbietet, diese Ziel-API definiert allerdings Typen mit anderen Schnittstellen. Diese Typen werden hier als *Adaptees*, also als anzupassende Typen bezeichnet. Sie definieren eine oder mehrere plattform-spezifische Operationen, die die benötigte Funktionalität bereitstellen. Solche Operationen sind potenziell anders benannt und erwarten andere Parameter als die Operation des ursprünglich eingesetzten Typs. In einigen Fällen ist die benötigte Funktionalität sogar auf mehrere Adaptees verteilt, sodass eine klare Abbildung des ursprünglich eingesetzten Typs auf einen einzelnen Adaptee nicht möglich ist.

Hinzu kommt, dass möglicherweise Code außerhalb des Portierungsprojekts vom zu portierenden Client abhängt. Solcher Code, der außerhalb des Portierungsprojektes entwickelt wird, wird hier als *externer Code* bezeichnet. Der Client stellt gegebenenfalls plattform-spezifische Operationen für externen Code zur Verfügung. In der Abbildung sind dies die Operationen `publicOperationA(API-Typ)` und `publicOperationB()`. Sie definieren Parameter- oder Rückgabewerte mit plattform-spezifischen Typen. In der Zielimplementation soll der Client ebenfalls plattform-spezifische Operationen für externen Code anbieten. Diese Operationen sollen den jeweils passenden Adaptee als Parameter- bzw. Rückgabetyp definieren, damit der externe Code der Zielplattform sie in plattform-typischer Weise aufrufen kann. Diese Anforderung wird zusätzlich

erschwert, wenn keine klare Abbildung vom ursprünglich eingesetzten `API-Typ` auf einen einzelnen Adaptee möglich ist.

5.2.1 Portierungs-spezifische Kriterien für den Vergleich strukturangleichender Entwurfsmuster

Die geschilderte Problemsituation kann sehr unterschiedlich ausgeprägt sein. In manchen Fällen ist externer Code relevant, insbesondere wenn eine Programmbibliothek portiert wird, die native Schnittstellen für ihre Nutzer anbieten muss. In anderen Fällen gibt es keinen externen Code, etwa wenn eine geschlossene Anwendung portiert wird. Auch die Konstellation von API-Typen, Adaptees und deren Schnittstellen kann unterschiedlich komplex sein. Zudem haben die Programmiersprachen der Ausgangs- und Zielimplementation einen Einfluss darauf, welche Entwurfsmuster eingesetzt werden können. Die Auswahl eines passenden Strukturmusters muss sich dementsprechend an der konkreten Problemsituation ausrichten. Dafür werden die Folgenden Kriterien aufgestellt, anhand derer die Entwurfsmuster verglichen werden.

Anwendbarkeit
Grundlegend ist zu prüfen, welche Vorbedingungen gelten müssen, um das jeweilige Entwurfsmuster anzuwenden. Manche Entwurfsmuster können zum Beispiel nicht mit allen Programmiersprachen implementiert werden, weil sie bestimme Sprachkonstrukte einsetzen. Außerdem kann nicht jedes Muster angewendet werden, um beliebige Unterschiede zwischen den APIs zu behandeln. Beispielsweise können einige Entwurfsmuster nur zur Abbildung eines ursprünglich genutzten Typs auf genau einen Typ der Ziel-API genutzt werden, während andere auch mehrere Typen hinter einer gemeinsamen Schnittstelle verbergen können [Bartolomei u. a. 2010].

Bereitstellung plattform-typischer Schnittstellen
Wie oben erwähnt, kann externer Code eine Rolle spielen, der den portierten Code auf der Zielplattform über native Schnittstellen nutzen soll. Die Entwickler des externen Codes kennen die plattform-spezifischen APIs, die sie innerhalb ihres eigenen Projekts verwenden. Sie sind es gewohnt, deren Typen und Operationen zu nutzen. Die öffentliche Schnittstelle des portierten Codes sollte folglich Parameter und Rückgabewerte mit den nativen Typen der plattform-spezifischen APIs definieren.

Die hier diskutierten Entwurfsmuster stehen in Konflikt mit dieser Anforderung, denn sie passen die Schnittstellen der nativen Typen der Zielplattform an die Schnittstellen der Ausgangsplattform an, oder verbergen die nativen Typen hinter nachgebildeten Schnittstellen. Es ist daher zu bewerten, inwiefern sie verhindern, dass die Typen der plattform-spezifischen APIs als Parameter- und Rückgabetypen für Schnittstellen zu externem Code genutzt werden.

Wiederverwendbarkeit des angleichenden Codes
Tritt dieselbe Plattformabhängigkeit an verschiedenen Stellen im Code auf, so kann die vorgenommene Anpassung idealerweise an all diesen Stellen eingesetzt werden, ohne dass der angleichende Code dupliziert wird. Die untersuchten Entwurfsmuster unterstützen diese Wiederverwendung in unterschiedlichem Maße.

Umfang notwendiger Refactorings am ursprünglichen Code
In Abschnitt 1.3 wurde bereits dargelegt, dass etwaige Änderungen an der ursprünglichen Codebasis aus verschiedenen Gründen nur begrenzt möglich oder sinnvoll sind. Um manche der betrachteten Entwurfsmuster einzuführen, muss der ursprüngliche Code allerdings restrukturiert werden. Es ist zu bewerten, wie umfangreich diese erforderlichen Refactorings sind.

5.2.2 Beschreibung und Bewertung des Entwurfsmusters *Object Adapter*

Das weitverbreitete Strukturmuster *Object Adapter* wird häufig in Portierungsprojekten eingesetzt, um die Schnittstellen zur ursprünglichen Plattform auch in der Zielimplementation bereitzustellen. Sein Einsatz in Portierungsprojekten ist in Abbildung 5.2 dargestellt.

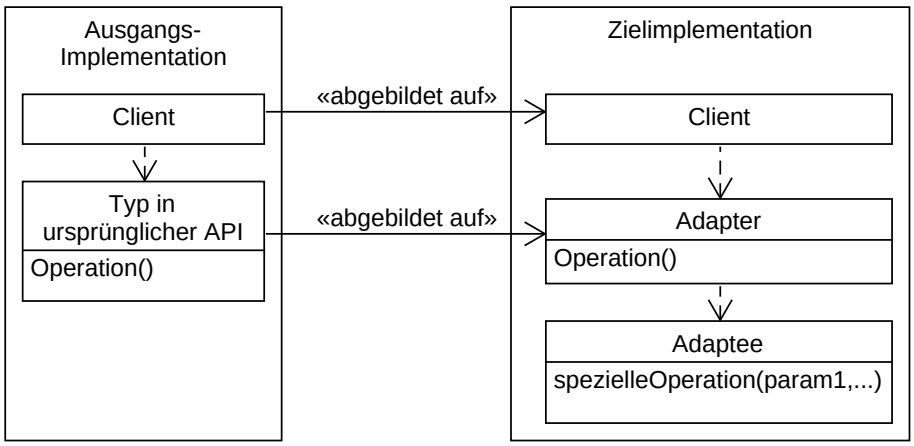

Abbildung 5.2: Das Entwurfsmuster *Object Adapter*

In der portierten Implementation wird eine neue Klasse `Adapter` definiert, deren Schnittstelle exakt die Schnittstelle des ursprünglich eingesetzten Typs nachbildet. Dieser Adapter delegiert die Operationsaufrufe intern an ein Exemplar der anzupassenden `Adaptee`-Klasse. Der ursprünglich eingesetzte Typ wird auf den Adapter abgebildet. Dies kann in Quellcode-Konvertoren als einfaches Typ-Mapping konfiguriert werden.

Ein Adapter kann potenziell auch mehrere Adaptees gleichzeitig hinter seiner Schnittstelle verbergen und sie zur Realisierung der angebotenen Plattformfunktionalität nutzen [Seiffert u. Hummel 2013]. Man kann argumentieren, dass es sich bei solchen

Ausprägungen des Musters mit mehreren Adaptees eigentlich um das Muster *Facade* handelt. *Facade* kapselt die komplizierten Schnittstellen mehrerer Typen [Gamma u. a. 1995, p.185]. Der Zweck der hier betrachteten Anwendungen von Entwurfsmustern ist allerdings nicht die Vereinfachung der Schnittstellen sondern ihre Anpassung an einen ursprünglich eingesetzten Typ. Daher wird hier auch dann von *Object Adaptern* gesprochen, wenn der Adapter mehrere Adaptees kapselt.

Einsatz in den untersuchten Projekten
Das Entwurfsmuster *Object Adapter* findet in allen untersuchten Projekten Anwendung. Beispielsweise definiert die Java-Implementation von *docopt* Adapter für das ursprünglich eingesetzte Python-Modul `Python.re`, das reguläre Ausdrücke verarbeitet. Dazu werden im entsprechenden Adapter die Klassen `java.util.regex.Pattern` und `java.util.regex.Matcher` genutzt.

In der ursprünglichen Form des Musters nach Gamma u. a. [1995, S. 141] gibt es ein `Target`-Interface, das die gemeinsame Schnittstelle des Adapters und des ursprünglich eingesetzten Typs definiert. Ein solches Interface ist in der ursprünglichen Form des Musters notwendig, da Adapter und ursprünglich eingesetzter Typ sich in *derselben Codebasis* befinden und gegeneinander austauschbar sein sollen. In statisch getypten Programmiersprachen ist dazu das `Target`-Interface als gemeinsamer Supertyp nötig. In den untersuchten Portierungsprojekten hingegen befinden sich der ursprünglich eingesetzte Typ und der Adapter in getrennten Codebasen, sodass das Interface ausgelassen wird. Stattdessen nutzen die Client-Klassen den Adapter direkt. Die ursprünglich eingesetzten Typen werden unmittelbar auf entsprechende Adapter abgebildet.

Im Gegensatz dazu gibt es in den Umsetzungen des Musters in den Xamarin-Plugins sehr wohl explizite `Target`-Interfaces. Beispielsweise definiert das Interface `IAudioPlayer` Operationen um Audio-Dateien abzuspielen. Es wird für jede Plattform von einer plattform-spezifischen Klasse implementiert, die jeweils `AudioPlayerImplementation` heißt.

Bewertung der Anwendbarkeit
Das Entwurfsmuster *Object Adapter* kann in jeder objektorientierten Programmiersprache umgesetzt werden. Es ist auch dann anwendbar, wenn mehr als ein Adaptee genutzt werden muss, um den ursprünglich eingesetzten Typ nachzubilden.

Bewertung der Bereitstellung plattform-typischer Schnittstellen
Nach Einführung des Musters hängt der portierte Client von der `Adapter`-Klasse ab. Entwickler von externem Code sind an die nativen Typen der Plattform, also an die Adaptees, gewöhnt. Für sie sollte die intern genutzte Angleichung der Ziel-API transparent sein. Ein Adapter sollte folglich nicht an der Schnittstelle zu externem Code angeboten werden.

In den untersuchten Portierungsprojekten definieren Schnittstellen zu externem Code daher niemals Parameter oder Rückgabewerte vom Typ des Adapters, sondern des Adaptees. Dazu muss der Adaptee aus dem Adapter entpackt werden, ehe er an externe Aufrufer übergeben wird. Umgekehrt müssen Adaptees, die als Parameter an den portierten Client übergeben werden, zunächst in einem Adapter „verpackt“werden, um sie im portierten Code zu nutzen.

Für den Einsatz von Konvertern ist das hinderlich, da der ursprünglich eingesetzte Typ nicht eindeutig auf den Adapter oder den Adaptee abgebildet werden kann. Hinzu kommt, dass Quellcode-Konvertoren den Code zum Ver- und Entpacken des Adaptees in der Regel nicht automatisch generieren. Dies muss manuell implementiert werden.

Insgesamt ist es bei Einsatz des Entwurfsmusters *Object Adapter* also möglich, den Adaptee als Parameter- und Rückgabetyp einzusetzen. Dies ist aber mit zusätzlichem, manuell zu verfassendem Code verbunden.

Bewertung der Wiederverwendbarkeit des isolierten plattform-spezifischen Quellcodes
Der Adapter kann wiederverwendet werden, um den Adaptee an beliebig vielen Stellen im Code anzupassen. Er kann auch wiederverwendet werden, um beliebige Subtypen des Adaptees anzupassen. Dazu muss der Adapter lediglich einen Konstruktor oder eine Zugriffsoperation definieren, die die konkrete Adaptee-Instanz setzt.

Bewertung des Umfangs notwendiger Refactorings am ursprünglichen Code
Wird der Object Adapter in der Zielimplementation eingeführt, so muss die Ausgangsimplementation keinem Refactoring unterzogen werden.

5.2.3 Beschreibung und Bewertung des Entwurfsmusters *Class Adapter*

Abbildung 5.3 zeigt den Einsatz des Entwurfsmusters *Class Adapter*, das in zwei der untersuchten Portierungsprojekte auftritt.

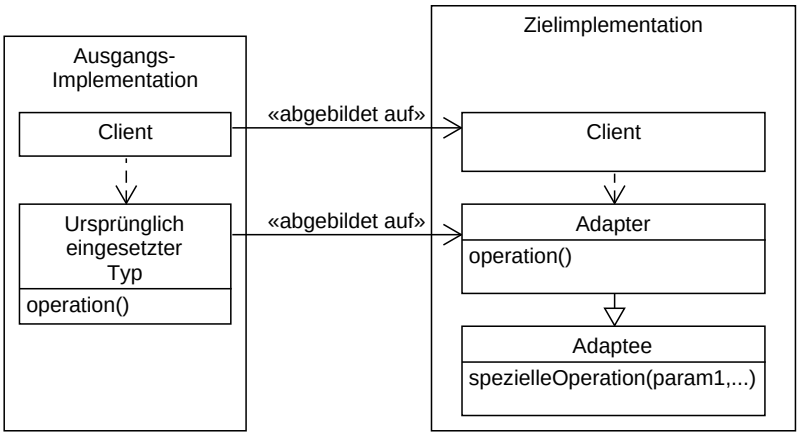

Abbildung 5.3: Das Entwurfsmuster *Class Adapter*

Ähnlich wie beim Object Adapter wird in der portierten Implementation ein zusätzlicher Typ `Adapter` eingeführt, der dem ursprünglich eingesetzten Typ in Bezug auf seine Schnittstelle entspricht. Im Gegensatz zum *Object Adapter* nutzt der Adapter den anzupassenden Adaptee allerdings nicht, sondern erbt von ihm. Um die angebotene Funktionalität auszuführen, ruft er die plattform-spezifischen Operationen des Adaptees zur Laufzeit an sich selbst auf.

Einsatz in den untersuchten Projekten
Class Adapter wird nur in zwei der untersuchten Projekte eingesetzt. In der Zielimplementation von Lucene.Net adaptiert die Klasse `Lucene.Net.Support.IO.ByteArrayOutputStream` den Adaptee `MemoryStream`, um die Schnittstelle der Java-Klasse `java.io.ByteArrayOutputStream` nachzubilden, die in der ursprünglichen Implementation in Java genutzt wurde.

Im Projekt NGit fungiert beispielsweise die Klasse `AList<T>` als Adapter für den Adaptee `System.Collections.Generic.List<T>`. Sie bietet Operationen wie `TrimToSize()` an, die der ursprünglich eingesetzte Typ `java.util.ArrayList` definiert.

Bewertung der Anwendbarkeit
Das Entwurfsmuster kann in beliebigen objektorientierten Programmiersprachen eingesetzt werden. Ein Class Adapter kann allerdings in den meisten Programmiersprachen nicht von mehreren Adaptees erben. Er ist folglich nur dann einsetzbar, wenn der ursprünglich eingesetzte Typ auf genau einen Typ in der Zielplattform abgebildet werden kann. Das Muster kann außerdem nur dann angewendet werden, wenn der Adaptee die

Vererbung zulässt und einen passenden Konstruktor definiert, der vom Adapter als Superkonstruktor aufgerufen werden kann. Beispielsweise kann das Muster in Java nicht auf einen Adaptee angewendet werden, der die Vererbung mit dem Modifikator `final` verhindert. Gleiches gilt für Adaptees, die ausschließlich Konstruktoren mit dem Zugriffsmodifikator `private` definieren.

Bewertung der Bereitstellung plattform-typischer Schnittstellen
Da der Class Adapter vom Adaptee erbt, bietet er auch sämtliche im Adaptee definierte plattform-spezifische Operationen an. Der Adapter kann daher als Rückgabewert problemlos an externen Code übergeben werden. Entwickler des externen Codes können ihn genau so nutzen, wie sie sonst den Adaptee nutzen würden. Der Rückgabewert kann sogar als `Adaptee` deklariert werden, sodass es für externe Aufrufer völlig transparent ist, dass zur Laufzeit eine `Adapter`-Instanz zurückgegeben wird. Im Gegensatz zum Object Adapter ist es dazu weder möglich noch notwendig, den Adaptee aus seinem Adapter zu „entpacken“.

Allerdings ist das Muster Class Adapter ungeeignet, wenn der Adaptee als Parametertyp an Schnittstellen zu externem Code eingesetzt werden soll. Ein solcher, als Parameter übergebener Adaptee kann nicht nachträglich in einen Class Adapter „eingepackt“ werden.

Bewertung der Wiederverwendbarkeit des isolierten plattform-spezifischen Quellcodes
Ein Class Adapter kann zur Abbildung beliebig vieler Abhängigkeiten zum ursprünglich eingesetzten Typ wiederverwendet werden. Im Unterschied zum Object Adapter kann der Class Adapter allerdings nicht wiederverwendet werden, um Subtypen des Adaptees anzupassen, denn der Adapter erbt explizit von einem konkreten Adaptee [Gamma u. a. 1995]. Um Subtypen des Adaptees anzupassen, muss der Code des Adapters folglich dupliziert werden.

Bewertung des Umfangs notwendiger Refactorings am ursprünglichen Code
Wie schon beim Object Adapter sind keine Anpassungen des ursprünglichen Codes notwendig, wenn das Entwurfsmuster in der Zielimplementation eingesetzt wird.

5.2.4 Beschreibung und Bewertung des Entwurfsmusters *Extension Method*

Das Entwurfsmuster *Extension Method* ist in Abbildung 5.4 zu sehen.

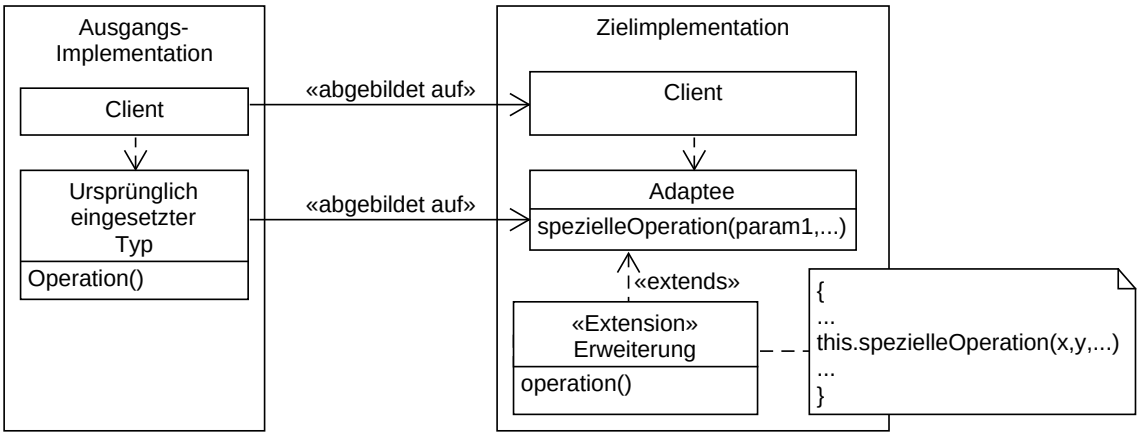

Abbildung 5.4: Das Entwurfsmuster *Extension Method*

Eine Extension Method erweitert einen bestehenden Typ um eine Methode. Zur Angleichung von Schnittstellen wird das Muster eingesetzt, indem der Adaptee um die Methoden erweitert wird, die ihm im Vergleich zum ursprünglich eingesetzten Typ fehlen. Ähnlich wie die zuvor diskutierten Adapter-Muster delegiert die ergänzte Methode den Aufruf an die nativen Methoden des Adaptees. Der entscheidende Unterschied liegt darin, dass kein zusätzlicher Typ in der portierten Implementation eingeführt wird, sondern der bestehende Adaptee *erweitert* wird. Nach Einführung der Extension Method kann die ergänzte Methode unmittelbar an allen Instanzen von `Adaptee` aufgerufen werden, auch wenn diese Instanzen außerhalb des portierten Codes erzeugt wurden.

Einsatz in den untersuchten Projekten
Das Entwurfsmuster *Extension Method* kommt in allen untersuchten Projekten bis auf die Xamarin-Erweiterungen zum Einsatz.

Im Projekt NGit werden Extension Methods für 24 verschiedene Typen in einer gemeinsamen statischen Klasse `Sharpen.Extensions` definiert. Diese Klasse ist dementsprechend groß und unübersichtlich. Außerdem ist es bei dieser gesammelten Ablage schwer, Extension Methods für einzelne Typen in anderen Projekten wiederzuverwenden, ohne sie zu duplizieren. Im Projekt Lucene.Net werden die Erweiterungen verschiedener Typen stattdessen jeweils in einer eigenen statischen Klasse abgelegt, was die Übersichtlichkeit und Wiederverwendbarkeit erhöht.

Grundsätzlich sind bei der Umsetzung des Musters zwei Varianten zu unterscheiden. Die erste Variante implementiert das Entwurfsmuster mit einem spezifischen Sprachkonstrukt, das die Erweiterung eines bestehenden Typs erlaubt. Sämtliche der untersuchten Portierungsprojekte, in denen die Zielsprache ein solches Sprachkonstrukt definiert,

nutzen dieses Konstrukt. In Swift trägt es beispielsweise den Namen *Extension*. Es wird in der Swift-Implementation von docopt unter anderem eingesetzt, um den Typ `String` zu erweitern. `String` wird beispielsweise um die Operation `partition(separator)` ergänzt, die Strings in Python anbieten. Sie berechnet für einen String das Präfix und Postfix in Bezug auf die übergebene Zeichenkette `separator`.

Die zweite Variante des Musters simuliert die Erweiterung des Adaptees lediglich. Dazu wird eine global zugreifbare, statische Methode definiert, die als ersten Parameter eine Instanz des Adaptees entgegennimmt. Diese Methode kann im Gegensatz zu richtigen Extension Methods nicht am Adaptee selbst aufgerufen werden. Stattdessen ruft der Entwickler die statische Methode auf und übergibt den Adaptee als ersten Parameter, gefolgt von den übrigen Parametern der Methode. Ein Beispiel dafür ist die Methode `goog.str.startsWith(string, prefix)`, die in der JavaScript-Implementation von libphonenumbers verwendet wird. Sie prüft, ob ein String mit dem übergebenen Präfix beginnt. Der ursprünglich eingesetzte Typ `String` in Java bietet dazu die Methode `startsWith(prefix)` als Instanzmethode an. Diese Nachbildung verfehlt offensichtlich das Ziel, die Schnittstellen anzugleichen, da der statische Aufruf sich vom Aufruf einer Instanzmethode unterscheidet.

Bewertung der Anwendbarkeit
Soll ein Adaptee um eine Methode erweitert werden, so bedarf es eines entsprechenden Sprachkonstrukts, das die tatsächliche Erweiterung ermöglicht. Beispiele dafür sind *Extensions* in der Sprache Swift oder *Extension Methods* in C#. In JavaScript können Extension Methods als *Function* dem *Prototype*-Objekt eines Typs hinzugefügt werden.

In anderen Sprachen wie beispielsweise Java existiert ein solches Konstrukt nicht. Dann kann das Entwurfsmuster wie oben beschrieben durch eine statische Methode simuliert werden, die den Adaptee als ersten Parameter entgegennimmt. Diese Option hat den Nachteil, dass der Adaptee nicht tatsächlich erweitert wird und somit der Unterschied zwischen den Methodenaufrufen im ursprünglichen Quelltext und denen im portierten Quelltext bestehen bleibt. Insbesondere wenn ein Quellcode-Konverter eingesetzt wird, ist diese Option nur zielführend, wenn Mappings von Instanzmethoden auf statische Methoden konfiguriert werden können.

Je nach Sprache können zudem nicht alle Operationen als Extension Method definiert werden. C# erlaubt es beispielsweise nicht, zusätzliche Konstruktoren als Extension Method zu definieren. Zudem kann nur ein Adaptee in derselben Extension Method erweitert werden. Werden zur Erfüllung der benötigten Funktionalität weitere Adaptees benötigt, so müssen diese innerhalb der Extension Method erzeugt werden.

Bewertung der Bereitstellung plattform-typischer Schnittstellen
Durch die ergänzten Operationen wird die Schnittstelle des Adaptees verändert. Die zusätzlichen Operationen sind den Entwicklern des externen Codes nicht bekannt und

verfremden den Adaptee somit in gewisser Weise. Im Gegensatz zu den Adapter Patterns kann der Adaptee aber problemlos als Parameter und Rückgabetyp an den Schnittstellen zu externem Code eingesetzt werden, da seine nativen Operationen weiterhin zur Verfügung stehen. Ein Ver- bzw Entpacken des Adaptees wie beim Object Adapter ist dazu nicht notwendig.

Bewertung der Wiederverwendbarkeit des isolierten plattform-spezifischen Quellcodes
Die Erweiterung des Adaptees gilt automatisch projektweit für alle vom Adaptee abhängigen Code-Elemente. Extension Methods gelten darüber hinaus auch für sämtliche Subtypen des Adaptees, sodass auch hier keine explizite Wiederverwendung oder wiederholte Anpassung notwendig ist. Auch die projektübergreifende Wiederverwendung von Extension Methods ist unproblematisch. Im Gegensatz zu Adaptern können Extension Methods auch zur Ergänzung einzelner Methoden ohne Anpassung des Quellcodes in andere Projekte übernommen werden.

Bewertung des Umfangs notwendiger Refactorings am ursprünglichen Code
Um das Entwurfsmuster einzusetzen, sind keine Änderungen am ursprünglichen Code notwendig.

5.2.5 Beschreibung und Bewertung des Entwurfsmusters *Generation Gap*

Das Entwurfsmuster *Genration Gap* wurde ursprünglich im Kontext domänenspezifischer Sprachen eingesetzt, um generierten Code von manuell zu entwickelndem Code zu trennen [Vlissides 1996]. Abbildung 5.5 stellt es im Kontext der Portierung dar.

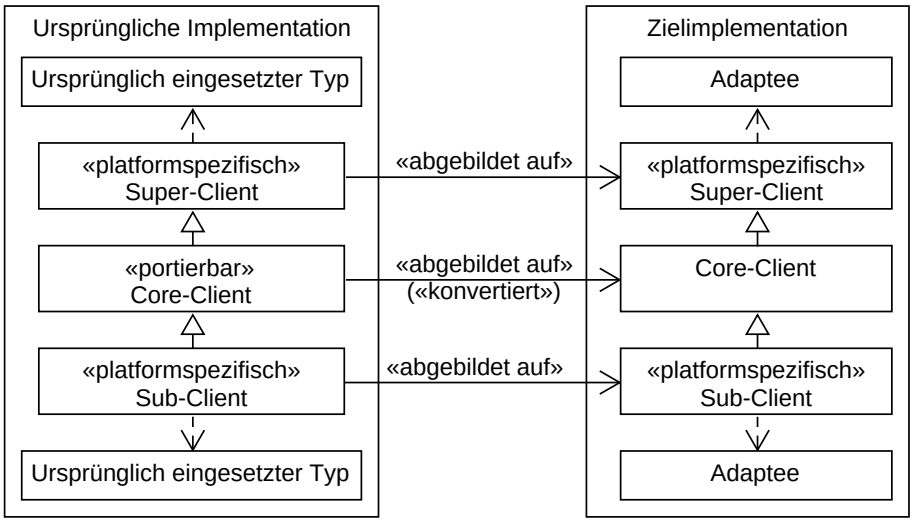

Abbildung 5.5: Das Entwurfsmuster *Generation Gap*

Es spaltet den ursprünglichen Client in portierbare und nicht portierbare Klassen. Die plattform-spezifischen Abhängigkeiten zu Typen der Ausgangsplattform werden dazu

in einen Supertyp oder Subtyp des Clients ausgelagert [Fowler 2010, S. 572]. Diese plattform-spezifischen Typdefinitionen werden hier als Super-Client bzw. Sub-Client bezeichnet. Der verbleibende portierbare Client wird in dieser Arbeit als Core-Client bezeichnet. Die portierenden Entwickler überführen diesen Core-Client in eine Entsprechung in der Zielimplementation. Sie setzen dazu gegebenenfalls einen Quellcode-Konverter ein. Sub- und Super-Client werden in der Zielimplementation händisch re-implementiert. Ihre Schnittstelle zum Core-Client ist in beiden Implementationen gleich.

Um den plattformspezifischen Anteil in einen Sub-Client auszulagern, kann der Core-Client eine abstrakte Methode als Schablonenmethode definieren. Der Sub-Client implementiert diese dann plattform-spezifisch unter Nutzung der jeweiligen Plattform-API. Die portierte Version des Sub-Clients implementiert ebenfalls die abstrakte Methode, verwendet dabei allerdings die entsprechende API der Zielplattform. So abstrahiert der Core-Client von den plattform-spezifischen APIs, die er indirekt durch Nutzung des Sub-Clients verwendet.

Ganz ähnlich können API-Abhängigkeiten in den Super-Client ausgelagert werden. Dazu werden plattform-spezifische Methoden im Super-Client implementiert. Der Core-Client ruft diese auf, ohne selbst die plattform-spezifischen APIs zu nutzen.

Einsatz in den untersuchten Projekten
Generation Gap wird im Quellcode beider Xamarin-Erweiterungen eingesetzt, aber nicht in den übrigen untersuchten Projekten. Ursächlich dafür ist möglicherweise, dass die Entwickler der übrigen Projekte die ursprüngliche Implementation nicht verändern konnten, sodass die Aufspaltung der Client-Klassen nicht infrage kam.

Im Xamarin Media Manager Plugin fungiert beispielsweise die abstrakte Klasse `MediaManagerBase` als Core-Client. Sie implementiert die Abläufe, die etwa beim Starten, Anhalten und Auswählen wiederzugebender Mediendateien durchgeführt werden. Diese Abläufe werden für die Plattformen Android, iOS, MacOS und tvOS jeweils durch einen konkreten Sub-Client `MediaManagerImplementation` konkretisiert, der dazu plattform-spezifische APIs nutzt.

Bewertung der Anwendbarkeit
Generation Gap kann mittels jeder objektorientierten Programmiersprache realisiert werden, sofern die notwendigen Refactorings an der ursprüngliche Implementation durchgeführt werden können.

Bewertung der Bereitstellung plattform-typischer Schnittstellen
Plattform-typische Schnittstellen können vom Super- und Sub-Client angeboten werden. Es ist allerdings kritisch, das Entwurfsmuster einzusetzen, wenn externer Code bereits von plattform-spezifische Operationen des Clients abhängt. Solcher externer Code wird

gegebenenfalls fehlerhaft, wenn diese Operationen bei der Aufspaltung des Clients verschoben werden.

Bewertung der Wiederverwendbarkeit des isolierten plattform-spezifischen Quellcodes
Der plattform-spezifische Sub-Client erbt explizit vom Core-Client. Somit kann er nicht wiederverwendet werden, um andere Clients vom ursprünglich eingesetzten Typ zu entkoppeln.

Ein plattform-spezifischer Super-Client kann dagegen von mehreren Core-Clients wiederverwendet werden, die von ihm erben. Das ist allerdings nur dann möglich und sinnvoll, wenn diese Core-Clients dieselbe Abstraktion von der eingesetzten API benötigen. Beispielsweise könnte ein Super-Client `HttpClient` die Kommunikation mit einem Server auf Basis des Protokolls HTTP implementieren. Wenn mehrere Klassen der ursprünglichen Implementation eine solche Kommunikation durchführen, so kann aus diesen Klassen jeweils ein Core-Client abgespalten werden, der von `HttpClient` erbt. Die Core-Clients würden dann lediglich die anzusprechende URL definieren und die Antworten des Servers verarbeiten. Setzt der Super-Client hingegen ein fachliches Konzept um, das spezifisch für den ursprünglichen Client ist, so kann er nicht wiederverwendet werden. Beispielsweise könnte ein Super-Client, der Fahrzeugdaten von einem Webserver abruft und die Antworten des Servers deserialisiert, nicht in anderen fachlichen Kontexten wiederverwendet werden.

Bewertung des Umfangs notwendiger Refactorings am ursprünglichen Code
Um das Entwurfsmuster Generation Gap zu implementieren, muss der ursprüngliche Client in mindestens zwei Klassen aufgeteilt werden. Die Teilung des Clients ist eine kreative Aufgabe, bei der ein Entwickler entscheiden muss, welcher Teil des Codes im Core-Client verbleibt und welcher Teil in eine plattform-spezifische Klasse extrahiert wird. Sie kann dementsprechend nicht automatisiert werden.

5.2.6 Zusammenfassender Vergleich

Abbildung 5.6 bietet einen Überblick über die diskutierten Entwurfsmuster. Oben links in der Abbildung ist ein Ausschnitt der ursprünglichen Implementation dargestellt. Dort ist zu sehen, das der Client in der ursprünglichen Implementation unmittelbar von einem plattform-spezifischen Typ abhängt, der in einer API der ursprünglichen Plattform definiert ist. Rechts daneben ist die Angleichung der Ziel-API mit den Entwurfsmustern Object Adapter und Class Adapter dargestellt. Darunter ist die Angleichung mittels Extension Method zu sehen. Im unteren Bereich der Abbildung ist das Entwurfsmuster Generation Gap abgebildet, das plattform-spezifische Klassen aus dem ursprünglichen Client extrahiert. Diese werden in der Zielimplementation re-implementiert.

Anwendbarkeit
Einschränkungen bezüglich der Programmiersprache macht nur das Entwurfsmuster Extension Method. Es setzt ein Sprachkonstrukt zur Erweiterung eines bestehenden Typen voraus. Das Muster Class Adapter kann ohne Mehrfachvererbung nur angewendet werden, wenn der ursprünglich eingesetzte Typ auf genau einen Adaptee abgebildet werden kann. Außerdem muss der Adaptee es zulassen, dass der Adapter von ihm erbt. Die Entwurfsmuster Object Adapter und Generation Gap haben keine solchen Vorbedingungen.

Bereitstellung plattform-typischer Schnittstellen
Soll der Adaptee an der Schnittstelle zu externem Code als Parameter- oder Rückgabetyp eingesetzt werden, so ist dies mit dem Entwurfsmuster Extension Method am einfachsten. Es führt in der Zielimplementation keine zusätzlichen Typen ein, sondern erweitert den bestehenden Adaptee. Dieser kann ohne Vorverarbeitung an externe Aufrufer übergeben und von diesen nativ genutzt werden. Der einzige Nachteil besteht darin, dass die Schnittstelle des Adaptees durch die zusätzlichen Operationen an Übersichtlichkeit verliert.

Beim Einsatz eines Object Adapter ist es notwendig, den Adaptee an der Schnittstelle zu externem Code in einen Adapter zu verpacken, bzw. diesen zu entpacken, ehe er an externe Aufrufer zurückgegeben werden kann. Das ist insbesondere beim Einsatz automatischer Quellcode-Konvertoren kritisch, da sie den Code für das Ver- und Entpacken des Adaptees nicht automatisch erzeugen.

Wird ein Class Adapter eingesetzt, so kann dieser problemlos an externen Code übergeben werden. Der externe Code kann den Class Adapter genau so nutzen wie den Adaptee, da es sich um einen Subtypen handelt. Ein Class Adapter sollte an Schnittstellen zu externem Code allerdings nicht als Parametertyp eingesetzt werden, da die Adaption für Entwickler von externem Code transparent sein sollte.

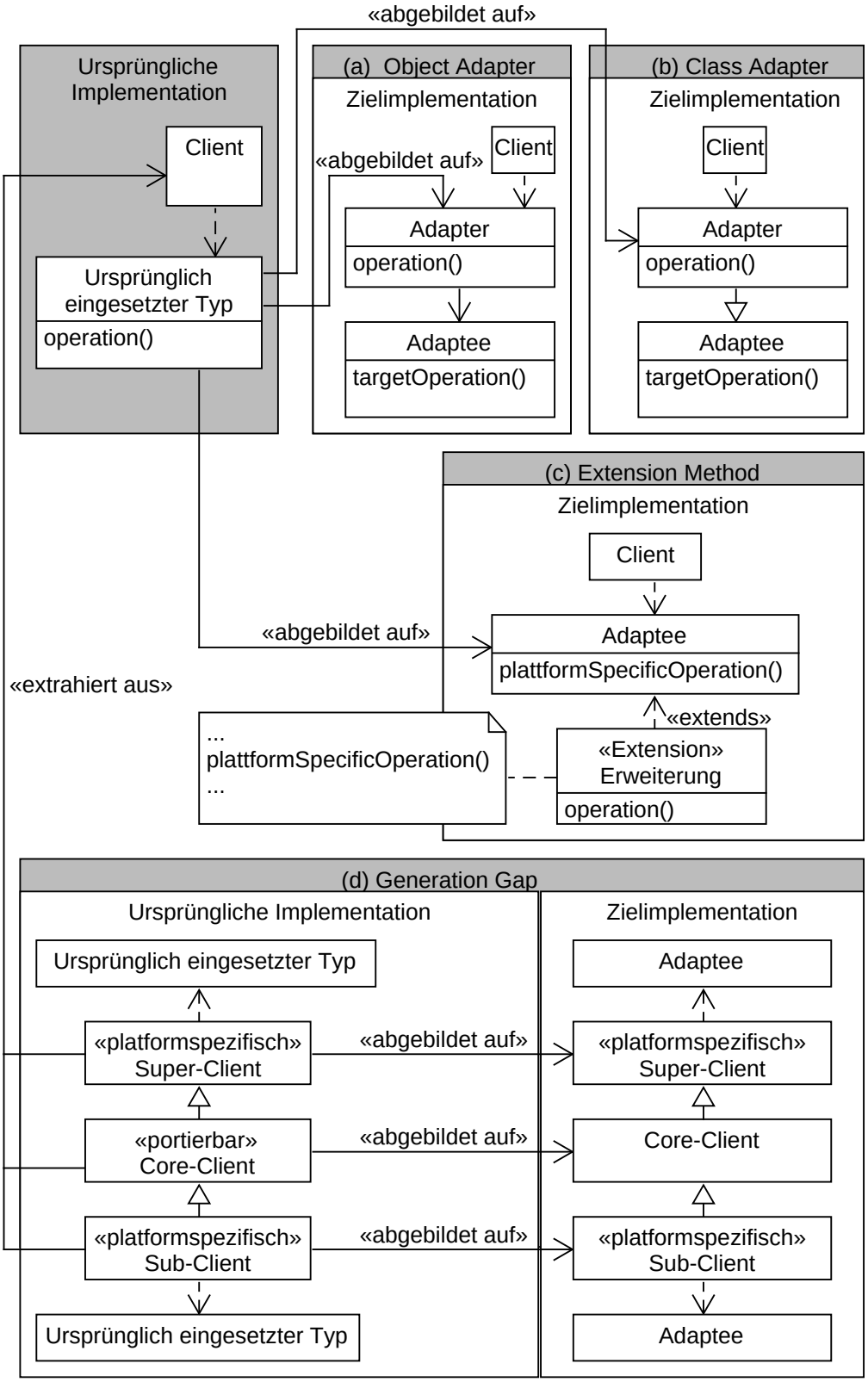

Abbildung 5.6: Synoptische Darstellung der Strukturmuster Object Adpter, Class Adapter, Extension Method und Generation Gap, jeweils in der für Portierungsprojekte typischen Ausführung

Wird entsprechend des Entwurfsmusters Generation Gap ein plattform-spezifischer Sub- oder Super-Client vom ursprünglichen Client abgespalten, so kann der abgespaltene Typ eine beliebige plattform-spezifische Schnittstelle definieren. Dies ist allerdings mit einer Änderung des ursprünglichen Codes verbunden, der möglicherweise bereits von externem Code eingesetzt wird. Dies birgt die Gefahr, dass die bislang von externem Code genutzte Schnittstelle zum Client nicht mehr existiert.

Wiederverwendbarkeit des isolierten plattform-spezifischen Quellcodes
Die Anpassungen, die durch Extension Methods vorgenommen werden, gelten automatisch für alle Teile des Codes, die den Adaptee nutzen. Auch projektübergreifend ist die Wiederverwendung einzelner Extension Methods problemlos möglich.

Bei Einsatz eines Object Adapters oder eines Class Adapters ist es ebenfalls möglich, alle Abhängigkeiten zum Adaptee durch denselben adaptierenden Code zu entkoppeln. Anders als bei Extension Methods müssen dazu allerdings die Abhängigkeiten zum Adaptee einzeln durch Abhängigkeiten zum Adapter ersetzt werden.

Die Anpassung, die ein Class Adapter vornimmt, ist zudem auf einen konkreten Adaptee beschränkt, während ein Object Adapter auch sämtliche Subtypen des Adaptees anpassen kann.

Wird das Muster Generation Gap eingesetzt, so kann ein etwaiger Sub-Client nicht wiederverwendet werden, um weitere Clients von ihren Abhängigkeiten zur Plattform zu entkoppeln. Der Super-Client ist dagegen wiederverwendbar, sofern seine Zuständigkeit sich auf die Abstraktion von der eingesetzten API beschränkt. Dann können Core-Clients mit verschiedenen fachlichen Aufgaben von ihm erben.

Umfang notwendiger Refactorings am ursprünglichen Code
Die Muster Class Adapter, Object Adapter und Extension Method lassen den ursprünglichen Code unberührt. Generation Gap hingegen erfordert, dass plattform-spezifischer Code aus dem ursprünglichen Client in einen Sub- oder Super-Client extrahiert wird.

5.3 Richtlinie für den Einsatz von Strukturmustern in Portierungsprojekten

Aus den obigen Untersuchungen der Entwurfsmuster wurde eine Richtlinie abgeleitet, die besagt, unter welchen Voraussetzungen der Einsatz des jeweiligen Entwurfsmusters möglich und angemessen ist [Stehle u. Riebisch 2018a]. Sie gibt darüber hinaus Hinweise zur Umsetzung einzelner Muster. Tabelle 5.2 bietet eine Übersicht über diese Richtlinie.

Kriterium	Object Adapter	Class Adapter	Extension Method	Generation Gap
Mehrere Adaptees	✓	✗	✗	✓
Externer Code	(✗)	(✗)	✓	(✗)
Benötigt Sprachkonstrukt für Typ-Erweiterungen	✗	✗	✓	✗
Ursprünglicher Code soll unverändert bleiben	✓	✓	✓	✗
Abhängigkeit tritt mehrfach auf	✓	✓	✓	(✓)

Tabelle 5.2: Tabellarische Übersicht über die Richtlinie zum Einsatz der vier strukturangleichenden Entwurfsmuster

Die erste Spalte der Tabelle erfasst die Entscheidungskriterien zur Wahl eines Entwurfsmusters. Ein Haken in den folgenden Spalten drückt aus, dass das jeweilige Muster das Kriterium erfüllt, ein Kreuz markiert entsprechend die Nichterfüllung des Kriteriums. Klammern um das jeweilige Symbol schränken dieses Urteil entsprechend der folgenden Ausführungen ein.

Mithilfe des Musters **Object Adapter** kann ein einzelner oder auch mehrere Adaptees an eine vereinheitlichte Schnittstelle angepasst werden. Dafür ist im Gegensatz zu den Mustern Class Adapter und Generation Gap keine Vererbung notwendig und auch kein spezifisches Sprachkonstrukt wie beim Entwurfsmuster Extension Method. Der Nachteil seines Einsatzes besteht darin, dass der Adaptee zur Laufzeit in den Adapter „verpackt“werden muss. Daher sind Object Adapters in folgender Situation anwendbar und angemessen:

- Die Schnittstellen eines oder mehrerer Adaptees und ggf. auch ihrer Subtypen sollen an den ursprünglich eingesetzten Typ angeglichen werden **und**

  (a) Der Adaptee soll nicht als Parameter oder Rückgabetyp an Schnittstellen zu externem Code eingesetzt werden

  oder

(b) Die Portierung des Clients wird manuell durchgeführt, sodass das Ver- und Entpacken des Adaptees an Schnittstellen zu externem Code händisch implementiert werden kann. Der Adaptee kann dann als Parameter- bzw. Rückgabetyp eingesetzt werden.

Im Gegensatz dazu kann der Adaptee beim Einsatz des Musters **Class Adapter** nicht nachträglich, also erst zur Laufzeit in den Adapter „verpackt“werden. Zudem kann in der Regel nur von einem Adaptee geerbt werden, um seine Schnittstelle anzupassen. Vorteilhaft ist, dass der Adapter unmittelbar wie ein Adaptee einsatzbar ist und somit als Rückgabewert an externen Code übergeben werden kann. Der Einsatz von Class Adaptern ist dementsprechend in folgender Situation möglich und angemessen:

- Ein Adaptee soll an einen ursprünglich eingesetzten Typ angepasst werden, nicht aber etwaige Subtypen des Adaptees
- Der Adaptee lässt die Vererbung zu und hat einen Konstruktor, der für den Adapter als Superkonstruktor zugreifbar ist.
- Der Adaptee soll nicht als Parameter an Schnittstellen zu externem Code eingesetzt werden.

Das Muster **Extension Method** erspart den Entwicklern das Ver- und Entpacken des Adaptees und führt keinen zusätzlichen Typ in der Zielplattform ein. Es ist daher als einfachste Option für die Anpassung der Schnittstelle des Adaptees zu betrachten. Der Einsatz des Musters ist in folgender Situation möglich und angemessen:

- Die Schnittstellen eines Adaptees und seiner etwaigen Subtypen sollen an die des ursprünglich eingesetzten Typs angepasst werden.
- Die Zielsprache bietet ein Sprachkonstrukt zur Ergänzung zusätzlicher Methoden zu einem bestehenden Typ.

Die Entwickler sollten Extension Methods für verschiedene Typen in separaten Dateien definieren, um Wartbarkeit und Wiederverwendbarkeit der Erweiterungen zu gewährleisten.

Der Einsatz des Musters **Generation Gap** ist komplizierter und erfordert, dass der Entwickler eine sinnvolle Abstraktion als Super-Client extrahieren kann. Sein Einsatz ist daher in folgender Situation möglich und angemessen:

- Es soll eine Abhängigkeit zu einem einzelnen, konkreten Adaptee oder eine Kombination solcher Abhängigkeiten isoliert werden.
- Die portierenden Entwickler haben schreibenden Zugriff auf die ursprüngliche Codebasis.

- Diese Codebasis kann geändert werden, ohne die Funktionsfähigkeit von externem Code zu gefährden.

Insbesondere bei Clients, die selbst als Abstraktion von der ursprünglich eingesetzten Plattform zu verstehen sind, ist der Einsatz von Generation Gap angemessen. Beispiele dafür finden sich in den untersuchten Xamarin-Plugins, die die Funktionalitäten der konkreten Plattformen durch spezialisierte Sub-Clients zur Verfügung stellen. Das Muster sollte allerdings nur angewendet werden, wenn die entstehende Vererbungsbeziehung zwischen Super-Client und Core-Client, bzw. Core-Client und Sub-Client angemessen ist. Das ist insbesondere dann der Fall, wenn die abgespaltenen Oberklassen ein allgemeines Konzept umsetzen, welches die Unterklassen spezialisieren. Ansonsten ist das Muster Object Adapter vorzuziehen, mit dem ebenfalls mehrere Adaptees entkoppelt werden können. Damit folgt diese Richtlinie dem zweiten Prinzip des objektorientierten Entwurfs nach Gamma u. a. [1995, S. 20] : „Favor object composition over class inheritance“.

In den untersuchten Portierungsprojekten wurden Object Adapter, Class Adapter bzw. Extension Method in der jeweiligen Zielimplementation eingesetzt, um dort einen Adaptee, wie oben beschrieben, an einen ursprünglich genutzten Typ anzupassen. Grundsätzlich können Adapter und Erweiterungen stattdessen auch in der Ausgangsimplementation eingesetzt werden, um die ursprünglich genutzte API an die Ziel-API anzupassen. Dies kann dann sinnvoll sein, wenn die Schnittstelle der Zielplattform leichter zu nutzen ist als die der Ausgangsplattform, oder wenn das Muster Extension Method eingesetzt werden soll und nur die Ausgangssprache ein entsprechendes Erweiterungskonstrukt enthält. Wie bereits in Abschnitt 5.2.1 dargelegt, sollte der Quellcode der Ausgangsimplementation allerdings nach Möglichkeit unangetastet bleiben. Die portierenden Entwickler sollten folglich die Anpassungen nach Möglichkeit in der Zielimplementation vornehmen.

5.4 Vermeidung plattform-spezifischer Instanziierungen durch Kombination von Struktur- und Erzeugungsmustern

Alle oben diskutierten Strukturmuster zielen darauf ab, die Schnittstelle des ursprünglich eingesetzten Typs auf der Zielplattform nachzubilden. Dadurch kann die Nutzungs- oder Vererbungsbeziehung des Clients zum ursprünglich eingesetzten Typen in der Zielimplementation nachgebildet werden. Die Entwurfsmuster definieren allerdings nicht, wie der Client Zugriff auf eine Instanz des Adapters, des erweiterten Adaptees, bzw. des Sub-Clients erhält. Diese angeglichenen Typen müssen aber instanziiert werden, ehe sie verwendet werden können. Auch die Instanziierung kann plattform-spezifisch sein. Beispielsweise kann der ursprünglich genutzte Typ einen Konstruktor mit einem Parameter

definieren, während der Konstruktor des Adaptees in der Zielplattform zwei Parameter erfordert. Ist die Angleichung der Konstruktoren nicht mithilfe der diskutierten Strukturmuster möglich, so können zwei Strategien eingesetzt werden, um die Instanziierung an zusätzliche plattform-spezifische Klassen zu delegieren.

Zum einen können der ursprüngliche Client und seine portierte Entsprechung jeweils per *Dependency Injection* Zugriff auf die benötigten Instanzen erhalten. Dazu definiert der Client beispielsweise einen Konstruktorparameter, der eine Instanz des benötigten Typs entgegennimmt. Diese Taktik verschiebt das Problem der Instanziierung an die Klassen, die den Client erzeugen. Sie ist daher nur sinnvoll, wenn diese Klassen nicht selbst zum portierten Code gehören.

Zum anderen können plattform-spezifische Instanziierungen durch den Einsatz von Erzeugungsmustern verborgen werden. Solche Erzeugungsmuster definieren ein zusätzliches Code-Element, beispielsweise eine Klasse, die ausschließlich für die Bereitstellung von Instanzen zuständig ist. Sie führen eine plattform-übergreifend einheitliche Schnittstelle ein, die der ursprüngliche und der portierte Client nutzen, um die benötigten Instanzen zu erhalten. Beispiele für solche Erzeugungsmuster sind *Service Locator*, *Abstract Factory*, *Factory Method*, *Prototype*, *Object Pool* oder *Builder*.

Von den analysierten Projekten setzen nur die Xamarin-Plugins solche Erzeugungsmuster ein. Beispielsweise definiert die Klasse `CrossMediaManager` im Medien-Plugin für Xamarin die Fabrikmethode `CreateMediaManager()`. Sie erzeugt Instanzen der jeweils plattform-spezifischen Implementation von `MediaManagerImplementation`.

Im Gegensatz zu den diskutierten Strukturmustern unterscheiden Erzeugungsmuster sich nicht wesentlich im Hinblick auf ihren Einfluss auf die Portierung. Wann welches Erzeugungsmuster einzusetzen ist, hängt vielmehr von den Charakteristika der zu instanziierenden Klasse ab [Gamma u. a. 1995][Freeman u. a. 2004]. Ein Vergleich von Erzeugungsmustern ist daher nicht Gegenstand dieser Arbeit.

6 Trace Recovery für plattform-übergreifende Trace Links

Nach Abschluss des Portierungsprozesses sind nur die vollständig konvertierten Code-Elemente durch Trace Links mit ihren Entsprechungen verknüpft. Diese Links existieren nur, wenn der eingesetzte Konverter sie generiert. Manuell oder teil-automatisch übersetzte Code-Elemente wurden bislang nicht mit ihren Entsprechungen verknüpft.

In Abschnitt 4.1.2 wurde entschieden, dass ein Trace-Recovery-Mechanismus zu entwickeln ist, der diese Lücke schließt. Er soll als Suchmaschine arbeiten, die Trace Links zwischen einem gegebenen Code-Element und dessen Entsprechungen ermittelt und in einer Ergebnisliste präsentiert. Er soll alle tatsächlichen Entsprechungen identifizieren und diese möglichst weit vorne in die Ergebnisliste einsortieren, damit ein Entwickler die gesuchte Entsprechung möglichst schnell identifizieren kann.

Die ermittelten Trace Links werden nicht gespeichert und gewartet, sondern erneut erhoben, wenn ein Entwickler sie benötigt. Der Mechanismus muss somit effizient arbeiten, damit ein Entwickler möglichst ohne spürbare Verzögerung auf die Trace Links zugreifen kann. Um den Entwickler nicht in seinem Gedankenfluss zu unterbrechen, muss der Mechanismus die Ergebnisse in weniger als einer Sekunde erheben [Nielsen 1993, S. 135].

Ansatz
Wie in Abschnitt 3.5 argumentiert, eignet sich für die Zuordnung der ursprünglichen und portierten Code-Elemente ein token-basiertes Verfahren, wie es auch zur Erkennung von Code-Duplikaten innerhalb einer Programmiersprache eingesetzt wird. Dazu werden die Code-Elemente beider Codebasen anhand der enthaltenen Bezeichner miteinander verglichen und Code-Elemente mit ähnlichen Bezeichnern werden einander zugeordnet. Um zusätzlich strukturelle Ähnlichkeiten zu erfassen, sind die erhobenen Bezeichner gemäß ihrer Position in der Struktur des Quelltextes zu gewichten. Beispielsweise wird bei der Repräsentation einer Klasse der Klassenname höher gewichtet als der Name einer lokalen Variable innerhalb der Klasse. So nutzt der Trace-Recovery-Mechanismus die bei der Portierung systematisch herbeigeführten Ähnlichkeiten als Indikatoren für Entsprechungsbeziehungen.

Um der Forderung nach Effizienz gerecht zu werden, wird das rechenintensive Parsen der Quelltexte vorab durchgeführt und nur bei Änderungen einer Datei für eben diese Datei wiederholt. Dabei werden die Bezeichner jedes Code-Elements in einem effizient durchsuchbaren Index abgelegt. Auf Basis dieses Index werden die Trace Links im zweiten Teilprozess ad hoc ermittelt, wenn sie benötigt werden.

6.1 Ablauf des Trace-Recovery-Prozesses

Der Mechanismus vollzieht die Suche nach Trace Links in zwei Teilprozessen. Der erste Teilprozess parst die Code-Elemente der ursprünglichen und der portierten Implementation und legt die extrahierten Bezeichner in einem Index ab. Dieser Teilprozess wird einmal für sämtliche Code-Elemente ausgeführt. Werden einzelne Code-Elemente geändert, so werden die entsprechenden Einträge im Index aktualisiert. Der zweite Teilprozess ermittelt auf Basis dieses Index die Entsprechungen zu einem gegebenen Code-Element.

Teilprozess 1: Parsen und Indexieren des ursprünglichen und portierten Quellcodes Der Ablauf des ersten Teilprozesses ist in Abbildung 6.1 zu sehen.

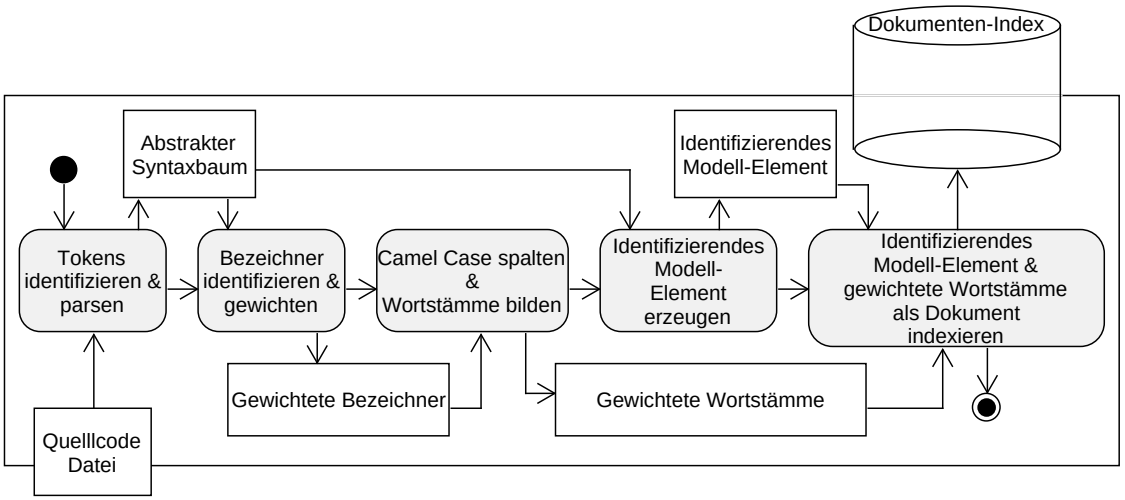

Abbildung 6.1: Teilprozess 1: Parsen und Indexieren der Quellcode-Elemente in der ursprünglichen und portierten Implementation [Stehle u. Riebisch 2018b]

Im ersten Schritt werden die Quellcode-Dateien sowohl der ursprünglichen als auch der portierten Implementation geparst. Zu jeder eingelesenen Quellcode-Datei wird ein abstrakter Syntaxbaum erstellt.

Aus diesem abstrakten Syntaxbaum werden für jedes Code-Element dessen eigener Bezeichner sowie die Bezeichner sämtlicher enthaltenen Code-Elemente erfasst und in einer

Multimenge gespeichert. Beispielsweise werden zu einer Klasse neben dem Klassennamen auch sämtliche Namen von Methoden, Feldern und Variablen erhoben, die innerhalb der Klasse verwendet werden. Die so erzeugte Multimenge repräsentiert das geparste Code-Element und dient später dem Vergleich mit potenziellen Entsprechungen. Sie werden hier im Sinne des Information Retrieval als *Dokumente* bezeichnet, da sie später in einem Suchindex abgelegt und mit Suchanfragen verglichen werden [Manning u. a. 2008, S. 1f.].

Der Inhalt dieser Dokumente soll berücksichtigen, dass Bezeichner auf verschiedenen Ebenen des Syntaxbaumes unterschiedlich stark zur Bedeutung eines Code-Elements beitragen. Beispielsweise ist der Klassenname für die Beschreibung einer Klasse aussagekräftiger als der Name einer lokalen Variable. Um diese Strukturinformation des Quelltextes zu erhalten, werden die Bezeichner entsprechend des programmiersprachlichen Konzepts gewichtet, das sie benennen. Klassennamen erhalten beispielsweise ein höheres Gewicht als lokale Variablen und werden entsprechend häufiger in ein Dokument aufgenommen, das eine Klasse repräsentiert. Wird dem Konstrukt *Klassenname* beispielsweise das Gewicht 3 zugeordnet, so werden Klassennamen beim parsen dreifach in das Dokument eingefügt. Die konkreten Werte der Gewichte werden in Abschnitt 6.2 festgelegt.

Die in das Dokument aufgenommenen gewichteten Bezeichner werden im nächsten Schritt in ihre Bestandteile zerlegt, auf ihren Wortstamm reduziert und Großbuchstaben werden durch Kleinbuchstaben ersetzt, sodass die spätere Suche auch Ähnlichkeiten erkennt, wenn Bezeichner sich nur in Teilen gleichen. Der Bezeichner *DateConverter* würde beispielsweise in *date* und *convert* zerlegt. Um Bezeichner wie *ServiceLocation* und *LocationService* trotz dieser Zerlegung voneinander unterscheiden zu können, wird zusätzlich der ursprüngliche, zusammengesetzte Bezeichner in das Dokument aufgenommen.

Um das analysierte Code-Element später wieder auffinden zu können, wird ein Modell-Element erzeugt, das es identifiziert. Dazu wird das in Abschnitt 4.5.1 eingeführte Metamodell für Traceability-Modelle verwendet. Zu einer Klasse wird beispielsweise ein Modell-Element vom Typ `Typdefinition` erzeugt, wie es in Abschnitt 4.5.1 eingeführt wurde. Es erfasst unter anderem den vollständigen Bezeichner der Klasse und den Pfad der Datei, in der die Klasse definiert wird (vgl. Abbildung 4.7).

Schließlich wird dem identifizierenden Modell-Element in einem Index das erzeugte Dokument mit den gewichteten Bezeichnern zugeordnet. Nach Abschluss des ersten Teilprozesses enthält dieser Dokumenten-Index für jedes analysierte Code-Element ein identifizierendes Modell-Element und ordnet ihm das zugehörige Bezeichner-Dokument zu.

Teilprozess 2: Erzeugung von Trace Links zu einem gegebenen Quellcode-Element auf Basis des erzeugten Index

Im zweiten Teilprozess wird der erstellte Dokumenten-Index anhand eines gegebenen Quellcode-Elements nach dessen Entsprechungen durchsucht. Zu den gefundenen Entsprechungen werden Trace Links erzeugt, die das gegebene Element mit ihnen verbinden. Abbildung 6.2 zeigt den Ablauf, der diese Suche realisiert.

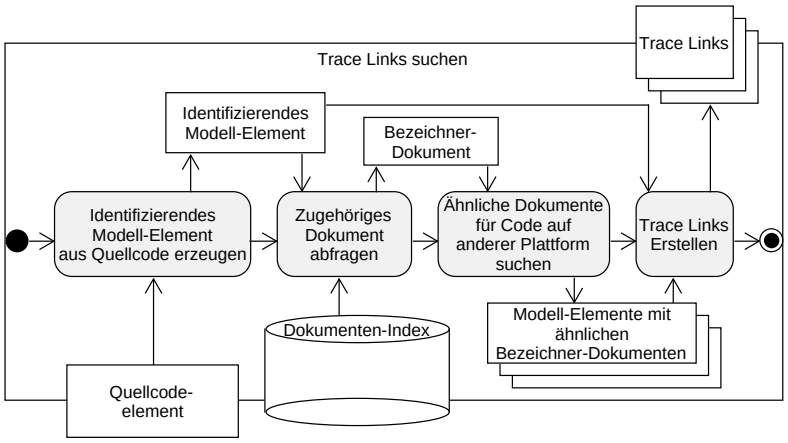

Abbildung 6.2: Erzeugung von Trace Links zu einem gegebenen Quellcode-Element anhand des erzeugten Index [Stehle u. Riebisch 2018b]

Der Teilprozess wird durch eine Anfrage ausgelöst, die zu einem gegebenen Code-Element nach Entsprechungen in der zweiten Implementation fragt. Das Code-Element der Anfrage kann sowohl aus der Ausgangs- als auch aus der Zielimplementation stammen, sodass Entsprechungen in der jeweils anderen Implementation gesucht werden.

Dazu wird im ersten Schritt das identifizierende Modell-Element erzeugt. Anhand dieses Modell-Elements wird im zweiten Schritt das zugehörige Dokument Q mit den gewichteten Bezeichner-Wortstämmen q_i aus dem Dokumenten-Index abgefragt. Um potenzielle Entsprechungen aufzufinden, wird Q im dritten Schritt mit jedem Dokument D verglichen, das aus der zweiten Codebasis stammt. Zum Vergleich wird die Ähnlichkeitsfunktion Okapi BM25 [Robertson u. Walker 1994] eingesetzt, die als eine der erfolgreichsten Text-Retrieval Algorithmen gilt [Robertson u. Zaragoza 2009]. Okapi BM25 berechnet einen numerischen Wert für die Ähnlichkeit zwischen den Dokumenten Q und D. Dazu werden die Dokumente als Tupel $Q = (q_1, q_2, q_3, ...)$ und $D = (d_1, d_2, d_3, ...)$ betrachtet, deren Elemente die gewichteten, auf Wortstämme reduzierten Bezeichner sind. Okapi BM25 bestimmt den Ähnlichkeitswert anhand folgender Formel:

$$BM25(Q,D) = \sum_{i=1}^{|Q|} IDF(q_i) \cdot \frac{f(q_i,D) \cdot 2.2}{f(q_i,D) + 1.2 \cdot (0.25 + 0.75 \cdot \frac{|D|}{avgdl})}$$

Die Funktion $IDF(q_i)$ in der Gleichung ist die umgekehrte Dokumentenhäufigkeit

(**I**nverse **D**ocument **F**requency) des Bezeichners q_i. Sie berechnet einen Wert für den Informationsgehalt des Bezeichners q_i, indem sie misst, wie stark das Enthaltensein von q_i ein Dokument von anderen unterscheidet. Dahinter steckt die Annahme, dass Bezeichner, die im Quellcode eines Projektes selten auftreten, ein Code-Element gut charakterisieren, während häufig auftretende Bezeichner wie `i` oder `result` kaum repräsentativ sind. IDF wird berechnet als:

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

N is dabei die Zahl aller Dokumente im Index und $n(q_i)$ die Anzahl der Dokumente, die q_i enthalten.

Der Bruch auf der rechten Seite der Funktion $BM25$ berechnet einen Wert dafür, wie wichtig ein Bezeichner q_i des Abfragedokuments für das zu vergleichende Dokument D ist. $f(q_i, D)$ misst dazu, wie häufig der Bezeichner q_i im Dokument D auftritt. Durch die Gewichtung der Bezeichner bei der Indexierung wird dieser Wert für wichtige Bezeichner wie Klassennamen erhöht. $avgdl$ ist die durchschnittliche Anzahl der Wortstämme in allen Dokumenten des Index, also die durchschnittliche Dokumentlänge. $|D|$ ist die absolute Dokumentlänge von D. Somit ist $\frac{|D|}{avgdl}$ die relative Dokumentlänge von D. Im Nenner wird sie eingesetzt, um lange Dokumente niedriger zu bewerten als kurze. Dahinter verbirgt sich die Annahme, dass lange Dokumente typischerweise weniger spezifisch sind.

$BM25$ erhebt somit zu jedem Bezeichner q_i des Anfrage-Dokuments Q einen numerischen Wert, der erfasst, wie wichtig er für die Bedeutung des zu vergleichenden Dokuments D ist. Die Summer dieser Werte bildet den zusammenfassenden Ähnlichkeitswert zwischen Q und D.

So wird für jedes indexierte Dokument D der Ähnlichkeitswert $BM25(Q, D)$ berechnet, der angibt wie ähnlich D dem Anfrage-Dokument Q ist. Ist dieser Wert größer als Null, so wird ein Trace Link erzeugt, der die identifizierenden Modell-Elemente für Q und D verbindet. Der Ähnlichkeitswert ist ein Indikator dafür, wie wahrscheinlich es sich bei den verknüpften Elementen tatsächlich um einander entsprechende Elemente handelt. Er wird dem Trace Link daher als Konfidenzwert zugeordnet. Die erzeugten Trace Links werden nach diesem Wert geordnet und als Ergebnisse der Suchanfrage ausgegeben.

6.2 Bestimmung von Bezeichnergewichten

Um den beschriebenen Trace-Recovery-Mechanismus zu konkretisieren, müssen die Bezeichnergewichte festgelegt werden, die bestimmen, wie oft ein Bezeichner im ersten

Teilprozess in ein Dokument aufgenommen wird. Sie haben maßgeblichen Einfluss auf den Vergleich der Dokumente und damit auf die Qualität der Suchergebnisse.

Die Häufigkeit eines Bezeichners im Quelltext hängt unter anderem von der untersuchten Programmiersprache ab sowie vom Stil des Entwicklers und von etwaigen Quelltextkonventionen. Beispielsweise können Variable in der Programmiersprache Swift ohne explizite Nennung des Typbezeichners deklariert werden, wenn sie direkt bei ihrer Deklaration initialisiert werden. Der Typ der Variablen wird dann aus dem Ausdruck abgeleitet, mit dem die Variable initialisiert wird. Dies bezeichnet man als Typinferenz. Codebeispiel 6.1 demonstriert dies anhand der Variablen `fahrzeug`. Sie wird ohne Nennung ihres statischen Typs deklariert. Dieser wird stattdessen durch den Typ des Konstruktoraufrufs `Lastwagen()` bestimmt.

In Java ist Typinferenz erst seit Version 10 möglich. Der größte Anteil des existierenden Java-Quellcodes nutzt dieses Konstrukt daher nicht und deklariert jede Variable mit Typ und Namen wie in Codebeispiel 6.2 zu sehen. Der Typbezeichner wird dadurch in Java-Dokumenten häufiger genannt als in Swift-Dokumenten. Um dies auszugleichen, wäre das Gewicht von Typbezeichnern für Java-Quelltexte niedriger zu wählen als für Swift-Quelltexte, die Gebrauch von Typinferenz machen.

```
var fahrzeug = Lastwagen()
```

Codebeispiel 6.1: Deklaration einer Variablen vom Typ Fahrzeug in Swift mit Type Inference

```
Fahrzeug fahrzeug = new Lastwagen();
```

Codebeispiel 6.2: Deklaration einer Variablen vom Typ Fahrzeug in Java ohne Type Inference

Die Bezeichnergewichte könnten daher für jedes Projekt spezifisch optimiert werden. Um diesen wiederholten Aufwand zu vermeiden, wird in den folgenden Abschnitten ein Satz von Bezeichnergewichten für die Ähnlichkeitsanalyse zwischen Typdefinitionen (Klassen, Interfaces, Enumerations etc.) anhand eines Portierungsprojekts optimiert. Es wird zunächst die Annahme getroffen, dass dieser Satz von Gewichten im allgemeinen, also auch in anderen Portierungsprojekten eine Verbesserung im Vergleich zur Suche ohne Gewichtung erwirkt. In Abschnitt 8.6 wird geprüft, inwiefern dieser Satz von Gewichten tatsächlich auch in Projekten mit anderen Programmiersprachen und Konventionen eine Verbesserung bewirkt.

6.2.1 Relevante Bezeichnergewichte für Typdefinitionen

Tabelle 6.1 erfasst in der ersten Spalte, welche Bezeichner in die Dokumente zur Repräsentation einer Typdefinition, beispielsweise einer Klasse oder eines Interfaces einfließen. In

der zweiten Spalte der Tabelle wird zu jeder Art von Bezeichner das Bezeichnergewicht benannt, das mit einem Wert zu versehen ist.

Kategorie von Bezeichnern	Zu bestimmendes Bezeichnergewicht
Der Bezeichner des Typs selbst, beispielsweise der Klassenname	$g_{typName}$
Bezeichner der vom Typ definierten Attribute	$g_{attributName}$
Bezeichner der Typen der Attribute	$g_{attributTyp}$
Bezeichner der vom Typ definierten Operationen	$g_{operationsName}$
Bezeichner der Rückgabetypen der Operationen	$g_{rückgabeTyp}$
Bezeichner der Parameter der Operationen	$g_{parameterName}$
Bezeichner der Typen der Parameter	$g_{parameterTyp}$
Bezeichner lokaler Variablen bei ihrer Deklaration	$g_{variablenName}$
Bezeichner der Typen lokaler Variablen	$g_{variablenTyp}$
Bezeichner beliebiger Variablen bei ihrer Verwendung	$g_{variablenNutzung}$
Bezeichner aufgerufener Operationen	$g_{operationsAufruf}$

Tabelle 6.1: Überblick über die aus Typdeklarationen erhobenen Bezeichner

Die Arten erhobener Bezeichner sind durch den Funktionsumfang der Analyse-werkzeuge beschränkt, die sie aus den Code-Elementen extrahieren. Theoretisch könnten beispielsweise auch Bezeichner von Parametern in Lambda-Ausdrücken oder auch mathematische oder logische Operatoren erfasst werden. Darauf wurde hier verzichtet, da nicht alle eingesetzten Parser ausreichend feingranular arbeiten, um weitere Bezeichner zu erkennen.

6.2.2 Zielfunktion zur Bewertung eines Satzes von Bezeichnergewichten

Es ist eine Zielfunktion zu definieren, anhand derer ein Satz von Bezeichnergewichten bewertet werden kann. Die Suche nach Trace Links soll möglichst vollständige, gut sortierte Ergebnislisten liefern, sodass Entwickler möglichst wenige der Ergebnisse nacheinander verfolgen müssen, bis sie die für sie relevanten Entsprechungen gefunden haben. Als Kriterium zur Bewertung von Suchergebnis-Listen schlagen Manning u. a. [2008] die

Metrik *Mean Average Precision* (MAP) vor, die in der TREC-Community die meist verwendete Metrik zur Bewertung von Suchverfahren ist [Manning u. a. 2008, S. 159] [Zhou u. a. 2013]. Im Gegensatz zu den in anderen Kontexten üblichen Metriken *Precision* und *Recall* bewertet die Mean Average Precision auch die Reihenfolge der Suchergebnisse, weshalb sie hier als Zielfunktion für die Optimierung des Suchverfahrens genutzt wird. MAP bewertet die durchschnittliche Qualität der Ergebnislisten für eine Menge Q von Suchanfragen q_j anhand folgender Formel:

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

Angenommen, Q sei eine Menge von Anfragen nach Entsprechungen für Java Code-Elemente, dann sucht die Anfrage q_j nach der Menge aller m_j Trace-Links, die ein gegebenes Java-Code-Element mit seinen Swift-Entsprechungen verknüpfen.

Für jede Anfrage q_j wird die Ergebnisliste in m_j Teillisten R_{jk} zerlegt. Die k-te dieser Teillisten enthält dabei alle Ergebnisse bis zum k-ten Treffer. Für diese Teillisten wird im rechten Teil der Formel die *Average Precision* berechnet, also der durchschnittliche Anteil der Treffer in den Teillisten. Abbildung 6.3 veranschaulicht dies anhand einer Ergebnisliste, die zwei von zwei gesuchten Elementen enthält.

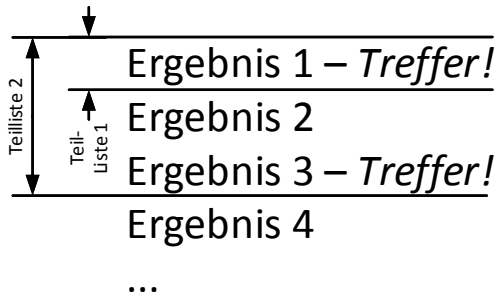

Abbildung 6.3: Exemplarische Ergebnisliste zur Veranschaulichung der Metrik Mean Average Precision

Die Treffer sind an Position eins und drei der Ergebnisliste. Die Average Precision berechnet die Precision für die beiden eingezeichneten Teillisten bis zum ersten bzw. zweiten Treffer und bildet den Durchschnitt:

$$AveragePrecision = \frac{1}{2} \sum_{k=1}^{2} Precision(R_k) = \frac{1}{2} * (1 + 0,75) = 0,875$$

MAP bildet für alle $|Q|$ Suchanfragen den Durchschnitt ihrer Average Precisions.

6.2.3 Datensatz korrekter Trace Links zum Abgleich mit den Ergebnissen des Trace-Recovery-Mechanismus

Wie zuvor entschieden, wird die Optimierung des Trace-Recovery-Mechanismus zunächst anhand eines konkreten Projekts durchgeführt. Um den Mechanismus in Bezug auf dieses Projekt zu bewerten, muss ein Satz korrekter Trace Links gefunden werden. Auf Basis dieser korrekten Trace links können die vom Mechanismus vorgeschlagenen Ergebnislisten mit der Zielfunktion MAP bewertet werden. Dazu ist zunächst ein passendes Portierungsprojekt zu wählen, anhand dessen der Mechanismus optimiert werden kann.

Der Trace-Recovery-Mechanismus soll insbesondere für händisch portierten Code optimiert werden. Trace Links zwischen ursprünglichen Code-Elementen und ihren händisch portierten Entsprechungen sind schwerer aufzufinden, da der portierende Entwickler unterschiedliche Bezeichner und Strukturen nutzen kann. Im Gegensatz zu automatisch übersetztem Code gibt es bei händischer Übersetzung auch keine formalen Abbildungen zwischen den ursprünglich genutzten externen APIs und den APIs der Zielplattform, die für die Gewinnung der Trace Links genutzt werden könnten. Der Mechanismus soll daher anhand eines Projekts optimiert werden, das für händisch übersetzten Code typisch ist. Die ursprüngliche und die portierte Implementation sollten möglichst realistische Unterschiede in Bezug auf die verwendeten Strukturen und gewählten Bezeichner aufweisen.

Ein solches Projekt ist *Twidere*[1]. Twidere ist ein alternativer mobiler Client für Twitter, der zunächst für das Betriebssystem Android entwickelt und später nach iOS portiert wurde. Die Android-Implementation besteht aus 1128 Typdefinitionen in Java und 606 Typdefinitionen in der Programmiersprache Kotlin. Die entsprechende iOS-Implementation definiert 132 Typen in der Sprache Swift. Die deutlich kleinere Anzahl von Typen in der iOS-Implementation ist darauf zurückzuführen, dass nicht alle Funktionen der ursprünglichen Version portiert wurden. Die Portierung ist gänzlich ohne automatische Code-Konversion vollzogen worden. Die Verwendung gleicher Bezeichner und ähnlicher Konzepte ist ausschließlich darauf zurückzuführen, dass der portierende Entwickler auch die ursprüngliche Implementation entwickelt hat. Es ist nicht zu erkennen, dass systematisch Ähnlichkeiten zwischen der ursprünglichen und der portierten Implementation herbeigeführt wurden. Damit ist Twidere ein geeignetes Projekt, um den Trace-Recovery-Mechanismus für händisch portierten Code zu optimieren.

Manning u. a. [2008, S. 152] geben als Faustregel an, dass die Bewertung von Ergebnislisten für 50 Anfragen ausreicht, um ein Suchverfahren aussagekräftig zu bewerten. Dementsprechend wurden zufällig 50 Typdefinitionen der Android-Implementation ausgewählt, die aus unterschiedlichen Bereichen der Anwendung stammen. Darunter

[1] `https://github.com/mariotaku/twidere`

sind sowohl Klassen des Domänenmodells als auch fachliche Service-Klassen, technische Utility-Klassen und Controller für die Benutzerschnittstelle. Zu ihnen wurden händisch die Entsprechungen in der iOS-Implementation ermittelt. Einige der 50 Typdefinitionen werden in der iOS-Implementation durch mehr als einen Typ re-implementiert, sodass sich für die 50 Typdefinitionen 62 Trace Links zu ihren Entsprechungen ergeben. Tabelle A.3 im Anhang erfasst diese Links. Um die Korrektheit dieser manuell ermittelten Liste von Trace Links abzusichern, wurden sie von einer am Projekt nicht beteiligten Wissenschaftlerin geprüft, wobei sich lediglich eine Ergänzung ergab.

6.2.4 Wahl des Optimierungsverfahrens

Jeder der oben genannten Arten von Bezeichnern ist bei der Optimierung mit einem ganzzahligen Bezeichnergewicht zu versehen. Der Lösungsraum L dieser Optimierung besteht damit aus Tupeln, die zu jeder Kategorie von Bezeichnern ein anzuwendendes Bezeichnergewicht als natürliche Zahl enthalten.

Aufgrund der hohen Anzahl zu bestimmender Bezeichnergewichte und der jeweils hohen Anzahl möglicher Belegungen kann eine optimale Lösung nicht durch vollständiges Ausprobieren ermittelt werden. Stattdessen soll hier systematisch eine Näherung für das optimale Bezeichnertupel bestimmt werden. Algorithmen, die diesen Zweck erfüllen, bezeichnet man als Metaheuristiken. Zu ihrer Anwendung wird eine Nachbarschaftsrelation definiert, die angibt, welche Lösungen als direkte Nachbarn einer gegebenen Lösung gelten. Eine Lösung $l = (g_{typName}, g_{attributName}, g_{attributTyp}, \dots, g_{operationsAufruf})$ besteht hier aus einem Tupel, das zu jedem Bezeichnergewicht dessen Wert festlegt. Als Nachbarn eines solchen Tupels werden hier diejenigen Lösungen $l_n \in L$ definiert, die durch Inkrementieren oder Dekrementieren eines der Gewichte in l erzeugt werden können. Das Inkrementieren oder Dekrementieren eines Gewichts wird als Zug bezeichnet.

Basierend auf dieser Nachbarschaftsrelation und einer Ausgangslösung können verschiedene Metaheuristiken zur Anwendung kommen. Ein Beispiel ist das *Stochastic Hill Climbing*, bei dem iterativ ein zufällig gewählter Nachbar der Ausgangslösung bewertet wird und als neue Ausgangslösung akzeptiert wird, falls er ein besseres Ergebnis erzielt als die aktuelle Ausgangslösung. Ähnlich agiert das Verfahren *Steepest Ascent Hill Climbing*, das in jedem Schritt die gesamte Nachbarschaft der aktuellen Lösung bewertet und diejenige Nachbarlösung als neue Ausgangslösung akzeptiert, die die größte Verbesserung im Vergleich zur aktuellen Lösung bietet. Der Nachteil dieser beiden Verfahren ist, dass sie die Suche beenden, wenn keine unmittelbar benachbarte Lösung ein besseres Ergebnis erzielt als die aktuelle. Dadurch endet das Verfahren gegebenenfalls in einem lokalen Optimum, das kein globales Optimum ist.

Ein Verfahren, das dieses Problem überwindet, ist die *Tabu-Suche*. Dabei handelt es sich um ein iteratives Verfahren, das insbesondere bei diskreten Optimierungsproblemen erfolgreich angewendet wird, zu denen auch das vorliegende gehört [Laguna 2018, S. 741]. Die Tabu-Suche bewertet zunächst die Ausgangslösung anhand der Zielfunktion. Zusätzlich definiert sie eine zunächst leere Tabu-Liste, in der verbotene Lösungen gespeichert werden. Anschließend geht sie iterativ wie folgt vor: Sie bewertet alle benachbarten Lösungen, die nicht in der Tabu-Liste enthalten sind. Diejenige Nachbarlösungen, die die beste Bewertung erhält, wird in der nächsten Iteration als Ausgangslösung verwendet. Dies geschieht unabhängig davon, ob sie besser ist als die Ausgangslösung. Dadurch können lokale Optima bei der Tabu-Suche überwunden werden, um über den Umweg einer Verschlechterung eine neue beste Lösung zu finden. Eine Lösung, die einmal als Ausgangslösung gedient hat, wird in die Tabu-Liste eingefügt, sodass sie nicht erneut geprüft wird. Die Suche endet, wenn alle Nachbarn der aktuellen Ausgangslösung in der Tabu-Liste enthalten sind. Da dies ggf. nicht in angemessener Zeit eintritt, wird als alternatives Abbruchkriterium der Ablauf von acht Stunden festgelegt.

Erweiterte Varianten der Tabu-Suche nutzen keine Tabu-Liste mit konkreten Lösungen. Stattdessen wird ein Ringspeicher angelegt, in dem *Züge* für eine begrenzte Anzahl von Iterationen verboten werden [Gendreau u. Potvin 2005]. Es werden dabei solche Züge verboten, die eine kürzlich erreichte Verbesserung rückgängig machen würden. Dadurch werden mehr Lösungen von der Prüfung ausgeschlossen. Das erhöht die Effizienz des Algorithmus. Diese Weiterentwicklung wird auch hier angewendet. Wird die Trace Link-Suche beispielsweise durch das Erhöhen des Gewichts $g_{typName}$ erzielt, so wird die Verringerung dieses Gewichts für die folgenden Iterationen verboten. Die Größe des Ringspeichers legt dabei fest, wie lange die entsprechenden Züge verboten werden. Es ist ein passender Wert für die Größe des Ringspeichers zu ermitteln. Da es elf Bezeichnergewichte gibt, die jeweils inkrementiert oder dekrementiert werden können, gibt es maximal 22 mögliche Züge zu gültigen Nachbarn, die die Grenzen der Wertebereiche einhalten. Die Tabu-Suche wird dementsprechend 22 mal mit Speichergrößen zwischen eins und 22 ausgeführt. Die beste dabei berechnete Lösung wird hier als Ergebnis der näherungsweisen Optimierung akzeptiert.

6.2.5 Durchführung und Ergebnis der Optimierung

Die Optimierung wurde unter Einsatz des Frameworks *James* durchgeführt, das verschiedene metaheuristische Verfahren in Java implementiert [De Beukelaer u. a. 2015]. James bietet auch eine Implementation der Tabu-Suche, die um folgende konkrete Klassen ergänzt wurde, um sie für diese Optimierung nutzbar zu machen:

- eine Klasse zur Repräsentation einer Lösung, in diesem Fall ein Tupel von Bezeichnergewichten,

- eine Klasse, die die Nachbarschaftsrelation beschreibt und zu einer gegebenen Lösung die möglichen Züge zu benachbarten Lösungen berechnet,
- eine Klasse zur Definition des Tabu-Speichers, der prüft, ob ein gegebener Zug aktuell erlaubt ist
- sowie eine Klasse, die eine gegebene Lösung bewertet.

Diese Implementation der Optimierung ist im Repository mit dem Code des Retrieval-Frameworks im gesonderten Branch *factor-optimization* zu finden[2].

Als initiale Ausgangslösung wurden alle Gewichte mit dem Wert eins belegt, sodass die Suche sich genau verhält wie ohne Gewichtung. Nach Ablauf der festgelegten acht Stunden wurde die Optimierung mit der jeweiligen Speichergröße beendet. Der Tabu-Speicher der Größe fünf erzielte dabei das beste Optimierungsergebnis. Tabelle 6.2 hält die entsprechende Kombination von Bezeichnergewichten fest. Diese Kombination erzielte eine Mean Average Precision von 0.75 für die in Twidere identifizierten Entsprechungen.

Name des Gewichts	Wert
$g_{typName}$	5
$g_{attributName}$	3
$g_{attributTyp}$	2
$g_{operationsName}$	3
$g_{rückgabeTyp}$	1
$g_{parameterName}$	2
$g_{parameterTyp}$	0
$g_{variablenName}$	1
$g_{variablenTyp}$	0
$g_{variablenNutzung}$	1
$g_{operationsAufruf}$	1

Tabelle 6.2: Näherungsweise optimierte Bezeichnergewichte für das Portierungsprojekt *Twidere*

6.3 Ein erweiterbares Trace-Recovery-Framework

Der hier entwickelte Trace-Recovery-Mechanismus wurde auf Basis des Such-Rahmenwerkes Lucene[3] realisiert, welches die Dokumente effizient speichert und zu deren Vergleich die Ähnlichkeitsfunktion Okapi BM25 wie oben beschrieben einsetzt. Aus dieser

[2] https://github.com/TilStehle/Java-Kotlin-Swift-Trace-Link-Recovery
[3] http://lucene.apache.org/

Entwicklung ist ein erweiterbares Trace-Recovery-Framework entstanden, das den Trace-Recovery-Mechanismus für die Sprachen Swift, C#, Java, JavaScript und Kotlin umsetzt. Der zugehörige Quellcode ist gemeinsam mit einem Plugin für die Entwicklungsumgebung IntelliJ IDEA veröffentlicht[4]. Dieses Plugin nutzt die ermittelten Trace Links. Abschnitt 8.7 demonstriert seinen Einsatz während der Co-Evolution.

Um das Framework für zusätzliche Sprachen zu erweitern, müssen entsprechende Parser für diese Sprachen integriert werden. Um den Aufwand für solche Erweiterungen möglichst gering zu halten, wurde das Parsen der Code-Elemente von der Erstellung und Indexierung der Dokumente entkoppelt. Besonderes Augenmerk lag darauf, dass erweiternde Entwickler sich nicht in die Nutzung von Lucene einarbeiten müssen, um Code-Elemente einer weiteren Programmiersprache zu parsen und zu indexieren. Abbildung 6.4 zeigt die Exemplarische Erweiterung des Frameworks um die Sprache TypeScript.

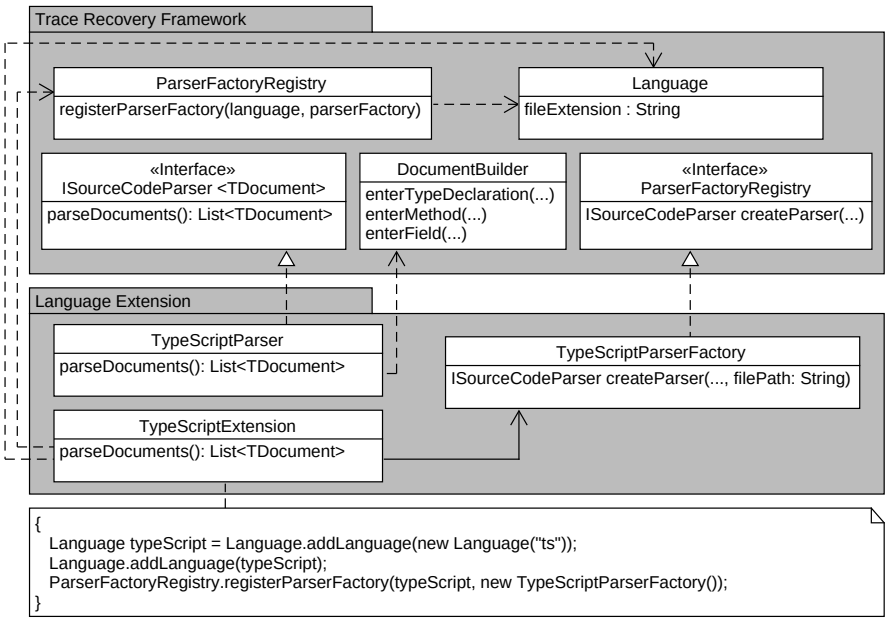

Abbildung 6.4: Klassendiagramm einer exemplarischen Erweiterung des Trace-Recovery-Frameworks um eine zusätzliche Sprache

Die dafür relevanten Klassen des Frameworks sind im oberen Teil der Abbildung zu sehen. Im unteren Bereich sind die Klassen zu sehen, die die erweiternden Entwickler selbst implementieren. Dazu zählen

- der Parser `TypeScriptParser`, der das Interface `ISourceCodeParser` implementiert und Code-Elemente der zusätzlichen Programmiersprache in abstrakte Syntaxbäume überführt
- die Klasse `TypeScriptParserFactory`, die Instanzen des Parsers erzeugt

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

- die Klasse `TypeScriptExtension`, die die ParserFactory bei der `ParserFactory-Registry` des Frameworks registriert und der Sprache `TypeScript` als Exemplar der Klasse `Language` zuordnet.

Um den Parser zu entwickeln, kann beispielsweise ein Parser-Generator wie ANTLR (ANother Tool for Language Recognition)[5] eingesetzt werden, der auf Basis der formalen Grammatik einer Programmiersprache einen Parser generiert. Ein solcher Parser liest Code-Elemente der Sprache ein und erzeugt einen entsprechenden abstrakten Syntaxbaum. Zur Erstellung der im Index abzulegenden Dokumente traversiert die Erweiterung den erzeugten abstrakten Syntaxbaum und übermittelt die identifizierten Bezeichner an ein Exemplar der Klasse `DocumentBuilder`. `DocumentBuilder` definiert dazu Operationen wie `enterTypeDeclaration(...)`, die beispielsweise aufzurufen ist, wenn eine Klassendeklaration im Syntaxbaum erkannt wird. Sie nimmt den Klassennamen und identifizierende Informationen zur Klasse entgegen, delegiert die Verarbeitung der Bezeichner zu gewichteten Wortstämmen, legt sie in einem Dokument ab und ordnet diesem ein identifizierendes Modell-Element zu. Werden dem `DocumentBuilder` Bezeichner zu untergeordneten Elementen wie Methoden im Syntaxbaum übermittelt, so ergänzt er das Dokument zur entsprechenden Klassendeklaration um die untergeordneten Bezeichner. So erzeugt das Framework Dokumente für Typdefinitionen, Methoden und Attribute.

Das Framework wurde wie oben beschrieben um die Programmiersprachen C#, JavaScript und Kotlin erweitert, nachdem es zunächst nur Java und Swift verarbeiten konnte. Je nachdem, ob die Grammatik der zusätzlichen Sprache für einen Parser-Generator verfügbar war, hat die Erweiterung des Tools ein bis drei Tage in Anspruch genommen. Kenntnisse des Suchprozesses waren dazu nicht notwendig, sodass erweiternde Entwickler sich lediglich mit den oben beschriebenen Schnittstellen und der Erstellung eines Parsers befassen müssen, um das Trace-Recovery-Framework selbstständig zu erweitern.

[5] `https://www.antlr.org/`

7 Nutzung von Trace Links in der plattform-übergreifenden Co-Evolution

Mit dem Abschluss der initialen Portierung ist der Lebenszyklus der portierten Software nicht beendet. Das viel zitierte und inzwischen gut mit Daten untermauerte erste Gesetz der Software-Evolution besagt, dass ein Programm, welches einen realen Zweck erfüllt, kontinuierlichen Änderungen unterzogen werden muss, oder seine Nützlichkeit einbüßt [Lehman 1980; Herraiz u. a. 2013]. Das systematische Herbeiführen konzeptueller Ähnlichkeiten und Trace Links in der vorgestellten Portierungsmethode zielt dementsprechend darauf ab, die Co-Evolution der ursprünglichen und portierten Implementation durch konzeptuelle Angleichung sowie durch Trace Links zu vereinfachen. Abbildung 7.1 zeigt, in welchen Phasen des Änderungsprozesses nach Rajlich [2012, S. 74] die systematisch erzeugten Ähnlichkeiten und Trace Links ausgenutzt werden können.

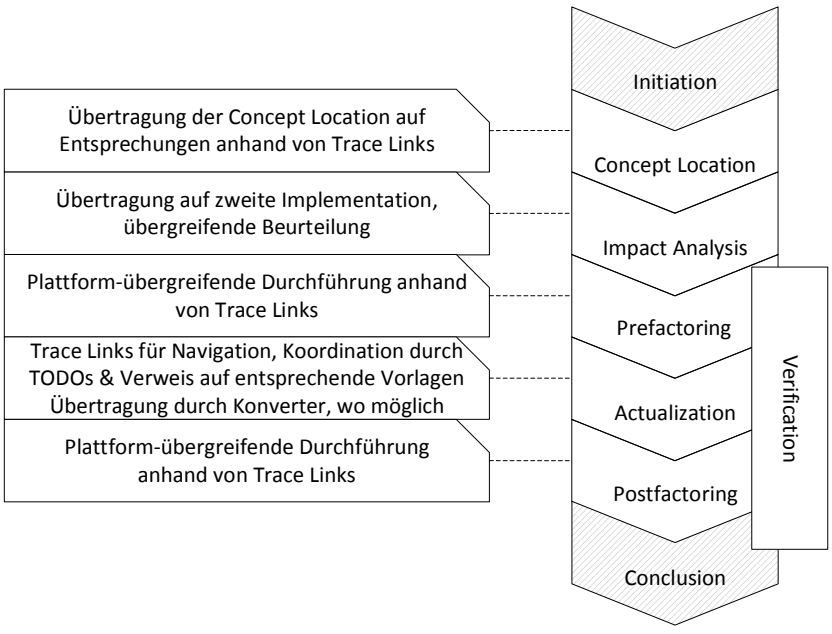

Abbildung 7.1: Nutzung der erzeugten Trace Links im Änderungsprozess nach Rajlich [2012, S. 74]

In den folgenden Abschnitten werden die Konzepte für die vereinheitlichte und koordinierte Durchführung dieser Phasen unter Nutzung der plattform-übergreifenden Trace Links beschrieben.

7.1 Plattform-übergreifende Concept Location

Die Concept Location, also das Auffinden einer zu ändernden Funktionalität im Quellcode, ist eine der Aufgaben, die bei portierter Software für beide Implementationen anfällt. Dazu müssen Entwickler den Entwurf jeder Implementation zunächst verstehen. Singh u. a. [2016] sowie Fritz u. a. [2014] stellen fest, dass Entwickler für das Verstehen des Codes und das Identifizieren der zu ändernden Code-Teile erheblichen Aufwand in das Navigieren entlang von Quellcode-Abhängigkeiten investieren. Entwickler verbringen bei der Wartung von fremdem Code etwa 35% ihrer Zeit mit dieser Tätigkeit [Ko u. a. 2006; Piorkowski u. a. 2013]. Sie stellen gedankliche Bezüge zwischen Code-Elementen her, indem sie entlang deren Abhängigkeiten navigieren und dadurch verstehen, welcher Teil des Quellcodes welche Funktionalität implementiert. Die Angleichung der Strukturen beider Implementationen trägt zu einer Vereinfachung bei, da ein Entwickler sein Verständnis der ursprünglichen Strukturen auf die Zielimplementation übertragen kann.

Darüber hinaus kann ein Entwickler die ermittelten Trace Links wie folgt nutzen, um die Concept Location plattform-übergreifend durchzuführen: Zunächst ermittelt er das zu ändernde Code-Element in einer der beiden Implementationen. Ist das richtige Code-Element in der ersten Implementation gefunden, so lässt er sich die plattform-übergreifenden Trace Links zwischen dem ermittelten Element und seinen Entsprechungen anzeigen. Auf Basis seiner Kenntnis des gemeinsamen Entwurfs kann er das korrespondierende Element in der zweiten Implementation ohne doppelten Aufwand für Concept Location bestimmen. Ist er sich nicht sicher, so kann er die Trace Links nutzen, um zu den verknüpften Elementen zu navigieren und zu prüfen, welches von ihnen die zu ändernde Funktionalität umsetzt.

Um dieses Vorgehen zu ermöglichen, sollten die eingesetzten Entwicklungsumgebungen so erweitert werden, dass Entwickler zwischen den Entsprechungen in beiden Implementationen navigieren können. Wird je Plattform eine unterschiedliche Entwicklungsumgebung eingesetzt, so ist die anzuzeigende Entsprechung in der passenden Entwicklungsumgebung zu präsentieren. So ist eine angemessene Darstellung des Codes gewährleistet und der Entwickler kann seine Suche ggf. mit den Mitteln der passenden Entwicklungsumgebung fortsetzen.

Um die Realisierbarkeit dieses Vorgehens zu demonstrieren, wurde ein Plugin für die Entwicklungsumgebung IntelliJ IDEA entwickelt, dessen Einsatz in Abschnitt 8.7.1 demonstriert wird. Es nutzt das zuvor beschriebene Trace-Recovery-Framework zur

Gewinnung plattformübergreifender Trace Links und erlaubt Entwicklern die Navigation anhand dieser Trace Links über die Grenzen einer Entwicklungsumgebung hinaus.

7.2 Plattform-übergreifende Durchführung von Impact-Analyse

Bei der Impact-Analyse bestimmt der Entwickler alle Code-Elemente, auf die sich eine Änderung auswirkt. Soll eine Änderung plattform-übergreifend konsistent durchgeführt werden, so muss die Impact-Analyse für die Ausgangs- und Zielimplementation durchgeführt werden, um das vollständige Ausmaß der Änderung umfassend bewerten zu können.

Diverse Ansätze für die Impact-Analyse nutzen Abhängigkeitsgraphen, die Code-Elemente als Knoten und Abhängigkeiten zwischen ihnen als Kanten darstellen [Lehnert 2011]. Beispielsweise kann es sich bei den Knoten um Pakete, Klassen oder Methoden handeln, je nachdem, in welcher Granularität die Fortpflanzung der Änderung untersucht werden soll. Diese Graphen werden dazu genutzt, ausgehend von einem zu ändernden Element schrittweise die Fortpflanzung der Änderung anhand der erfassten Abhängigkeiten zu ermitteln. Der entsprechende Prozess nach Rajlich [2012, S. 113f] ist in Abbildung 7.2 zu finden.

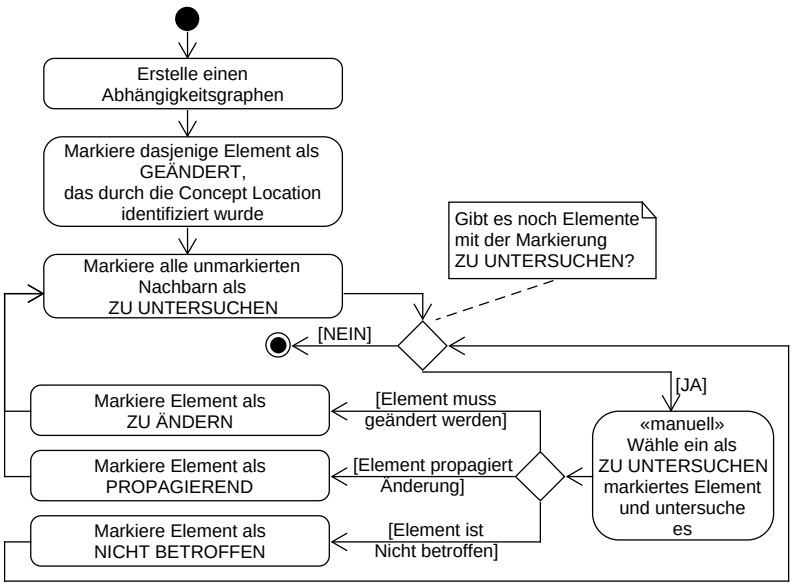

Abbildung 7.2: Ablauf der Impact-Analyse nach Rajlich [2012, S. 113]

Zunächst wird der Abhängigkeitsgraph erstellt und ein zu änderndes Element wird im Rahmen der Concept Location vor der eigentlichen Impact-Analyse bestimmt. Ausgehend von diesem Element werden alle im Graphen benachbarten Elemente als Kandidaten für eine Änderung mit **ZU UNTERSUCHEN** markiert. Im nächsten Schritt prüft ein Entwickler einen der Kandidaten dahingehend, ob er selbst geändert werden muss

(Markierung **ZU ÄNDERN**), die Änderung an seine Nachbarn propagiert, ohne selbst betroffen zu sein (Markierung **PROPAGIEREND**) oder überhaupt nicht von der Änderung betroffen ist (Markierung: **NICHT BETROFFEN**). Elemente, die als **ZU ÄNDERN** oder **PROPAGIEREND** markiert wurden, geben die Änderung potenziell an ihre Nachbarn weiter, sodass deren Nachbarn als **ZU UNTERSUCHEN** markiert werden. Dieser Prozess wird wiederholt, bis keine Elemente mehr als **ZU UNTERSUCHEN** markiert sind.

Die bei der Portierung erzielte konzeptuelle Ähnlichkeit und die ermittelten Trace Links können dabei helfen, das vollständige Ausmaß der Änderung plattform-übergreifend zu bewerten. Dazu wird für jedes zu ändernde Element geprüft, ob es eine Entsprechung in der zweiten Implementation hat. Dies kann anhand der ermittelten Trace Links erfolgen. Ist der zu ändernde Aspekt des betroffenen Code-Elements in die Zielimplementation überführt worden, so muss auch das entsprechende Element in der Zielimplementation geändert werden. Angenommen eine Klasse ist in Bezug auf ihre Schnittstelle exakt in die Zielimplementation überführt worden und eine ihrer Operationen muss für die Änderung angepasst werden. Dann muss diese Anpassung auch in der entsprechenden portierten Operation durchgeführt werden. Trace Links sollten daher im Sinne des oben beschriebenen Ablaufs als Kante den Abhängigkeitsgraphen ergänzen. So können die Elemente der ursprünglichen und portierten Implementation anhand eines zusammengeführten Graphen gemeinsam untersucht werden.

Dieser Ansatz wurde in einer Masterarbeit umgesetzt, die im Rahmen dieser Arbeit betreut wurde [Berger 2019]. Sie nutzt das in Abschnitt 6.3 beschriebene Trace-Recovery-Framework, um Trace Links zu ermitteln und erweitert die Abhängigkeitsgraphen um Trace Links. Sie ermöglicht die plattform-übergreifende Impact-Analyse auf der Ebene von Typdefinitionen. Das Konzept wurde im Zuge der Masterarbeit in einem Plugin für die Entwicklungsumgebung IntelliJ IDEA realisiert, das in Abschnitt 8.7.2 demonstriert wird. Zwar arbeitet das Plugin ausschließlich auf der Ebene von Typdefinitionen, das Konzept plattform-übergreifender Impact-Analyse ist jedoch nicht auf diese relativ grobe Granularität beschränkt. Genauso können Abhängigkeitsgraphen analysiert werden, die Operationen oder Codezeilen mit ihren Abhängigkeiten erfassen. Voraussetzung dafür ist lediglich, dass die Abhängigkeitsgraphen für Ausgangs- und Zielimplementation auf dieser Ebene erstellt werden können und Trace Links zwischen Code-Elementen dieser Ebene ermittelt werden. Das in Abschnitt 6.3 beschriebene Trace-Recovery-Framework kann in diesem Kontext eingesetzt werden, um Trace Links zwischen Typdefinitionen, Methoden und Attributen zu gewinnen.

Verschiedene Ansätze haben das grundlegende Vorgehen der Impact-Analyse erweitert, beispielsweise um eine automatische Sortierung der zu prüfenden Elemente [Briand u. a. 1999] oder um den dynamischen Wechsel der betrachteten Art von Code-Elementen [Petrenko u. Rajlich 2009]. Diese Weiterentwicklungen wirken sich nicht darauf aus, dass die

zugrundeliegenden Graphen um Trace Links als Abhängigkeiten ergänzt werden können, um eine plattformübergreifende Betrachtung zu ermöglichen.

7.3 Plattform-übergreifende Durchführung von Refactorings

Die hier entwickelte Portierungsmethode führt systematisch dazu, dass sich die Implementationen für die ursprüngliche und die zusätzliche Plattform in Bezug auf eingesetzte Strukturen und Konzepte entsprechen. Um diese Entsprechungen aufrecht zu erhalten und die gemeinsame Wartbarkeit beider Implementationen zu verbessern, sollten notwendige Refactorings in den Phasen Pre- und Postfactoring plattform-übergreifend synchronisiert werden.

Ein Beitrag dazu liegt in der sprachübergreifenden Implementation von Refactoring-Werkzeugen. Werkzeugentwickler beschreiben dazu den Ablauf eines Refactorings zunächst abstrakt und konkretisieren ihn dann in sprachspezifischen Erweiterungen [Jemerov 2008]. Zusätzlich können Refactorings wie folgt plattform-übergreifend durchgeführt werden: Der Entwickler wählt zunächst wie gewohnt die zu restrukturierenden Code-Elemente einer Implementation und das anzuwendende Refactoring. Im Falle des Refactorings *Extract Interface* muss er beispielsweise die Operationen eines Typs wählen, die in ein neues Interface extrahiert werden sollen. Anschließend legt er die entsprechend zu restrukturierenden Code-Elemente der zweiten Implementation fest. Dazu nutzt er den Mechanismus zum Auffinden plattform-übergreifender Trace Links, der in Kapitel 6 entwickelt wurde. Zu jedem zu restrukturierenden Code-Elemente wählt er die zugehörige Entsprechung in der zweiten Implementation. Daraufhin wird das Refactoring in beiden Implementationen angewendet. Schlägt es in einer Codebasis fehl, so wird es in beiden Codebasen rückgängig gemacht, sodass kein inkonsistenter Zustand entsteht. Um diesen Ablauf zu vereinfachen, können Werkzeuge entwickelt werden, die den restrukturierenden Entwickler durch die Auswahl der betroffenen Elemente führen, das Refactoring automatisiert in beiden Implementationen anwenden und fehlgeschlagene Restrukturierungen gegebenenfalls rückgängig machen.

Dies wurde im Rahmen der vorliegenden Arbeit exemplarisch in Form eines Plugins für die Entwicklungsumgebung IntelliJ IDEA umgesetzt. Es erlaubt die plattformübergreifende Durchführung von Rename-Refactorings. Seine Anwendung wird in Abschnitt 8.7.4 am Beispiel des Codes einer portierten mobilen App demonstriert.

7.4 Plattform-übergreifende Durchführung von Änderungen

Auch bei der Durchführung der Änderungen können die ermittelten Trace Links und die erzielten Ähnlichkeiten genutzt werden. Vollständig konvertierbarer Quellcode muss dazu lediglich in der ursprünglichen Implementation angepasst werden. Die durchgeführte Änderung kann dann auf die Zielplattform übertragen werden, indem der zur Portierung eingesetzte Quellcode-Konverter erneut angewendet wird.

Darüber hinaus können händisch zu übertragende Änderungen anhand der Trace Links koordiniert werden. Dazu setzt der Entwickler die Änderung zunächst in einer der Implementationen um. Er kann nun die Trace Links nutzen, um die entsprechend zu ändernden Quellcode-Elemente in der zweiten Implementation zu identifizieren. Diese versieht er mit einem TODO-Kommentar, der die Änderung beschreibt. Er kann darüber hinaus den geänderten Quellcode der ersten Implementation als Vorlage für die entsprechenden Änderungen in der zweiten Implementation mit in den Kommentar aufnehmen. Der Entwickler, der die entsprechende Änderung in der zweiten Implementation vornimmt, kann diese Vorlage nutzen, um die Änderung äquivalent umzusetzen, anstatt ein eigenes, potenziell anderes Konzept für die Änderung zu entwickeln. Er muss das zu ändernde Element nicht selbstständig identifizieren und die notwendige Änderung nicht erneut konzipieren.

Durch die Wiederverwendung des Änderungskonzepts wird erstens verhindert, dass Inkonsistenzen bei Änderungen entstehen. Zweitens wird der Entwickler der zweiten Implementation das Konzept und gegebenenfalls die Code-Vorlage der Änderung verstehen, prüfen und eventuell Fehler finden. Er fungiert somit gleichzeitig als Reviewer der ursprünglichen Änderung.

Um diesen Einsatz von Trace Links zu vereinfachen, können Werkzeuge entwickelt werden, die das Erstellen von Kommentaren an verknüpften Code-Elementen ohne Wechsel der Entwicklungsumgebung ermöglichen. In Abschnitt 8.7.3 wird die Anwendung eines solchen Werkzeugs demonstriert, das als Plugin für die Entwicklungsumgebung IntelliJ IDEA umgesetzt wurde.

8 Evaluation und Demonstration der Ergebnisse

8.1 Kriterien und Metriken für die Evaluation

Um die Erfüllung des in Abschnitt 1.4 definierten Ziels objektiv zu bewerten, werden in diesem Abschnitt Kriterien aufgestellt, anhand derer die Ergebnisse dieser Arbeit evaluiert werden.

Der Hauptbeitrag dieser Arbeit besteht in der entwickelten Portierungsmethode inklusive der Richtlinien zur Anwendung von Entwurfsmustern und der Mechanismen zur Erhebung plattform-übergreifender Trace Links. Ein grundlegendes Kriterium für die Evaluation der Methode ist ihre **Anwendbarkeit** für die Portierung objektorientierter Software.

Die Methode verfolgt drei maßgebliche Ziele:

1. Portierbare Quellcode-Elemente, insbesondere Typdefinitionen sollen systematisch in konzeptuelle Entsprechungen in der Zielimplementation übertragen werden.
2. Es sollen Trace Links zwischen einander entsprechenden Code-Elementen eingeführt werden.
3. Evolutionäre Aufgaben sollen plattform-übergreifend zusammengeführt und koordiniert werden.

Aus diesen Zielen leiten sich weitere Kriterien ab.

Zur Bewertung des ersten Ziels ist der Umfang des Codes zu erheben, den die portierten **Typdefinitionen** ausmachen, **die vollständig ihren Vorbildern entsprechen**. Entsprechungen werden dabei in drei Abstraktionsebenen bewertet.

- **Typ-Ebene:** Eine Typdefinition entspricht ihrem Vorbild in der ursprünglichen Implementation in Bezug auf ihre Zuständigkeiten.
- **Schnittstellen-Ebene:** Die Schnittstelle dieser Typdefinition entspricht darüber hinaus der Schnittstelle des Vorbilds. Sie definiert also Operationen mit gleichen Namen, entsprechenden Rückgabetypen und Parametern.

- **Anweisungs-Ebene:** Die in der Typdefinition enthaltenen Anweisungen und Kontrollstrukturen können jeweils eindeutig einer zweckgleichen Anweisung oder Kontrollstruktur in der Entsprechung zugeordnet werden.

Das übergeordnete Ziel ist, dass ein möglichst großer Anteil des Codes plattformübergreifend co-evolviert werden kann. Zu messen ist daher der **Anteil der Typ-Definitionen**, die auf der jeweiligen Abstraktionsebene vollständig ihren Vorbildern entsprechen. Da der Aufwand der Evolution durch den Umfang der Typdefinitionen beeinflusst wird, ist der Anteil dieser Typdefinitionen anhand ihrer **Codezeilen** zu messen.

Das zweite Ziel der Methode besteht darin, plattform-übergreifende Trace Links zwischen einander entsprechenden Code-Elementen einzuführen. Dazu wurden zwei Mechanismen entwickelt. Der erste Mechanismus sieht eine Erweiterung von Quellcode-Konvertoren vor, die die Eingabe-Elemente eines Konverters mit den automatisch erzeugten Übersetzungen durch Trace Links verknüpft. Um zu zeigen, dass dieser Mechanismus **realisierbar** ist, wurde er prototypisch in einem Quellcode-Konverter implementiert. Als Kriterium für die Qualität des Mechanismus ist zu erheben, inwiefern die erzeugten Traceability-Modelle korrekt und vollständig sind. Dazu eignen sich die Metriken **Precision** und **Recall**.

Der zweite Mechanismus verfolgt die Strategie des Trace Recovery, also der nachträglichen Erhebung von Trace Links. Er agiert als Suchmaschine für Entsprechungsbeziehungen und verfolgt als solche den Zweck, möglichst gut sortierte Ergebnislisten zu erzeugen. Das Kriterium zur Bewertung dieser Ergebnislisten ist, ob korrekte Entsprechungen in der **Sortierung der Suchergebnisse** möglichst weit vorne auftreten. Die Metrik **M**ean **A**verage **P**recision fasst diese Eigenschaft in einer Zahl zusammen. Sie wurde bereits in Abschnitt 6.2.2 eingeführt und zur Optimierung des Mechanismus als Zielfunktion genutzt. Im Gegensatz zu einer Betrachtung anhand von Precision und Recall bewertet MAP auch die Reihenfolge der Suchergebnisse und ist auf potentiell unendliche Ergebnislisten anwendbar. Der Mechanismus zeichnet sich insbesondere durch die Anwendung von Bezeichnergewichten aus. Es ist folglich anhand der Metrik MAP zu erfassen, inwiefern der Mechanismus eine Verbesserung im Vergleich zu anderen Ansätzen ohne Gewichtung der Bezeichner erzielt. Der Mechanismus muss zudem effizient sein, sodass Entwickler die vorgeschlagenen Trace Links nutzen können, ohne ihren Gedankenfluss zu unterbrechen. Nielsen [1993, S. 135] nennt für eben diese Anforderung einen **Grenzwert von einer Sekunde für die Antwortzeit**. Zwischen der Anfrage nach Trace Links zu einem gegebenen Code-Element und der Vorlage der Suchergebnisse darf dementsprechend höchstens eine Sekunde vergehen.

Das dritte Ziel der Methode besteht darin, evolutionäre Aufgaben plattform-übergreifend anhand der erzielten Ähnlichkeiten und der erhobenen Trace Links zusammenzuführen

und zu koordinieren. Es ist zu prüfen, inwiefern die dafür entwickelten Ansätze realisiert werden können und welche Vereinfachungen sie bieten. Als Kriterium für den Erfolg dieser Ansätze ist ihre **Realisierbarkeit durch prototypische Implementationen** zu zeigen. Darüber hinaus ist zu erheben, inwiefern durch die Anwendung der Ansätze **Arbeitsschritte** im Vergleich zur nicht synchronisierten Evolution **entfallen** oder **koordinierende Tätigkeiten ermöglicht** werden.

8.2 Vorgehen

Wie oben definiert, ist zu evaluieren, ob die hier entwickelte Portierungsmethode anwendbar ist und inwiefern ihre Anwendung dazu führt, dass ein erhöhter Anteil isolierter Typdefinitionen vollständig in Entsprechungen überführt wird. Dazu wurde die Portierungsmethode auf drei Fallstudien angewendet. Abschnitt 8.3 bietet einen Überblick über die portierten Systeme. In Abschnitt 8.4 werden die Abläufe der Fallstudien beschrieben und die erzielten Entsprechungsbeziehungen bewertet.

Um die Realisierbarkeit des Trace-Capture-Mechanismus zu demonstrieren, wird in Abschnitt 8.5 die Erweiterung zweier Quellcode-Konvertoren beschrieben, die im Rahmen eines studentischen Projektes umgesetzt wurden. Der umfangreichere der beiden Konvertoren wurde exemplarisch bei der Portierung einer mobilen App angewendet. Für das dabei erzeugte Traceability-Modell werden Precision und Recall bestimmt.

Die Implementation des Mechanismus zur nachträglichen Suche von Trace Links wurde bereits in Abschnitt 6.3 beschrieben. Die Qualität der erzeugten Ergebnislisten wird in Abschnitt 8.6 geprüft. Dazu wurden für drei Portierungsprojekte je 50 Typdefinitionen und ihre Entsprechungen in der portierten Implementation erhoben. Um die vom Mechanismus vorgeschlagenen Suchergebnisse zu bewerten, wurden sie mit diesen manuell erhobenen Trace Links anhand der Metrik Mean Average Precision verglichen. Um zu prüfen, inwiefern die Anwendung der Bezeichnergewichte den Mechanismus im Vergleich zu aktuellen Clone-Detection-Mechanismen verbessert, wird zu jedem Projekt auch die MAP bei gleicher Gewichtung aller Bezeichner erhoben. Anhand eines statistischen Tests wird geprüft, ob die beobachteten Unterschiede signifikant sind.

Die Anwendung der Ansätze zur Nutzung von Trace Links während der Co-Evolution werden in Abschnitt 8.7 demonstriert. Dort wird jeweils eine prototypische Implementation jedes Ansatzes beschrieben und anhand eines Szenarios zur Co-Evolution an der Fallstudie *Metropole des Wissens* veranschaulicht. Es wird beurteilt, welche Arbeitsschritte durch Anwendung des Ansatzes entfallen oder hinzukommen und welche koordinierenden Tätigkeiten ermöglicht werden.

Jede dieser Maßnahmen zur Evaluation schließt mit einer Diskussion ihrer Validität ab. Dabei werden jeweils folgende drei Fragen diskutiert [Wright u. a. 2010]:

- Interne Validität: Gibt es Einflüsse auf das Ergebnis, die nicht der Kontrolle der Untersuchung unterliegen?
- Externe Validität: Inwiefern können die Ergebnisse auf andere Fälle übertragen bzw. verallgemeinert werden?
- Konstruktvalidität: Reflektieren die erhobenen Daten tatsächlich, ob das angestrebte Ziel erreicht wurde?

8.3 Charakterisierung der durchgeführten Fallstudien

Tabelle 8.1 bietet einen Überblick über die Softwareprojekte, die auf Basis der hier entwickelten Methode portiert wurden.

Fallstudie	LoC (Original Platform)	Ausgangs-Sprache	Ziel-Sprache
App eines Paketdienstleisters	7907	Java	C#
DESMO-JS	12.609	Java	JavaScript
Metropole des Wissens	3084	Swift	Java

Tabelle 8.1: Überblick über die durchgeführten Fallstudien

In der ersten Spalte ist jeweils die Bezeichnung der Fallstudie aufgeführt. Die zweite Spalte enthält die jeweilige Größe der zu portierenden Quellcodebasis in Codezeilen. Die dritte und vierte Spalte enthalten die ursprünglich genutzte Programmiersprache sowie die Sprache der Zielimplementation.

8.3.1 Portierung der mobilen App eines großen deutschen Paketdienstleisters von Android nach Windows Phone

In der ersten Fallstudie wurde die hier entwickelte Portierungsmethode auf die mobile App eines großen deutschen Paketdienstleisters angewendet. Diese bietet ihren Nutzern Informationen zu den Paketshops und Services des Dienstleisters sowie zum Status aktueller Paketsendungen.

Die Fallstudie eignet sich gut für die Evaluation der Methode, da sie eine Anwendung mit kommerziellem Einsatz portiert, sodass die Portierungsmethode unter realistischen Bedingungen angewendet wurde. Die ursprüngliche Implementation der App wurde

von der Firma ICNH[1] in der Programmiersprache Java für das Betriebssystem Android erstellt und wird bis heute dort weiterentwickelt. Im Rahmen der Fallstudie sollte sie für das Betriebssystem Windows Phone unter Nutzung der Sprache C# portiert werden.

Die zu portierende App bot zum Zeitpunkt der Portierung vier Hauptfunktionalitäten:

1. **Sendungsverfolgung:** Der Nutzer konnte sich anhand einer Sendungsnummer den Verlauf des Versandprozesses anzeigen lassen.

2. **Paketshop-Suche:** Er konnte anhand einer Postleitzahl nach Paketshops suchen, die Sendungen entgegennehmen. Die gefundenen Paketshops wurden auf einer Karte dargestellt.

3. **Paketpreis-Rechner:** Der Anwender konnte darüber hinaus den Preis eines Pakets anhand der Paketmaße und der Zieladresse berechnen lassen.

4. **News:** Ferner konnte er Neuigkeiten des Dienstleisters lesen.

Der ursprüngliche Quellcode der Fallstudie ist nicht öffentlich, sodass keine Codebeispiele aus der Fallstudie gegeben werden. Er umfasst 7907 Zeilen Java-Code.

8.3.2 Portierung der Simulationsbibliothek DESMO-J von der Java Laufzeitumgebung nach JavaScript für Webbrowser

Um die Effektivität der Methode auch bei der Anwendung durch Dritte zu prüfen, wurde sie im Rahmen einer Masterarbeit auf die Programmbibliothek DESMO-J angewendet [Greiert 2018].

DESMO-J ist eine Programmbibliothek, die Komponenten zur Definition und Ausführung von Simulationsmodellen sowie zur Auswertung durchgeführter Simulationsexperimente bietet. Bei der Definition der Simulationsmodelle unterstützt DESMO-J zwei Paradigmen: Die zu simulierenden Abläufe können zum einen als Abfolge von Ereignissen modelliert werden, die den globalen Gesamtzustand des Modells verändern. Zum anderen können die Abläufe als Prozesse modelliert werden, die das Verhalten einzelner Entitäten beschreiben, die miteinander interagieren können [Page u. Kreutzer 2005, S. 263 ff.].

DESMO-J bietet außerdem Implementationen verschiedener Zufallsverteilungen an, die die modellierenden Entwickler nutzen, um die Auswirkungen und die zeitliche Abfolge der Ereignisse und Prozesse zufallsbasiert zu modellieren. Die so definierten Modelle können in Simulationsexperimenten wiederholt ausgeführt werden.

[1] www.icnh.de

In DESMO-J sind darüber hinaus Statistik-Komponenten enthalten, die ein simulierender Entwickler einsetzen kann, um während eines Experiments Kennzahlen zu erheben. Basierend auf diesen Statistik-Komponenten wird im Anschluss an das Experiment ein Bericht über die erhobenen Kennzahlen erstellt.

Diese Funktionalitäten sollen perspektivisch auch für Entwickler von Web-Anwendungen mit Simulationsfunktionen verfügbar sein. Dazu wurde ein Großteil von DESMO-J für Webbrowser unter Nutzung der Skriptsprache JavaScript portiert.

8.3.3 Portierung der iOS-App *Metropole des Wissens* für die Android-Plattform

Als dritte Fallstudie dient die Portierung einer mobilen App der Hamburger Behörde für Wissenschaft, Forschung und Gleichstellung, die vor der Portierung ausschließlich in einer Swift-Implementation für das Betriebssystem iOS vorlag. Um sie einem größeren Nutzerkreis zugänglich zu machen, sollte sie in eine Java-Implementation für das Betriebssystem Android überführt werden.

Die Aufgabe eignete sich gut zur Evaluation der hier entwickelten Portierungsmethode, da für die Überführung von Java-Code in die Programmiersprache Swift nur sehr eingeschränkte Quellcode-Konvertoren verfügbar waren. Damit erlaubte es die Fallstudie, die Methode auch unter Abwesenheit geeigneter Konvertoren zu bewerten.

Die ursprüngliche Version der App wurde am Arbeitsbereich *Applied Software Technology* an der Universität Hamburg entwickelt[2]. Sie versorgt ihre Nutzer mit aktuellen Informationen zu den Bildungsinstituten im Großraum Hamburg. Sie bietet die folgenden vier Hauptfunktionalitäten:

1. **News:** Dem Nutzer werden Berichte und Neuigkeiten zu den Instituten angezeigt, die er über diverse Social Media Apps veröffentlichen kann.
2. **Veranstaltungen:** Er kann darüber hinaus Informationen zu bevorstehenden Veranstaltungen der Institute einsehen. Diese kann er mittels Social Media Apps teilen und Kalendereinträge zu den Veranstaltungen in der Kalender-App seiner Wahl generieren lassen.
3. **Karte der Institute:** Der Nutzer kann sich außerdem eine Karte anzeigen lassen, auf dem alle teilnehmenden Institute verzeichnet sind. Wählt er eines der Institute aus, so werden nähere Informationen angezeigt. Von dieser Perspektive aus können die Kontaktinformationen und Social-Media-Kanäle des Instituts unmittelbar mit den geeigneten Apps auf dem Gerät genutzt werden.

[2] `https://mast.informatik.uni-hamburg.de/m-lab-app-hafen-der-wissenschaft-auf-der-landespressekonferenz-vorgestellt/`

4. **Hamburg Open Online University:** Dem Nutzer wird als vierte Funktionalität der Zugang zur *Hamburg Open Online University* geboten. Dabei handelt es sich um ein Portal für Lehr-Materialien zu interdisziplinären Themen sowie für Forschungsergebnisse der Institute[3].

Die ursprüngliche Implementation der App für iOS umfasst 3084 Zeilen Swift Code.

8.4 Evaluation hinsichtlich Anwendbarkeit und erzielter Entsprechungsbeziehungen

Es ist zu prüfen, ob die entwickelte Portierungsmethode anwendbar ist und welche konzeptuellen Entsprechungen sie erzielt. In den folgenden Abschnitten wird dazu die Anwendung der Methode in den drei oben skizzierten Fallstudien beschrieben und der Anteil der Typdefinitionen erfasst, der vollständig in konzeptuelle Entsprechungen überführt wurde.

8.4.1 Anwendung der Methode auf die mobile App eines Paketdienstleisters

In der ersten Fallstudie wurde die Portierungsmethode auf die in Abschnitt 8.3.1 beschriebene mobile App eines großen deutschen Paketdienstleisters angewendet. Die ursprüngliche Implementation der App ist grob in die fünf Ebenen zu unterteilen, die in Abbildung 8.1 zu sehen sind.

[3] `https://www.hoou.de/`

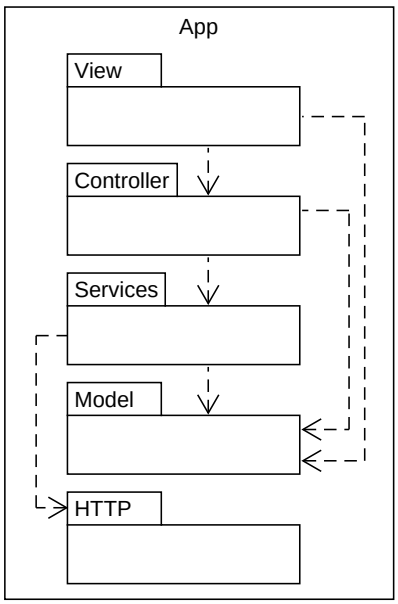

Abbildung 8.1: Architektur der Paketdienstleister-App

Die `View`-Ebene enthält plattformspezifischen XML-Code, der die Elemente der Benutzeroberfläche auf dem Bildschirm anordnet. Die `Controller`-Ebene enthält Klassen, die auf Nutzerinteraktionen reagieren und darunter liegende Schichten nutzen, um fachliche Aufgaben wie die Preisberechnung anzustoßen. Sie aktualisieren darüber hinaus die Benutzeroberfläche, wenn die Services ihren Zustand ändern. Außerdem prüfen sie Nutzereingaben auf Korrektheit und verwalten die anzuzeigenden Daten. Die Klassen der Schicht `Services` bieten geschäftslogische Dienstleistungen wie das Berechnen von Paketpreisen. Objekte der Anwendungsdomäne wie Paketmaße, Preise und Paketshops werden durch die Klassen der `Model`-Ebene repräsentiert. Um asynchron mit dem Server des Paketdienstleisters zu kommunizieren, bietet die `HTTP`-Ebene eine abstrakte Klasse, die Zeichenketten in der JavaScript Object Notation (JSON) herunterlädt und diese zu Instanzen der `Model`-Klassen deserialisiert. Ihre konkreten Subklassen in der `Services`-Ebene legen dazu lediglich die anzusprechende URL sowie den Typ der zu deserialisierenden Objekte fest.

Ablauf der Portierung

Phase I

Im **ersten Schritt** der Methode waren die zu portierenden Funktionalitäten festzulegen. Es wurde beschlossen, sämtliche der vier Hauptfunktionalitäten *Sendungsverfolgung*, *Paketshop-Suche*, *Paketpreis-Rechner* und *News* zu portieren.

Im **zweiten Schritt** wurden die Abhängigkeiten der Implementation zur Android-Plattform untersucht. Dazu bietet die Entwicklungsumgebung Android Studio ein Analysewerkzeug. Mit diesem können die Abhängigkeiten zu festgelegten Paketen ermittelt werden. So wurden Abhängigkeiten der ursprünglichen Implementation zu den Paketen der Android-API, zur Java Runtime Environment sowie zu sämtlichen in der Build-Konfiguration erfassten externen Bibliotheken erhoben. Das Ergebnis wird im Folgenden skizziert.

Durch alle Ebenen hinweg wurden die Typen der Java Runtime Environment genutzt. Dazu gehörten unter anderem primitive Datentypen im Package `Java.lang` sowie Sammlungstypen, die vom Package `java.util` bereitgestellt werden. Die darüber hinausgehenden Abhängigkeiten sind in Abbildung 8.2 auf Paket-Ebene dargestellt.

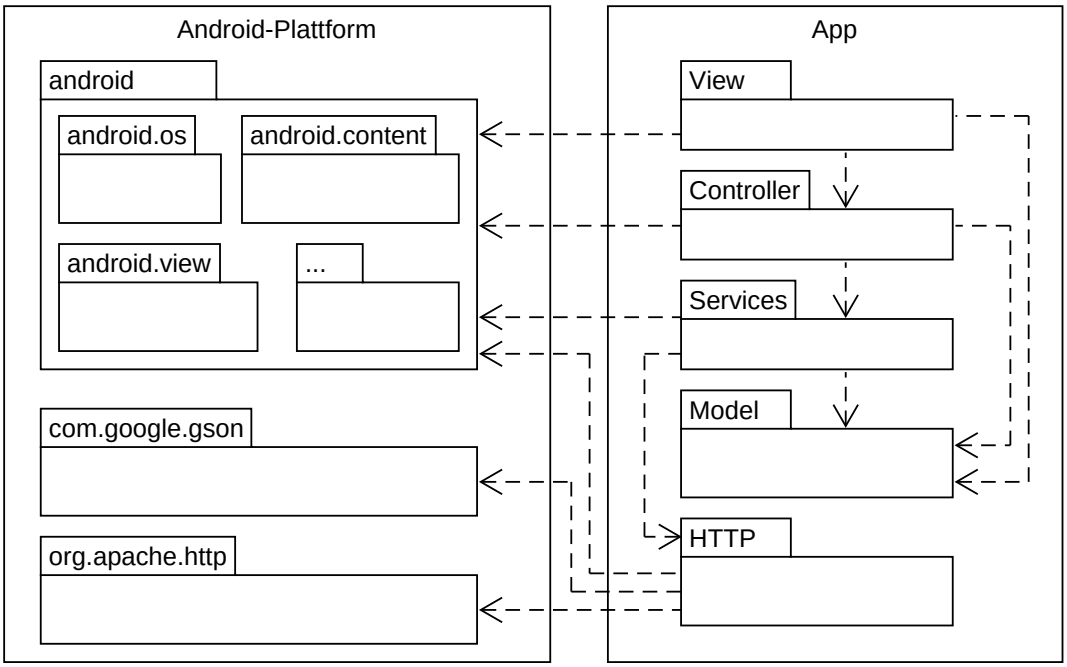

Abbildung 8.2: Abhängigkeiten der Paketdienstleister-App

Insbesondere die Klassen der `View`-Ebene machten intensiven Gebrauch von der Android-API. Die Anordnung der Elemente zur Benutzerinteraktion war darüber hinaus in XML-Dateien definiert, die ausschließlich plattform-spezifische Elemente und Layout-Komponenten verwendeten.

Auch sämtliche Klassen der `Controller`-Ebene erbten von Typen wie `android.app.Activity` oder `android.app.Fragment`, um ihren Lebenszyklus vom Betriebssystem verwalten zu lassen.

Die Klassen der Ebene `Services` nutzten ebenfalls die Android-API. Beispielsweise wurde die Klasse `android.content.SharedPreferences` genutzt, um Einstellungen zu verwalten.

Die in der Ebene `Model` definierten Typen des Domänenmodells waren nicht abhängig vom Betriebssystem oder externen Bibliotheken.

Die Ebene `HTTP` umfasste nur eine abstrakte Klasse, die die asynchrone Kommunikation mit dem Server realisierte. Basierend auf der abstrakten Klasse `android.os.AsyncTask`

lud sie Instanzen der Klassen aus der `Model`-Ebene herunter. Sie nutzte zur Kommunikation das Package `org.apache.http`, das zum Zeitpunkt der ursprünglichen Entwicklung in der Android-API enthalten war. Für die Deserialisierung der JSON-Zeichenketten wurde die externe Bibliothek *GSON*[4] eingesetzt.

Der **dritte Schritt** der Methode befasste sich mit der Bewertung und Auswahl von Quellcode-Konvertoren. Die Wahl fiel auf den Konverter *Sharpen*, der bezüglich des unterstützten Sprachumfangs zu den besten Konvertoren gehört. Zudem handelt es sich bei Sharpen um ein kostenloses und quelloffenes Werkzeug, das somit erweiterbar war [Khanji 2015]. Ein weiterer Vorteil bestand darin, dass Mappings für die Konversion konfiguriert werden konnten, um viele der Abhängigkeiten zur Java Runtime Environment und zur Android-API korrekt auf ihre Entsprechungen abzubilden. Darüber hinaus war in einem vorausgegangenen studentischen Projekt[5] eine Erweiterung für Sharpen entwickelt worden, die beim Einsatz von Sharpen zusätzlich zum konvertierten Code ein Traceability-Modell erzeugt. Sie verknüpft die Code-Elemente der Eingabe vollständig und korrekt mit ihren Übersetzungen.

Der **vierte Schritt** der Methode legte darauf aufbauend fest, welche Teile des ursprünglichen Quellcodes unter Anwendung welcher Abstraktionen portiert werden sollten. Wie in Abschnitt 4.4.4 beschrieben, wurde entschieden, stark plattformabhängigen Code mit technischer Ausrichtung händisch zu portieren, dabei aber die Aufteilung der Typen und ihre Zuständigkeiten zu übertragen. Dies galt für sämtliche Klassen der Ebenen `View`.

In der Ebene `Controller` war insbesondere der Code zur Aktualisierung der Benutzeroberfläche plattform-spezifisch, da er die Schnittstellen der Anzeigeelemente nutzte, die Teil der Android-API sind. Die Verwaltung der anzuzeigenden Daten und die Verarbeitung von Benutzereingaben war dagegen weitgehend frei von Abhängigkeiten zum Betriebssystem. Um diese Vermischung von plattform-spezifischen und portierbaren Elementen zu beheben, wurde entschieden, die `Controller`-Ebene aufzuspalten. Dazu wurde die zusätzliche Ebene `ViewModels` definiert, deren Klassen aus den bestehenden Controllern zu extrahieren waren. Sie verwalten die Benutzereingaben und interagieren mit den Services. Der übrige plattformabhängige Teil der `Controller`-Klassen aktualisiert lediglich die Darstellung der Daten an der Benutzeroberfläche. Dazu wird er von den `ViewModel`-Klassen über Zustandsänderungen benachrichtigt.

Die Klassen der Ebenen `Model` und `Services` hatten nur wenige Plattformabhängigkeiten. Vor dem Hintergrund der Leistungsfähigkeit von Sharpen wurde beschlossen, diesen Code vollständig automatisch zu konvertieren und ihn somit Anweisung für Anweisung in Entsprechungen zu überführen.

[4] `https://github.com/google/gson`

[5] Mobile Anwendungen für mehrere Plattformen - Portierung, Architektur- und Tool-Entwicklung (Masterprojekt 2016/17), vgl. Tabelle 9.4 auf Seite 193 dieser Arbeit

Es wurde entschieden, die Klassen der `HTTP`-Ebene händisch zu re-implementieren, da sie viele Plattformabhängigkeiten aufwiesen. Da ihre Schnittstelle von den darüber liegenden Ebenen genutzt wird, sollte diese in den Entsprechungen exakt nachgebildet werden. Dies war unproblematisch, da ihre Schnittstellen im Gegensatz zu den Klassen der `View`-Ebene nicht durch Vererbungsbeziehungen vorgegeben waren.

Im **fünften Schritt** der Methode erfolgte die Einteilung des Portierungsvorhabens entsprechend der Hauptfunktionalitäten *Sendungsverfolgung*, *Paketshop-Suche*, *Paketpreis-Rechner* und *News* in vier Inkremente. Als erstes Inkrement sollte der Paketpreis-Rechner portiert werden.

Im **sechsten Schritt** war die in Schritt vier beschriebene Aufspaltung der `Controller`-Ebene zunächst für das erste Inkrement zu realisieren. Dazu wurden zunächst View-Models aus den Controllern extrahiert. Diese benachrichtigen die beobachtenden `Controller`-Klassen per Methodenaufruf über Änderungen an den anzuzeigenden Daten. In der Zielplattform werden solche Benachrichtigungen allerdings mit einem plattformspezifischen Event-Mechanismus umgesetzt. Somit bestand die Notwendigkeit, eine Abstraktion von der plattform-spezifischen Benachrichtigung zu schaffen. Darüber hinaus sollten die ViewModels ihren Zustand asynchron initialisieren. Auch dies geschieht unterschiedlich auf den beiden Plattformen. Die in Abschnitt 5.3 entwickelte Richtlinie definiert, dass das Muster *Generation Gap* angemessen ist, da mehrere Abhängigkeiten zu isolieren waren und eine rein technische Abstraktion als Oberklasse extrahiert werden konnte. Dementsprechend wurde die abstrakte Oberklasse `AbstractViewModel<ModelType>` als wiederverwendbarer Super-Client abgespalten. Die Umsetzung des Entwurfsmusters und der Einsatz der ViewModel-Klassen sind in Anhang A.1 anhand eines Beispiels beschrieben. Der Quellcode von `AbstractViewModel` ist öffentlich zugänglich[6].

Phase II

Im **siebten Schritt** der Methode wurden die Abhängigkeiten zur Plattform durch API-Mappings auf ihre Pendants in der Windows Phone Plattform abgebildet. Die Definition dieser Mappings für den Konverter ist online veröffentlicht[7]. Außerdem wurden die notwendigen Adaptionen der Ziel-Plattform entsprechend der in Abschnitt 5.3 definierten Richtlinie umgesetzt. Das betraf beispielsweise Abhängigkeiten zum Interface `java.util.List<T>`, das die Operation `SubList(startIndex, endIndex)` definiert. Diese Operation liefert eine Teilliste, die die Elemente der ursprünglichen Liste vom angegebenen Startindex bis zum Endindex enthält. Die Entsprechung des Typs auf der

[6] `https://github.com/TilStehle/.NetAdapters/tree/master/ViewModelFramework`

[7] `https://swk-www.informatik.uni-hamburg.de/~stehle/sharpen-all-options_nda_conform.txt`
Um einer Verschwiegenheitserklärung gerecht zu werden, wurde der ursprüngliche Name der App in den Package-Bezeichnern durch `GermanParcelService` ersetzt und einige Mappings wurden entfernt. Dies ist entsprechend gekennzeichnet.

Windows Phone-Plattform ist `System.Collections.Generic.IList<T>`. Sie bietet die entsprechende Operation `GetRange(index, count)` an, die ebenfalls eine Teilliste erzeugt. Allerdings erwartet sie als zweiten Parameter die Länge der Teilliste anstatt eines Endindex. Das Interface `IList<T>` sowie alle Klassen, die es implementieren, waren dementsprechend um eine Operation zu ergänzen, die der in Java genutzten Operation `SubList(startIndex, endIndex)` entspricht. Die in Abschnitt 5.3 aufgestellte Richtlinie empfiehlt für diesen Fall die Anwendung des Musters *Extension Method*, da der anzupassende Typ inklusive seiner Subtypen adaptiert werden soll und C# einen entsprechenden Mechanismus zur Definition von *Extension Methods* anbietet. Der Code zur Adaption von `IList<T>` mittels Extension Method ist in Codebeispiel 8.1 zu sehen.

```
public static List<T> SubList<T>(this IList<T> self,
                                 int startIndex, int endIndex)
{
        return self.GetRange(startIndex, endIndex - startIndex);
}
```

Codebeispiel 8.1: Extension Method SubList() zur Erweiterung des Typs System.Collections.Generic.IList

Auf gleiche Weise wurden zwölf weitere Extension Methods implementiert. Darüber hinaus wurden gemäß der Richtlinie zwei *Object Adapter* implementiert. Der Code dieser Extension Methods[8] und Object Adapter[9] ist öffentlich zugänglich. Die extrahierten ViewModels sowie die Services und Klassen des Domänenmodells wurden durch diesen Einsatz strukturangleichender Entwurfsmuster vollständig konvertierbar. Um den isolierten konvertierbaren Quellcode zu übersetzen, wurde Sharpen entsprechend der aufgestellten Mappings konfiguriert und auf die Klassen der Ebenen `Service` und `Model` angewendet.

Im **achten Schritt** der Portierung wurden die Klassen der Ebenen `View`, `Controller` und `Http` re-implementiert, um das erste Inkrement zu vervollständigen, ehe die übrigen Inkremente portiert wurden.

Bemerkenswert ist, dass 79% des adaptierenden Codes bereits bei der Portierung des ersten Inkrements entstand. Sämtliche Adaptionen wurden im späteren Verlauf der Portierung mehrfach verwendet. Einmal vorgenommene Adaptionen werden also für spätere Portierungs-Inkremente wiederverwendet. Tendenziell nimmt der Aufwand für Adaption somit von Inkrement zu Inkrement ab, da die bereits angeglichenen Plattform-Schnittstellen nicht erneut angeglichen werden müssen. Dies legt nahe, dass sich der

[8] `https://github.com/TilStehle/.NetAdapters/blob/master/Extensions`
[9] `https://github.com/TilStehle/.NetAdapters/tree/master/Adapters/Windows%20Phone`

Aufwand für die Anpassung der Zielplattform umso mehr rentiert, je mehr Funktionalitäten zu portieren sind, die dieselben Plattform-APIs nutzen.

Erhebung des Anteils vollständig in Entsprechungen überführter Typdefinitionen

Die gesamte portierte Implementation der App umfasst 6324 Codezeilen in der Programmiersprache C#. Abbildung 8.3 bietet einen Überblick über die erzielten Entsprechungsbeziehungen der Typdefinitionen der Zielimplementation zu ihren Vorbildern. Die Säule repräsentiert 100% des Codes der Zielimplementation. Die Beschriftungen der Säulenabschnitte nennen den Anteil der Typdefinitionen in Codezeilen und Prozent, die ihren Originalen in Bezug auf die Aspekte *Zuständigkeiten*, *Zuständigkeiten und Schnittstellen* bzw. *Zuständigkeiten, Schnittstellen und Anweisungen vollständig* entsprechen. Exakte Entsprechungen zwischen Anweisungen und Methodensignaturen sind wie in Abschnitt 8.1 beschlossen also nur dann in dieser Betrachtung berücksichtigt, wenn ganze Typdefinitionen diese Entsprechungsbeziehungen vollständig erfüllen.

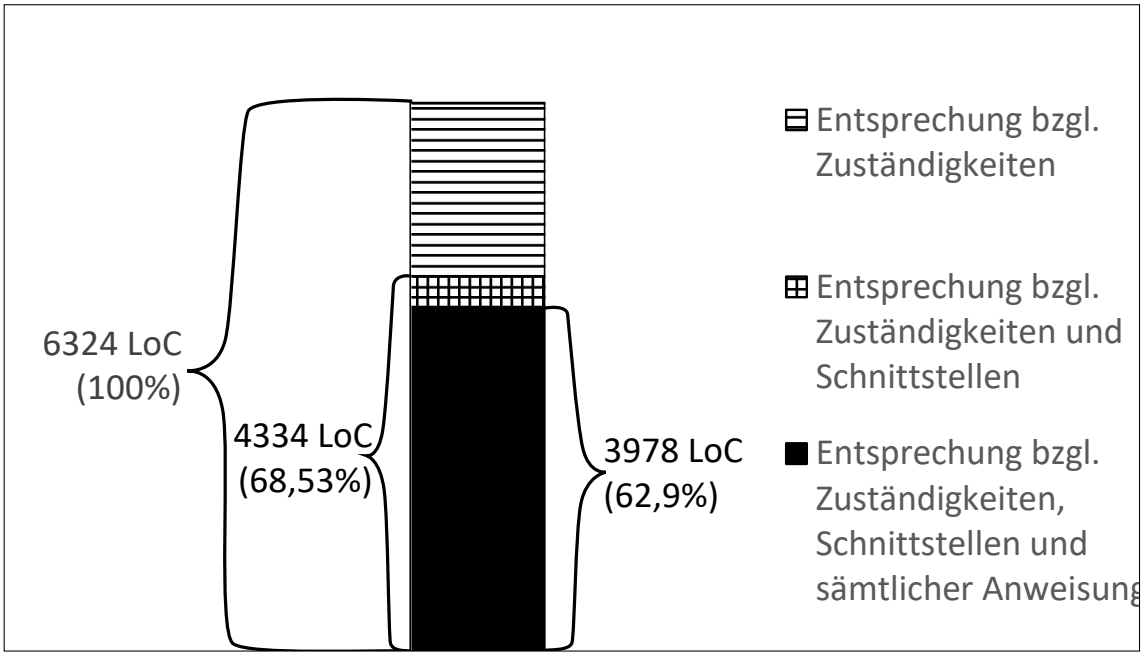

Abbildung 8.3: Anteile der Typdefinitionen, die ihren Vorbildern vollständig entsprechen

62,9% (3978 Zeilen) des Codes liegen in vollständig automatisch konvertierten Typdefinitionen. Jede darin enthaltene Anweisung hat folglich eine Entsprechung im Original.

4334 Zeilen und damit 68,53% des Codes machen solche Typdefinitionen aus, die ihren Vorbildern in Bezug auf Zuständigkeiten und Schnittstellen entsprechen. Dies beinhaltet neben den automatisch konvertierten Typdefinitionen den Code der Adapter, der seinen Vorbildern lediglich bezüglich ihrer Schnittstellen und Zuständigkeiten entspricht.

Diese Adapter umfassen 356 Codezeilen und damit 6% des Codes (mittlerer Säulenabschnitt).

Insgesamt entsprechen sämtliche Typdefinitionen ihren Vorbildern bezüglich ihrer Zuständigkeiten. Die im obersten Säulenabschnitt repräsentierten 1990 Zeilen (31,47%) sind in den portierten Code-Elementen der `View`- und `Controller`-Ebenen enthalten, die den Klassenentwurf der ursprünglichen Implementation übernehmen, ohne die Schnittstellen ihrer Vorbilder exakt nachzubilden. Abschnitt 8.4.4 bewertet dieses Ergebnis gemeinsam mit den folgenden zwei Fallstudien.

8.4.2 Anwendung der Methode auf DESMO-J

In der zweiten Fallstudie wurde die Portierungsmethode auf die Simulationsbibliothek DESMO-J angewendet. Diese Fallstudie wurde im Rahmen einer Masterarbeit durchgeführt [Greiert 2018]. Dem Bearbeiter war die hier entwickelte Portierungsmethode aus einem studentischen Projekt bekannt. Er war vor der Fallstudie ein erfahrener Java-Entwickler, hatte allerdings keine Vorkenntnisse in der Programmierung mit JavaScript. Auch DESMO-J hatte er zuvor weder mitentwickelt noch genutzt.

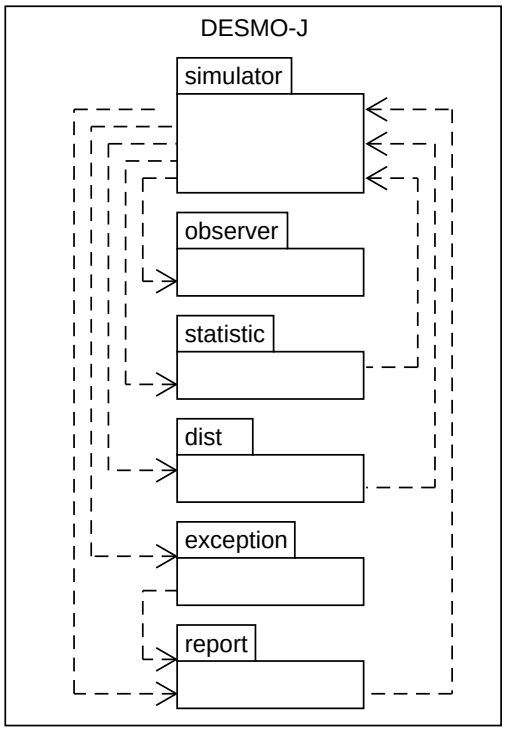

Abbildung 8.4: Die Pakete der Simulationsbibliothek DESMO-J

Abbildung 8.4 zeigt die Paketstruktur der in Java programmierten Ausgangsimplementation von DESMO-J. Basisklassen zur Definition von Simulationsmodellen sowie die Ablaufsteuerung der Simulation liegen im Paket `simulator`. Im Paket `observer` sind lediglich zwei Typdefinitionen enthalten, die die generischen Anteile des Entwurfsmusters *Observer* implementieren. Die Aufzeichnung von Kennzahlen während der Simulation ist in den Klassen des Pakets `statistic` umgesetzt. Das Paket `dist` enthält die von DESMO-J bereitgestellten Zufallsverteilungen. Klassen zur Repräsentation von Fehlern bei der Simulation sind im Paket `exception` definiert. Das Paket `report` enthält die Klassen zur Erzeugung von Simulationsberichten.

Ablauf der Portierung

Phase I

Im **ersten Schritt** der Methode waren die zu portierenden Funktionalitäten zu bestimmen. Um die Portierung im Rahmen einer Masterarbeit beherrschbar zu machen, wurde sie auf die Komponenten beschränkt, die zur Definition, Ausführung und Berichterstattung eines häufig verwendeten Simulationsbeispiels notwendig sind. Das Beispiel simuliert den Transport von Containern in einem Containerterminal[10]. Dazu setzt es sämtliche ereignisbasierten Modellierungs- und Simulationsklassen sowie verschiedene Zufallsverteilungen und Statistik-Komponenten ein. Dieser Querschnitt durch alle Funktionalitäten der Bibliothek ist repräsentativ für DESMO-J, sodass die beobachtete Effektivität der Methode auf eine Portierung weiterer Inkremente übertragbar ist. Die zu portierenden Komponenten umfassten insgesamt 12609 Codezeilen.

Im **zweiten Schritt** wurden die Abhängigkeiten der Bibliothek zur ursprünglichen Plattform untersucht [Greiert 2018, S. 60ff.]. Dazu gehören die Abhängigkeiten zur Java Runtime Environment sowie zu drei externen Bibliotheken. Das Ergebnis der Analyse ist in Abbildung 8.5 auf Paketebene zu sehen.

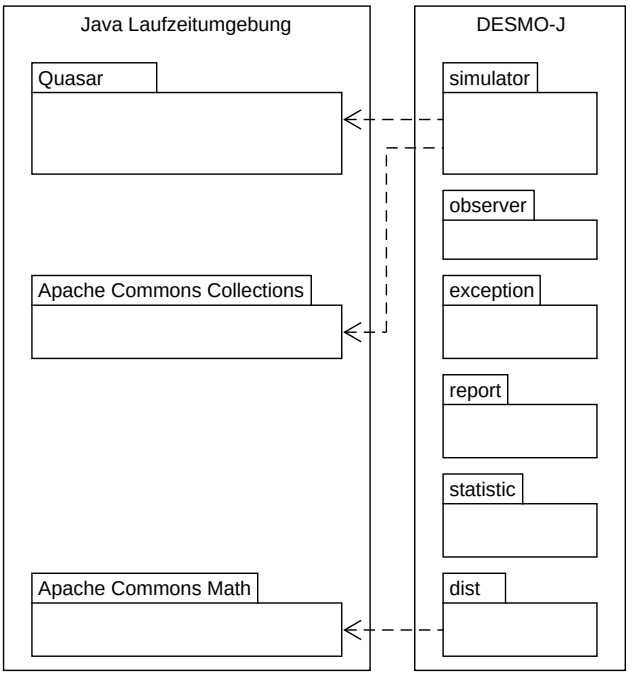

Abbildung 8.5: Abhängigkeiten des zu portierenden Ausschnitts von DESMO-J

Abhängigkeiten der Pakete untereinander sowie zur Java Runtime Environment sind nicht abgebildet, um die Übersichtlichkeit zu bewahren. Die Java Runtime Environment

[10] `http://desmoj.sourceforge.net/tutorial/events/1.html`

mit ihren primitiven Datentypen, Sammlungen und anderen grundlegenden Funktionalitäten wird in allen Klassen von DESMO-J genutzt.

Darüber hinaus bestehen Abhängigkeiten zu den drei externen Bibliotheken *Quasar*[11], *Apache Commons Collections*[12] und *Apache Commons Math*[13].

Quasar bietet eine hochperformante Implementation von Threads, die bei der Ausführung von Simulationsexperimenten nach dem prozessbasierten Paradigma genutzt wird. Zwar ist sie für den zu portierenden Teil von DESMO-J nicht notwendig, da dieser ausschließlich das ereignisorientierte Paradigma nutzt. Allerdings unterstützen einige Klassen aus dem Package `simulator` beide Simulationsparadigmen, sodass Abhängigkeiten zu *Quasar* bestehen. Die zweite externe Bibliothek ist *Apache Commons Collections*, die Sammlungstypen anbietet. Diese werden in zwei Klassen des Pakets `simulator` eingesetzt. Als dritte externe Bibliothek kommt *Apache Commons Math* zum Einsatz. Vier Klassen des Pakets `dist` nutzen diese Bibliothek, um Zufallsverteilungen zu berechnen.

Im **dritten Schritt** der Methode wurden acht verfügbare Konvertoren unter anderem in Bezug auf Funktionsumfang, Korrektheit, Verständlichkeit des konvertierten Codes, Lizenzkosten und Erweiterbarkeit verglichen [Greiert 2018, S. 17ff.]. Am besten geeignet war das Eclipse-Plugin *JSweet*[14], das den vollständigen Umfang der in DESMO-J eingesetzten Java-Syntax übersetzen kann und gut lesbaren JavaScript-Code erzeugt.

Im **vierten Schritt** der Methode wurde festgelegt, welche Abstraktionen bei der Portierung der Klassen von DESMO-J angewendet werden sollte. Da DESMO-J als Programmbibliothek nur wenige Abhängigkeiten zu externen Bibliotheken aufweist und frei von plattform-spezifischen Benutzeroberflächen ist, wurde beschlossen, die zu portierenden Klassen möglichst vollständig automatisch zu portieren und somit Entsprechungen auf der Ebene einzelner Anweisungen zu erzielen.

Die Portierung wurde im **fünften Schritt** in zwei Inkremente eingeteilt: Das erste sollte die Zufallsverteilungen und einen zugehörigen Test portieren, um den gewählten Konverter zu testen. Das zweite Inkrement sollte im Anschluss das Beispielmodell und die darin verwendeten Klassen der Bibliothek überführen.

Schritt sechs der Methode entfiel in diesem Fall, da es sich bei DESMO-J um eine breit eingesetzte Programmbibliothek handelt. Dementsprechend sollte die Struktur der Ausgangsimplementation nicht verändert werden, um externe, von DESMO-J abhängende

[11] `http://docs.paralleluniverse.co/quasar/`
[12] `https://commons.apache.org/proper/commons-collections/`
[13] `https://commons.apache.org/proper/commons-math/`
[14] `http://www.jsweet.org/eclipse-plugin/`

Software nicht zu beeinträchtigen. Abgesehen davon war der gesamte Code des zu portierenden Anteils mit dem verfügbaren Werkzeug konvertierbar, sodass eine Trennung von konvertierbaren und nicht konvertierbaren Anteilen nicht nötig war.

Phase II

Im **siebten Schritt** wurde der Konverter JSweet angewendet, der als Eclipse-Plugin zur Verfügung steht. Zur Konfiguration des Konverters ist lediglich das entsprechende Eclipse-Projekt für JSweet als Input für die Konversion zu markieren. Das Plugin prüft daraufhin den im Projekt enthaltenen Java-Quellcode auf Konvertierbarkeit und gibt gegebenenfalls Warnungen aus. Anschließend führt es den Konversionsvorgang selbstständig durch. Viele Abhängigkeiten zur Java Runtime Environment werden ohne zusätzliche Mappings korrekt in die Zielsprache übersetzt. Die übrigen Mappings wurden in einer JavaScript-Datei definiert[15]. Um die in Browsern verfügbaren JavaScript-Schnittstellen an die ursprünglich genutzten Schnittstellen der Java Runtime Environment anzupassen, wurden die in Abschnitt 5 diskutierten Entwurfsmuster eingesetzt. Beispielsweise wurde ein Object Adapter für die Schnittstelle der Klasse `java.util.random` definiert, der intern die JavaScript-Implementation eines Zufallszahlengenerators benutzt[16].

Die Entwurfsmuster wurden durch den portierenden Entwickler zunächst eingesetzt, ohne der in Abschnitt 5.3 definierten Richtlinie zu folgen. Um die Wirksamkeit der Richtlinie zu prüfen, wurden die Adaptionen der Zielplattform nachträglich an diese Richtlinie angepasst. Beispielsweise musste der Typ `Number` in der Zielplattform um drei Operationen erweitert werden. Diese Operationen waren ursprünglich als statische Methoden implementiert worden, die eine Erweiterung von `Number` auf Basis des Musters *Extension Method* simulieren. Infolge dessen mussten die Klassen `AbstractChartDataTable` und `NumericalDist` in der Zielplattform händisch angepasst werden, sodass sie diese statischen Methoden aufriefen. Dieser Ansatz zur Adaption der Zielplattform verstieß gegen die Empfehlung, Extension Methods nur durch entsprechende Erweiterungsmechanismen umzusetzen. Durch die nachträgliche Implementierung der `Extension Methods` als JavaScript-Properties wurde die Richtlinie nachträglich befolgt[17]. Die zuvor notwendigen händischen Änderungen an den Klassen `AbstractChartDataTable` und `NumericalDist` entfielen dadurch. Sie konnten durch die Anwendung der Richtlinie vollständig automatisiert in eine korrekte Übersetzung konvertiert werden.

Um das portierte Beispiel im **achten Schritt** vollständig ausführbar zu machen, mussten drei Klassen der Zielimplementation manuell angepasst werden, die fehlerhaft

[15] Die zusätzlichen Mappings sind auf der beiliegenden CD unter `DESMO-JS/editedJS/Mappings_Platform.js` zu finden.

[16] Der Adapter ist auf der beiliegenden CD unter folgendem Pfad zu finden: `DESMO-JS/editedJS/Random.js`

[17] Die entsprechenden Erweiterungen sind online unter folgender URL verfügbar: `https://github.com/TilStehle/DESMO-JS/blob/master/webapp/editedJS/Extensions_Number.js`

konvertiert worden waren [Greiert 2018, S. 81]. Dies wäre nur durch eine Anpassung der Ausgangsimplementation oder des Konverters vermeidbar gewesen. Auf eine Änderung der Ausgangsimplementation wurde verzichtet, um externen Code mit Abhängigkeiten zu DESMO-J nicht zu beeinträchtigen. Zur Anpassung des Konverters wäre ein unverhältnismäßig hoher Aufwand für die Einarbeitung in dessen Code entstanden.

JSweet produziert keine Trace Links zwischen dem zu konvertierenden Java-Quellcode und der entsprechenden Übersetzung. Um plattform-übergreifende Trace Links zu erhalten, wurde daher der in Abschnitt 6 entwickelte Trace-Recovery-Mechanismus eingesetzt. Die Qualität der ermittelten Trace Links wird in Abschnitt 8.6 untersucht.

Eine ausführbare Version des portierten Beispiel-Experiments inklusive der Zielimplementation von DESMO-J ist auf der beiliegenden CD zu finden[18].

Erhebung des Anteils vollständig in Entsprechungen überführter Typdefinitionen

Der Quellcode der Zielimplementation umfasst 14230 Zeilen JavaScript-Code. Abbildung 8.6 bietet einen Überblick über den Umfang der erzielten Entsprechungsbeziehungen zwischen Typdefinitionen der Zielimplementation und ihren Vorbildern.

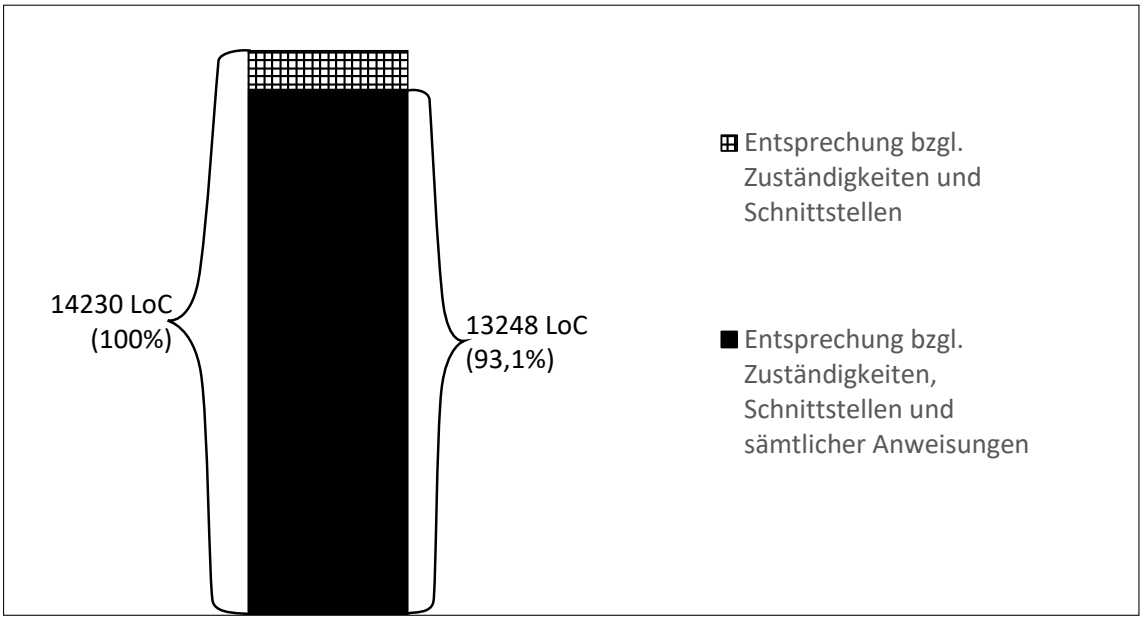

Abbildung 8.6: Anteile der Typdefinitionen, die ihren Vorbildern vollständig entsprechen

13248 Zeilen und damit 93,1% der Zielimplementation liegen in vollständig automatisch übersetzten Typdefinitionen, die Anweisung für Anweisung ihren Pendants der Ausgangsimplementation entsprechen.

[18] Zum Starten des Beispielmodells muss lediglich die Datei `DESMO-JS/EventsExample.html` im Webbrowser geöffnet werden.

Sämtliche Typdefinitionen der Zielimplementation entsprechen ihren Vorbildern bezüglich ihrer Zuständigkeiten und Schnittstellen. Über die automatisch konvertierten Typen hinaus weisen also auch die händisch bearbeiteten Typdefinitionen im Umfang von 982 Codezeilen diese Entsprechungen auf, ohne ihren Vorbildern Anweisung für Anweisung zu entsprechen. Sie sind im oberen Balkenabschnitt repräsentiert. Dazu zählen zum einen die drei manuell angepassten und damit manuell zu evolvierenden Typdefinitionen `MersenneTwisterRandomGeneratorJS`, `BrowserOutputWriter` und `EventTreeList`, die 316 Codezeilen (2,2%) ausmachen. Zum anderen sind darin die adaptierenden Typdefinitionen im Umfang von 666 Codezeilen (4,7%) enthalten, die die ursprünglich genutzten Schnittstellen zur Plattform nachbilden. Abschnitt 8.4.4 bewertet dieses Ergebnis gemeinsam mit den anderen beiden Fallstudien.

8.4.3 Anwendung der Methode auf die App *Metropole des Wissens*

Die in dieser Arbeit entwickelte Portierungsmethode wurde auf die in Abschnitt 8.3.3 beschriebene mobile App angewendet, die ihre Nutzer mit aktuellen Informationen zu den Bildungsinstituten im Großraum Hamburg versorgt. Um die Zielgruppe der Nutzer zu vergrößern, sollte die ursprünglich nur für das Betriebssystem iOS verfügbare App auf die Android-Plattform portiert werden. Die Architektur der Ausgangsimplementation für iOS ist in Abbildung 8.7 dargestellt.

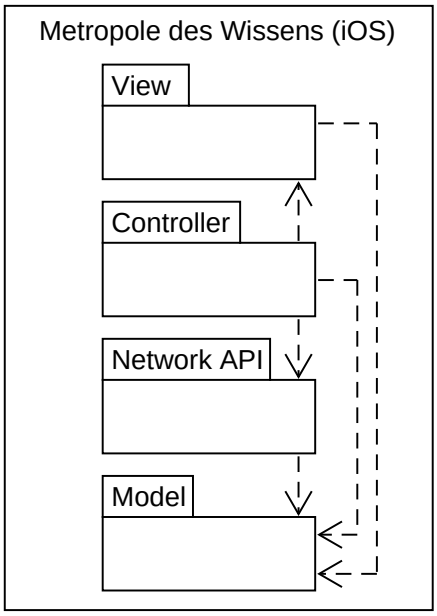

Abbildung 8.7: Paketstruktur der Ausgangsimplementation von *Metropole des Wissens*

Sie definiert die Pakete `Model`, `View` und `Controller` in Anlehnung an das weit verbreitete Muster `MVC` sowie ein zusätzliches Paket `NetworkAPI`.

Die Klassen des `Model-Pakets` modellieren die Entitäten der Domäne. Dazu gehören etwa Institute, Veranstaltungen und Neuigkeiten.

Das Paket `Network-API` definiert zwei Klassen, die diese Informationen von einem Webserver abrufen. Die erste dieser zwei Klassen besteht lediglich aus Konstanten, die die Adressen zur Kommunikation mit dem Webserver beinhalten. Die zweite Klasse kommuniziert asynchron mit diesem Server, um Veranstaltungen, Neuigkeiten und

Informationen zu den Instituten herunterzuladen und als Instanzen der `Model`-Klassen bereitzustellen.

Die Klassen des `View`-Pakets stellen die Informationen zu Instituten, Veranstaltungen und Neuigkeiten grafisch dar. Die Anordnung der Elemente der Benutzeroberfläche ist dabei in einer iOS-spezifischen Datei beschrieben, die als *Storyboard* bezeichnet wird.

Die Klassen des `Controller`-Pakets weisen den `View`-Klassen die aktuell darzustellenden Instanzen der `Model`-Ebene zu und verarbeiten Benutzerinteraktionen.

Ablauf der Portierung

Phase I
Im **ersten Schritt** des Projektes wurde in Absprache mit dem Kunden festgelegt, dass die Ausgangsimplementation vollumfänglich zu portieren war.

Im **zweiten Schritt** der Portierung wurden die Abhängigkeiten der ursprünglichen Implementation zur iOS-Plattform analysiert. Das Ergebnis ist in Abbildung 8.8 zu sehen.

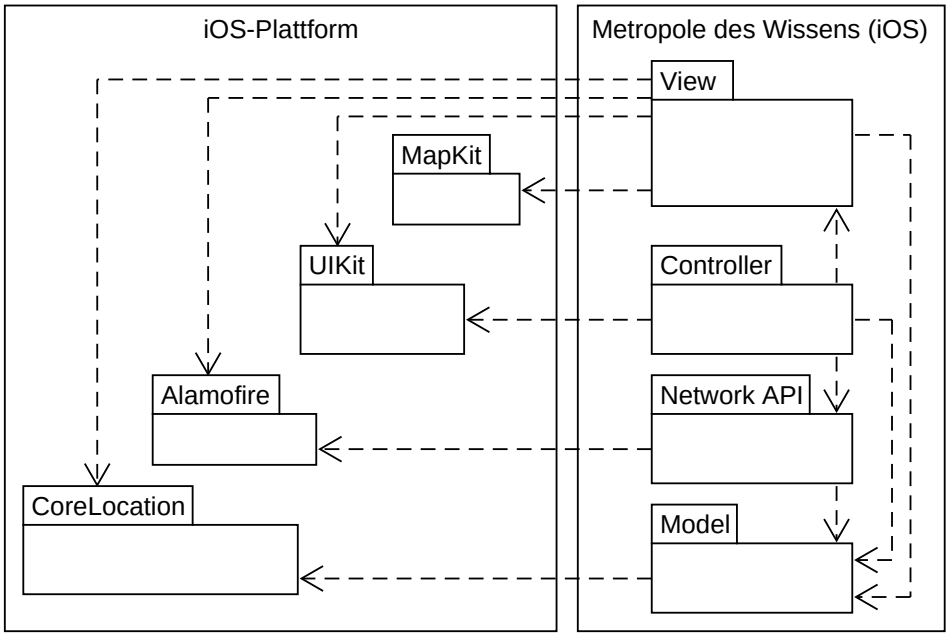

Abbildung 8.8: Abhängigkeiten der Ausgangsimplementation von *Metropole des Wissens* zur iOS-Plattform

Die relevanten APIs der Plattform sind auf der linken Seite der Abbildung dargestellt, die Ausgangsimplementation auf der rechten. Um die Lesbarkeit der Abbildung zu gewährleisten, sind Abhängigkeiten zum grundlegenden Framework *Foundation* nicht abgebildet. Dieses Framework definiert unter anderem die Basisdatentypen, die in sämtlichen Paketen genutzt wurden.

Die Klassen des `View`-Pakets nutzten darüber hinaus das Framework `MapKit`, das die Darstellung einer Karte mit Positionsmarkierungen ermöglicht. Die dazu benötigte Positionsangabe wurde durch einen Typ des Frameworks `CoreLocation` modelliert. Sämtliche Klassen der Pakete `View` und `Controller` waren außerdem vom Framework `UIKit` abhängig, das grafische Oberflächenelemente für iOS definiert. Das Paket `View` nutzte darüber hinaus das Framework `AlamoFire`, um darzustellende Bilder anhand einer URL herunterzuladen. Im Paket `Network API` wurde das Framework *AlamoFire* zur asynchronen Kommunikation mit dem Webserver eingesetzt. Die Klassen des Pakets `Model` hingen von der Programmbibliothek *SwiftyJSON* ab, die die vom Server übertragenen Objekte aus der JavaScript Object Notation (JSON) deserialisiert.

Im **dritten Schritt** waren die verfügbaren Quellcode-Konvertoren zu bewerten. Zum Zeitpunkt der Portierung war kein Konverter verfügbar, der die Swift-Quelltexte direkt in Java-Code übersetzen konnte. Allerdings stand der Konverter SwiftKotlin[19] zur Verfügung, der Swift-Quellcode in die Sprache Kotlin übersetzt. Kotlin kann wiederum mit der Entwicklungsumgebung Android Studio zu Java-Quellcode konvertiert werden. Somit war eine transitive Übersetzung des Swift-Quelltextes über Kotlin nach Java möglich. SwiftKotlin konnte zum Zeitpunkt der Fallstudie allerdings nur einen begrenzten Ausschnitt der Syntax von Swift korrekt übersetzen. Außerdem wurden nur Abhängigkeiten zu Basisdatentypen korrekt auf ihre Kotlin-Entsprechungen abgebildet. Weitere Abhängigkeiten konnten nicht durch API-Mappings auf etwaige Entsprechungen abgebildet werden, da der Konverter keine Konfiguration solcher Mappings anbot. Infolge dieser Einschränkungen war keine der ursprünglichen Typdefinitionen vollständig konvertierbar. Damit stand auch fest, dass der Konverter nicht zur vollständigen Übertragung von Änderungen während der Co-Evolution eingesetzt werden konnte. Trotz dieser Einschränkung sollte SwiftKotlin genutzt werden, um Java-Vorlagen für sämtliche Klassen der `Model`-Schicht und weitere simple Typdefinitionen zu erzeugen, die ausschließlich Konstanten enthielten oder Schnittstellen definierten.

Aufbauend auf dieser Entscheidung wurde im **vierten Schritt** festgelegt, unter welchen Abstraktionen die ursprünglichen Quellcode-Elemente in ihre Entsprechungen für Android portiert werden sollten. Abbildung 8.9 stellt die angestrebten Entsprechungsbeziehungen zwischen den Elementen der vier Schichten dar.

[19] `https://github.com/angelolloqui/SwiftKotlin`

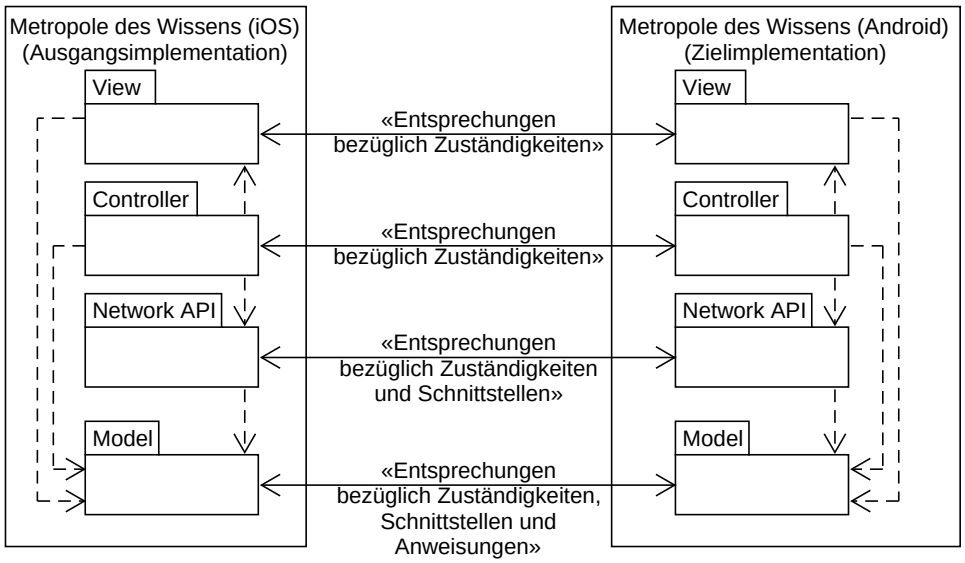

Abbildung 8.9: Entsprechungsbeziehungen zwischen Ausgangs- und Zielimplementation der App *Metropole des Wissens*

Die Anweisungen der `Model`-Klassen sollten eins zu eins portiert werden. Bei der Portierung der technischen Typen aus den Schichten `View` und `Controller` sollten plattformübergreifende Entsprechungen zwischen Typdefinitionen mit gleicher Zuständigkeit entstehen. Auf exakte Entsprechungen auf der Ebene von Anweisungen oder Schnittstellen wurde, wie in Abschnitt 4.4.4 empfohlen, verzichtet, da sie die Benutzeroberfläche der App realisieren und dazu ausgiebigen Gebrauch von den APIs der Plattform machen. Die in der Schicht `Network API` definierten Typen sollten in Pendants mit exakt entsprechenden Schnittstellen überführt werden, da die Typen zwar stark plattformabhängig implementiert waren, aber ihre Schnittstellen nicht plattform-spezifisch waren. Mit dieser Festlegung der Abstraktionsebenen anhand der bestehenden Paketstruktur war keine Änderung der Architektur notwendig. Die Soll-Architektur entsprach damit der Ist-Architektur auf Paketebene.

Im **fünften Schritt** der Methode wurde die Portierung der App in vier Inkremente aufgeteilt, die nacheinander die Hauptfunktionalitäten *News*, *Veranstaltungen*, *Karte der Institute* und *Hamburg Open Online University* in die Zielplattform überführen sollten.

Im **sechsten Schritt** wurden einige simple Refactorings durchgeführt, um deplatzierte Code-Elemente aus den Paketen `Network API`, `View` und `Controller` in das Paket `Model` zu verschieben. Dadurch wurde der Anteil des Codes erhöht, der Anweisung für Anweisung portiert werden konnte. Ein Beispiel dafür ist die neu definierte Methode `append(EventFeed)`, die einer bestehenden Sammlung von Veranstaltungen zusätzliche Elemente hinzufügt. Sie wurde aus der Klasse `EventViewController` im Paket `Controller` extrahiert und in die Klasse `EventFeed` des Domänenmodells

verschoben. Als Teil einer weitgehend plattform-unabhängigen Klasse konnte sie sogar vollständig automatisiert übersetzt und damit Anweisung für Anweisung in Entsprechungen überführt werden.

Phase II

Im **siebten Schritt** wurden die Quellcode-Konvertoren eingesetzt, um Vorlagen beispielsweise für die Java-Klassen des Domänenmodells zu generieren. Um dabei die vorgesehenen Entsprechungen auf Anweisungsebene zu ermöglichen, wurden einige Schnittstellen der Android-Plattform an die ursprünglich genutzten Schnittstellen von iOS angepasst. So musste etwa die Klasse `java.util.Date` adaptiert werden, die dem Struct `Date` in der iOS-Plattform entspricht. `Date` wird in den Modell-Klassen genutzt, um den Beginn und das Ende einer Veranstaltung in einem Attribut zu erfassen. Die ursprüngliche Implementation der App erweitert dieses Struct um die Methode `weekday()`, die den Wochentag des repräsentierten Datums als `String` bestimmt. Laut der in Abschnitt 5.3 definierten Richtlinie ist das Muster *Class Adapter* hier angemessen, da externer Quellcode in der Fallstudie nicht relevant ist und `java.util.Date` die Vererbung zulässt. Dementsprechend wurde ein solcher Class Adapter implementiert. Codebeispiel A.1 im Anhang zeigt einen Ausschnitt des Quellcodes, der diesen Adapter realisiert.

Die mit den Konvertoren generierten Vorlagen wurden im **achten Schritt** wo nötig vervollständigt. Im Anschluss wurden die Pakete `View` und `Controller` händisch portiert. Dabei konnten viele Anweisungen durch einfache Anpassung der Syntax aus ihren Vorbildern der ursprünglichen Implementation gewonnen werden. Zur Gewinnung von Trace Links wurde der in Abschnitt 6 entwickelte Mechanismus eingesetzt. Die Qualität der gewonnenen Trace Links wird in Abschnitt 8.6 bewertet. Abschnitt 8.7.1 greift das Ergebnis dieser Fallstudie auf und demonstriert anhand des ursprünglichen und portierten Quellcodes verschiedene Werkzeuge für die plattform-übergreifende Co-Evolution.

Erhebung des Anteils vollständig in Entsprechungen überführter Typdefinitionen

Die Zielimplementation von *Metropole des Wissens* umfasst insgesamt 4802 Zeilen Quellcode in der Programmiersprache Java. Abbildung 8.10 bietet einen Überblick über die erzielten Entsprechungsbeziehungen der Typdefinitionen der Zielimplementation zu ihren Vorbildern.

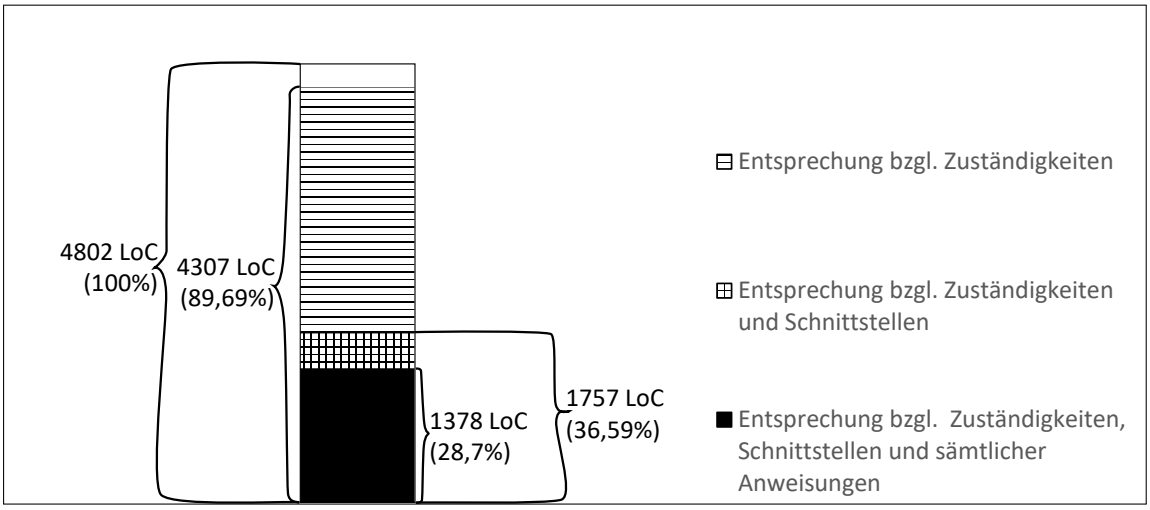

Abbildung 8.10: Anteile der Typdefinitionen, die ihren Vorbildern vollständig entsprechen

1378 Zeilen und damit 28,7% der Zielimplementation liegen in Typdefinitionen, die ihren Vorbildern Anweisung für Anweisung entsprechen. Diese sind im untersten Balkenabschnitt repräsentiert.

Darüber hinaus wurden Typdefinitionen im Umfang von 379 Codezeilen (7,98%) unter exakter Nachbildung der ursprünglichen Schnittstellen portiert. Diese repräsentiert der karierte Balkenabschnitt. Insgesamt umfassen die Typdefinitionen mit exakt abgebildeten Schnittstellen also 1757 Zeilen und damit 36,59% der Zielimplementation. Dazu zählen auch die Adapter zur Anpassung der Schnittstellen zur Zielplattform, die 106 Zeilen (3%) ausmachen.

Weitere 2550 Codezeilen bilden Typdefinitionen, die ihrem ursprünglichem Pendant lediglich in Bezug auf ihre Zuständigkeit entsprechen (horizontal gestreifter Balkenabschnitt). Insgesamt entsprechen damit Typdefinitionen im Umfang von 4307 Zeilen und damit 89,69% ihren Vorbildern bezüglich ihrer Zuständigkeiten.

Hinzu kommen 495 Zeilen (10,31%) in Typdefinitionen der `Controller`-Ebene, zwischen denen eine plattform-spezifische Aufteilung von Zuständigkeiten herrscht. Dies kommt insbesondere dadurch zustande, dass die Android-Plattform spezifische Klassen

zur Erzeugung grafischer Listenelemente vorsieht. Diese Zuständigkeit wird in der iOS-Implementation von den Controller-Klassen übernommen, die auch das Laden der anzuzeigenden Elemente anstoßen und die Darstellung der gesamten Listenansicht steuern.

8.4.4 Zusammenfassende Bewertung der Fallstudien

Abbildung 8.11 stellt die erzielten Anteile von Typdefinitionen mit Entsprechungen zu ihren Vorbildern für die drei oben beschriebenen Fallstudien noch einmal im Vergleich dar.

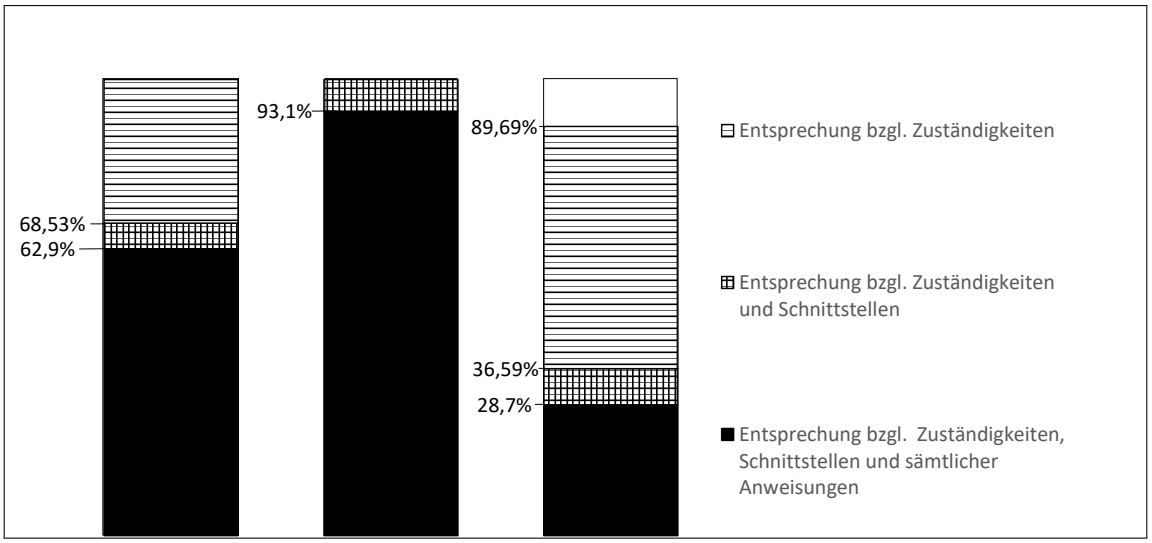

Abbildung 8.11: Anteile der Typdefinitionen, die ihren Vorbildern vollständig entsprechen

In allen drei Fallstudien entspricht mit 100%, 100% und 89,69% ein Großteil der portierten Typdefinitionen ihren Vorbildern der Ausgangsimplementation bezüglich ihrer Zuständigkeiten.

Der Anteil der Typdefinitionen, die ihren Vorbildern auch in Bezug auf Schnittstellen und Anweisungen entsprechen, unterscheidet sich dagegen stark. Der geringere Anteil in der ersten und dritten Fallstudie ist auf die Notwendigkeit zurückzuführen, eine grafische Benutzerschnittstelle zu portieren, die typischerweise viele Plattformabhängigkeiten aufweist. Dies stellt bereits Terekhov [2001] fest. Schließt man die Implementationen der Benutzeroberfläche von der Betrachtung aus, so beläuft sich der Anteil der auf Anweisungsebene portierten Typdefinitionen in der ersten und dritten Fallstudie auf 92% (Fallstudie 1) bzw. 75% (Fallstudie 3). Die Methode ist somit bezüglich der erwirkten Entsprechungsbeziehungen erfolgreich.

Tabelle 8.2 erfasst die jeweiligen Anteile des automatisch übersetzten Codes in der zweiten Spalte.

Fallstudie	Automatische Quellcode-Konversion
App eines Paketdienstleisters	63% automatisch konvertiert
DESMO-JS	93% automatisch konvertiert
Metropole des Wissens	40% basiert auf generierten Java-Vorlagen

Tabelle 8.2: Überblick über erreichte plattform-übergreifende Entsprechungen in den Fallstudien

In den Fallstudien, in denen ein ausgereifter Konverter zum Einsatz kam, wurden 63% (Fallstudie 1) bzw. 93% (Fallstudie 2) des Quellcodes automatisch übersetzt. Der vergleichsweise hohe Wert von 93% ist darauf zurückzuführen, dass die zu portierende Bibliothek DESMO-J keine interaktive Benutzeroberfläche definiert und nur wenige Abhängigkeiten zu anderen APIs aufweist.

Auch für Portierungsprojekte, in denen kein ausgereifter Konverter verfügbar ist, bietet die Anwendung der Methode einen Vorteil gegenüber einem Ad-hoc-Vorgehen. Das deutet zum Beispiel der Vergleich der dritten Fallstudie mit dem Projekt *Twidere* an, bei dem es sich um eine Ad-hoc-Portierung ohne den Einsatz der beschriebenen Methode handelt. Eine Beschreibung des Projekts findet sich in Abschnitt 6.2.3. Wie in der dritten Fallstudie adressiert die Portierung von Twidere die Betriebssysteme iOS und Android mit den Programmiersprachen Swift und Java. Im Gegensatz zur Fallstudie, bei der die Methode angewendet wurde, unterscheiden sich die Entwürfe von Twidere für Android und iOS stark in Bezug auf die Zuständigkeiten der Typen. Selbst in den Klassen des Domänenmodells unterscheidet sich der Zuschnitt der Typen, obwohl abgesehen von einer Bibliothek zur Deserialisierung von JSON-Objekten keine Abhängigkeiten bestehen. Keine der Typdefinitionen in der Zielimplementation von Twidere entspricht ihrem Vorbild Anweisungen für Anweisung. In der dritten Fallstudie hingegen trifft dies auf Typdefinitionen im Umfang von 28,7% des Codes zu.

Die in Abschnitt 5.1 beleuchteten Portierungsprojekte, die ohne Anwendung der hier entwickelten Portierungsmethode durchgeführt wurden, erreichen durchaus einen hohen Automatisierungsgrad. Dieser kann allerdings aufgrund der vielen verteilten manuellen Anpassungen am konvertierten Code nicht numerisch erfasst werden. Ein wesentlicher Unterschied dieser Portierungsprojekte zu den oben beschriebenen Fallstudien besteht darin, dass durch den Einsatz der Portierungsmethode konvertierbarer Quellcode vollständig, also für ganze Typdefinitionen von manuell zu wartendem Quellcode

getrennt wurde. Manuelle Änderungen an den entsprechenden konvertierten Typdefinitionen sind somit im Gegensatz zu den Projekten Lucene.Net[20] und NGit[21] nicht notwendig. In Bezug auf die Isolation konvertierbarer Elemente von plattform-spezifischen Elementen bietet die Methode somit ebenfalls einen Vorteil.

8.4.5 Diskussion der Validität

Interne Validität
Zwei der Fallstudien wurden vom Autor selbst durchgeführt. Es besteht die Gefahr, dass der portierende Entwickler einen maßgeblichen Einfluss auf die erzielten Entsprechungsbeziehungen hat. Um dieser Gefahr zu begegnen, wurde die zweite Fallstudie durch einen anderen Entwickler durchgeführt.

Externe Validität
Grundsätzlich ist jedes Portierungsprojekt individuell. Insbesondere die Verfügbarkeit eines Konverters, die Kombination aus ursprünglich eingesetzter Programmiersprache und Zielsprache sowie das Ausmaß der Plattformabhängigkeiten unterscheiden sich von Fall zu Fall. Alle drei Faktoren haben einen Einfluss auf den Aufwand für die Isolation portierbarer Elemente. Insofern ist die Übertragbarkeit der Ergebnisse einer einzelnen Fallstudie begrenzt.

Um dies zu berücksichtigen, wurden zur Evaluation der Methode drei Fallstudien durchgeführt, die sich in Bezug auf diese Voraussetzungen stark unterschieden: Die ersten beiden Fallstudien setzten einen ausgereiften Konverter ein, die dritte nicht. Die beiden Fallstudien, in denen mobile Apps portiert wurden, portieren Benutzeroberflächen mit vielen Plattformabhängigkeiten, Fallstudie 2 weist vergleichsweise wenige Abhängigkeiten zur ursprünglichen Plattform auf. Alle drei Fallstudien unterscheiden sich in der Kombination von ursprünglich eingesetzter Programmiersprache und Zielsprache. Fallstudien für alle Kombinationen hätten den Rahmen dieser Arbeit überstiegen. Die aufgeführten Unterschiede zwischen den Fallstudien steigern aber die Übertragbarkeit der Ergebnisse für zukünftige Portierungen.

Konstruktvalidität
Die Entsprechungen zielen darauf ab, die Co-Evolution durch Erfahrungsportabilität zu vereinfachen. Da der Umfang einer Typdefinition erheblichen Einfluss auf den Evolutionsaufwand hat, wurde nicht die Anzahl der Typdefinitionen mit Entsprechungsbeziehungen zu ihren Vorbildern gemessen, sondern deren Umfang in Codezeilen. Gemessen wurde jeweils der **Anteil der Typ-Definitionen**, die ihren Vorbildern *eindeutig* und *vollständig* in Bezug auf ihre Zuständigkeit, ihre Schnittstelle bzw. die in ihnen

[20] `https://github.com/apache/lucenenet/`
[21] `https://github.com/mono/ngit`

enthaltenen Anweisungen entsprechen. Die Entsprechungen sind nur dann in der Co-Evolution wertvoll, wenn die Entwickler sie erkennen und sie nutzen. Insbesondere sollen sie bereits anhand der Typdefinition und ihrer Zuordnung zum Vorbild erkennen, welche Entsprechungsbeziehungen vorliegen. Entsprechungen zwischen einzelnen Anweisungen wurden daher nicht erfasst, wenn sie nur einen Teil einer Typdefinition ausmachen.

8.5 Demonstration und Bewertung des Trace-Capture-Mechanismus für Quellcode-Konvertoren

In Abschnitt 4.5.1 wurde vorgeschlagen, Trace Links bereits bei der Konversion des ursprünglichen Quelltextes zu erzeugen, sofern der übersetzte Code ohne manuelle Anpassung genutzt werden kann. Um die Realisierbarkeit dieses Ansatzes zu demonstrieren, wurden die Quellcode-Konvertoren *Sharpen*[22] und *J2Swift*[23] im Rahmen eines studentischen Projekts[24] unter Leitung des Autors um die automatische Erzeugung von Trace Links erweitert. Die Konvertoren übersetzen Java-Quellcode in die Programmiersprachen C# bzw. Swift. Die erweiterte Version von J2Swift ist öffentlich zugänglich[25], während die Erweiterung von Sharpen aus lizenzrechtlichen Gründen nicht veröffentlicht wurde.

Beide Erweiterungen erzeugen bei jeder Konversion ein Traceability-Modell und speichern es als XML-Datei. Das Modell verknüpft sämtliche ursprünglichen Typdefinitionen, Methoden und Attribute mit ihren Entsprechungen in der Übersetzung. Die Qualität der dabei generierten Traceability-Modelle ist, wie in Abschnitt 8.1 festgelegt, anhand der Metriken Precision und Recall zu bewerten.

8.5.1 Einsatz des Trace-Capture-Mechanismus bei der Portierung einer mobilen App

Im Rahmen des oben erwähnten studentischen Projekts[26] wurden Teile der Android-App *Se-Manager* auf das Betriebssystem Windows Phone portiert. Bei dieser Portierung kam der in Abschnitt 4.5.1 beschriebene Trace-Capture-Mechanismus zum Einsatz.

[22] `https://github.com/mono/sharpen`
[23] `https://github.com/patniemeyer/j2swift`
[24] Mobile Anwendungen für mehrere Plattformen - Portierung, Architektur- und Tool-Entwicklung (Masterprojekt 2016/17), vgl. Tabelle 9.4 auf Seite 193 dieser Arbeit
[25] `https://github.com/TilStehle/Cross-Platform-Traceability`
[26] Mobile Anwendungen für mehrere Plattformen - Portierung, Architektur- und Tool-Entwicklung (Masterprojekt 2016/17), vgl. Tabelle 9.4 auf Seite 193 dieser Arbeit

Die App ermöglicht es den Betreuern universitärer Übungsgruppen, den Fortschritt der Übungsteilnehmer zu dokumentieren. Die Betreuer tragen dazu die Erledigung einer Aufgabe in der App gemeinsam mit einer entsprechenden Bewertung ein. Auf dieser Basis bietet die App zu jedem Teilnehmer der Übung einen Überblick über den individuellen Bearbeitungsstand der Aufgaben. Die Ablage der Übungsergebnisse in einer zentralen Datenbank ermöglicht darüber hinaus statistische Auswertungen zum Lernfortschritt der Teilnehmer über alle Übungsgruppen hinweg. So können die verantwortlichen Lehrenden beispielsweise Rückschlüsse darüber ziehen, welche Themen häufig zu Schwierigkeiten bei den studierenden führen und diese Themen in der zugehörigen Vorlesung wiederholen.

Abbildung 8.12 stellt die Architektur der Ausgangsimplementation der App dar.

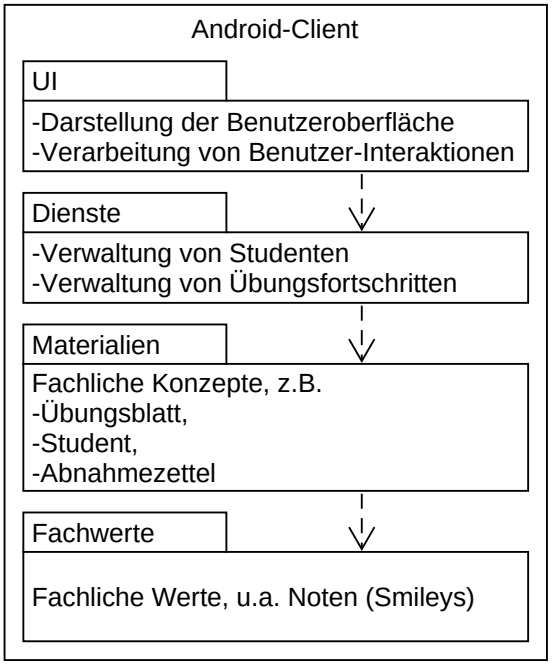

Abbildung 8.12: Architektur der Android-App *Se-Manager*

Grundlegende fachliche Werte wie Übungstermine und Smileys zur Bewertung von Übungsergebnissen werden von den Klassen des Pakets `Fachwerte` umgesetzt. Darauf aufbauend implementieren die Klassen im Paket `Materialien` fachliche Konzepte wie *Übungsblatt*, *Student* oder *Abnahmezettel*. Die Typdefinitionen im Paket `Dienste` verwalten die Studenten und ihren aktuellen Übungsfortschritt unter Zugriff auf einen Webservice. Mit diesem kommunizieren sie über eine REST-Schnittstelle. Im Paket `UI` wird eine Benutzerschittstelle realisiert, über die der Nutzer die App bedient.

Zur Portierung auf das Betriebssystem Windows Phone wurde ein Teil der Typdefinitionen automatisch in die Programmiersprache C# übersetzt. Dazu wurde eine im Projekt erweiterte Version von Sharpen auf 39 vollständig konvertierbare Typdefinitionen der Ausgangsimplementation angewendet. Diese Typdefinitionen sind gemeinsam mit ihren Übersetzungen und der Konfigurationsdatei zur Anwendung des Konverters auf der beiliegenden CD im Ordner `Materialien_Trace_Capture` zu finden.

Die im Projekt umgesetzte Erweiterung generiert bei der Übersetzung ein Traceability-Modell auf Basis des in Abschnitt 4.5.1 eingeführten Metamodells. Es wird in einer

XML-Datei abgelegt, sodass es bei der Co-Evolution der Ausgangs- und Zielimplementation automatisiert genutzt werden kann. Die erzeugte XML-Datei ist auf der beiliegenden CD ebenfalls im Ordner `Materialien_Trace_Capture` unter dem Namen `TraceabilityModel.xml` zu finden. Sie enthält 245 Trace Links, die die konvertierten Typdefinitionen sowie ihre Attribute und Operationen mit den generierten Übersetzungen verknüpfen. Die erzeugten Trace Links sind zur besseren Lesbarkeit zusätzlich in Tabelle A.5 im Anhang erfasst.

8.5.2 Bewertung der Ergebnisse

Um die generierten Trace Links bezüglich Korrektheit und Vollständigkeit zu bewerten, wurden sie durch einen manuellen Abgleich mit den tatsächlichen Entsprechungen zwischen Ein- und Ausgabe der Konversion geprüft. Diese Prüfung ergibt, dass die Trace Links vollständig und korrekt sind. Precision und Recall haben damit jeweils den Wert 1. Das Ergebnis kann durch den Abgleich der generierten Trace Links in Tabelle A.5 im Anhang mit den tatsächlichen Entsprechungsbeziehungen zwischen den Code-Elementen im Ordner `Materialien_Trace_Capture` der beiliegenden CD nachvollzogen werden. Dieses optimale Ergebnis ist wenig überraschend, da der für die Übersetzung zuständige Code-Konverter die entstehenden Entsprechungsbeziehungen unmittelbar bei der Konversion als Trace Links protokolliert.

8.5.3 Diskussion der Validität

Interne Validität
Es ist möglich, dass nicht alle Übersetzungsregeln des Konverters bei der Übersetzung des Beispiels mit nur 39 Typdefinitionen zum Einsatz kommen. Für diese Übersetzungsregeln besteht die Gefahr, dass sie noch nicht um die Generierung von Trace Links ergänzt wurden. Dies würde allerdings nicht zu einer grundsätzlich anderen Bewertung des Ansatzes führen, da es sich um eine unvollständige Implementation und nicht um einen Fehler im Ansatz handelte. Die Menge des übersetzten Quellcodes ist hier nicht ausschlaggebend, da die Regeln des erweiterten Konverters und damit auch die Erzeugung der Trace Links auch auf größere Codebasen in derselben Weise angewendet würden wie im untersuchten Fall.

Externe Validität
Grundsätzlich kann aus der prototypischen Erweiterung zweier Konvertoren nicht darauf geschlossen werden, dass alle Konvertoren in gleicher Weise erweiterbar sind. Beispielsweise können Konvertoren möglicherweise nicht um einen Trace-Capture-Mechanismus erweitert werden, wenn sie den eingegebenen Quellcode nicht unmittelbar

übersetzen, sondern als Eingabe ein Kompilat des zu übersetzenden Codes entgegennehmen und dieses übersetzen. Im zu übersetzenden Kompilat sind die ursprünglichen Code-Elemente gegebenenfalls nicht eindeutig identifizierbar, sodass nicht für alle Elemente des Ausgangscodes auch Trace Links generiert werden können. Für Konvertoren wie Sharpen und J2Swift, die unmittelbar den abstrakten Syntaxbaum der Eingabe verarbeiten, ist eine Erweiterung der Ausgabe um Traceability-Modelle möglich, wie hier demonstriert wurde. Für solche Erweiterungen sind auch die erzielten Werte für Precision und Recall repräsentativ, da die Zuordnung zwischen den Elementen der Eingabe und Ausgabe eindeutig in den Abbildungsregeln des Konverters definiert ist.

Konstruktvalidität
Die vom Trace-Capture-Mechanismus erzeugten Trace Links sollen von Entwicklern während der Co-Evolution genutzt werden. Dementsprechend waren die erzeugten Modelle anhand zweier Kriterien zu bewerten. Das erste Kriterium ist, wie häufig Entwickler bei der Nutzung der Trace Links zum korrekten Code-Element geführt werden. Dies wird durch die Metrik *Precision* erfasst, die hier angewendet wurde. Das zweite Kriterium ist, ob die korrekte Entsprechung überhaupt durch einen Trace Link dokumentiert wird. Ist dies nicht der Fall, so müssen Entwickler händisch nach einer Entsprechung suchen. Die Metrik *Recall* wurde dementsprechend eingesetzt, um zu messen, welcher Anteil der erzeugten impliziten Entsprechungsbeziehungen auch durch Trace Links dokumentiert wird.

8.6 Evaluation des Trace-Recovery-Mechanismus zur Erhebung plattform-übergreifender Trace Links

Der in dieser Arbeit entwickelte Trace-Recovery-Mechanismus wird eingesetzt, um Trace Links zu einem gegebenen Code-Element vorzuschlagen, die dieses mit seinen Entsprechungen in einer zweiten Codebasis verknüpfen. In Abschnitt 8.1 wurde beschlossen, die erzeugten Ergebnislisten anhand der Metrik Mean Average Precision zu bewerten. Eine ausführliche und mit einem Beispiel veranschaulichte Erläuterung dieser Metrik ist in Abschnitt 6.2.2 zu finden.

Um den Mechanismus aussagekräftig zu bewerten, waren Projekte auszuwählen, auf die er für eine aussagekräftige Evaluation angewendet wurde. Bei dieser Auswahl ist das angestrebte Einsatzgebiet des Mechanismus ausschlaggebend. Er soll Entsprechungsbeziehungen sowohl für händisch portierten Code als auch für automatisch übersetzten Code ermitteln, falls der eingesetzte Konverter nicht selbst Trace Links generiert. Der Mechanismus wurde daher sowohl anhand des Codes der Fallstudie *Metropole des Wissens* bewertet, in der hauptsächlich händische Übersetzungen vorgenommen wurden, als

auch anhand des Quelltextes aus der Fallstudie *DESMO-J*, der nahezu vollständig automatisch übersetzt wurde. Um den Einsatz auch unabhängig von der hier entwickelten Portierungsmethode zu untersuchen, wurde der Mechanismus darüber hinaus auf das Projekt Lucene.Net angewendet, das automatische und händische Übersetzungen mischt (siehe dazu Abschnitt 5.1.4).

Wie bereits bei der Optimierung der Bezeichnergewichte in Abschnitt 6.2 wurde zunächst für jedes der untersuchten Projekte ein Datensatz korrekter Trace Links manuell erhoben, der als Soll-Wert für den Vergleich der generierten Trace Links dient. Manning u. a. [2008, S. 152] geben als Faustregel an, dass die Bewertung von Ergebnislisten für 50 Informationsbedarfe hinreichend ist, um ein Suchverfahren aussagekräftig zu bewerten. Dementsprechend wurden je Projekt 50 Typdefinitionen einer Implementation ausgewählt und ihre Entsprechungen in der zweiten Implementation identifiziert. Um Klassen aus allen Schichten der Architektur in die Bewertung einzubeziehen, wurde bei der Auswahl der Typdefinitionen über alle Pakete der ersten Implementation iteriert und der alphabetisch nächste noch nicht aufgenommene Vertreter des Paketes gewählt bis die Zahl von 50 Typdefinitionen erreicht war. Zu diesen wurden dann sämtliche Entsprechungen in der zweiten Implementation identifiziert. Die Listen der so erhobenen Entsprechungen sind in den Tabellen A.2, A.3 und A.4 im Anhang zu finden.

Der Trace-Recovery-Mechanismus wurde anschließend eingesetzt, um zu jeder der manuell zugeordneten Typdefinitionen eine Liste von Trace Links vorzuschlagen. Für jedes der Projekte wurden die so erzeugten Ergebnislisten anhand der Metrik Mean Average Precision bewertet. Tabelle 8.3 erfasst die in diesen Untersuchungen beobachteten MAP-Werte.

Projekt	MAP mit Gewichtung der Bezeichner gemäß Abschnitt 6.2	MAP ohne Gewichtung der Bezeichner	Differenz der MAP-Werte	t	$\|t\| > t(0,95,49)$
DESMO-J	0,95	0,817	0,133	-3.792	✓
Metropole des Wissens	0,9	0,846	0,054	-2.796	✓
Lucene.Net	0,86	0,82	0,04	-1.445	

Tabelle 8.3: Gegenüberstellung der Mean Average Precision für den Trace-Recovery-Mechanismus mit und ohne Gewichtung der Bezeichner

Das jeweilige Projekt ist in der ersten Spalte der Tabelle genannt. Die Werte der erzielten Mean Average Precision sind in der zweiten Spalte aufgeführt. Die Werte zwischen 0,86 und 0,95 sind so zu interpretieren, dass ein Entwickler, der die erzeugten Ergebnislisten zur Suche nach einer konkreten Entsprechung nutzt, in den untersuchten Projekten

durchschnittlich nur fünf bis 14% falscher Vorschläge prüft, bis er die für ihn relevante Entsprechung findet. Es ist anzunehmen, dass ein mit dem jeweiligen Projekt vertrauter Entwickler in der Regel noch weniger falsche Vorschläge tatsächlich verfolgt und prüft, da er einige der Ergebnisse anhand ihres Namens bereits als inkorrekte oder für ihn nicht relevante Ergebnisse erkennt und sie überspringt.

Um diese Ergebnisse qualitativ einordnen zu können, ist ein Vergleich mit alternativen Mechanismen sinnvoll. Einen vergleichbaren Trace-Recovery-Mechanismus gibt es allerdings nicht. Zwar gibt es sprachunabhängige Mechanismen zur Suche nach Paaren ähnlicher Code-Elemente. Diese sind aber für einen vergleichenden Test entweder nicht verfügbar und wurden bislang nur zur Suche nach Klonen innerhalb derselben Programmiersprache eingesetzt [Mishne u. De Rijke 2004], sie sind nicht auf ganze Typdefinitionen anwendbar [Flores u. a. 2012] oder können die Suche nur auf festgelegten Quellen im Internet durchführen [Byalik u. a. 2015]. Damit ist das hier entwickelte Werkzeug zur Suche nach Trace Links das einzige im Kontext der Portierung einsetzbare.

Um dennoch eine aussagekräftige Vergleichsbasis zu schaffen, wurde eine Version des hier entwickelten Mechanismus implementiert, die die erhobenen Bezeichner nicht anhand der Position im abstrakten Syntaxbaum gewichtet. Dies ist auch bei anderen sprachunabhängigen Vergleichsmechanismen der Fall [Flores u. a. 2012] [Byalik u. a. 2015]. Diese abgewandelte Version wurde zum Vergleich ebenfalls auf die gewählten Projekte angewendet. Die erzielten Ergebnisse ohne Gewichtung der Bezeichner sind in der dritten Spalte von Tabelle 8.3 erfasst. Die Verbesserung der Mean Average Precision durch die Anwendung der in Abschnitt 6.2 bestimmten Bezeichnergewichte ist in der vierten Spalte aufgeführt. Zu sehen ist, dass die Anwendung der dort bestimmten Gewichte für alle drei Stichproben zu einer Erhöhung der Mean Average Precision führt, dass sich die Sortierung der Ergebnislisten also verbessert.

Es ist zu prüfen, ob die beobachteten Verbesserungen statistisch signifikant sind. Die Mean Average Precision bildet, wie in Abschnitt 6.2.2 erläutert, den Durchschnitt über die Average Precisions der Ergebnislisten. Bei statistischer Signifikanz kann also allgemein die Aussage getroffen werden, dass der Erwartungswert der Average Precision einer Ergebnisliste im jeweiligen Projekt durch die Anwendung der Bezeichnergewichte steigt. Um dies zu prüfen, wurde die Average Precision jeder generierten Ergebnisliste mit Anwendung der Bezeichnergewichte der entsprechenden Average Precision ohne Anwendung der Bezeichnergewichte zugeordnet. So ergibt sich für jedes der drei Projekte eine paarweise verbundene Stichprobe mit 50 Wertepaaren. Diese Wertepaare enthalten jeweils zu einer Suchanfrage die Average Precisions beider Ergebnislisten, die der Mechanismus mit und ohne Gewichtung der Bezeichner erzeugt. Um zu prüfen, ob der Erwartungswert der Average Precision im jeweiligen Projekt tatsächlich durch die Anwendung der Bezeichnergewichte steigt, wurde der Zweistichproben-t-Test für abhängige Stichproben auf diese Paare angewendet [Cleff 2019, S. 157]. Dieser Test ist hier an-

wendbar, da die Stichprobengröße $n = 50$ größer als 30 ist und die untersuchte Variable metrisch skaliert ist [Cleff 2019, S. 144].

Es wird folgende Nullhypothese aufgestellt:

Der Erwartungswert der Average Precision für das Verfahren ohne Gewichtung der Bezeichner ist mindestens so hoch wie der Erwartungswert des Verfahrens mit Gewichtung der Bezeichner. Die Gewichtung verbessert die Ergebnisse der Suche also nicht:

$$E(AveragePrecision_{ohneGewichtung}) \geq E(AveragePrecision_{mitGewichtung})$$

Bei einer zulässigen Irrtumswahrscheinlichkeit $\alpha = 5\%$ und einer Stichprobengröße $n = 50$ ist die Nullhypothese dann abzulehnen, wenn der Betrag $|t|$ des berechneten t-Wertes das 95%-Quantil der t-Verteilung überschreitet. Für $n = 50$ beträgt dieser kritische Wert $t(0,95,49) = 1,68$. Der t-Wert für das jeweilige Projekt ist in der fünften Spalte von Tabelle 8.3 erfasst. In der letzten Spalte ist ein Haken verzeichnet, wenn die Nullhypothese für das jeweilige Projekt abzulehnen ist und damit die Signifikanz der Verbesserung festgestellt wurde. Dies ist für die Projekte *DESMO-J* und *Metropole des Wissens* der Fall. Für das Projekt *Lucene.Net* ist die Verbesserung nicht signifikant.

Antwortzeit
Im Zuge der oben beschriebenen Experimente wurden die Antwortzeiten des Mechanismus bei seinem Einsatz erhoben. Abbildung 8.13 skizziert die Verteilung der Antwortzeiten für die jeweils 50 Anfragen in den Projekten Metropole des Wissens, Lucene.Net und DESMO-J in Form von Boxplots.

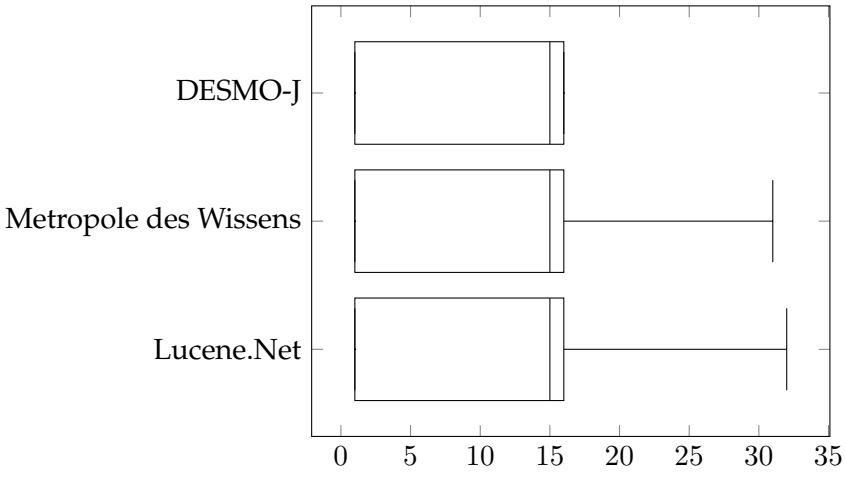

Abbildung 8.13: Boxplots für die Verteilungen der Antwortzeiten des Trace-Recovery-Mechanismus für die jeweils 50 Anfragen in den Projekten *DESMO-J*, *Metropole des Wissens* und *Lucene.Net*

Zu sehen ist, dass die Antwortzeit des Mechanismus für die drei Projekte ähnlich verteilt war, obgleich eine ganz unterschiedliche Anzahl von Typdefinitionen mit den Anfragen

zu vergleichen war. Während der Mechanismus bei Anfragen im Projekt Metropole des Wissens 76 Typdefinitionen mit der Anfrage vergleichen musste, waren in den Projekten DESMO-J und Lucene.Net 1832 bzw. 2039 Vergleiche notwendig. Ungeachtet dessen lag in allen drei Fällen der Median bei 15 Millisekunden und drei Viertel der Anfragen wurden in maximal 16 Millisekunden beantwortet. Lediglich der Maximalwert der Anfragen liegt bei den Projekten DESMO-J und Lucene.Net mit 31 bzw. 32 Millisekunden höher als im Projekt Metropole des Wissens. Das zu Beginn dieses Kapitels aufgestellte Maximum von einer Sekunde für die Antwortzeit unterschritt der Mechanismus in den Experimenten damit deutlich[27].

8.6.1 Diskussion der Validität

Interne Validität
Grundsätzlich können bei der manuellen Erhebung der korrekten Trace Links Fehler gemacht worden sein. Dies würde die gemessenen Werte der Mean Average Precision verfälschen. Um diese Gefahr zu mindern, wurden die Trace Links manuell durch eine zweite Wissenschaftlerin geprüft. Die Antwortzeit des Mechanismus kann darüber hinaus durch andere Prozesse beeinflusst worden sein, die gleichzeitig auf dem Rechner durchgeführt wurden. Dies hat sich jedoch, falls überhaupt, stets zu Ungunsten des Mechanismus ausgewirkt, sodass die beobachteten Maximalwerte gültig sind.

Externe Validität
Die Signifikanz der Verbesserung durch die Anwendung von Bezeichnergewichten ist ausschließlich in Bezug auf die drei untersuchten Projekte *DESMO-J*, *Lucene.Net* und *Metropole des Wissens* geprüft worden. Die Rahmenbedingungen anderer Projekte können sich von denen der untersuchten Projekte unterscheiden. Insbesondere die Kombination von ursprünglicher Programmiersprache und Zielsprache sowie projektspezifische Quelltextkonventionen haben einen Einfluss auf die Wahl und die Häufigkeiten von Bezeichnern im Quelltext. Dementsprechend ist anzunehmen, dass die Bezeichnergewichte nicht für jedes Projekt optimal sind. Dennoch sprechen die hohen beobachteten MAP-Werte dafür, dass der Trace-Recovery-Mechanismus mit den in Abschnitt 6.2 bestimmten Gewichten effektiv ist.

Konstruktvalidität
Entwickler setzen im Rahmen der Co-Evolution den untersuchten Trace-Recovery-Mechanismus als Suchmaschine ein. Als solche sollte der Mechanismus möglichst alle korrekten Entsprechungen auffinden und diese in der Ordnung der Ergebnisse möglichst weit vorne einsortieren. Die absolute Länge der Ergebnislisten ist dabei weniger relevant, da der Entwickler aufhört, die Ergebnisse zu durchsuchen, sobald er die für ihn

[27] Die Experimente wurden auf einem Notebook mit 16 GB Arbeitsspeicher und einem Prozessor des Typs Intel Core i7-3740QM durchgeführt.

relevante Entsprechung identifiziert hat. Durch die Bildung von Teillisten, die mit einem korrekten Trace Link abschließen, bewertet MAP den Mechanismus genau entsprechend dieses Verhaltens (vgl. hierzu Abschnitt 6.2.2). Einfachere Metriken wie Precision und Recall sind dagegen ungeeignet, da sie die Sortierung der Ergebnisse vernachlässigen.

Die gemessene Antwortzeit ist relevant, da der Einsatz des Mechanismus den Gedankenfluss des Entwicklers nicht durch zu lange Wartezeiten unterbrechen sollte [Nielsen 1993, S. 135]. Beim Einsatz des Mechanismus in Entwicklungswerkzeugen kommt zusätzlicher Rechenaufwand für die Darstellung der Ergebnisse hinzu, der im Experiment nicht gemessen wurde. Bis zum gesetzten Maximum der Antwortzeit bleiben für diese Darstellung allerdings selbst im schlechtesten Falle noch 968 Millisekunden. Es ist nicht anzunehmen, dass dieser Wert überschritten wird.

8.7 Demonstration und Bewertung prototypischer Werkzeuge für die plattform-übergreifende Co-Evolution am Fallbeispiel der portierten mobilen App *Metropole des Wissens*

8.7.1 Werkzeug zur Navigation entlang automatisch erhobener Trace Links

In Abschnitt 7.1 wurde ein Ansatz zur Unterstützung der plattform-übergreifenden Concept Location aufgezeigt. Er basiert auf der Nutzung plattform-übergreifender Trace Links zur Navigation zwischen einander entsprechenden Quellcode-Elementen. Um seine Realisierbarkeit zu zeigen, wurde ein Plugin für die Entwicklungsumgebung IntelliJ IDEA entwickelt. Es erlaubt Entwicklern von Quelltexten in den Programmiersprachen Java und Kotlin, ausgehend von einer Methode, einem Attribut oder einer Typdefinition, direkt zu den Entsprechungen in einer verknüpften Implementation zu navigieren. Handelt es sich bei den Zielen der Navigation um Code-Elemente in Swift, so wird die Entwicklungsumgebung Xcode geöffnet und das Code-Element wird im Kontext seines Xcode-Projektes zur Bearbeitung geöffnet. Das Plugin wurde initial in einem Projekt und einer studentischen Studie unter Betreuung des Autors entwickelt und im Rahmen dieser Arbeit weiterentwickelt. Es basiert auf dem in Abschnitt 6.3 entwickelten Werkzeug zur Suche nach plattform-übergreifenden Trace Links zwischen einander entsprechenden Code-Elementen. Der Code des Plugins wurde unter der MIT-Lizenz veröffentlicht[28].

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

Exemplarische Demonstration

Das Plugin wird hier im Kontext der Fallstudie *Metropole des Wissens* demonstriert, die in Abschnitt 8.4.3 beschrieben wurde. Als exemplarische Anforderung soll nach Abschluss der Portierung das Modell von Veranstaltungen um eine maximale Teilnehmerzahl erweitert werden, die den Nutzern der App angezeigt wird. Diese Information soll sowohl bei der Darstellung in der Veranstaltungsübersicht als auch in der Detailansicht einer Veranstaltung zu sehen sein. Abbildung 8.14 zeigt die Detailansicht von *Metropole des Wissens* in der entsprechend erweiterten Version mit Anzeige der maximalen Teilnehmerzahl.

Abbildung 8.14: Bildschirmaufnahme der iOS-Applikation *Metropole des Wissens*, abgewandelt von [Berger 2019, S. 74f.]

Der Entwickler beginnt seine Arbeit in der Entwicklungsumgebung Android Studio, die er für die Android-Implementation der App nutzt. Dort konfiguriert er das Plugin, indem er den Pfad angibt, unter dem der Code der entsprechenden iOS-Implementation gespeichert ist. Um die Anforderung umzusetzen, muss der Entwickler nun zunächst den zu ändernden Quelltext identifizieren. Dazu prüft er die Namen der Klassen, die Teil der Codebasis sind und stößt auf die Klasse `EventModel`, die die zu erweiternden Veranstaltungen repräsentiert. Damit hat er die Aufgabe der Concept Location für die Android-Implementation abgeschlossen.

Ausgehend von diesem Punkt kann der Entwickler das Plugin nutzen, um nach Entsprechungen in der iOS-Implementation zu suchen. Dazu setzt er den Cursor auf den Namen der Klasse und führt die Tastenkombination Cmd + Alt + U aus, oder wählt aus

dem Navigationsmenü die Option *Show corresponding elements in linked implementation*. Das Plugin zeigt daraufhin das in Abbildung 8.15 zu sehende Popup-Fenster an.

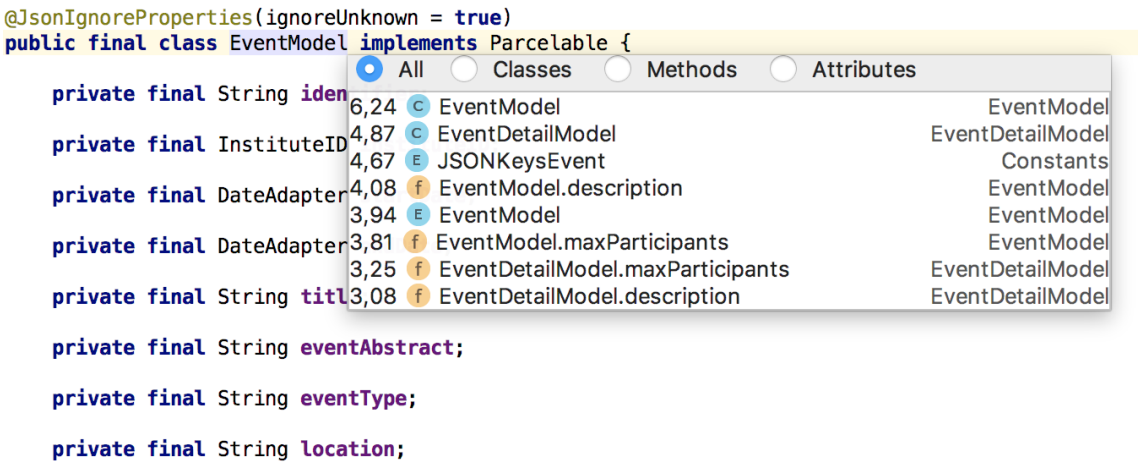

Abbildung 8.15: Präsentation der Vorschläge für Entsprechungen zur Klasse `EventModel` in der iOS-Implementation von *Metropole des Wissens*

Es enthält eine sortierte Ergebnisliste mit potenziellen Entsprechungen zur Klasse EventModel. In der Kopfzeile des Fensters kann der Entwickler die Vorschläge nach der Art der verknüpften Elemente filtern und so ausschließlich Klassen, Methoden oder Attribute als Suchergebnisse anzeigen lassen. Da das gleichnamige Struct `EventModel` der iOS-Implementation als erstes Suchergebnis aufgeführt wird, ist dies hier nicht erforderlich. Der Entwickler wählt den Eintrag `EventModel` aus, da er die naheliegende Vermutung hat, dass es sich um die korrekte Entsprechung zur Java-Klasse `EventModel` handelt.

Das Plugin öffnet daraufhin das verknüpfte Projekt in der Entwicklungsumgebung Xcode und wechselt zum Swift-Code des Structs `EventModel`. Dies geschieht mittels eines Skripts in der Sprache AppleScript, das als virtueller Benutzer mit dem Betriebssystem interagiert und Mausklicks und Tastatureingaben automatisch vornimmt. Es automatisiert die Navigation zur Datei, in der die iOS-Version von `EventModel` definiert ist und setzt den Cursor dort auf den Klassennamen. Ein Bildschirmvideo des hier beschriebenen Ablaufs ist auf der beiliegenden CD unter `Bildschirmvideos/Navigation.mov` und online[29] zu finden. Nachdem der Entwickler das Struct `EventModel` in Augenschein genommen hat, erkennt er, dass es sich tatsächlich um die passende Entsprechung handelt und die Erweiterung von Veranstaltungen auch hier vorzunehmen ist. Auf die gleiche Art und Weise identifiziert der Entwickler auch die Klasse `EventDetailModel` und ihre Entsprechung in der iOS-Implementation als zu ändernde Code-Elemente. Damit schließt der die Phase der plattformübergreifenden Concept Location ab.

[29] `https://swk-www.informatik.uni-hamburg.de/~stehle/Navigation.mov`

Bewertung des Ansatzes

Ohne das Vorhandensein plattform-übergreifender Trace Links hätte der Entwickler manuell nach dem Quellcode suchen müssen, der in der iOS-Implementation die Veranstaltungen modelliert. Die Anwendung der hier entwickelten Portierungsmethode hat bereits dazu geführt, dass dieses Konzept strukturgleich in einer gleichnamigen Klasse umgesetzt wurde. Das Auffinden der entsprechenden Klasse hätte also auch ohne explizite Trace Links nur wenige Sekunden gedauert. Die Nutzung des Trace Links beschleunigt diesen Vorgang allerdings, da die passende Entwicklungsumgebung nicht manuell geöffnet und die Klasse `EventModel` nicht im Projekt gesucht werden muss.

8.7.2 Werkzeug für die plattform-übergreifende Impact-Analyse

In Abschnitt 7.2 wurde ein Ansatz zur Nutzung von Trace Links für die plattform-übergreifende Impact-Analyse vorgestellt. Der Ansatz wurde im Rahmen einer Masterarbeit als Plugin für die Entwicklungsumgebung IntelliJ IDEA bzw. Android Studio umgesetzt [Berger 2019]. Es ermöglicht Entwicklern, die Fortpflanzung einer Änderung auf weitere Typdefinitionen in der ursprünglichen und der portierten Implementation anhand eines plattform-übergreifenden Abhängigkeitsgraphen zu identifizieren. Es wurde im Rahmen der erwähnten Masterarbeit eingesetzt, um, ausgehend von den Java-Klassen `EventModel` und `EventDetailModel`, die Auswirkung der oben beschriebenen Erweiterung der App *Metropole des Wissens* zu analysieren. Dieses Fallbeispiel wird im Folgenden wiedergegeben.

Exemplarische Demonstration

Der Entwickler sucht zunächst im Quelltext nach Typdefinitionen, die die Veranstaltungsinformationen enthalten und identifiziert somit `EventModel` und `EventDetail-Model` als zu erweiternde Klassen. Um die Impact-Analyse zu starten, markiert er beide Klassen über ein Kontextmenü als *Changed* und öffnet die Ansicht des Analyse-Plugins. Diese zeigt die beiden betroffenen Klassen und all jene Typdefinitionen, die durch Abhängigkeiten wie Nutzung, Vererbung oder Deklaration eines Rückgabetyps mit ihnen verbunden sind. Abbildung 8.16 zeigt die Darstellung der beiden Klassen im Plugin.

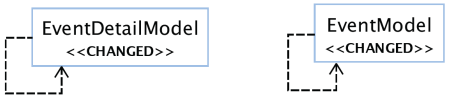

Abbildung 8.16: Darstellung der zu ändernden Klassen `EventModel` und `EventDetailModel` im Plugin für die plattform-übergreifende Impact-Analyse entnommen aus [Berger 2019, S. 77]

Zur Verbesserung der Übersichtlichkeit wurden ihre 30 direkten Nachbarn ausgeblendet. Letztere sind im Abhängigkeitsgraphen mit dem Schlüsselwort *Next* als zu untersuchende Typdefinitionen gekennzeichnet. Um die Übersicht über den Graphen zu verbessern, kann der Entwickler sämtliche Klassen ausblenden, die nicht Teil des Entwicklungsprojekts sind, sondern in externen Bibliotheken definiert sind. Im Anschluss daran untersucht der Entwickler die übrigen Nachbarn von `EventModel` und `EventDetailModel`. Darunter befinden sich bereits sämtliche zu ändernden Typdefinitionen der Android-Implementation. Der zugehörige Abhängigkeitsgraph ist in Abbildung 8.17 zu sehen. Auch hier sind die weiteren Nachbarn der markierten Klassen zu Zwecken der Übersichtlichkeit ausgeblendet.

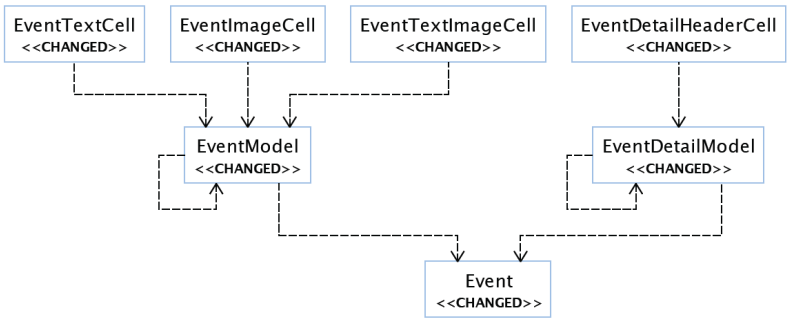

Abbildung 8.17: Darstellung des Abhängigkeitsgraphen für die zu ändernden Klassen der Android-Implementaion, entnommen aus [Berger 2019, S. 78]

Nachdem der Entwickler die Impact-Analyse für die Android-Implementation abgeschlossen hat, wählt er nacheinander jeweils eine der identifizierten Klassen im Abhängigkeitsgraphen an und wählt im Kontextmenü die Option *Show corresponding element in connected codebase*. Daraufhin stößt das Plugin eine Suche nach Entsprechungen in der iOS-Implementation der App an, die durch das in Abschnitt 6.3 beschriebene Trace-Recovery-Framework durchgeführt wird. Der Entwickler prüft die Vorschläge und selektiert jeweils die passende Entsprechung. Im konkreten Beispiel ist dies sehr einfach, da die korrespondierenden Swift-Klassen bis auf eine Ausnahme die gleichen Namen tragen wie die zu ändernden Java-Klassen. Der Mechanismus schlägt für alle Klassen bis auf `EventModel` die passende Entsprechung als erstes Suchergebnis vor. Wählt der Entwickler eine verknüpfte Typdefinition aus, so wird diese in den Abhängigkeitsgraphen mit einer roten Linie eingefügt, die die Entsprechungsbeziehung darstellt. Auf diese Weise identifiziert der Entwickler sämtliche betroffenen Entsprechungen in der iOS-Implementation. Im Anschluss setzt er die Impact-Analyse für die iOS-Implementation fort, um etwaige plattformspezifische Änderungsbedarfe auszumachen. In diesem Beispiel beschränkt sich dieser Schritt darauf, dass er lediglich die direkten Nachbarn der betroffenen Klassen als nicht von der Änderung betroffen markiert. Abbildung 8.18 zeigt den im Plugin dargestellten Abhängigkeitsgraphen nach Abschluss dieses Vorgangs.

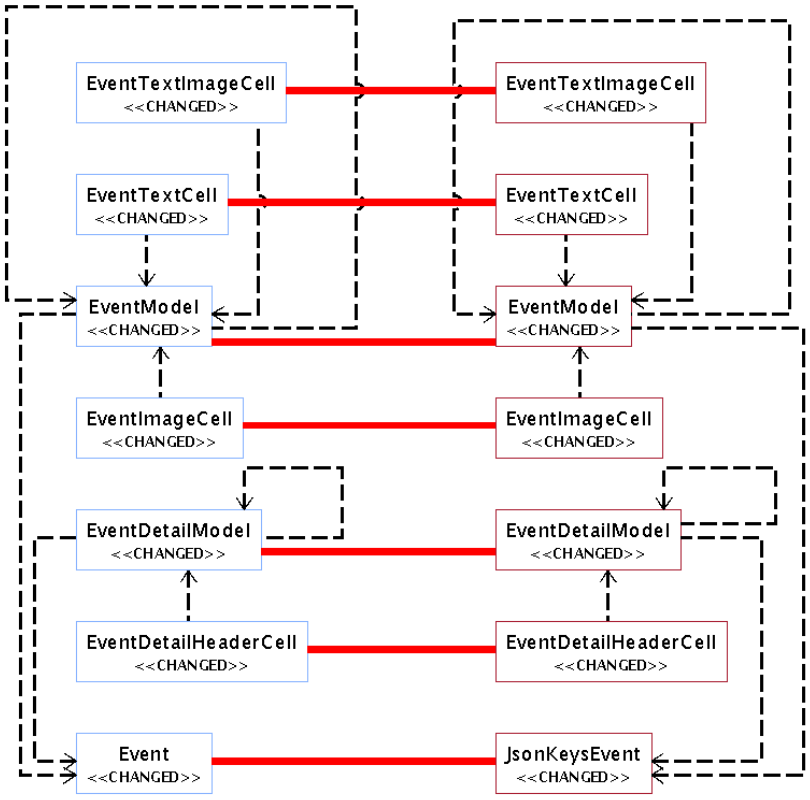

Abbildung 8.18: Darstellung des plattform-übergreifenden Abhängigkeitsgraphen im Plugin für die plattform-übergreifende Impact-Analyse, entnommen aus [Berger 2019, S. 79]

Er enthält sämtliche zu ändernden Klassen der Android- und iOS-Implementation der App. Um dieses Ergebnis der Impact-Analyse zu prüfen, wurde die Änderung von einem zweiten Entwickler ohne Kenntnis der Analyse-Ergebnisse umgesetzt. Die notwendigen Änderungen sind in Tabelle A.6 im Anhang erfasst. Ein Abgleich der tatsächlich geänderten Klassen mit dem Ergebnis der Impact-Analyse ergibt keine Abweichungen.

Bewertung des Ansatzes

Das oben beschriebene Vorgehen erspart es dem Entwickler, die Impact-Analyse in der zweiten Implementation mit der manuellen Concept Location zu beginnen. Stattdessen kann er ausgehend von den betroffenen Elementen der zuerst untersuchten Implementation anhand der vorgeschlagenen Trace Links auf deren ebenfalls zu ändernde Entsprechungen in der zweiten Implementation schließen. Dies wäre auch ohne das Vorhandensein von Trace Links mit nur wenig Aufwand verbunden, da bereits der Klassenentwurf durch die Anwendungen der Portierungsmethode übertragen wurde. Dadurch muss die fast immer gleichnamige Entsprechung einer zu ändernden Klasse auch in der zweiten Implementation geändert werden.

Impact-Analysen werden unter anderem zum Schätzen des Änderungsaufwands genutzt [Tanveer 2017]. Ist den Entwicklern in dieser Situation bekannt, dass die Implementationen einander konzeptuell sehr ähnlich sind, so wie im beschriebenen Beispiel, dann kann die Impact-Analyse auf eine der Implementationen beschränkt werden. Die Schätzung für diese eine untersuchte Implementation kann dann genutzt werden, um den Gesamtaufwand für die plattform-übergreifende Änderung abzuleiten. Somit wird durch die plattform-übergreifende Impact-Analyse durch die gezielte Übertragung der ursprünglichen Konzepte auf die Zielimplementation erheblich erleichtert.

Unabhängig vom eingesparten Aufwand ist der Vorteil des Ansatzes, dass eine plattformübergreifende Übersicht über die betroffenen Elemente in einem gemeinsamen Abhängigkeitsgraphen ermöglicht wird. Dies ermöglicht unabhängig von der Aufgabe der Impact-Analyse, dass Entwickler die Strukturen der Implementationen visuell gegenüberstellen können. Unterschiede zwischen den Entwürfen offenbaren sich somit durch den Einsatz des Plugins bei der Impact-Analyse. Dadurch können angleichende Restrukturierungen vor der eigentlichen Durchführung der Änderung geplant und vorgenommen werden. Das erarbeitete Änderungskonzept kann dann in beiden Implementationen gleichartig umgesetzt werden.

8.7.3 Werkzeug zur Koordination von Änderungen anhand von TODO-Kommentaren

In Abschnitt 7.4 wurde ein Ansatz zur Koordination von Änderungen eingeführt, dessen Anwendung hier demonstriert wird. Er sieht vor, dass Entwickler nach der Änderung eines Code-Elements dessen Entsprechung mit einem Kommentar versehen, der die gleichartig anzupassende Stelle markiert und die notwendige Änderung beschreibt. Ein solcher Kommentar kann gegebenenfalls auch den zuerst geänderten Quelltext als Vorlage für die Umsetzung der entsprechenden Änderung enthalten. Die Nutzung dieser Vorlage erleichtert es dem Entwickler, die Änderung der zweiten Implementation konzeptuell und funktional konsistent zur Änderung an der ersten Implementation vorzunehmen.

Das in Abschnitt 8.7.1 beschriebene Plugin für IntelliJ IDEA bzw. Android Studio wurde dementsprechend so erweitert, dass es die Übertragung von ToDo-Kommentaren ermöglicht. Diese Funktionalität wird hier exemplarisch im Kontext der oben beschriebenen Erweiterung an der App *Metropole des Wissens* eingesetzt.

Exemplarische Demonstration

Bei der Impact-Analyse wurde erkannt, dass die Klasse `JSONKeys.Event` der Android-Implementation und das entsprechende Enum `JSONKeysEvent` der iOS-Implementation

zu ändern sind. Sie enthalten jeweils die Namen aller Attribute der JSON-Darstellung von Veranstaltungen als *String*-Konstanten. Sie müssen jeweils um eine zusätzliche Konstante erweitert werden, die den Namen des Attributs für die maximale Teilnehmerzahl enthält. Der im folgenden beschriebene Ablauf stellt dar, wie der Entwickler der Android-Implementation mit dem Plugin interagiert, um die Änderung zu koordinieren. Ein Bildschirmvideo dieses Ablaufs ist auf der beiliegenden CD im Ordner `Bildschirmvideos/CommentsForLinkedImplementation.mov` und online[30] zu finden.

Zunächst fügt der Entwickler der Klasse `JSONKeys.Event` die Konstante `maxParticipants` hinzu, die in Codebeispiel 8.2 zu sehen ist.

```
public static final String maxParticipants = "maxParticipants";
```

Codebeispiel 8.2: Die Konstante `maxParticipants` im Java-Quelltext

Anschließend möchte er einen Kommentar in der zugehörigen iOS-Entsprechung hinterlassen, der dokumentiert, wie sie konsistent zu ändern ist. Dazu setzt er den Cursor auf den Namen der Klasse und führt die Tastenkombination Cmd + Alt + D aus. Das Plugin sucht daraufhin nach Entsprechungen der Klasse `JSONKeys.Event` in der iOS-Implementation und öffnet das in Abbildung 8.19 zu sehende Popup-Fenster, aus dem eine Entsprechung gewählt werden kann.

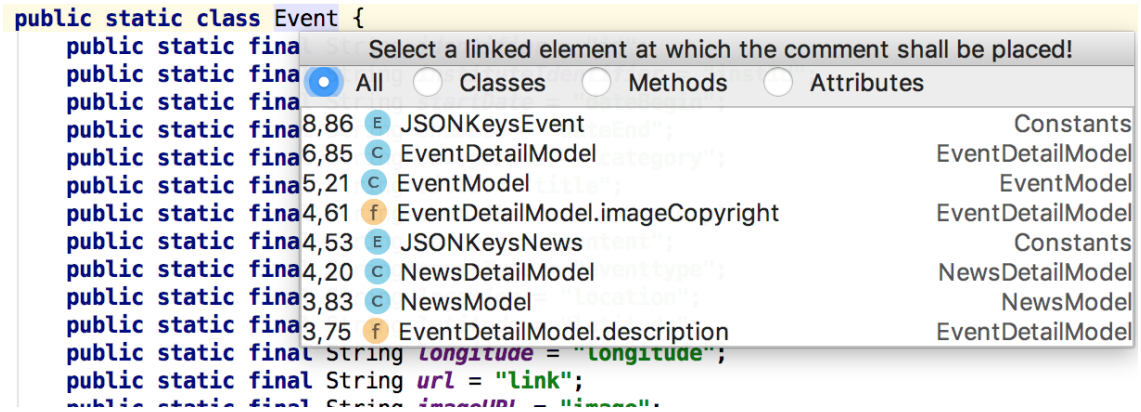

Abbildung 8.19: Initiation des plattform-übergreifenden Rename-Refactorings in Android Studio

Der Entwickler wählt in diesem Dialog das Enum `JSONKeysEvent`, das als erstes Ergebnis angezeigt wird. Er wird daraufhin aufgefordert, einen Text für den zu erstellenden ToDo-Kommentar zu verfassen. Dazu öffnet sich der in Abbildung 8.20 dargestellte Dialog.

[30] `https://swk-www.informatik.uni-hamburg.de/~stehle/CommentsForLinkedImplementation.mov`

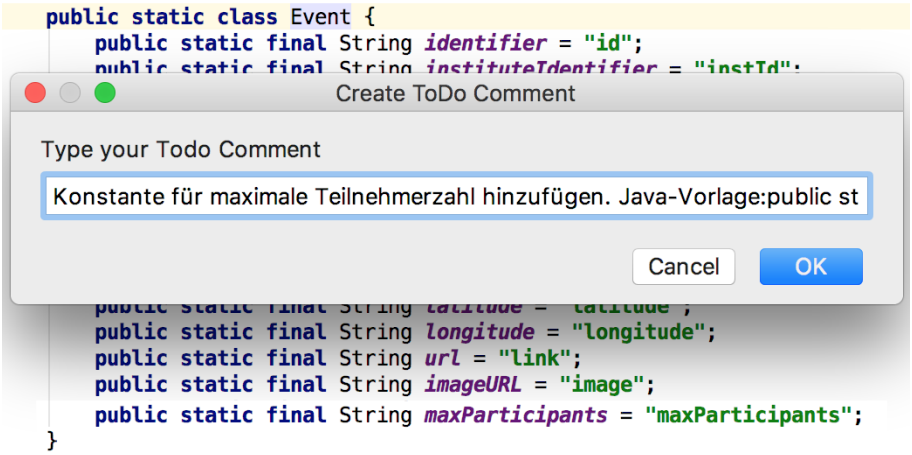

Abbildung 8.20: Eingabe eines ToDo Kommentars, der am Enum `JSONKeysEvent` hinterlassen werden soll

Da er den Java-Code der angelegten Konstante als Vorlage für die entsprechende Anpassung der iOS-Implementation zur Verfügung stellen möchte, formuliert er den Kommentar wie folgt: *„Konstante für maximale Teilnehmerzahl hinzufügen. Java-Vorlage: `public static final String maxParticipants = "maxParticipants";` “*. Das Plugin erstellt daraufhin einen ToDo-Kommentar im Quellcode des gewählten Swift-Enums `JSONKeysEvent`.

Der Entwickler der iOS-Implementation kann nun den ToDo-Kommentar nutzen, um die Erweiterung äquivalent im Swift-Quelltext vorzunehmen. Aus dem Kommentar ist ersichtlich, wie das Attribut in der JSON-Repräsentation benannt ist. Darüber hinaus ist aus dem Kommentar ersichtlich, welchen Namen die zusätzliche Konstante in der Android-Implementation trägt, sodass der iOS-Entwickler denselben Namen verwendet und damit die Konsistenz zwischen beiden Implementationen bewahrt.

Bewertung des Ansatzes

Das oben beschriebene Vorgehen ist immer dann anwendbar, wenn ein zu änderndes Code-Element und seine Entsprechung dasselbe Konzept umsetzen. Bei der Umsetzung der Änderung in der zweiten Implementation entfallen durch dieses Vorgehen aufwendige Schritte: Der Entwickler der zweiten Implementation muss die anzupassende Stelle nicht manuell im Quellcode identifizieren. Er muss kein eigenes Konzept für die Änderung entwerfen, wenn der Kommentar die notwendige Änderung beschreibt. Ferner kann er die Vorlage im Kommentar nutzen, um seine Änderung daraus abzuleiten. Infolge des Wegfalls dieser Schritte ist anzunehmen, dass der Aufwand für die konsistente Änderung sinkt und dass Entwickler mit höherer Wahrscheinlichkeit die konzeptuelle Konsistenz zwischen Ausgangs- und Zielimplementation bewahren.

8.7.4 Werkzeug für die plattform-übergreifende Durchführung von Rename-Refactorings

In Abschnitt 7.3 wurde ein Ansatz zur plattform-übergreifenden Durchführung von Refactorings vorgestellt. Er beschreibt die teilautomatisierte Durchführung von Refactorings auf Basis der Trace Links, die in Phase 2 der Portierungsmethode erhoben wurden. Um zu zeigen, dass diese automatische Unterstützung realisierbar ist, wurde sie im Rahmen dieser Arbeit in einem prototypischen Werkzeug für Entwickler realisiert. Es ermöglicht die plattform-übergreifende Durchführung der Refactorings *Rename Method* bzw. *Rename Type* für Methoden oder Typen, die in Java programmiert wurden und eine Entsprechung in einer verbundenen Swift-Implementation haben. Das Werkzeug wurde auf Basis des in Abschnitt 6.3 eingeführten Trace-Recovery-Frameworks realisiert und ist Teil des IntelliJ-Plugins, das im vorherigen Abschnitt 8.7.1 vorgestellt wurde.

Es wird zur Demonstration hier auf den Code der Fallstudie *Metropole des Wissens* angewendet, um dort eine Klasse der Android-Implementation gemeinsam mit ihrer Entsprechung in der iOS-Implementation konsistent umzubenennen.

Exemplarische Demonstration

Die Klasse `MapFeed` soll plattform-übergreifend umbenannt werden. Ihr Zweck ist die Speicherung einer Liste von Instituten, die die App auf einem Stadtplan darstellt. Unabhängig von ihrer Verwendung hat sie keinen Bezug zu Stadtplänen, sodass der Klassenname `MapFeed` nicht angemessen erscheint. Sie soll daher in `InstitutesList` umbenannt werden. Dazu führt der Entwickler einen Rechtsklick auf dem Klassennamen aus und wählt aus dem Kontextmenü die Option *Refactor → Rename element and linked counterparts*. Alternativ kann ein Tastenkürzel für dieses Refactoring definiert werden. Das Werkzeug fragt nun nach dem gewünschten neuen Namen der Klasse, wie in Abbildung 8.21 zu sehen.

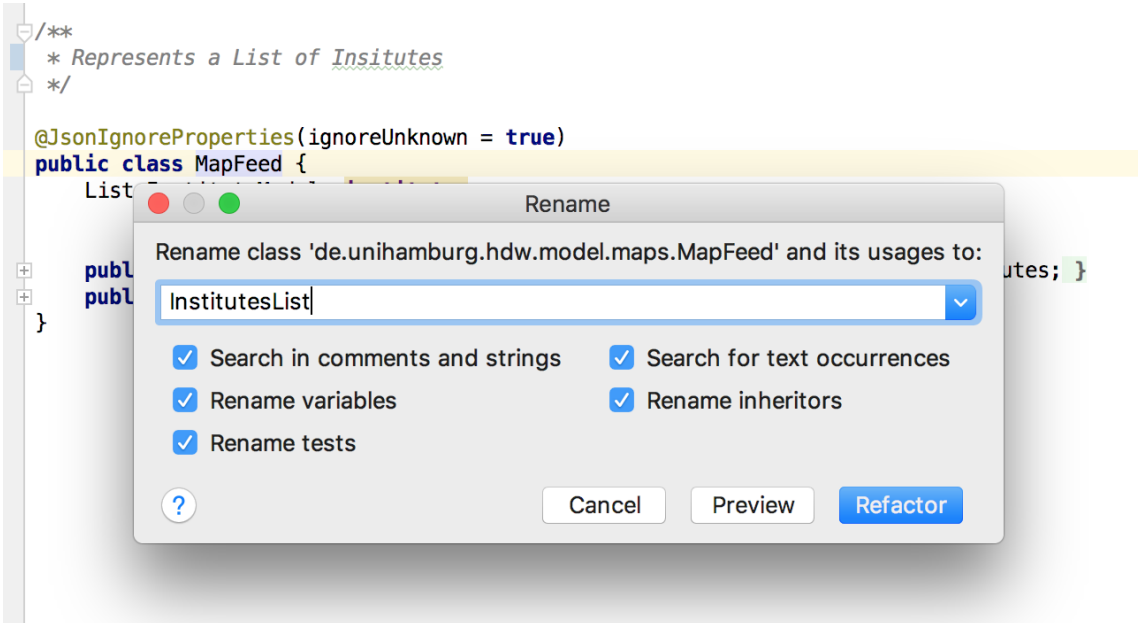

Abbildung 8.21: Abfrage des neuen Namens für die Klasse `MapFeed`

Im nächsten Schritt nutzt das Werkzeug den in Abschnitt 6 eingeführten Trace-Recovery-Mechanismus, um Trace Links zu Code-Elementen zu erhalten, die der Klasse `MapFeed` in der iOS-Implementation der App entsprechen. Es präsentiert dem Entwickler die ersten 50 Ergebnisse, aus denen er wählen kann, welche Elemente konsistent umbenannt werden sollen. Dies ist in Abbildung 8.22 zu sehen.

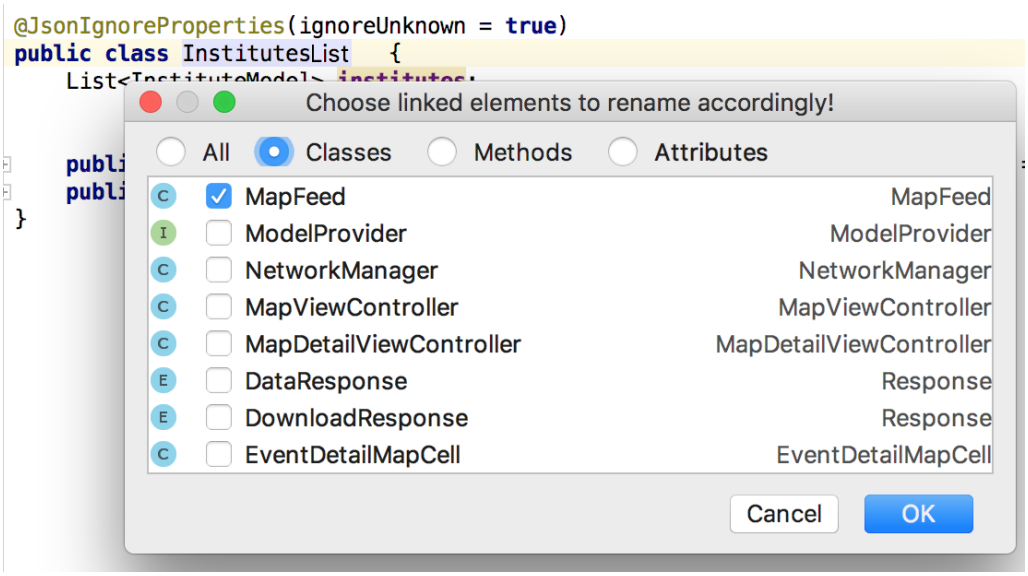

Abbildung 8.22: Abfrage der entsprechend umzubenennenden Elemente der iOS-Implementation

Der Entwickler wählt die entsprechende, gleichnamige Klasse in der iOS-Implementation, um diese konsistent umzubenennen. Nachdem er seine Auswahl bestätigt, führt das

Werkzeug das Refactoring für beide Klassen aus.

Für die Durchführung des Refactorings in der Android-Implementation nutzt das Werkzeug die Refactoring-API der Entwicklungsumgebungen IntelliJ IDEA und Android Studio. Um das entsprechende Refactoring in der iOS-Implementation umzusetzen, wird die Refactoring-Funktionalität der Entwicklungsumgebung Xcode genutzt, mit der der Swift-Code entwickelt wurde. Diese bietet keine öffentliche API zur unmittelbaren Durchführung von Refactorings. Stattdessen führt das Plugin ein Programm in der Skriptsprache AppleScript aus. Dieses Skript simuliert durch Tastatureingaben einen Benutzer, der das Refactoring in Xcode durchführt. Der Entwickler sieht somit, wie Xcode geöffnet wird und das Skript die entsprechende Datei öffnet und das Refactoring-Werkzeug automatisch anwendet. Nach Abschluss des entsprechenden Refactorings in der iOS-Implementation sind beide Klassen konsistent umbenannt.

Bewertung des Ansatzes

Durch die plattform-übergreifende Durchführung des Refactorings muss der neue Name des umzubenennenden Elements nicht für jede der Implementationen einzeln eingegeben werden. Dies ist ein kleiner Gewinn in Bezug auf den Aufwand für plattform-übergreifende Refactorings. Der entscheidende Vorteil gegenüber der separaten Durchführung der Refactorings ist, dass der Aufwand für die Koordination der einzelnen Refactorings entfällt. Insbesondere wird automatisch verhindert, dass das Refactoring nur in einer Implementation durchgeführt und in der zweiten Implementation unterlassen wird. Schlägt das plattform-übergreifende Refactoring in einer der Implementationen fehl, so wird es auch in der zweiten Implementationen rückgängig gemacht, sodass auch in diesem Fall die Konsistenz erhalten bleibt.

Die im Plugin verfügbaren Rename-Refactorings sind wenig komplex, weshalb sie hier zu Zwecken der Demonstration genutzt wurden. Komplexere Refactorings auf Swift-Code wurden zudem von Xcode zum Zeitpunkt der Plugin-Entwicklung nicht zuverlässig unterstützt. Der zugrundeliegende Ansatz ist aber grundsätzlich auf komplexere Refactorings übertragbar. Voraussetzung für die effektive Anwendung plattform-übergreifender Refactorings ist, dass die gemeinsam zu restrukturierenden Code-Elemente sich in Bezug auf die zu ändernde Struktur gleichen. Bei komplexeren Refactorings ist es relativ aufwendig diese Voraussetzung vorab zu prüfen. Diese Prüfung ist allerdings nicht nur für die Durchführung von Refactorings relevant. Sie dient auch dazu, strukturelle Unterschiede unabhängig von der gewünschten Restrukturierung zu erkennen.

8.7.5 Diskussion der Validität

In den obigen Abschnitten wurden die Konzepte zur Unterstützung der plattformübergreifenden Co-Evolution anhand prototypischer Werkzeuge und deren Anwendung bewertet. Die Gültigkeit und Übertragbarkeit der identifizierten Vorteile wird im folgenden diskutiert.

Interne Validität
Die demonstrierten Beispiele wurden mit Ausnahme der Impact-Analyse vom Autor dieser Arbeit durchgeführt. Es ist möglich, dass andere Entwickler die jeweilige Aufgabe anders umgesetzt hätten. Dementsprechend ist es auch möglich, das manche der beobachteten Vorteile der Ansätze und Werkzeuge ausbleiben, wenn sie von anderen Entwicklern eingesetzt werden. Daher ist ein Test der Werkzeuge in weiteren Entwicklungsprojekten zu prüfen.

Externe Validität
Ein Aspekt, der den präsentierten Nutzen in anderen Fällen schmälern könnte, ist die Qualität der genutzten Trace Links. Vor allem, wenn die Entwürfe und die Namenswahl der ursprünglichen Implementation und der Zielimplementation stärker voneinander abweichen, ist es wahrscheinlich, dass sich die Sortierung der vorgeschlagenen Trace Links verschlechtert. In diesen Fällen muss der Entwickler mehr Trace Links prüfen, ehe er die korrekte Entsprechung identifizieren und nutzen kann, was zusätzlichen Aufwand verursacht. Es handelt sich zudem bei den präsentierten Änderungen um relativ einfache Fälle, was der Verständlichkeit der Ausführungen dient. Insbesondere das plattform-übergreifende Rename-Refactoring ist sehr einfach. Grundsätzlich ist der Ansatz auf komplexere Refactorings übertragbar, allerdings sind nicht für alle Sprachen auch Werkzeuge verfügbar, die komplexe Refactorings umsetzen. Zum Zeitpunkt der Implementierung des Refactoring-Werkzeugs war das Rename-Refactoring das einzige, für das eine korrekt arbeitende Automatisierung in Xcode verfügbar war.

Konstruktvalidität
Die Vereinfachung der Co-Evolution kann an den Bewertungen bezüglich entfallender Arbeitsschritte und der Ermöglichung koordinierender Tätigkeiten abgelesen werden. Denkbar wäre alternativ eine langfristige Beobachtung des Entwicklungsaufwands in Projekten, die der hier vorgeschlagenen Methode folgen. Um einen aussagekräftigen Vergleich anzubieten, müsste die Portierung und Co-Evolution allerdings zusätzlich ohne Anwendung der hier vorgestellten Methode und Werkzeuge durchgeführt werden. Zusätzlich wären die präsentierten Werkzeuge auszureifen, um ihren Einsatz durch andere Entwickler zu vereinfachen. Beides würde den Rahmen dieser Arbeit sprengen.

8.8 Zusammenfassende kritische Reflexion der Fallstudien, Experimente und Demonstrationen

In den durchgeführten Fallstudien wurde durch die Anwendung der Portierungsmethode ein durchweg hoher Anteil von Typdefinitionen in vollständige Entsprechungen überführt. Die Fallstudien stellen allerdings keinen sicheren Beweis dafür dar, dass der Einsatz der Methode immer zu einem größeren Ausmaß von Entsprechungen führt als ein nicht angeleitetes Vorgehen der Entwickler. Um eine quantifizierbare Verbesserung zu messen, wäre außerdem ein Vergleich der Portierung jeweils mit und ohne Einsatz der Methode notwendig gewesen. Im Rahmen dieser Arbeit war ein solcher Vergleich nicht möglich, da keine Entwickler für die wiederholte Portierung einer der Fallstudien ohne Einsatz der Methode zu gewinnen waren. Alternativ wäre ein Vergleich unterschiedlicher Portierungsprojekte denkbar gewesen. Ein solcher Vergleich wäre nur begrenzt aussagekräftig, da Portierungsprojekte sich stark unterscheiden. Zum einen haben die Kenntnisse und Fähigkeiten der portierenden Entwickler einen großen Einfluss auf den Aufwand und die Ergebnisse des Portierungsprojekts. Zum anderen variiert der mögliche Automatisierungsgrad extrem in Abhängigkeit von der Kombination von Ausgangs- und Zielsprache, der genutzten APIs und des Umfangs der zu portierenden Benutzeroberfläche. Dies wird sehr deutlich, wenn man das in Abschnitt 6.2.3 beschriebene Portierungsprojekt *Twidere* mit den in Abschnitt 5.1 betrachteten Projekten vergleicht, die bezüglich des Einsatzes von Entwurfsmustern untersucht wurden. Letztere streben bewusst die Angleichung von Ausgangs- und Zielimplementation an und setzen die Entwurfsmuster ein, die in dieser Arbeit verglichen wurden. Sie portieren Programmbibliotheken, die naturgemäß wenige Plattformabhängigkeiten haben und keine grafische Benutzerschnittstelle anbieten. Twidere hingegen ist eine mobile App, bei der ein großer Teil des Codes die plattform-spezifische Benutzeroberfläche implementiert. Die Portierung von Twidere ist zudem ohne die explizite Zielsetzung durchgeführt worden, Ausgangs und Zielimplementation aneinander anzugleichen. Um die breite Anwendbarkeit der Methode über dieses Spektrum hinweg zu zeigen, wurden die Fallstudien bewusst so gewählt, dass mal eine hohe Automatisierung möglich war, wie bei *DESMO-J*, und mal die Automatisierung stark begrenzt war, wie bei der Portierung der App *Metropole des Wissens*.

Die in Abschnitt 8.6 beschriebenen Experimente zur Evaluation des Trace-Recovery-Mechanismus zeigen, dass dieser in drei Projekten mit unterschiedlichen Kombinationen von Ausgangs- und Zielsprache sehr gut sortierte Suchergebnisse liefert. Sie zeigen allerdings nicht, dass die in Abschnitt 6.2 bestimmten Bezeichnergewichte für alle Kombinationen von Ausgangs- und Zielsprache eine Verbesserung der Ergebnisse bewirkt. Für zwei der drei untersuchten Projekte konnte aber eine statistisch signifikante Verbesserung gezeigt werden. Unabhängig davon bietet das zugehörige Trace-Recovery-

Rahmenwerk eine Möglichkeit, plattform-übergreifende Trace Links zu erheben, die bislang ohne Alternative ist. Es gibt zwar verwandte Ansätze aus der Forschung zur Duplikaterkennung, die für diesen Zweck angepasst werden könnten, diese sind aber entweder nicht in öffentlich verfügbaren Werkzeugen implementiert [Mishne u. De Rijke 2004], nicht auf ganze Typdefinitionen anwendbar [Flores u. a. 2012] oder können die Suche nur auf festgelegten Quellen im Internet durchführen [Byalik u. a. 2015].

Das übergeordnete Ziel der hier entwickelten Portierungsmethode ist es, die Co-Evolution von Ausgangs- und Zielimplementation durch die Übertragung von Konzepten und durch plattform-übergreifende Trace Links zu vereinfachen und zusammenzuführen. In Abschnitt 8.7 wurde der Wegfall von Arbeitsschritten bei der exemplarischen Durchführung evolutionärer Aufgaben gezeigt. Dort wurde auch demonstriert, wie Arbeitsabläufe auf Basis der erzielten Entsprechungsbeziehungen und Trace Links plattformübergreifend zusammengeführt werden können. Ein quantitativer Vergleich des *Aufwands* der Portierung und Co-Evolution mit und ohne Einsatz der Portierungsmethode wurde allerdings nicht durchgeführt. Darüber hinaus wäre die langfristige Beobachtung des Einsatzes der entwickelten Werkzeuge für die Co-Evolution in der Industrie wertvoll, um zu prüfen, ob sie den Aufwand für die Co-Evolution tatsächlich senken. Dies hat bislang nicht stattgefunden, da es keine Änderungsanforderungen für die portierten Fallstudien gibt. Die portierte Version der App des Paketdienstleisters wird zudem nicht weiter genutzt, da die Zielplattform Windows Phone nicht länger betrieben wird. Ferner handelt es sich bei den entwickelten Werkzeugen zur Unterstützung der Co-Evolution um Prototypen, die nicht den notwendigen Reifegrad aufweisen, um industriell eingesetzt zu werden.

Die grundsätzlichen Vorteile werden allerdings dadurch sichtbar, dass Arbeitsschritte durch ihren Einsatz entfallen und koordinierende Tätigkeiten ermöglicht werden (vgl. Abschnitt 8.7).

8.9 Diskussion bezüglich der Aufwandsreduktion bei der Portierung und Co-Evolution

In Abschnitt 8.7 wurde exemplarisch demonstriert, wie der Einsatz plattform-übergreifender Trace Links dazu führt, dass Arbeitsschritte bei der Co-Evolution entfallen. So entfällt etwa die wiederholte Konzeption von Änderungen, das doppelte Umsetzen von Refactorings und die Entwickler können Änderungen mit geringem Aufwand übertragen, indem sie den zuerst geänderten Quelltext als Vorlage für die Umsetzung der Änderung in der zweiten Plattform nutzen. Auch während der Portierung entsteht eine Ersparnis gegenüber einer doppelten Implementation durch den Einsatz von Quelltext-Konvertoren und durch die Wiederverwendung von Konzepten. Beides wird durch die

systematische Isolation plattform-spezifischer Elemente anhand der hier entwickelten Richtlinie erleichtert, die Entwickler beim Einsatz isolierender Entwurfsmuster anleitet.

Um diese Ersparnis zu ermöglichen, investieren die Entwickler allerdings zusätzlichen Aufwand während der Portierung. Sie müssen die Isolation der plattform-spezifischen Code-Elemente in Schritt fünf der Methode durchführen, die Handhabung des eingesetzten Quellcode-Konverters erlernen und die notwendigen Mappings konfigurieren. Dieser Aufwand ist kritisch zu reflektieren. Die darauf verwendete Zeit konnte während der Fallstudien nicht klar bestimmt werden, da andere Tätigkeiten häufig im schnellen Wechsel mit der Isolation einzelner Elemente durchgeführt wurden. Allerdings wurde jeweils der Gesamtaufwand für die Fallstudien erhoben. Im Falle der Portierung von *Metropole des Wissens* wurde dieser von den ursprünglichen Entwicklern geschätzt, ehe die Portierung auf Basis der hier entwickelten Methode in Betracht gezogen wurde. Diese Schätzung findet sich in Anhang A.6. Abzüglich des Aufwands für abschließende Tests wurde ein Minimum von 11 und ein Maximum von 18 Personentagen angesetzt. Der tatsächliche Aufwand für die Portierung nach der vorgestellten Methode beläuft sich auf nur 10 Personentage. Das spricht dafür, dass der zusätzliche Aufwand für die Isolation plattform-spezifischer Anteile in dieser Fallstudie nicht größer war als der Aufwand, der durch die Wiederverwendung von Konzepten und durch die Teilautomatisierung der Quellcode-Übersetzung eingespart werden konnte. Für die anderen beiden Fallstudien liegt eine solche unabhängige Schätzung nicht vor. Allerdings ist der jeweils benötigte Aufwand nicht auffällig hoch. Für die Portierung von DESMO-J benötigte der durchführende Student zehn Arbeitstage. Die Portierung der mobilen App des Paketdienstleisters hat 16 Arbeitstage in Anspruch genommen. Grundsätzlich lässt die Methode den portierenden Entwicklern die Freiheit, den Aufwand für die Isolation plattform-spezifischer Code-Elemente selbst zu steuern. Stellen sie fest, dass ein angestrebtes Maß an Automatisierung unwirtschaftlich ist, so können sie dem im Rahmen des iterativen Vorgehens entgegensteuern, indem sie den Abstraktionsgrad der angestrebten Entsprechungsbeziehungen erhöhen (siehe Abschnitt 4.5.3).

Die oben beschriebenen, einmalig zu betreibenden Aufwände während der Portierung bewirken erstens eine Aufwandsersparnis bereits für das Portierungsprojekt selbst, denn die portierenden Entwickler können einen erhöhten Anteil des Codes automatisch in die Zielplattform übertragen und müssen weniger plattformspezifische Implementationskonzepte entwerfen. Zweitens wird bei jeder evolutionären Änderung wiederholt Aufwand eingespart: Etwa die Hälfte des Aufwands während der Evolution von Software investieren Entwickler in das wiederholte Verstehen des bestehenden Quellcodes, ehe sie Änderungen planen und umsetzen können [Bennett u. Rajlich 2000] [Nurvitadhi u. a. 2003]. Durch die erzielten konzeptuellen Entsprechungen müssen Entwickler bei der Weiterentwicklung großer Teile des Codes nur noch ein Implementationskonzept

verstehen und nicht mehr ein Konzept je Plattform. Der Aufwand für das Verstehen des Codes wird somit für gleichartig strukturierte Elemente halbiert, da dieser Schritt für die Zielimplementation entfällt.

Auch der Aufwand für die Umsetzung von Änderungen während der Co-Evolution wird erheblich reduziert, da vollständig konvertierbare Code-Elemente von plattformspezifischen, händisch zu ändernden Elementen bereits bei der Portierung isoliert wurden. Änderungen an konvertierbarem Quellcode können dadurch ohne nennenswerten Aufwand auf die Zielimplementation übertragen werden, indem die Entwickler den Code der Ausgangsimplementation anpassen und durch erneutes Konvertieren in die Zielplattform überführen. Für die Entwickler entfallen somit die konzeptuelle Übertragung der Änderung in die Zielplattform und die händische Wiederholung der Umsetzung.

Zusammenfassend kann also festgehalten werden, dass der steuerbare potentielle Mehraufwand für die Isolation plattform-spezifischer Elemente erstens einer Aufwandsminderung durch erhöhte Automatisierung und durch die erhöhte Wiederverwendung von Konzepten bereits während der Portierung gegenübersteht. Zweitens wird bei der anschließenden Co-Evolution Aufwand durch die vereinfachte Übertragung von Änderungen eingespart. Diese zuletzt genannte Aufwandsersparnis tritt im Verlauf der Co-Evolution immer wieder auf, während der investierte Aufwand für die Isolation nur einmalig anfällt.

9 Fazit und Ausblick

9.1 Fazit

Im Rahmen dieser Arbeit wurde eine Methode zur Portierung von Software entwickelt, die darauf abzielt, die synchronisierte Co-Evolution von ursprünglicher und portierter Implementation zu vereinfachen. Der Vorteil gegenüber anderen Ansätzen besteht darin, dass große Teile des portierten Codes dieselben Konzepte und Strukturen implementieren wie der Code für die ursprüngliche Plattform. Diese Teile werden außerdem von plattform-spezifischen Code-Elementen isoliert, sodass die synchronisierte Weiterentwicklung und der Einsatz etwaiger automatischer Code-Konvertoren vereinfacht wird.

Es wurden Entwurfsmuster bewertet und verglichen, die diese Isolation durch die Einführung plattform-übergreifend gleicher Schnittstellen herbeiführen. Basierend auf diesem Vergleich wurde eine Richtlinie entwickelt, die portierende Entwickler bei der Wahl und Implementation solcher Entwurfsmuster in ihrem konkreten Szenario anleitet [Stehle u. Riebisch 2018a].

Die entwickelte Portierungsmethode wurde inklusive dieser Richtlinie auf drei Portierungsprojekte angewendet, zwei Portierungsprojekte mobiler Anwendungen und eine Portierung einer Software-Bibliothek für die Implementation und Ausführung von Simulationsmodellen. Die Typdefinitionen des portierten Quellcodes entsprechen ihren Vorbildern in allen drei Fallstudien zu großen Anteilen bezüglich ihrer Zuständigkeiten, Schnittstellen und der Anweisungen, mit denen sie implementiert sind.

Darüber hinaus wurden Mechanismen für Trace Capture und Trace Recovery im Kontext der Portierung entwickelt. Sie ermitteln plattform-übergreifende Trace Links zwischen einander entsprechenden Quellcode-Elementen der Ausgangs- und Zielimplementation. Die vom Trace-Capture-Mechanismus generierten Trace Links sind vollständig und korrekt. Auch die Ergebnisse des entwickelten Trace-Recovery-Mechanismus waren in den durchgeführten Experimenten vollständig, enthielten allerdings auch falsche Trace Links, wobei letztere in der Regel hinter den korrekten Trace Links aufgeführt werden, was die erzielten hohen Werte der Mean Average Precision belegen.

Basierend auf dieser Portierungsmethode und den Mechanismen zur Ermittlung von Trace Links wurden Ansätze zur Vereinfachung und Koordination der plattform-übergreifenden Co-Evolution von Ausgangs- und Zielimplementation formuliert. Sie nutzen die erhobenen Trace Links zur Navigation zwischen einander entsprechenden Quellcode-Elementen, zur plattform-übergreifenden Impact-Analyse, zur Koordination von Änderungsaktivitäten und für die Übertragung von Refactorings. Diese Ansätze wurden prototypisch in Plugins für weit verbreitete Entwicklungsumgebungen realisiert und ihr Einsatz wurde exemplarisch demonstriert. Dabei konnte gezeigt werden, dass einzelne Arbeitsschritte im Vergleich zur separaten Weiterentwicklung von Ausgangs- und Zielimplementation entfallen und dass koordinierende Tätigkeiten ermöglicht werden. Somit bleibt die konzeptuelle Ähnlichkeit zwischen Ausgangs- und Zielimplementation erhalten und die langfristige Co-Evolution wird vereinfacht.

9.2 Grenzen der Portierungsmethode

Die hier vorgestellte Portierungsmethode ist nicht auf jedes Portierungsprojekt anwendbar. Insbesondere setzt sie voraus, dass die Ausgangsimplementation in einer objektorientierten Programmiersprache verfasst ist und dass auch die Zielimplementation objektorientiert umgesetzt wird. Ferner ist die Portierungsmethode vor allem dann sinnvoll einsetzbar, wenn die Ausgangsimplementation einen gewissen Reifegrad aufweist und einen hohen Geschäftswert hat. Ansonsten kann auch die Re-Implementierung auf Basis eines Cross-Platform-Frameworks in Erwägung gezogen werden.

Darüber hinaus beziehen sich die entwickelte Vorgehensweise und die Techniken zur Ermittlung und Nutzung plattform-übergreifender Trace Links auf Situationen, in denen sich Ausgangs- und Zielsprache unterscheiden. Kann die ursprünglich genutzte Programmiersprache hingegen auch für die Zielimplementation verwendet werden, so sollten die portierenden Entwickler diese Möglichkeit nutzen. Sie sollten dann eine gemeinsame Codebasis für Ausgangs- und Zielimplementation anstreben, in der plattformunabhängige Code-Elemente verwaltet werden. Die hier entwickelte Portierungsmethode ist dann nicht in ihrer Gesamtheit anzuwenden. Jedoch kann zumindest die in Abschnitt 5.3 entwickelte Richtlinie Anwendung finden, um möglichst viele Code-Elemente in beiden Plattformen trotz unterschiedlicher APIs nutzen zu können. Sie leitet Entwickler dabei an, die Schnittstellen zur Ausgangs- und Zielplattform durch den Einsatz von Entwurfsmustern aneinander anzugleichen.

Die vorgestellte Portierungsmethode behandelt nicht alle potenziell auftretenden Herausforderungen eines Portierungsprojektes. Sie legt den Fokus auf die Portierung und Co-Evolution des Quellcodes. Darüber hinaus kann es notwendig sein, dass Ausgangs-

und Zielimplementation auf einer gemeinsamen Datenbasis operieren. Diese Herausforderung tritt in den betrachteten Fallstudien nicht auf und sie wird von der hier entwickelten Methode nicht behandelt. Weiterführende Arbeiten können diese Herausforderung einbeziehen, indem sie die bestehenden Ansätze für die Datenmigration [Sneed u. a. 2010] in den Kontext der Portierung übertragen.

9.3 Ausblick

9.3.1 Übertragung der Methode auf abgeschlossene Portierungsprojekte

Die vorgestellte Portierungsmethode definiert Richtlinien und Mechanismen, die zu konzeptuellen Ähnlichkeiten zwischen der ursprünglichen und portierten Implementation führen. Auch bei bereits abgeschlossenen Portierungsprojekten besteht grundsätzlich der Bedarf nach solchen Ähnlichkeiten und nach Traceability, um die plattformübergreifende Co-Evolution zu vereinfachen.

Am Beispiel des Twidere-Projektes ist zu sehen, dass der Mechanismus zur Suche nach Trace Links auch auf abgeschlossene Portierungsprojekte angewendet werden kann, in denen die ursprünglichen Konzepte nur unsystematisch übertragen wurden. In diesen Kontexten könnte der Trace-Recovery-Mechanismus als Basis für ein Recommender-System genutzt werden. Ein solches System könnte Refactorings vorschlagen, die die strukturellen Unterschiede zwischen ursprünglicher und portierter Implementation entfernen. Insbesondere wäre es sinnvoll, Bezeichner und Strukturen schrittweise plattformübergreifend anzugleichen. Die hier entwickelte Richtlinie zur Anwendung strukturangleichender Entwurfsmuster kann dabei genutzt werden, um angemessene Entwurfsmuster vorzuschlagen, die durch Refactorings eingeführt werden.

9.3.2 Übertragung der Methode auf Migrationsprojekte

Teile der hier entwickelten Portierungsmethode können auch im Kontext von Migrationsprojekten nützlich sein. Beispielsweise ist es auch in Migrationsprojekten grundsätzlich wünschenswert, dass die migrierte Version der Software der ursprünglichen Version konzeptuell ähnelt, sodass die ursprünglichen Entwickler sich leicht in die migrierte Version einarbeiten können. Wird beispielsweise eine Programmbibliothek ausgetauscht, so können die in Abschnitt 5 diskutierten Entwurfsmuster und Richtlinien eingesetzt werden, um möglichst wenige der Abhängigen Code-Elemente an die neue API anpassen zu müssen.

Darüber hinaus kann der Trace-Recovery-Mechanismus hilfreich sein, um Code-Elemente, die der Codebasis während der Migration hinzugefügt wurden, ihren Entsprechungen in

der ursprünglichen Version der Software zuzuordnen. Das kann insbesondere hilfreich sein, um Fehler in migrierten Code-Elementen zu lokalisieren, indem man sie mit ihrer ursprünglichen, fehlerfreien Entsprechung vergleicht.

9.3.3 Entwicklung einer Quelloffenen Adapter-Bibliothek

Im Rahmen der Fallstudien hat sich herausgestellt, dass einmal entwickelte Adapter in der Regel wiederverwendet werden können, wenn weitere Inkremente portiert werden. Um diese Wiederverwendung auch projektübergreifend zu erlauben, können Adapter in einer quelloffenen Adapter-Bibliothek gesammelt und anderen Entwicklern zur Verfügung gestellt werden.

9.3.4 Erweiterungen der Portierungsmethode

Diese Arbeit konzentriert sich auf die Portierung von Quellcode. Darüber hinaus sind in einigen Fällen auch Datenbanken so zu vereinheitlichen, dass sie von Ausgangs- und Zielimplementation gemeinsam genutzt werden können. Auch die zugehörige Dokumentation kann vereinheitlicht und plattform-übergreifend co-evolviert werden. Weiterführende Arbeiten können diese Aspekte auf Basis der entwickelten Portierungsmethode behandeln. Die vorgestellte Methode macht zudem keine Vorgaben bezüglich der Arbeitsteilung bei der Co-Evolution. Durch die Trennung von plattform-spezifischen und plattformübergreifend äquivalenten Code-Elementen ist aber der Grundstein dafür gelegt, auch die Zuständigkeiten von Entwicklern zwischen diesen Elementen aufzuteilen. Eine weiterführende Arbeit kann auf dieser Basis die möglichen Teamstrukturen diskutieren und Richtlinien zur Aufteilung von Aufgaben bei der plattform-übergreifenden Co-Evolution entwickeln.

9.4 Weitere Nutzungspotentiale des Trace-Recovery-Mechanismus

Mit Hilfe des hier entwickelten Trace-Recovery-Mechanismus können Entwickler nach Entsprechungen zu einem gegebenen Code-Element auch in öffentlichen Codebasen suchen. Auf diesem Wege wäre es möglich, portierenden Entwicklern aus einer großen Basis öffentlicher Quellcode-Elemente potenzielle Vorlagen für die Übersetzung von Quellcode in eine andere Sprache vorzuschlagen.

Sprachübergreifende Entsprechungen zwischen Code-Elementen sind nicht nur in Portierungsprojekten relevant. Auch bei der Entwicklung einer Software mit klassischer Client-Server-Architektur werden häufig die Klassen des Domänenmodells jeweils für

den Client und den Server definiert. Auch bei der Entwicklung sicherheitskritischer Systeme werden gelegentlich Teile des Codes redundant implementiert, um Fehler in einer der Implementationen zu erkennen und auszugleichen. Der Mechanismus zur Erkennung einander entsprechender Code-Elemente kann auch in diesen Kontexten eingesetzt werden. Bei der Weiterentwicklung solcher redundanten Implementationen können Trace Links helfen, die Konsistenz zwischen den Entsprechungen zu wahren. Dabei können auch die in Kapitel 7 vorgestellten Konzepte zur synchronisierten Co-Evolution auf Basis von Trace Links zum Einsatz kommen.

Literaturverzeichnis

[Abd-El-Hafiz 2012] ABD-EL-HAFIZ, S. K.: A Metrics-Based Data Mining Approach for Software Clone Detection. In: *2012 IEEE 36th Annual Computer Software and Applications Conference*, 2012. – ISSN 0730–3157, S. 35–41

[Abou-Saleh u. a. 2018] ABOU-SALEH, Faris; CHENEY, James; GIBBONS, Jeremy; MCKINNA, James; STEVENS, Perdita: Introduction to Bidirectional Transformations. In: *Bidirectional Transformations*. Springer, 2018, S. 1–28

[Al-Omari u. a. 2012] AL-OMARI, F.; KEIVANLOO, I.; ROY, C. K.; RILLING, J.: Detecting Clones Across Microsoft .NET Programming Languages. In: *2012 19th Working Conference on Reverse Engineering*, 2012. – ISSN 2375–5369, S. 405–414

[Ali u. a. 2001] ALI, Mir F.; PÉREZ-QUIÑONES, Manuel A.; SHELL, Eric; ABRAMS, Marc: Building Multi-Platform User Interfaces with UIML. In: *Proceedings of the Fourth International Conference on Computer-Aided Design of User Interfaces* Bd. cs.HC/0111024, 2001, S. 255–266

[Alves u. a. 2005] ALVES, V.; CARDIM, I.; VITAL, H.; SAMPAIO, P.; DAMASCENO, A.; BORBA, P.; RAMALHO, G.: Comparative Analysis of Porting Strategies in J2ME Games. In: *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005. – ISSN 1063–6773, S. 123–132

[Antoniol u. a. 2001] ANTONIOL, G.; CANFORA, G.; CASAZZA, G.; DE LUCIA, A.: Maintaining Traceability Links During Object-Oriented Software Evolution. In: *Software: Practice and Experience* 31, Nr. 4, John Wiley & Sons 2001, S. 331–355. – ISSN 1097–024X

[Antoniol u. a. 2002] ANTONIOL, G.; CANFORA, G.; CASAZZA, G.; LUCIA, A. D.; MERLO, E.: Recovering Traceability Links between Code and Documentation. In: *IEEE Transactions on Software Engineering* 28, Nr. 10, IEEE 2002, Oct, S. 970–983. – ISSN 0098–5589

[Apt u. Olderog 1994] APT, Krzysztof R.; OLDEROG, Ernst-Rüdiger: *Programmverifikation*. Springer Lehrbuch, 1994

[Arango u. a. 1985] ARANGO, G; BAXTER, I; FREEMAN, P; PIDGEON, C: Maintenance and Porting of Software by Design Recovery. In: *Conference on Software Maintenance–1985*. Washington, D.C, USA : IEEE Press, 1985. – ISBN 0–8186–0648–7, S. 42–49

[Arrighi u. a. 2014] ARRIGHI, Pablo; GIRARD, Johan; LEZAMA, Miguel; MAZET, Kévin: The GOOL System: A Lightweight Object-oriented Programming Language Translator. In: *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*. Uppsala, Sweden : ACM, 2014 (ICOOOLPS '14). – ISBN 978–1–4503–2914–9, S. 1–7

[Asuncion u. a. 2010] ASUNCION, Hazeline U.; ASUNCION, Arthur U.; TAYLOR, Richard N.: Software Traceability with Topic Modeling. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. Cape Town, South Africa : ACM, 2010 (ICSE '10). – ISBN 978–1–60558–719–6, S. 95–104

[Avetisyan u. a. 2015] AVETISYAN, A.; KURMANGALEEV, S.; SARGSYAN, S.; ARUTUNIAN, M.; BELEVANTSEV, A.: LLVM-based code clone detection framework. In: *2015 Computer Science and Information Technologies (CSIT)*, 2015, S. 100–104

[Baker 1993] BAKER, Brenda S.: A Theory of Parameterized Pattern Matching: Algorithms and Applications. In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*. San Diego, California, USA : ACM, 1993 (STOC '93). – ISBN 0–89791–591–7, S. 71–80

[Balaban u. a. 2005] BALABAN, Ittai; TIP, Frank; FUHRER, Robert: Refactoring Support for Class Library Migration. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. San Diego, CA, USA, 2005 (OOPSLA '05). – ISBN 1–59593–031–0, S. 265–279

[Bartolomei u. a. 2010] BARTOLOMEI, Thiago T.; CZARNECKI, Krzysztof; LÄMMEL, Ralf: Swing to SWT and back: Patterns for API migration by wrapping. In: *2010 IEEE International Conference on Software Maintenance*, 2010. – ISSN 1063–6773, S. 1–10

[Baxter u. a. 1998] BAXTER, I. D.; YAHIN, A.; MOURA, L.; SANT'ANNA, M.; BIER, L.: Clone detection using abstract syntax trees. In: *Proceedings of the International Conference on Software Maintenance*, 1998. – ISSN 1063–6773, S. 368–377

[Bennett u. Rajlich 2000] BENNETT, Keith H.; RAJLICH, Václav T.: Software Maintenance and Evolution: A Roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*. Limerick, Ireland, 2000. – ISBN 1–58113–253–0, S. 73–87

[Berger 2019] BERGER, Nils-Hendrik: *Plattfromübergreifende Software Change Impact Analyse für portierte Software*, Universität Hamburg, Masterarbeit, 2019

[Botturi u. a. 2013] BOTTURI, G.; EBEID, E.; FUMMI, F.; QUAGLIA, D.: Model-driven design for the development of multi-platform smartphone applications. In: *2013 Forum on Specification Design Languages (FDL)*, 2013. – ISSN 1636–9874, S. 1–8

[Briand u. a. 1999] BRIAND, L.C.; WUST, J.; LOUNIS, H.: Using coupling measurement for impact analysis in object-oriented systems. In: *IEEE International Conference on Software Maintenance - 1999 (ICSM'99)*, 1999. – ISSN 1063–6773, S. 475–482

[Broeksema 2010] BROEKSEMA, Bertjan: *A Visual Tool-Based Approach to Porting C++ Code*, Rijksuniversiteit Groningen, Diplomarbeit, 2010

[Byalik u. a. 2015] BYALIK, Antuan; CHADHA, Sanchit; TILEVICH, Eli: Native-2-native: Automated Cross-platform Code Synthesis from Web-based Programming Resources. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. Pittsburgh, PA, USA : ACM, 2015 (GPCE 2015). – ISBN 978–1–4503–3687–1, S. 99–108

[Cerny u. Donahoo 2015] CERNY, Tomas; DONAHOO, Michael J.: On Separation of Platform-Independent Particles in User Interfaces. In: *Cluster Computing* 18, Nr. 3, Springer Science+Business Media 2015, S. 1215–1228

[Chapman 2011] CHAPMAN, Cameron: *Review Of Cross-Browser Testing Tools*. `http://www.smashingmagazine.com/2011/08/a-dozen-cross-browser-testing-tools/`. Version: August 2011

[Cleff 2019] CLEFF, Thomas: *Angewandte Induktive Statistik und Statistische Testverfahren*. Gabler Verlag, 2019. – ISBN 978–3–8349–0753–0

[Cleland-Huang u. a. 2010] CLELAND-HUANG, J.; CZAUDERNA, A.; GIBIEC, M.; EMENECKER, J.: A machine learning approach for tracing regulatory codes to product specific requirements. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 2010, S. 155–164

[Cleland-Huang u. a. 2012] CLELAND-HUANG, Jane; GOTEL, Orlena; ZISMAN, Andrea: *Software and Systems Traceability*. Springer Publishing Company, Incorporated, 2012. – ISBN 978–1–4471–2238–8

[Clements u. Northrop 2001] CLEMENTS, Paul; NORTHROP, Linda: *Software Product Lines: Practices and Patterns*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2001. – ISBN 0–201–70332–7

[Czarnecki u. Helsen 2006] CZARNECKI, K.; HELSEN, S.: Feature-based Survey of Model Transformation Approaches. In: *IBM Systems Journal* 45, Nr. 3 2006, S. 621–645. – ISSN 0018–8670

[Czarnecki u. a. 2009] CZARNECKI, Krzysztof; FOSTER, J. N.; HU, Zhenjiang; LÄMMEL, Ralf; SCHÜRR, Andy; TERWILLIGER, James F.: Bidirectional Transformations: A Cross-Discipline Perspective. In: PAIGE, Richard F. (Hrsg.): *Theory and Practice of Model Transformations*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009. – ISBN 978–3–642–02408–5, S. 260–283

[De Beukelaer u. a. 2015] DE BEUKELAER, Herman; DAVENPORT, Guy; DE MEYER, Geert; FACK, Veerle: JAMES: A modern object-oriented Java framework for discrete optimization using local search metaheuristics. In: *4th International Symposium & 26th National Conference on Operational Research (HELORS)*, 2015

[De Lucia u. a. 2007] DE LUCIA, Andrea; FASANO, Fausto; OLIVETO, Rocco; TORTORA, Genoveffa: Recovering Traceability Links in Software Artifact Management Systems Using Information Retrieval Methods. In: *ACM Transactions on Software Engineering and Methodology* 16, Nr. 4, ACM 2007, September. – ISSN 1049–331X

[Dehlinger u. Dixon 2011] DEHLINGER, Josh; DIXON, Jeremy: Mobile Application Software Engineering: Challenges and Research Directions. In: *Workshop on Mobile Software Engineering* Bd. 2, 2011, S. 29–32

[Demeyer u. a. 2009] DEMEYER, Serge; DUCASSE, Stéphane; NIERSTRASZ, Oscar: *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2009 `http://scg.unibe.ch/download/oorp/`. – ISBN 978–3–9523341–2–6

[D'Hondt u. a. 2002] In: D'HONDT, Theo; DE VOLDER, Kris; MENS, Kim; WUYTS, Roel: *Co-Evolution of Object-Oriented Software Design and Implementation*. Boston, MA : Springer US, 2002. – ISBN 978–1–4615–0883–0, S. 207–224

[Di Martino u. Cretella 2013] *Kapitel* Semantic and Algorithmic Recognition Support to Porting Software Applications to Cloud. In: DI MARTINO, Beniamino; CRETELLA, Giuseppina: *International Workshop on Eternal Systems 2012: Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–45260–4, S. 1–15

[Dittberner 2007] DITTBERNER, Fabian: *Quelltextgestützte Softwaremigration*, Universität Hamburg, Diplomarbeit, 2007

[Ducasse u. a. 1999] DUCASSE, Stéphane; RIEGER, Matthias; DEMEYER, Serge: A Language Independent Approach for Detecting Duplicated Code. In: *Proceedings of the IEEE International Conference on Software Maintenance*. Washington, DC, USA : IEEE Computer Society, 1999 (ICSM '99), S. 109–119

[Eyal-Salman u. a. 2013] EYAL-SALMAN, H.; SERIAI, A. D.; DONY, C.: Feature-To-Code Traceability in a Collection of Software Variants: Combining Formal Concept Analysis and Information Retrieval. In: *IEEE 14th International Conference on Information Reuse Integration (IRI)*, 2013, S. 209–216

[Fiutem u. Antoniol 1998] FIUTEM, R.; ANTONIOL, G.: Identifying Design-Ccode Inconsistencies in Object-Oriented Software: A Case Study. In: *Proceedings of the International Conference on Software Maintenance*, 1998, S. 94–102

[Flores u. a. 2012] FLORES, Enrique; BARRÓN-CEDEÑO, Alberto; ROSSO, Paolo; MORENO, Lidia: DeSoCoRe: Detecting Source Code Re-Use across Programming Languages. In: *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2012, S. 1–4

[Forward u. Lethbridge 2008] FORWARD, Andrew; LETHBRIDGE, Timothy C.: Problems and Opportunities for Model-centric Versus Code-centric Software Development: A Survey of Software Professionals. In: *Proceedings of the 2008 International Workshop on Models in Software Engineering*. Leipzig, Germany, 2008, S. 27–32

[Fowler 2010] FOWLER, Martin: *Domain Specific Languages*. Addison-Wesley Professional, 2010. – ISBN 0321712943

[Freeman u. a. 2004] FREEMAN, Elisabeth; FREEMAN, Eric; BATES, Bert; SIERRA, Kathy: *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004. – ISBN 0596007124

[Fritz u. a. 2014] FRITZ, Thomas; SHEPHERD, David C.; KEVIC, Katja; SNIPES, Will; BRÄUNLICH, Christoph: Developers' Code Context Models for Change Tasks. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014

[Gamma u. a. 1995] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph E.: *Design Patterns. Elements of Reusable Object-Oriented Software.* Amsterdam : Addison-Wesley Longman, 1995. – ISBN 0201633612

[Gendreau u. Potvin 2005] *Kapitel* 6: Tabu Search. In: GENDREAU, Michel; POTVIN, Jean-Yves: *Search Methodologies*. Boston, MA : Springer US, 2005. – ISBN 978–0–387–28356–2, S. 165–186

[Gethers u. a. 2011] GETHERS, M.; KAGDI, H.; DIT, B.; POSHYVANYK, D.: An Adaptive Approach to Impact Analysis from Change Requests to Source Code. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering*. Lawrence, Kansas, USA, 2011. – ISSN 1938–4300, S. 540–543

[Gorschek u. a. 2014] GORSCHEK, Tony; TEMPERO, Ewan; ANGELIS, Lefteris: On the Use of Software Design Models in Software Development Practice: An Empirical Investigation. In: *Journal of Systems and Software* 95, Elsevier Science Inc. 2014, S. 176–193

[Graham 1992] GRAHAM, D. R.: Incremental Development and Delivery for Large Software Systems. In: *IEE Colloquium on Software Prototyping and Evolutionary Development*. London, UK, 1992, S. 156–195

[Grammel u. a. 2012] GRAMMEL, Birgit; KASTENHOLZ, Stefan; VOIGT, Konrad: Model Matching for Trace Link Generation in Model-Driven Software Development. In: FRANCE, Robert B. (Hrsg.); KAZMEIER, Jürgen (Hrsg.); BREU, Ruth (Hrsg.); ATKINSON,

Colin (Hrsg.): *Model Driven Engineering Languages and Systems*, Springer Berlin Heidelberg, 2012. – ISBN 978–3–642–33666–9, S. 609–625

[Greiert 2018] GREIERT, G.: *Systematische Portierung zur langfristigen Weiterentwicklung von Software auf mehreren Plattformen anhand von Desmo-J*, Universität Hamburg, Masterarbeit, 2018. `https://github.com/TilStehle/DESMO-JS/blob/master/MastersThesisGreiert.pdf`

[Greifenberg u. a. 2015] GREIFENBERG, Timo; HÖLLDOBLER, Katrin; KOLASSA, Carsten; LOOK, Markus; MIR SEYED NAZARI, Pedram; MÜLLER, Klaus; NAVARRO PEREZ, Antonio; PLOTNIKOV, Dimitri; REISS, Dirk; ROTH, Alexander; RUMPE, Bernhard; SCHINDLER, Martin; WORTMANN, Andreas: Integration of Handwritten and Generated Object-Oriented Code. In: *Model-Driven Engineering and Software Development*, Springer, 2015. – ISBN 978–3–319–27868–1, S. 112–132

[Guo u. a. 2017] GUO, J.; CHENG, J.; CLELAND-HUANG, J.: Semantically Enhanced Software Traceability Using Deep Learning Techniques. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, S. 3–14

[Halman u. a. 2006] *Kapitel* 3: Platform-Driven Development of Product Families. In: HALMAN, Johannes I. M.; HOFER, Adrian P.; VAN VUUREN, Wim: *Product Platform and Product Family Design*. Springer US, 2006, S. 27–47

[Hayes u. a. 2006] HAYES, Jane H.; DEKHTYAR, Alex; SUNDARAM, Senthil K.: Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. In: *IEEE Transactions on Software Engineering* 32, Nr. 1, IEEE Press 2006, S. 4–19. – ISSN 0098–5589

[Heitkötter u. a. 2012] HEITKÖTTER, Henning; HANSCHKE, Sebastian; MAJCHRZAK, Tim A.: Evaluating Cross-Platform Development Approaches for Mobile Applications. In: *Web Information Systems and Technologies*, Springer Berlin Heidelberg, 2012. – ISBN 978–3–642–36607–9, S. 120–138

[Heitkötter u. a. 2013] HEITKÖTTER, Henning; MAJCHRZAK, Tim A.; KUCHEN, Herbert: Cross-platform Model-driven Development of Mobile Applications with Md2. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. Coimbra, Portugal, 2013, S. 526–533

[Herraiz u. a. 2013] HERRAIZ, Israel; RODRIGUEZ, Daniel; ROBLES, Gregorio; GONZALEZ-BARAHONA, Jesus M.: The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review. In: *ACM Computing Surveys* 46, Nr. 2, ACM 2013, Dezember, S. 1–28

[Hevner u. a. 2008] HEVNER, Alan R.; MARCH, Salvatore T.; PARK, Jinsoo; RAM, Sudha: Design Science in Information Systems Research. In: *Management Information Systems Quarterly* 28, Nr. 1 2008, S. 6

[Hook 2005] HOOK, B.: *Portabler Code - Einführung in die plattformunabhängige Softwareentwicklung*. San Francisco, USA : No Starch Press, 2005. – ISBN 9781593270568

[Hübner u. Paech 2017] HÜBNER, Paul; PAECH, Barbara: Using Interaction Data for Continuous Creation of Trace Links Between Source Code and Requirements in Issue Tracking Systems. In: *Requirements Engineering: Foundation for Software Quality: 23rd International Working Conference*, 2017, S. 291–307

[Institute of Electrical and Electronics Engineers 1998] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS: *1219 - Standard for Software Maintenance*. 1998. – Standard

[International Organization for Standardization 2011] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. 2011. – Standard

[Janzen 1980] JANZEN, Daniel H: When is it coevolution. In: *Evolution* 34, Nr. 3 1980, S. 611–612

[Jemerov 2008] JEMEROV, Dmitry: Implementing Refactorings in IntelliJ IDEA. In: *Proceedings of the 2Nd Workshop on Refactoring Tools*. Nashville, Tennessee, 2008, S. 13:1–13:2

[Jiang u. a. 2007] JIANG, Lingxiao; MISHERGHI, Ghassan; SU, Zhendong; GLONDU, Stephane: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In: *Proceedings of the 29th International Conference on Software Engineering*, 2007, S. 96–105

[Joorabchi u. a. 2015] JOORABCHI, M. E.; ALI, M.; MESBAH, A.: Detecting inconsistencies in multi-platform mobile apps. In: *26th International Symposium on Software Reliability Engineering (ISSRE), 2015 IEEE*, 2015, S. 450–460

[Joorabchi u. a. 2013] JOORABCHI, M. E.; MESBAH, A.; KRUCHTEN, P.: Real Challenges in Mobile App Development. In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, S. 15–24

[Joorabchi 2016] JOORABCHI, Mona E.: *Mobile app development: challenges and opportunities for automated support*, University of British Columbia, Diss., 2016. `https://open.library.ubc.ca/cIRcle/collections/24/items/1.0228781`

[Kamiya u. a. 2002] KAMIYA, Toshihiro; KUSUMOTO, Shinji; INOUE, Katsuro: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. In: *IEEE Transactions on Software Engineering* 28 2002, S. 654–670

[Khanji 2015] KHANJI, Fares: *Evaluation der Codetransformatoren Sharpen und Tangible Java-to-C#-Converter anhand der Portierung eines Mediatheken-Verwaltungssystems von Java nach C#*. 2015. – Bachelorarbeit

[Knipp u. a. 2016] KNIPP, Eric; CLAYTON, Traverse; WATSON, Richard: A Guidance Framework for Architecting Portable Cloud and Multicloud Applications / Gartner Inc. Version: 2016. `https://www.gartner.com/binaries/content/assets/events/keywords/catalyst/catus8/a_guidance_framework_for_architecting.pdf`. 2016. – Forschungsbericht

[Ko u. a. 2006] KO, Andrew; MYERS, Brad; COBLENZ, Michael; AUNG, Htet: An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. In: *IEEE Transactions on Software Engineering* 32, Nr. 12, Institute of Electrical and Electronics Engineers (IEEE) 2006, S. 97–987

[Kodhai u. a. 2010] KODHAI, E.; KANMANI, S.; KAMATCHI, A.; RADHIKA, R.; SARANYA, B. V.: Detection of Type-1 and Type-2 Code Clones Using Textual Analysis and Metrics. In: *2010 International Conference on Recent Trends in Information, Telecommunication and Computing*, 2010, S. 241–243

[Kraft u. a. 2008] KRAFT, Nicholas A.; BONDS, Brandon W.; SMITH, RandyK.: Cross-language clone detection. In: *20th International Conference on Software Engineering and Knowledge Engineering*. San Francisco, USA, 2008

[Laguna 2018] *Kapitel* 2. In: LAGUNA, Manuel: *Tabu Search*. Cham : Springer International Publishing, 2018. – ISBN 978–3–319–07124–4, S. 741–758

[Lämmel 2004] LÄMMEL, Ralf: Coupled software transformations. In: *First international workshop on software evolution transformations*, 2004, S. 31–35

[Lehman 1980] LEHMAN, M. M.: Programs, life cycles, and laws of software evolution. In: *Proceedings of the IEEE* 68, Nr. 9 1980, September, S. 1060–1076

[Lehnert 2011] LEHNERT, Steffen: A Review of Software Change Impact Analysis / Technische Universität Ilmenau. Version: 2011. `https://pdfs.semanticscholar.org/87bd/a053d6e5b0cb06a508b21db39a1fe37503da.pdf`. 2011. – Forschungsbericht

[Liu u. a. 2006] LIU, Jia; BATORY, Don; LENGAUER, Christian: Feature Oriented Refactoring of Legacy Applications. In: *Proceedings of the 28th International Conference on Software Engineering*. Shanghai, China, 2006, S. 112–121

[Lodderstedt u. a. 2002] LODDERSTEDT, Torsten; BASIN, David; DOSER, Jürgen: SecureUML: A UML-Based Modeling Language for Model-Driven Security. In: *Proceedings of the 5th International Conference on the Unified Modeling Language: Model Engineering, Concepts, and Tools*. Dresden, Germany, 2002, S. 426–441

[Lopez-Herrejon u. a. 2011] LOPEZ-HERREJON, R. E.; MONTALVILLO-MENDIZABAL, L.; EGYED, A.: From Requirements to Features: An Exploratory Study of Feature-Oriented Refactoring. In: *15th International Software Product Line Conference*, 2011, S. 181–190

[Lucia u. a. 2012] LUCIA, A. D.; PENTA, M. D.; OLIVETO, R.; PANICHELLA, A.; PANICHELLA, S.: Using IR methods for labeling source code artifacts: Is it worthwhile? In: *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 2012, S. 193–202

[Mäder u. Gotel 2012] MÄDER, Patrick; GOTEL, Orlena: Towards automated traceability maintenance. In: *Journal of Systems and Software* 85, Nr. 10 2012, S. 2205–2227

[Mäder u. a. 2009] MÄDER, Patrick; GOTEL, Orlena; PHILIPPOW, Ilka: Enabling Automated Traceability Maintenance through the Upkeep of Traceability Relations. In: *Model Driven Architecture - Foundations and Applications*, Springer Berlin Heidelberg, 2009. – ISBN 978–3–642–02674–4, S. 174–189

[Majchrzak u. a. 2018] MAJCHRZAK, Tim A.; BIØRN-HANSEN, Andreas; GRØNLI, Tor-Morten: Progressive Web Apps: the Definite Approach to Cross-Platform Development? In: *Proceedings of the 51st Hawaii International Conference on System Sciences*, 2018, S. 5735 – 5744

[Manning u. a. 2008] MANNING, Christopher D.; RAGHAVAN, Prabhakar; SCHÜTZE, Heinrich: *Introduction to Information Retrieval*. Cambridge University Press, 2008. – ISBN 9780521865715

[Marcus u. Maletic 2001] MARCUS, A.; MALETIC, J. I.: Identification of high-level concept clones in source code. In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 2001, S. 107–114

[Maro u. a. 2016] MARO, S.; ANJORIN, A.; WOHLRAB, R.; STEGHÖFER, J.: Traceability maintenance: Factors and guidelines. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, S. 414–425

[Martin 2003] MARTIN, R.C.: *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, 2003 (Alan Apt series). – ISBN 9780135974445

[Mayer u. Schroeder 2012] MAYER, P.; SCHROEDER, A.: Cross-Language Code Analysis and Refactoring. In: *IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, 2012, S. 94–103

[McDonnell u. a. 2013] MCDONNELL, Tyler; RAY, Baishakhi; KIM, Miryung: An Empirical Study of API Stability and Adoption in the Android Ecosystem. In: *Proceedings of the 2013 IEEE International Conference on Software Maintenance*. Washington, DC, USA, 2013, S. 70–79

[Mens u. a. 2003] MENS, Tom; DEMEYER, Serge; BOIS, Bart D.; STENTEN, Hans; GORP, Pieter V.: Refactoring: Current Research and Future Trends. In: *Electronic Notes in Theoretical Computer Science* 82, Nr. 3 2003, S. 483–499. `http://www.sciencedirect.com/science/article/pii/S1571066105826246`

[Mesbah u. Prasad 2011] MESBAH, Ali; PRASAD, Mukul R.: Automated Cross-browser Compatibility Testing. In: *Proceedings of the 33rd International Conference on Software Engineering*. Waikiki, Honolulu, USA, 2011, S. 561–570

[Mishne 2003] MISHNE, Gilad: *Source Code Retrieval using Conceptual Graphs*, University of Amsterdam, Diplomarbeit, 2003

[Mishne u. De Rijke 2004] MISHNE, Gilad; DE RIJKE, Maarten: Source Code Retrieval using Conceptual Similarity. In: *Proceedings of the 2004 Conference on Computer Assisted Information Retrieval*, 2004, S. 539–554

[Mooney 1997] MOONEY, James D.: Bringing portability to the software process. Version: 1997. `https://pdfs.semanticscholar.org/ef08/695a3e437ea58f48e247f72b4bac61140c5c.pdf`. Citeseer, 1997. – Forschungsbericht

[Mooney 2004] MOONEY, James D.: Developing Portable Software. In: REIS, Ricardo (Hrsg.): *Information Technology*. Boston, MA : Springer US, 2004. – ISBN 978–1–4020–8159–0, S. 55–84

[Mooney 1990] MOONEY, J.D.: Strategies for supporting application portability. In: *Computer* 23, Nr. 11 1990, Nov, S. 59–70

[Moreno u. a. 2013] MORENO, L.; APONTE, J.; SRIDHARA, G.; MARCUS, A.; POLLOCK, L.; VIJAY-SHANKER, K.: Automatic generation of natural language summaries for Java classes. In: *21st International Conference on Program Comprehension (ICPC)*, 2013, S. 23–32

[Murta u. a. 2008] MURTA, Leonardo G. P.; HOEK, André van der; WERNER, Cláudia M. L.: Continuous and Automated Evolution of Architecture-to-Implementation Traceability Links. In: *Automated Software Engineering* 15, Nr. 1 2008, Mar, S. 75–107. – ISSN 1573–7535

[Nagappan u. Shihab 2016] NAGAPPAN, Meiyappan; SHIHAB, Emad: Future Trends in Software Engineering Research for Mobile Apps. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, S. 21–32

[Nguyen u. a. 2014] NGUYEN, Anh T.; NGUYEN, Hoan A.; NGUYEN, Tung T.; NGUYEN, Tien N.: Statistical Learning Approach for Mining API Usage Mappings for Code Migration. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. Vasteras, Sweden, 2014, S. 457–468

[Nielsen 1993] NIELSEN, J.: *Usability Engineering*. Academic Press, 1993. – ISBN 9780080520292

[Nita u. Notkin 2010] NITA, M.; NOTKIN, D.: Using Twinning to Adapt Programs to Alternative APIs. In: *32nd International Conference on Software Engineering*, 2010, S. 205–214

[Nurvitadhi u. a. 2003] NURVITADHI, E.; LEUNG, Wing W.; COOK, C.: Do class comments aid Java program understanding? In: *33rd Annual Frontiers in Education*, 2003, S. 13–17

[Oliveto u. a. 2010] OLIVETO, R.; GETHERS, M.; POSHYVANYK, D.; LUCIA, A. D.: On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery. In: *2010 IEEE 18th International Conference on Program Comprehension*, 2010, S. 68–71

[Page u. Kreutzer 2005] PAGE, Bernd; KREUTZER, Wolfgang: *The Java Simulation Handbook*. Shaker Verlag, 2005. – ISBN 978–3–8322–3771–4

[Panichella u. a. 2013] PANICHELLA, Annibale; DIT, Bogdan; OLIVETO, Rocco; DI PENTA, Massimiliano; POSHYVANYK, Denys; DE LUCIA, Andrea: How to Effectively Use Topic Models for Software Engineering Tasks? An Approach Based on Genetic Algorithms. In: *Proceedings of the 2013 International Conference on Software Engineering*. San Francisco, CA, USA, 2013, S. 522–531

[Parets u. Torres 1996] PARETS, Jose; TORRES, Juan C.: Software maintenance versus software evolution: an approach to software systems evolution. In: *Proceedings of the IEEE Symposium and Workshop on Engineering of Computer-Based Systems*, 1996, S. 134–141

[Peffers u. a. 2007] PEFFERS, Ken; TUUNANEN, Tuure; ROTHENBERGER, Marcus; CHATTERJEE, Samir: A Design Science Research Methodology for Information Systems Research. In: *Journal of Management Information Systems* 24, Nr. 3, M. E. Sharpe, Inc. 2007, S. 45–77

[Pehl 2012] PEHL, Joachim: *Design Patterns for API Analysis & Migration*, Diplomarbeit, 2012

[Petrasch u. Meimberg 2006] PETRASCH, Roland; MEIMBERG, Oliver: *Model Driven Architecture - eine praxisorientierte Einführung in die MDA*. dpunkt.verlag, 2006. – ISBN 3898643433

[Petrenko u. Rajlich 2009] PETRENKO, Maksym; RAJLICH, Vaclav: Variable Granularity for Improving Precision of Impact Analysis. In: *IEEE International Conference on Program Comprehension*. Vancouver, Canada, 2009, S. 10–19

[Petridis u. a. 2010] PETRIDIS, Panagiotis; DUNWELL, Ian; DE FREITAS, Sara; PANZOLI, David: An Engine Selection Methodology for High Fidelity Serious Games. In: *2010 Second International Conference on Games and Virtual Worlds for Serious Applications* IEEE, 2010, S. 27–34

[Pietrek u. Trompeter 2007] PIETREK, Georg; TROMPETER, J: *Modellgetriebene Softwareentwicklung - MDA und MDSD in der Praxis*. entwickler.press, 2007. – ISBN 3939084115

[Piorkowski u. a. 2013] PIORKOWSKI, David J.; FLEMING, Scott D.; KWAN, Irwin; BURNETT, Margaret M.; SCAFFIDI, Christopher; BELLAMY, Rachel K.; JORDAHL, Joshua: The Whats and Hows of Programmers' Foraging Diets. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Paris, France, 2013, S. 3063–3072

[Raheja u. Tekchandani 2013] RAHEJA, K; TEKCHANDANI, Rajkumar: An Emerging Approach towards Code Clone Detection : Metric Based Approach on Byte Code. In: *International Journal of Advanced Research in Computer Science and Software Engineering* 3, Nr. 5 2013, S. 881–888

[Rahimi u. a. 2016] RAHIMI, M.; GOSS, W.; CLELAND-HUANG, J.: Evolving Requirements-to-Code Trace Links across Versions of a Software System. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, S. 99–109

[Rajlich 2012] RAJLICH, Vaclav: *Software Engineering: The Current Practice*. CRC Press, 2012. – ISBN 978–1–43–984122–8

[Rajlich u. Bennett 2000] RAJLICH, Václav T; BENNETT, Keith H.: A staged model for the software life cycle. In: *Computer* 33, Nr. 7, IEEE 2000, S. 66–71

[Rieger u. Majchrzak 2016] RIEGER, Christoph; MAJCHRZAK, Tim A.: Weighted Evaluation Framework for Cross-Platform App Development Approaches. In: *EuroSymposium on Systems Analysis and Design*, 2016, S. 18–39

[Robertson u. Walker 1994] ROBERTSON, S. E.; WALKER, S.: Some Simple Effective Approximations to the 2-Poisson Model for Probabilistic Weighted Retrieval. In: *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Dublin, Ireland, 1994, S. 232–241

[Robertson u. Zaragoza 2009] ROBERTSON, Stephen; ZARAGOZA, Hugo: The Probabilistic Relevance Framework: BM25 and Beyond. In: *Foundations and Trends in Information Retrieval* 3, Nr. 4, Now Publishers Inc. 2009, S. 333–389

[Roy Choudhary 2014] ROY CHOUDHARY, Shauvik: Cross-platform Testing and Maintenance of Web and Mobile Applications. In: *Companion Proceedings of the 36th International Conference on Software Engineering*. Hyderabad, Indien, 2014, S. 642–645

[Roy Choudhary u. a. 2014] ROY CHOUDHARY, Shauvik; PRASAD, Mukul R.; ORSO, Alessandro: Cross-platform Feature Matching for Web Applications. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. San Jose, USA, 2014, S. 82–92

[Rumpe 2012] RUMPE, B.: *Xpert.press*. Bd. 2: *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer Berlin Heidelberg, 2012. – ISBN 978–3–54–020905–8

[Sangiovanni-Vincentelli u. Martin 2001] SANGIOVANNI-VINCENTELLI, A.; MARTIN, G.: Platform-Based Design and Software Design Methodology for Embedded Systems. In: *IEEE Design Test of Computers* 18, Nr. 6 2001, Nov, S. 23–33

[Schmitz 2014] SCHMITZ, Matthias: *Strategie für die Portierung von Desktop-Business-Anwendungen auf iOS-gestützte Endgeräte*. Springer Fachmedien Wiesbaden, 2014 (BestMasters). – ISBN 978–3–658–04768–9

[Seibel u. a. 2010] SEIBEL, Andreas; NEUMANN, Stefan; GIESE, Holger: Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. In: *Software & Systems Modeling* 9, Nr. 4 2010, S. 493–528

[Seiffert u. Hummel 2013] SEIFFERT, Dominic; HUMMEL, Oliver: Improving the Runtime-Processing of Test Cases for Component Adaptation. In: FAVARO, John (Hrsg.); MORISIO, Maurizio (Hrsg.): *Safe and Secure Software Reuse*, Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–38977–1, S. 81–96

[Shah u. a. 2013] SHAH, S. M. A.; DIETRICH, J.; MCCARTIN, C.: On the Automation of Dependency-Breaking Refactorings in Java. In: *2013 IEEE International Conference on Software Maintenance*, 2013, S. 160–169

[Sinai u. Yahav 2014] SINAI, Meital B.; YAHAV, Eran: *Code similarity via natural language descriptions*. 2014. – Masterarbeit

[Singh u. a. 2016] SINGH, Alka; HENLEY, Austin Z.; FLEMMING, Scott D.; LUONG, Maria V.: An Empirical Evaluation of Models of Programmer Navigation. In: *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. Raleigh, USA, 2016

[Sneed 2015] SNEED, Harry M.: Aufwandsschätzung der Softwarewartung und -evolution. In: *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI)*. Dresden, Deutschland, 2015, S. 386 – 402

[Sneed u. a. 2010] SNEED, H.M.; WOLF, E.; HEILMANN, H.: *Softwaremigration in der Praxis: Übertragung alter Softwaresysteme in eine moderne Umgebung*. dpunkt-Verlag, 2010. – ISBN 978–3–89–864564–5

[Spolsky 2000] SPOLSKY, Joel: *Things You Should Never Do, Part I*. Blogeintrag. `https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/`. Version: April 2000

[Stehle u. Riebisch 2015] STEHLE, Tilmann; RIEBISCH, Matthias: Establishing Common Architectures for Porting Mobile Applications to new Platforms. In: *17. Workshop Software-Reengineering und -Evolution der GI-Fachgruppe Software-Reengineering*, Gesellschaft für Informatik e.V., 2015, S. 21–22

[Stehle u. Riebisch 2017] STEHLE, Tilmann; RIEBISCH, Matthias: Erhebung von Trace Links für die koordinierte, plattformübergreifende Co-Evolution portierter Software. In: *Workshop des Arbeitskreises Traceability/Evolutionän der Technischen Universität Ilmenau: Aktuelle Methoden zur Gewinnung und Aktualisierung von Traceability-Modellen*, Gesellschaft für Informatik e.V., 2017, S. 14–16

[Stehle u. Riebisch 2018a] STEHLE, Tilmann; RIEBISCH, Matthias: Application of Design Patterns for Structural Alignment in Software Porting. In: *Proceedings of the 25th Australasian Software Engineering Conference (ASWEC)*, IEEE, 2018, S. 181 – 190

[Stehle u. Riebisch 2018b] STEHLE, Tilmann; RIEBISCH, Matthias: Modellierung plattformübergreifender Quellcode-Entsprechungen für die koordinierte Co-Evolution portierter Software-Systeme. In: *Modellierung 2018*. Braunschweig : Gesellschaft für Informatik e.V., 2018, S. 215–228

[Stehle u. Riebisch 2019] STEHLE, Tilmann; RIEBISCH, Matthias: A Porting Method for Coordinated Multiplatform Evolution. In: *Journal of Software: Evolution and Process* 31, Nr. 2, John Wiley and Sons 2019, S. 1–27

[Stellman u. Greene 2019] STELLMAN, Andrew; GREENE, Jennifer: *Agile Methoden von Kopf bis Fuß*. O'Reilly, 2019. – ISBN 978–3–96009–079–3

[Strein u. a. 2006] STREIN, D.; KRATZ, H.; LOWE, W.: Cross-Language Program Analysis and Refactoring. In: *6th IEEE International Workshop on Source Code Analysis and Manipulation*, 2006, S. 207–216

[Tanaka u. a. 1995] TANAKA, T.; HAKUTA, M.; IWATA, N.; OHMINAMI, M.: Approaches to Making Software Porting More Productive. In: *Proceedings of the 12th TRON Project International Symposium*, 1995, S. 73–85

[Tanveer 2017] TANVEER, Binish: Guidelines for Utilizing Change Impact Analysis when Estimating Effort in Agile Software Development. In: *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. Karlskrona, Sweden, 2017, S. 252–257

[Terekhov 2003] TEREKHOV, A. A.: Re-using software architecture in legacy transformation projects. In: *International Conference on Software Maintenance*. Amsterdam, Niederlande, Sept 2003, S. 462

[Terekhov u. Verhoef 2000] TEREKHOV, A. A.; VERHOEF, C.: The realities of language conversions. In: *IEEE Software* 17, Nr. 6 2000, Nov, S. 111–124. – ISSN 0740–7459

[Terekhov 2001] TEREKHOV, Andrey A.: Automating Language Conversion: A Case Study. In: *Proceedings of the IEEE International Conference on Software Maintenance*. Williamsburg, USA, 2001 (ICSM), S. 654–658

[Thalheim 2013] THALHEIM, Bernhard: The Conception of the Model. In: *Business Information Systems*. Poznań, Polen, 2013, S. 113–124

[Tichelaar 2001] TICHELAAR, Sander: *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*, University of Berne, Diss., 2001

[Tisdall u. a. 2018] TISDALL, Tom; CHOBHARKAR, Pankaj; KIM, Dae-Kyoo: Challenges in Porting Enterprise Applications to Mobile Platforms. In: *GetMobile: Mobile Computing and Communications* 22, Nr. 1, ACM 2018, S. 21–25

[Udagawa 2013] UDAGAWA, Yoshihisa: Source Code Retrieval Using Sequence Based Similarity. In: *International Journal of Data Mining & Knowledge Management Process (IJDKP)* 3, Nr. 4 2013, S. 57–74

[Valente u. a. 2012] VALENTE, M. T.; BORGES, V.; PASSOS, L.: A Semi-Automatic Approach for Extracting Software Product Lines. In: *IEEE Transactions on Software Engineering* 38, Nr. 4 2012, S. 737–754

[Vanhooff u. a. 2007] VANHOOFF, Bert; BAELEN, Stefan V.; JOOSEN, Wouter; BERBERS, E: Traceability as Input for Model Transformations. In: *Proceedings of the ECMDA Traceability Workshop*. Haifa, Israel, 2007

[Vernon 2013] VERNON, Vaughn: *Implementing Domain-Driven Design*. Upper Saddle River, USA : Addison-Wesley, 2013. – ISBN 978–0–321–83457–7

[Vislavski u. a. 2018] VISLAVSKI, T.; RAKIC, G.; CARDOZO, N.; BUDIMAC, Z.: LICCA: A tool for cross-language clone detection. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, S. 512–516

[Vlissides 1996] VLISSIDES, John: Generation Gap. In: *C++ Report* 8, Nr. 10 1996, S. 12–18

[Wasserman 2010] WASSERMAN, Anthony I.: Software Engineering Issues for Mobile Application Development. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. Santa Fe, New Mexico, USA : ACM, 2010 (FoSER '10). – ISBN 978–1–4503–0427–6, S. 397–400

[Wolf u. Bleek 2011] WOLF, Henning; BLEEK, Wolf-Gideon: *Agile Softwareentwicklung: Werte, Konzepte und Methoden*. Bd. 2. Heidelberg : dpunkt.verlag, 2011

[Wright u. a. 2010] WRIGHT, Hyrum K.; KIM, Miryung; PERRY, Dewayne E.: Validity Concerns in Software Engineering Research. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. Santa Fe, New Mexico, USA, 2010, S. 411–414

[Zager u. Verghese 2008] ZAGER, Laura A.; VERGHESE, George C.: Graph similarity scoring and matching. In: *Applied mathematics letters* 21, Nr. 1, Elsevier 2008, S. 86–94

[Zaidman u. a. 2008] ZAIDMAN, A.; ROMPAEY, B. V.; DEMEYER, S.; DEURSEN, A. v.: Mining Software Repositories to Study Co-Evolution of Production & Test Code. In: *2008 1st International Conference on Software Testing, Verification, and Validation*, 2008, S. 220–229

[Zellagui u. a. 2017] ZELLAGUI, Soumia; TIBERMACINE, Chouki; BOUZIANE, Lilia H.; SERIAI, Abdelhak-Djamel; DONY, Christophe: Refactoring Object-Oriented Applications towards a better Decoupling and Instantiation Unanticipation. In: *SEKE: Software Engineering and Knowledge Engineering*. Pittsburgh, United States, Juli 2017

[Zhong u. a. 2010] ZHONG, Hao; THUMMALAPENTA, Suresh; XIE, Tao; ZHANG, Lu; WANG, Qing: Mining API Mapping for Language Migration. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. Cape Town, South Africa, 2010, S. 195–204

[Zhou u. a. 2013] ZHOU, J.; LU, Y.; LUNDQVIST, K.: A Context-based Information Retrieval Technique for Recovering Use-Case-to-Source-Code Trace Links in Embedded Software Systems. In: *39th Euromicro Conference on Software Engineering and Advanced Applications*, 2013, S. 252–259

Vorausgegangene Veröffentlichungen des Autors

Stehle u. Riebisch 2015 STEHLE, Tilmann; RIEBISCH, Matthias: Establishing Common Architectures for Porting Mobile Applications to new Platforms. In: *17. Workshop Software-Reengineering und -Evolution der GI-Fachgruppe Software-Reengineering*, Gesellschaft für Informatik e.V., 2015, S. 21–22

Stehle u. Riebisch 2018a STEHLE, Tilmann; RIEBISCH, Matthias: Application of Design Patterns for Structural Alignment in Software Porting. In: *Proceedings of the 25th Australasian Software Engineering Conference (ASWEC)*, IEEE, 2018, S. 181 – 190

Stehle u. Riebisch 2017 STEHLE, Tilmann; RIEBISCH, Matthias: Erhebung von Trace Links für die koordinierte, plattformübergreifende Co-Evolution portierter Software. In: *Workshop des Arbeitskreises Traceability/Evolutionän der Technischen Universität Ilmenau: Aktuelle Methoden zur Gewinnung und Aktualisierung von Traceability-Modellen*, Gesellschaft für Informatik e.V., 2017, S. 14–16

Stehle u. Riebisch 2018b STEHLE, Tilmann; RIEBISCH, Matthias: Modellierung plattformübergreifender Quellcode-Entsprechungen für die koordinierte Co-Evolution portierter Software-Systeme. In: *Modellierung 2018*. Braunschweig : Gesellschaft für Informatik e.V., 2018, S. 215–228

Stehle u. Riebisch 2019 STEHLE, Tilmann; RIEBISCH, Matthias: A Porting Method for Coordinated Multiplatform Evolution. In: *Journal of Software: Evolution and Process* 31, Nr. 2, John Wiley and Sons 2019, S. 1–27

Betreute Lehrveranstaltungen und Abschlussarbeiten

Bearbeiter	Titel & Art der betreuten Arbeit	Beitrag zu dieser Arbeit
Andersen, Jakob S. Berger, Nils-Hendrik Fischer, Evelyn Greiert, Gerrit Jährling, Claas Khanji, Fares Schulz, Maike	Mobile Anwendungen für mehrere Plattformen - Portierung, Architektur- und Tool-Entwicklung (Masterprojekt 2016/17)	Anwendung der Portierungsmethode auf eine mobile App; Erweiterung der Konvertoren *Sharpen* und *J2Swift* zur Erzeugung von Trace Links
Maier, Thomas	Patterns und Designvarianten zur asynchronen Nutzung eines Webservices im Android-Framework am Beispiel der SE-Abnahme Applikation "SE-Manager" (Bachelorarbeit)	Untersuchung von Entwurfsmustern zur Entkopplung von Geschäftslogik und asynchroner Kommunikation
Kreutzmann, Martin	Patterns und Designvarianten zur Trennung von UI und Geschäftslogik im Android-Framework am Beispiel der SE-Abnahme Applikation „SE-Manager" (Bachelorarbeit)	Untersuchung von Entwurfsmustern zur Entkopplung von Benutzerschnittstelle und Geschäftslogik
Fortsetzung auf der nächsten Seite...		

Tabelle 9.1: Beiträge betreuter Lehrveranstaltungen und Abschlussarbeiten

Bearbeiter	Titel & Art der betreuten Arbeit	Beitrag zu dieser Arbeit
Khanji, Fares	Evaluation der Codetransformatoren Sharpen und Tangible Java-to-C#-Converter anhand der Portierung eines Mediatheken-Verwaltungssystems von Java nach C# (Bachelorarbeit)	Vergleich und Erweiterung von Quellcode-Konvertoren (Java - C#)
Greiert, Gerrit	Entwicklung eines Plugins in IntelliJ IDEA zum Auffinden von Quellcode-Entsprechungen (Studie)	Integration des Trace Recovery Frameworks in IntelliJ IDEA
Greiert, Gerrit	Systematische Portierung zur langfristigen Weiterentwicklung von Software auf mehreren Plattformen anhand von Desmo-J (Masterarbeit)	Anwendung der Portierungsmethode auf die Fallstudie DESMO-J
Andersen, Jakob S.	Zerlegen von Bezeichnern in Java und Swift- Quelltexten zwecks Ähnlichkeitsanalyse (Studie)	Automatische Extraktion von Bezeichnern aus Java- und Swift-Quelltexten im Trace Recovery-Framework
Berger, Nils-Hendrik	Plattformübergreifende Software Change Impact-Analyse für portierte Software (Masterarbeit)	Erweiterung einer Impact-Analyse-Methode um plattformübergreifende Trace Links; Realisierung der plattformübergreifenden Impact-Analyse als IntelliJ-Plugin

Beiträge betreuter Lehrveranstaltungen und Abschlussarbeiten

A Anhang

A.1 Abstrakte ViewModels für Android und WindowsPhone

Im Rahmen der Fallstudien in dieser Arbeit wurden generische abstrakte Klassen `AbstractViewModel<TModel>` entwickelt, die Modelle vom festzulegenden Typ `TModel` verwalten. Sie können ihr Modell asynchron initialisieren und informieren ihre Beobachter, wenn sich der Zustand des Modells ändert. In der Android-spezifischen Implementation von `AbstractViewModel<TModel>` ist die Benachrichtigung der Beobachter per Methodenaufruf implementiert, während die Implementation für .Net/WindowsPhone per Event umgesetzt ist. Der Code dieser Klassen ist öffentlich zugänglich[1].

Abbildung A.1 zeigt eine exemplarische Nutzung der Klassen zur Verwaltung eines Mitarbeiter-Objekts. Wird eine der Eigenschaften des Mitarbeiters durch die Methoden der konkreten Klasse `EmployeeViewModel` geändert, so ruft die Klasse an ihrer Oberklasse die Methode `super.raisePropertyChanged(propertyName)` auf. Die plattformspezifischen Implementationen der Methode benachrichtigen die beobachtenden Objekte.

[1] `https://github.com/TilStehle/.NetAdapters/tree/master/ViewModelFramework`

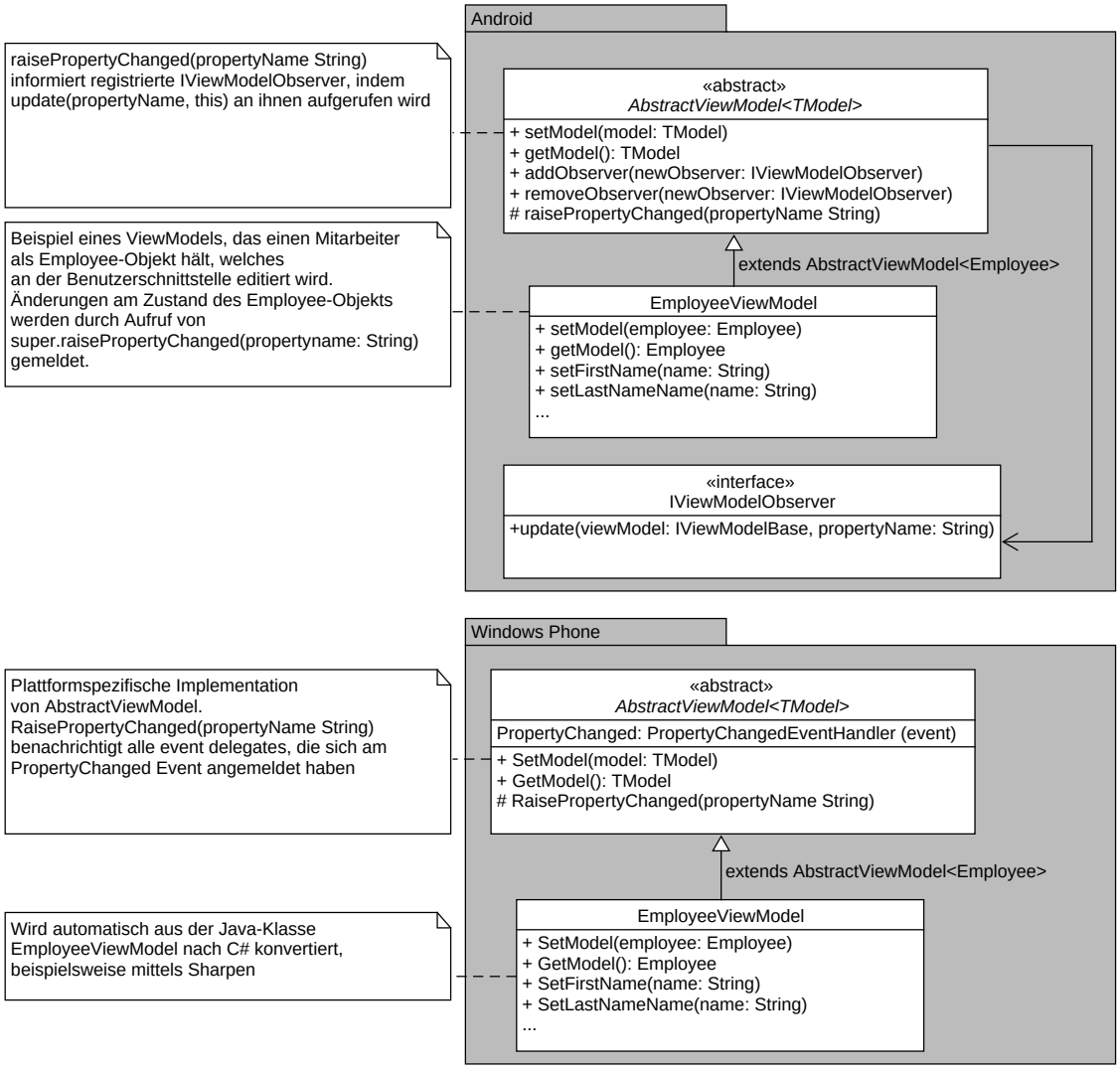

Abbildung A.1: Implementation des Generation Gap Patterns für ViewModel-Klassen in der App des Paketdienstleisters

A.2 Quellcodebeispiel zur Anpassung der Klasse `java.util.Date`

```
public class DateAdapter extends Date
{
   public DateAdapter() {}

   public DateAdapter(long millis)   { super(millis);}

   public String weekday()
   {
      SimpleDateFormat simpleDateformat =
      new SimpleDateFormat("EEEE");
      return simpleDateformat.format(this);
   }

   private static boolean isBetween(int x, int lower, int upper)
   {
      return lower <= x && x <= upper;
   }

   public String shortDate()
   {
      SimpleDateFormat simpleDateformat =
      new SimpleDateFormat("d. MMMM");
      return simpleDateformat.format(this);
   }

   public String formattedDurationUntil(DateAdapter endDate)
   {
      SimpleDateFormat formatter = new SimpleDateFormat("HH:mm");
      String  startString = formatter.format(this);
      String endString = formatter.format(endDate.getDate());
      return startString +" - "+endString;
   }
   ...
}
```

Codebeispiel A.1: Class Adapter zur Anpassung der Klasse `java.util.Date` an das erweiterte Struct `Date` der iOS-Plattform

A.3 Manuell erhobene Trace Links für das Projekt Twidere

Ursprüngliche Typdefinition (Java/Kotlin)	Portierte Typdefinition(Swift)
microblog.library.MicroBlogException	MicroBlogError.Error
microblog.library.twitter.api.FriendsFollowersResources	MicroBlogService
microblog.library.twitter.api.FavoritesResources	MicroBlogService
microblog.library.twitter.api.HelpResources	MicroBlogService
microblog.library.twitter.api.PrivateActivityResources	MicroBlogService
microblog.library.twitter.api.PrivateFriendsFollowersResources	MicroBlogService
microblog.library.twitter.api.TimelineResources	MicroBlogService
microblog.library.twitter.model.GeoPoint	GeoLocation
microblog.library.twitter.util.TwitterDateConverter	Date.Formatter
twidere.activity.HomeActivity	HomeController
twidere.activity.SignInActivity	SignInController
twidere.activity.ComposeActivity	ComposeController
twidere.activity.BrowserSignInActivity	BrowserSignInController
twidere.activity.MediaViewerActivity	MediaPreviewContainer
twidere.app.TwidereApplication	AppDelegate
twidere.fragment.StatusFragment	MicroBlogService
twidere.fragment.StatusFragment	DetailStatusCell
twidere.fragment.StatusFragment.StatusAdapter	DetailStatusCell
twidere.fragment.StatusFragment.StatusAdapter	StatusViewerController
twidere.model.AccountDetails	Account
twidere.model.account.cred.BasicCredentials	Account.Credentials
twidere.model.account.cred.Credentials	Account.Credentials
twidere.model.account.cred.Credentials.Type	Account.AuthType
twidere.model.account.cred.OAuthCredentials	Account.Credentials
twidere.model.account.AccountExtras	Account
twidere.model.account.AccountExtras	Account.Extras
twidere.model.account.StatusNetAccountExtras	Account.Extras
twidere.model.account.TwitterAccountExtras	Account.Extras
twidere.model.draft.UpdateStatusActionExtras	PendingStatusUpdate
twidere.model.draft.UpdateStatusActionExtras	Status.Metadata.InReplyTo
Fortsetzung auf nächster Seite...	

Tabelle A.1: Entsprechungen zwischen 50 Typen der Android-Implementation von Twidere und der entsprechenden iOS-Implementation

twidere.model.draft.UpdateStatusActionExtras	UpdateStatusResult
twidere.model.draft.UpdateStatusActionExtras	StatusUpdate
twidere.model.MediaUploadResult	MediaUploadResponse
twidere.model.MediaUploadResult	MediaUploadResult
twidere.model.ParcelableAccount	Account
twidere.model.ParcelableAccount	User
twidere.model.ParcelableActivity	Activity
twidere.model.ParcelableActivity	Activity.ObjectList
twidere.model.ParcelableCredentials	Account.Credentials
twidere.model.ParcelableLocation	GeoLocation
twidere.model.ParcelableMedia	MediaItem
twidere.model.ParcelableMedia	MediaItem.VideoInfo
twidere.model.ParcelableMedia.Type	MediaItem.MediaType
twidere.model.ParcelableMedia.VideoInfo	MediaItem.VideoInfo
twidere.model.ParcelableMedia.VideoInfo.Variant	MediaItem.VideoInfo.Variant
twidere.model.ParcelableMediaUpdate	MediaUpdate
twidere.model.ParcelableRelationship	User.Metadata
twidere.model.ParcelableStatus	Status
twidere.model.ParcelableStatusUpdate	PendingStatusUpdate
twidere.model.ParcelableStatusUpdate	StatusUpdate
twidere.model.ParcelableUser	User
twidere.model.ParcelableUserList	UserList
twidere.model.ParcelableUserMention	Activity.Action
twidere.model.StatusShortenResult	StatusShortenResult
twidere.model.UserKey	UserKey
twidere.service.AccountAuthenticatorService	OAuthService
twidere.service.AccountAuthenticatorService	SignInController
twidere.service.LengthyOperationsService	BackgroundOperationService
twidere.service.MediaUploaderService	BackgroundOperationService
twidere.task.CreateFriendshipTask	MicroBlogService
twidere.task.DestroyFriendshipTask	MicroBlogService
twidere.task.DestroyFriendshipTask	MicroBlogService
twidere.task.DestroyFriendshipTask	StatusViewerController
twidere.util.AsyncTwitterWrapper	MicroBlogService

Entsprechungen zwischen 50 Typen der Android-Implementation von Twidere und der entsprechenden iOS-Implementation

A.4 Manuell ermittelte Entsprechungen zur Evaluation des Trace-Recovery Mechanismus

Code-Element in ursprünglicher Implementation	Code-Element in Zielimplementation
AbstractChartDataTable	AbstractChartDataTable
AbstractNumericalChartCanvas	AbstractNumericalChartCanvas
AbstractNumericalCoorChartCanvas	AbstractNumericalCoorChartCanvas
AbstractNumericalCoorChartCanvasDouble	AbstractNumericalCoorChartCanvasDouble
AbstractNumericalCoorChartCanvasLong	AbstractNumericalCoorChartCanvasLong
AbstractTableFormatter	AbstractTableFormatter
CanvasHistogramDouble	CanvasHistogramDouble
CanvasHistogramLong	CanvasHistogramLong
CanvasTimeSeries	CanvasTimeSeries
ChartDataHistogramDouble	ChartDataHistogramDouble
ChartDataHistogramLong	ChartDataHistogramLong
ChartDataTimeSeries	ChartDataTimeSeries
Condition	Condition
ContDist	ContDist
ContDistExponReporter	ContDistExponReporter
ContDistExponential	ContDistExponential
ContDistUniform	ContDistUniform
ContDistUniformReporter	ContDistUniformReporter
ContDistWeibull	ContDistWeibull
DESMOJException	DESMOJException
DebugFileOut	DebugFileOut
DebugNote	DebugNote
Distribution	Distribution
DistributionManager	DistributionManager
DistributionReporter	DistributionReporter
Entity	Entity
ErrorFileOut	ErrorFileOut
ErrorMessage	ErrorMessage
Fortsetzung auf nächster Seite...	

Tabelle A.2: Entsprechungen zwischen Code-Elementen der Java-Implementation von DESMO-J und der portierten Implementation in JavaScript

Event	Event
EventAbstract	EventAbstract
EventNote	EventNote
EventOf2Entities	EventOf2Entities
EventOf3Entities	EventOf3Entities
EventTreeList	EventTreeList
EventsExample	EventsExample
FileOutput	FileOutput
HTMLDebugOutput	HTMLDebugOutput
HTMLErrorOutput	HTMLErrorOutput
LinearCongruentialRandomGenerator	LinearCongruentialRandomGenerator
MersenneTwisterRandomGenerator	MersenneTwisterRandomGeneratorJS
NumericalDist	NumericalDist
ServiceEndEvent	ServiceEndEvent
SimAbortedException	SimAbortedException
SimFinishedException	SimFinishedException
StatisticObject	StatisticObject
SubjectAdministration	SubjectAdministration
Truck	Truck
TruckArrivalEvent	TruckArrivalEvent
TruckGeneratorEvent	TruckGeneratorEvent
VanCarrier	VanCarrier

Tabelle A.2: Entsprechungen zwischen Code-Elementen der Java-Implementation von DESMO-J und der portierten Implementation in JavaScript

Source type (Java)	Target type (Swift)
hdw.ExtensionAdapters.DateAdapter	Date
hdw.model.events.EventDetailModel	EventDetailModel
hdw.model.events.EventFeed	EventFeed
hdw.model.events.EventModel	EventModel
hdw.model.events.FeedCategory	FeedCategory
hdw.model.maps.InstituteModel	InstituteModel
hdw.model.maps.MapFeed	MapFeed
hdw.model.news.NewsDetailModel	NewsDetailModel
hdw.model.news.NewsFeed	NewsFeed
hdw.model.news.NewsModel	NewsModel
hdw.model.shared.InstituteID	InstituteID
hdw.model.shared.ModelProvider	ModelProvider
hdw.networkApi.BackendURIConstant	BackendURIConstants
hdw.networkApi.NetworkManager	NetworkManager
hdw.view.Appearance	Appearance
hdw.view.Appearance.AppereanceDelegate	AppearanceDelegate
hdw.view.cells.events.AbstractEventCell	EventTextCell
hdw.view.cells.events.AbstractEventCell	EventTextImageCell
hdw.view.cells.events.AbstractEventCell	EventImageCell
hdw.view.cells.events.CategorieFilteredEventCellAdapter	EventCategoryViewController
hdw.view.cells.events.CategoryCell	CategoryCell
hdw.view.cells.events.EventCategoriesCell	EventCategoriesCell
hdw.view.cells.events.EventCategoriesCell	EventCategoriesCell
hdw.view.cells.events.EventCellAdapter	EventViewController
hdw.view.cells.events.EventImageCell	EventImageCell
hdw.view.cells.events.EventTextCell	EventTextCell
hdw.view.cells.events.EventTextImageCell	EventTextImageCell
hdw.view.cells.events.detail.EventDetailAttachmentsCell	EventDetailAttachmentsCell
hdw.view.cells.events.detail.EventDetailContentCell	EventDetailContentCell
hdw.view.cells.events.detail.EventDetailHeaderCell	EventDetailHeaderCell
hdw.view.cells.events.detail.EventDetailMapCell	EventDetailMapCell
hdw.view.cells.general.NoInternetCell	NoInternetCell
Fortsetzung auf nächster Seite...	

Tabelle A.3: Entsprechungen zwischen 50 Typen der Android-Implementation von Metropole des Wissens und der entsprechenden iOS-Implementation

hdw.view.cells.general.OpenInBrowserCell	OpenInSafariCell
hdw.view.cells.general.OpenUrlDelegate	OpenURLDelegate
hdw.view.cells.institutes.InstituteContentFragment	InstituteContentCell
hdw.view.cells.institutes.InstituteHeaderFragment	InstituteHeaderCell
hdw.view.cells.institutes.InstituteImageFragment	InstituteImageCell
hdw.view.cells.institutes.InstituteLinkFragment	InstituteLinkCell
hdw.view.cells.news.AbstractNewsCell	NewsImageCell
hdw.view.cells.news.AbstractNewsCell	NewsTextImageCell
hdw.view.cells.news.AbstractNewsCell	NewsTextCell
hdw.view.cells.news.NewsCellAdapter	NewsViewController
hdw.view.cells.news.NewsHeaderCell	NewsHeaderCell
hdw.view.cells.news.NewsImageCell	NewsImageCell
hdw.view.cells.news.NewsTextCell	NewsTextCell
hdw.view.cells.news.NewsTextImageCell	NewsTextImageCell
hdw.view.cells.news.detail.NewsDetailImageFragment	NewsDetailImageCell
hdw.view.cells.news.detail.NewsDetailTextFragment	NewsDetailTextCell
hdw.view.cells.paging.PagingCell	PagingCell
hdw.viewController.EventCategoryFragment	EventCategoryViewController
hdw.viewController.EventDetailFragment	EventDetailViewController
hdw.viewController.EventsFragment	EventViewController
hdw.viewController.EventsFragment	EventViewController
hdw.viewController.HOOUFragment	HOOUViewController
hdw.viewController.ImprintFragment	ImprintViewController
hdw.viewController.MapDetailFragment	MapDetailViewController
hdw.viewController.MapDetailFragment	MapDetailViewController
hdw.viewController.MapFragment	MapViewController
hdw.viewController.NewsDetailFragment	NewsDetailViewController
hdw.viewController.NewsDetailFragment	NewsDetailViewController
hdw.viewController.NewsFragment	NewsViewController
hdw.viewController.NewsFragment	NewsViewController
hdw.viewController.OptionsMenuCreator	PopoverFontChangerViewController
hdw.viewController.TabBarActivity	PopoverFontChangerViewController

Entsprechungen zwischen 50 Typen der Android-Implementation von Metropole des Wissens und der entsprechenden iOS-Implementation

Source type (Java)	Target type (C#)
analysis.Analyzer	Analysis.Analyzer
analysis.AnalyzerWrapper	Analysis.AnalyzerWrapper
analysis.CachingTokenFilter	Analysis.CachingTokenFilter
analysis.CharFilter	Analysis.CharFilter
analysis.ReusableStringReader	Analysis.ReusableStringReader
analysis.TokenFilter	Analysis.TokenFilter
analysis.TokenStream	Analysis.TokenStream
analysis.TokenStreamToAutomaton	Analysis.TokenStreamToAutomaton
analysis.Tokenizer	Analysis.Tokenizer
analysis.standard.StandardAnalyzer	Analysis.Standard.StandardAnalyzer
analysis.standard.StandardFilter	Analysis.Standard.StandardFilter
analysis.standard.StandardTokenizer	Analysis.Standard.StandardTokenizer
analysis.standard.StandardTokenizerImpl	Analysis.Standard.StandardTokenizerImpl
analysis.tokenattributes.CharTermAttribute	Analysis.TokenAttributes.ICharTermAttribute
analysis.tokenattributes.CharTermAttributeImpl	Analysis.TokenAttributes.CharTermAttribute
analysis.tokenattributes.FlagsAttribute	Analysis.TokenAttributes.IFlagsAttribute
analysis.tokenattributes.FlagsAttributeImpl	Analysis.TokenAttributes.FlagsAttribute
analysis.tokenattributes.KeywordAttribute	Analysis.TokenAttributes.IKeywordAttribute
analysis.tokenattributes.KeywordAttributeImpl	Analysis.TokenAttributes.KeywordAttribute
analysis.tokenattributes.OffsetAttribute	Analysis.TokenAttributes.IOffsetAttribute
analysis.tokenattributes.OffsetAttributeImpl	Analysis.TokenAttributes.OffsetAttribute
analysis.tokenattributes.PayloadAttribute	Analysis.TokenAttributes.IPayloadAttribute
analysis.tokenattributes.PayloadAttributeImpl	Analysis.TokenAttributes.PayloadAttribute
analysis.tokenattributes.PositionIncrementAttribute	Analysis.TokenAttributes.IPositionIncrementAttribute
analysis.tokenattributes.PositionIncrementAttributeImpl	Analysis.TokenAttributes.PositionIncrementAttribute
analysis.tokenattributes.PositionLengthAttribute	Analysis.TokenAttributes.IPositionLengthAttribute
analysis.tokenattributes.PositionLengthAttributeImpl	Analysis.TokenAttributes.PositionLengthAttribute
analysis.tokenattributes.TermToBytesRefAttribute	Analysis.TokenAttributes.ITermToBytesRefAttribute
analysis.tokenattributes.TypeAttribute	Analysis.TokenAttributes.ITypeAttribute
analysis.tokenattributes.TypeAttributeImpl	Analysis.TokenAttributes.TypeAttribute
codecs.Codec	Codecs.Codec
codecs.CodecUtil	Codecs.CodecUtil
codecs.DocValuesConsumer	Codecs.DocValuesConsumer
codecs.DocValuesFormat	Codecs.DocValuesFormat
codecs.DocValuesProducer	Codecs.DocValuesProducer
codecs.FieldInfosFormat	Codecs.FieldInfosWriter
codecs.FieldInfosFormat	Codecs.FieldInfosFormat
codecs.FieldInfosFormat	Codecs.FieldInfosReader
codecs.FieldsConsumer	Codecs.FieldsConsumer
Fortsetzung auf nächster Seite...	

Tabelle A.4: Entsprechungen zwischen 50 Typen der Java-Implementation von Lucene und Lucene.Net

codecs.FieldsProducer	Codecs.FieldsProducer
codecs.FilterCodec	Codecs.FilterCodec
codecs.LiveDocsFormat	Codecs.LiveDocsFormat
codecs.MultiLevelSkipListReader	Codecs.MultiLevelSkipListReader
codecs.MultiLevelSkipListWriter	Codecs.MultiLevelSkipListWriter
codecs.NormsFormat	Codecs.NormsFormat
codecs.PostingsFormat	Codecs.PostingsFormat
codecs.PostingsReaderBase	Codecs.PostingsReaderBase
codecs.PostingsWriterBase	Codecs.PostingsWriterBase
codecs.SegmentInfoFormat	Codecs.SegmentInfoFormat
codecs.StoredFieldsFormat	Codecs.StoredFieldsFormat
codecs.StoredFieldsReader	Codecs.StoredFieldsReader
codecs.StoredFieldsWriter	Codecs.StoredFieldsWriter

Entsprechungen zwischen 50 Typen der Java-Implementation von Lucene und Lucene.Net

A.5 Automatisch generierte Trace Links für das Projekt SE-Manager

Code-Element in ursprünglicher Implementation	Code-Element in Zielimplementation
AbmeldeAsyncTask	AbmeldeAsyncTask
AbmeldeAsyncTask._callback	AbmeldeAsyncTask._callback
AbmeldeAsyncTask._context	AbmeldeAsyncTask._context
AbmeldeAsyncTask._resources	AbmeldeAsyncTask._resources
AbmeldeAsyncTask._sslContext	AbmeldeAsyncTask._sslContext
AbmeldeAsyncTask._url	AbmeldeAsyncTask._url
AbmeldeAsyncTask.doInBackground	AbmeldeAsyncTask.DoInBackground
AbmeldeAsyncTask.onPostExecute	AbmeldeAsyncTask.OnPostExecute
AbmeldeAsyncTask.onPreExecute	AbmeldeAsyncTask.OnPreExecute
Abnahme	Abnahme
Abnahme._betreuer	Abnahme._betreuer
Abnahme._kommentar	Abnahme._kommentar
Abnahme._labor	Abnahme._labor
Abnahme._smiley	Abnahme._smiley
Abnahme._uebungsaufgabe	Abnahme._uebungsaufgabe
Abnahme._uebungstermin	Abnahme._uebungstermin
Abnahme.equals	Abnahme.Equals
Abnahme.hashCode	Abnahme.GetHashCode
AbnahmeAsyncTask	AbnahmeAsyncTask
AbnahmeAsyncTask._callback	AbnahmeAsyncTask._callback
AbnahmeAsyncTask._methoden	AbnahmeAsyncTask._methoden
AbnahmeAsyncTask._responseCodes	AbnahmeAsyncTask._responseCodes
AbnahmeAsyncTask._sslContext	AbnahmeAsyncTask._sslContext
AbnahmeAsyncTask._urls	AbnahmeAsyncTask._urls
AbnahmeAsyncTask._zielAdresse	AbnahmeAsyncTask._zielAdresse
AbnahmeAsyncTask.checkResponseCodes	AbnahmeAsyncTask.CheckResponseCodes
AbnahmeAsyncTask.doInBackground	AbnahmeAsyncTask.DoInBackground
AbnahmeAsyncTask.onPostExecute	AbnahmeAsyncTask.OnPostExecute
Fortsetzung auf nächster Seite...	

Tabelle A.5: Entsprechungen zwischen Code-Elementen der Java-Implementation des SE-Manager und der automatischen Übersetzung nach C#

AbnahmeAsyncTask.onPreExecute	AbnahmeAsyncTask.OnPreExecute
AbnahmeDienstImpl	AbnahmeDienstImpl
AbnahmeDienstImpl._alleRelevanten-AufgabenModel	AbnahmeDienstImpl._alleRelevanten-AufgabenModel
AbnahmeDienstImpl._context	AbnahmeDienstImpl._context
AbnahmeDienstImpl.fuege-AbnahmeHinzu	AbnahmeDienstImpl.Fuege-AbnahmeHinzu
AbnahmeDienstImpl.get-Betreuer	AbnahmeDienstImpl.Get-Betreuer
AbnahmeDienstImpl.gib-AbnahmezettelZuStudent	AbnahmeDienstImpl.Gib-AbnahmezettelZuStudent
AbnahmeDienstImpl.gib-RelevanteAufgaben	AbnahmeDienstImpl.Gib-RelevanteAufgaben
AbnahmeDienstImpl.gib-RelevanteAusnahmen	AbnahmeDienstImpl.Gib-RelevanteAusnahmen
AbnahmeDienstImpl.sBetreuer	AbnahmeDienstImpl.sBetreuer
AbnahmeDienstImpl.setBetreuer	AbnahmeDienstImpl.SetBetreuer
Abnahmezettel	Abnahmezettel
Abnahmezettel._abnahmeListe	Abnahmezettel._abnahmeListe
Abnahmezettel._anwesendeTermine	Abnahmezettel._anwesendeTermine
Abnahmezettel._student	Abnahmezettel._student
Abnahmezettel.equals	Abnahmezettel.Equals
Abnahmezettel.hashCode	Abnahmezettel.GetHashCode
Abnahmezettel.hatAktuelleKommentare	Abnahmezettel.HatAktuelleKommentare
Abnahmezettel.list2String	Abnahmezettel.List2String
AbnahmezettelAsyncTask	AbnahmezettelAsyncTask
AbnahmezettelAsyncTask._callback	AbnahmezettelAsyncTask._callback
AbnahmezettelAsyncTask._methode-Abnahme	AbnahmezettelAsyncTask._methode-Abnahme
AbnahmezettelAsyncTask._methode-Praesenz	AbnahmezettelAsyncTask._methode-Praesenz
AbnahmezettelAsyncTask._response-CodeAbnahme	AbnahmezettelAsyncTask._response-CodeAbnahme
Fortsetzung auf nächster Seite...	

Entsprechungen zwischen Code-Elementen der Java-Implementation des SE-Manager und der automatischen Übersetzung nach C#

AbnahmezettelAsyncTask._response-CodePraesenz	AbnahmezettelAsyncTask._response-CodePraesenz
AbnahmezettelAsyncTask._sslContext	AbnahmezettelAsyncTask._sslContext
AbnahmezettelAsyncTask._student	AbnahmezettelAsyncTask._student
AbnahmezettelAsyncTask._urlAbnahme	AbnahmezettelAsyncTask._urlAbnahme
AbnahmezettelAsyncTask._urlPraesenz	AbnahmezettelAsyncTask._urlPraesenz
AbnahmezettelAsyncTask._zielAdresse	AbnahmezettelAsyncTask._zielAdresse
AbnahmezettelAsyncTask.doInBackground	AbnahmezettelAsyncTask.DoInBackground
AbnahmezettelAsyncTask.onPostExecute	AbnahmezettelAsyncTask.OnPostExecute
AbnahmezettelAsyncTask.onPreExecute	AbnahmezettelAsyncTask.OnPreExecute
AnmeldeAsyncTask	AnmeldeAsyncTask
AnmeldeAsyncTask._betreuer	AnmeldeAsyncTask._betreuer
AnmeldeAsyncTask._callback	AnmeldeAsyncTask._callback
AnmeldeAsyncTask._context	AnmeldeAsyncTask._context
AnmeldeAsyncTask._resources	AnmeldeAsyncTask._resources
AnmeldeAsyncTask._sslContext	AnmeldeAsyncTask._sslContext
AnmeldeAsyncTask._url	AnmeldeAsyncTask._url
AnmeldeAsyncTask.doInBackground	AnmeldeAsyncTask.DoInBackground
AnmeldeAsyncTask.onPostExecute	AnmeldeAsyncTask.OnPostExecute
AnmeldeAsyncTask.onPreExecute	AnmeldeAsyncTask.OnPreExecute
AnmeldeDienstImpl	AnmeldeDienstImpl
AnmeldeDienstImpl._betreuer	AnmeldeDienstImpl._betreuer
AnmeldeDienstImpl._context	AnmeldeDienstImpl._context
AnmeldeDienstImpl.abmelden	AnmeldeDienstImpl.Abmelden
AnmeldeDienstImpl.anmelden	AnmeldeDienstImpl.Anmelden
AnwesenheitAsyncTask	AnwesenheitAsyncTask
AnwesenheitAsyncTask._callback	AnwesenheitAsyncTask._callback
AnwesenheitAsyncTask._methode	AnwesenheitAsyncTask._methode
AnwesenheitAsyncTask._responseCode	AnwesenheitAsyncTask._responseCode
AnwesenheitAsyncTask._sslContext	AnwesenheitAsyncTask._sslContext
AnwesenheitAsyncTask._url	AnwesenheitAsyncTask._url
AnwesenheitAsyncTask._zielAdresse	AnwesenheitAsyncTask._zielAdresse
Fortsetzung auf nächster Seite...	

Entsprechungen zwischen Code-Elementen der Java-Implementation des SE-Manager und der automatischen Übersetzung nach C#

AnwesenheitAsyncTask.doInBackground	AnwesenheitAsyncTask.DoInBackground
AnwesenheitAsyncTask.onPostExecute	AnwesenheitAsyncTask.OnPostExecute
AnwesenheitAsyncTask.onPreExecute	AnwesenheitAsyncTask.OnPreExecute
AufgabenAsyncTask	AufgabenAsyncTask
AufgabenAsyncTask.onPostExecute	AufgabenAsyncTask.OnPostExecute
Ausnahme	Ausnahme
Ausnahme._aufgabe	Ausnahme._aufgabe
Ausnahme._frist	Ausnahme._frist
Ausnahme._student	Ausnahme._student
Ausnahme.equals	Ausnahme.Equals
Ausnahme.hashCode	Ausnahme.GetHashCode
AusnahmenAsyncTask	AusnahmenAsyncTask
AusnahmenAsyncTask.onPostExecute	AusnahmenAsyncTask.OnPostExecute
AusnahmenAufgabenAsyncTask	AusnahmenAufgabenAsyncTask
AusnahmenAufgaben- AsyncTask._callback	AusnahmenAufgaben- AsyncTask._callback
AusnahmenAufgaben- AsyncTask._methode	AusnahmenAufgaben- AsyncTask._methode
AusnahmenAufgaben- AsyncTask._responseCode	AusnahmenAufgaben- AsyncTask._responseCode
AusnahmenAufgaben- AsyncTask._sslContext	AusnahmenAufgaben- AsyncTask._sslContext
AusnahmenAufgabe- nAsyncTask._url	AusnahmenAufgaben- AsyncTask._url
AusnahmenAufgaben- AsyncTask._zielAdresse	AusnahmenAufgaben- AsyncTask._zielAdresse
AusnahmenAufgaben- AsyncTask.doInBackground	AusnahmenAufgaben- AsyncTask.DoInBackground
AusnahmenAufgaben- AsyncTask.onPreExecute	AusnahmenAufgaben- AsyncTask.OnPreExecute
ContextHolder	ContextHolder
ContextHolder.context	ContextHolder.context
ContextHolder.getContext	ContextHolder.GetContext
Fortsetzung auf nächster Seite...	

Entsprechungen zwischen Code-Elementen der Java-Implementation des SE-Manager und der automatischen Übersetzung nach C#

ContextHolder.setContext	ContextHolder.SetContext
IAbnahmeDienst	IAbnahmeDienst
IAbnahmeDienst.fuege-AbnahmeHinzu	IAbnahmeDienst.Fuege-AbnahmeHinzu
IAbnahmeDienst.gib-AbnahmezettelZuStudent	IAbnahmeDienst.Gib-AbnahmezettelZuStudent
IAbnahmeDienst.gib-RelevanteAufgaben	IAbnahmeDienst.Gib-RelevanteAufgaben
IAbnahmeDienst.gib-RelevanteAusnahmen	IAbnahmeDienst.Gib-RelevanteAusnahmen
IAbnahmeDienstCallback	IAbnahmeDienstCallback
IAbnahmeDienstCallback.erfolgreich-AbnahmeHinzugefuegt	IAbnahmeDienstCallback.Erfolgreich-AbnahmeHinzugefuegt
IAbnahmeDienstCallback.erstelle-AbnahmezettelAnsicht	IAbnahmeDienstCallback.Erstelle-AbnahmezettelAnsicht
IAbnahmeDienstCallback.fehler-BeiAbnahme	IAbnahmeDienstCallback.Fehler-BeiAbnahme
IAbnahmeDienstCallback.fehler-BeiAufgabenSuche	IAbnahmeDienstCallback.Fehler-BeiAufgabenSuche
IAbnahmeDienstCallback.fehler-BeiAusnahmenSuche	IAbnahmeDienstCallback.Fehler-BeiAusnahmenSuche
IAbnahmeDienstCallback.fehler-BeiZettelSuche	IAbnahmeDienstCallback.Fehler-BeiZettelSuche
IAbnahmeDienstCallback.fuege-AufgabenInAuswahlEin	IAbnahmeDienstCallback.Fuege-AufgabenInAuswahlEin
IAbnahmeDienstCallback.fuege-AusnahmeInAuswahlEin	IAbnahmeDienstCallback.Fuege-AusnahmeInAuswahlEin
IAnmeldeDienst	IAnmeldeDienst
IAnmeldeDienst.abmelden	IAnmeldeDienst.Abmelden
IAnmeldeDienst.anmelden	IAnmeldeDienst.Anmelden
IAnmeldeDienstCallback	IAnmeldeDienstCallback
IAnmeldeDienstCallback.erfolgreiche-Abmeldung	IAnmeldeDienstCallback.Erfolgreiche-Abmeldung
Fortsetzung auf nächster Seite...	

Entsprechungen zwischen Code-Elementen der Java-Implementation des SE-Manager und der automatischen Übersetzung nach C#

IAnmeldeDienstCallback.erfolgreiche-Anmeldung	IAnmeldeDienstCallback.Erfolgreiche-Anmeldung
IAnmeldeDienstCallback.fehler-BeiAbmeldung	IAnmeldeDienstCallback.Fehler-BeiAbmeldung
IAnmeldeDienstCallback.fehler-BeiAnmeldung	IAnmeldeDienstCallback.Fehler-BeiAnmeldung
ILaborDienst	ILaborDienst
ILaborDienst.gibAlleLabore	ILaborDienst.GibAlleLabore
ILaborDienstCallback	ILaborDienstCallback
ILaborDienstCallback.fehlerBeiLaboreSuche	ILaborDienstCallback.FehlerBeiLaboreSuche
ILaborDienstCallback.fuege-AlleLaboreInListeEin	ILaborDienstCallback.Fuege-AlleLaboreInListeEin
IPraesenzDienst.fuege-WeitereStudentenHinzu	IPraesenzDienst.Fuege-WeitereStudentenHinzu
IPraesenzDienst.gib-AnwesendeStudenten	IPraesenzDienst.Gib-AnwesendeStudenten
IPraesenzDienst.gib-StudentenOhneAnwesenheit	IPraesenzDienst.Gib-StudentenOhneAnwesenheit
IPraesenzDienstCallback	IPraesenzDienstCallback
IPraesenzDienstCallback.erstelle-StudentenListe	IPraesenzDienstCallback.Erstelle-StudentenListe
IPraesenzDienstCallback.fehler-BeiStudentenSuche	IPraesenzDienstCallback.Fehler-BeiStudentenSuche
IPraesenzDienstCallback.fehler-BeimHinzufuegen	IPraesenzDienstCallback.Fehler-BeimHinzufuegen
IPraesenzDienstCallback.studenten-ErfolgreichHinzugefuegt	IPraesenzDienstCallback.Studenten-ErfolgreichHinzugefuegt
IStudentenDienst	IStudentenDienst
IStudentenDienst.gibAlleStudenten	IStudentenDienst.GibAlleStudenten
IStudentenDienst.gibRelevante-StudentenFuerAufgabe	IStudentenDienst.GibRelevante-StudentenFuerAufgabe
IPraesenzDienst	IPraesenzDienst
Fortsetzung auf nächster Seite...	

Entsprechungen zwischen Code-Elementen der Java-Implementation des SE-Manager und der automatischen Übersetzung nach C#

IStudentenDienst.gibRelevante-StudentenFuerAusnahme	IStudentenDienst.GibRelevante-StudentenFuerAusnahme
IStudentenDienst.sucheStudent	IStudentenDienst.SucheStudent
IStudentenDienstCallback	IStudentenDienstCallback
IStudentenDienstCallback.erstelle-StudentenListe	IStudentenDienstCallback.Erstelle-StudentenListe
IStudentenDienstCallback.fehler-BeiStudentenSuche	IStudentenDienstCallback.Fehler-BeiStudentenSuche
IUebungsterminDienst	IUebungsterminDienst
IUebungsterminDienst.gib-AlleUebungstermine	IUebungsterminDienst.Gib-AlleUebungstermine
IUebungsterminDienstCallback	IUebungsterminDienstCallback
IUebungsterminDienstCallback.fehler-BeiUebungsterminSuche	IUebungsterminDienstCallback.Fehler-BeiUebungsterminSuche
IUebungsterminDienstCallback.fuege-AlleUebungstermineInListeEin	IUebungsterminDienstCallback.Fuege-AlleUebungstermineInListeEin
LaborAsyncTask	LaborAsyncTask
LaborAsyncTask._callback	LaborAsyncTask._callback
LaborAsyncTask._methode	LaborAsyncTask._methode
LaborAsyncTask._responseCode	LaborAsyncTask._responseCode
LaborAsyncTask._sslContext	LaborAsyncTask._sslContext
LaborAsyncTask._url	LaborAsyncTask._url
LaborAsyncTask._zielAdresse	LaborAsyncTask._zielAdresse
LaborAsyncTask.doInBackground	LaborAsyncTask.DoInBackground
LaborAsyncTask.onPostExecute	LaborAsyncTask.OnPostExecute
LaborAsyncTask.onPreExecute	LaborAsyncTask.OnPreExecute
LaborDienstImpl	LaborDienstImpl
LaborDienstImpl._alleLabore	LaborDienstImpl._alleLabore
LaborDienstImpl._context	LaborDienstImpl._context
LaborDienstImpl.gibAlleLabore	LaborDienstImpl.GibAlleLabore
Praesenz	Praesenz
Praesenz._labor	Praesenz._labor
Praesenz._uebungstermin	Praesenz._uebungstermin
Fortsetzung auf nächster Seite...	

Entsprechungen zwischen Code-Elementen der Java-Implementation des SE-Manager und der automatischen Übersetzung nach C#

Praesenz.equals	Praesenz.Equals
Praesenz.hashCode	Praesenz.GetHashCode
PraesenzAsyncTask	PraesenzAsyncTask
PraesenzAsyncTask._callback	PraesenzAsyncTask._callback
PraesenzAsyncTask._methoden	PraesenzAsyncTask._methoden
PraesenzAsyncTask._responseCodes	PraesenzAsyncTask._responseCodes
PraesenzAsyncTask._sslContext	PraesenzAsyncTask._sslContext
PraesenzAsyncTask._urls	PraesenzAsyncTask._urls
PraesenzAsyncTask._zielAdresse	PraesenzAsyncTask._zielAdresse
PraesenzAsyncTask.check-ResponseCodes	PraesenzAsyncTask.Check-ResponseCodes
PraesenzAsyncTask.do-InBackground	PraesenzAsyncTask.Do-InBackground
PraesenzAsyncTask.on-PostExecute	PraesenzAsyncTask.On-PostExecute
PraesenzAsyncTask.on-PreExecute	PraesenzAsyncTask.On-PreExecute
PraesenzDienstImpl	PraesenzDienstImpl
PraesenzDienstImpl._context	PraesenzDienstImpl._context
PraesenzDienstImpl.fuege-WeitereStudentenHinzu	PraesenzDienstImpl.Fuege-WeitereStudentenHinzu
PraesenzDienstImpl.gib-AnwesendeStudenten	PraesenzDienstImpl.Gib-AnwesendeStudenten
PraesenzDienstImpl.gib-StudentenOhneAnwesenheit	PraesenzDienstImpl.Gib-StudentenOhneAnwesenheit
Smiley	Smiley
Smiley._zustand	Smiley._zustand
Smiley.equals	Smiley.Equals
Smiley.hashCode	Smiley.GetHashCode
Student	Student
Student._labor	Student._labor
Student._matrikelNr	Student._matrikelNr
Student._nachname	Student._nachname
Fortsetzung auf nächster Seite...	

Entsprechungen zwischen Code-Elementen der Java-Implementation des SE-Manager und der automatischen Übersetzung nach C#

Student._studiengang	Student._studiengang
Student._vorname	Student._vorname
Student.equals	Student.Equals
Student.hashCode	Student.GetHashCode
StudentenAsyncTask	StudentenAsyncTask
StudentenAsyncTask._callback	StudentenAsyncTask._callback
StudentenAsyncTask._methode	StudentenAsyncTask._methode
StudentenAsyncTask._responseCode	StudentenAsyncTask._responseCode
StudentenAsyncTask._sslContext	StudentenAsyncTask._sslContext
StudentenAsyncTask._url	StudentenAsyncTask._url
StudentenAsyncTask._zielAdresse	StudentenAsyncTask._zielAdresse
StudentenAsyncTask.doInBackground	StudentenAsyncTask.DoInBackground
StudentenAsyncTask.onPostExecute	StudentenAsyncTask.OnPostExecute
StudentenAsyncTask.onPreExecute	StudentenAsyncTask.OnPreExecute
StudentenDienstImpl	StudentenDienstImpl
StudentenDienstImpl._alleStudentenModel	StudentenDienstImpl._alleStudentenModel
StudentenDienstImpl._context	StudentenDienstImpl._context
StudentenDienstImpl.gibAlleStudenten	StudentenDienstImpl.GibAlleStudenten
StudentenDienstImpl.gibRelevante-StudentenFuerAufgabe	StudentenDienstImpl.GibRelevante-StudentenFuerAufgabe
StudentenDienstImpl.gibRelevante-StudentenFuerAusnahme	StudentenDienstImpl.GibRelevante-StudentenFuerAusnahme
StudentenDienstImpl.sucheStudent	StudentenDienstImpl.SucheStudent
Uebungsblatt	Uebungsblatt
Uebungsblatt._aufgaben	Uebungsblatt._aufgaben
Uebungsblatt._blattNr	Uebungsblatt._blattNr
Uebungsblatt._frist	Uebungsblatt._frist
Uebungsblatt.equals	Uebungsblatt.Equals
Uebungsblatt.hashCode	Uebungsblatt.GetHashCode
Uebungsblatt.list2String	Uebungsblatt.List2String
Uebungstermin._bezeichnung	Uebungstermin._bezeichnung
Uebungstermin._termin	Uebungstermin._termin
Uebungstermin	Uebungstermin
Fortsetzung auf nächster Seite...	

Entsprechungen zwischen Code-Elementen der Java-Implementation des SE-Manager und der automatischen Übersetzung nach C#

Uebungstermin.equals	Uebungstermin.Equals
Uebungstermin.hashCode	Uebungstermin.GetHashCode
UebungsterminAsyncTask	UebungsterminAsyncTask
UebungsterminAsyncTask._callback	UebungsterminAsyncTask._callback
UebungsterminAsyncTask._methode	UebungsterminAsyncTask._methode
UebungsterminAsyncTask._responseCode	UebungsterminAsyncTask._responseCode
UebungsterminAsyncTask._sslContext	UebungsterminAsyncTask._sslContext
UebungsterminAsyncTask._url	UebungsterminAsyncTask._url
UebungsterminAsyncTask._zielAdresse	UebungsterminAsyncTask._zielAdresse
UebungsterminAsyncTask.do-InBackground	UebungsterminAsyncTask.Do-InBackground
UebungsterminAsyncTask.on-PostExecute	UebungsterminAsyncTask.On-PostExecute
UebungsterminAsyncTask.on-PreExecute	UebungsterminAsyncTask.On-PreExecute
UebungsterminDienstImpl	UebungsterminDienstImpl
UebungsterminDienstImpl._alle-Uebungstermine	UebungsterminDienstImpl._alle-Uebungstermine
UebungsterminDienstImpl._context	UebungsterminDienstImpl._context
UebungsterminDienstImpl.gibAlle-Uebungstermine	UebungsterminDienstImpl.GibAlle-Uebungstermine
$1_Authenticator	$1__Authenticator_32
$1_Authenticator.get-PasswordAuthentication	$1__Authenticator_32.Get-PasswordAuthentication
$2_Authenticator	$2__Authenticator_47
$2_Authenticator.get-PasswordAuthentication	$2__Authenticator_47.Get-PasswordAuthentication

Entsprechungen zwischen Code-Elementen der Java-Implementation des SE-Manager und der automatischen Übersetzung nach C#

A.6 Aufwandsschätzung für die Portierung der mobilen App *Metropole des Wissens*

App	Min	Real	Max	Comment
News-Übersicht	2	3	4	Inkl. REST call, Layout
News-Details	1	1	2	Inkl. REST call, Layout
Veranstaltungs-Übersicht	2	2	3	Inkl. REST call, Layout. Design der Categories evtl. Aufwändig (kann man das wiederverwenden?).
Veranstaltungs-Details	1	2	3	Das Custom Design könnte deutlich Zeit einnehmen. Keine Erfahrung mit den Kalendereinträgen.
Kategorie-Übersicht (Liste)	1	1	1	Sehe hier keine technische Herausforderung.
Kartenansicht ink;. Institut Details	1	2	2	Eventuell nimmt das Anpassen der Usability Zeit ein.
Impressum	1	1	1	Da statisch, eher geringer Aufwand. Vermutlich maximal 2-3 Stunden.
HOOU-Integration	1	1	1	Das sollte innerhalb von 1-2 Stunden machbar sein.
Menüführung	1	1	1	Der Android Standard wandelt sich hier oft. Es sollte sich konkret auf die Art geeinigt werden.
Hands on Tests, Fehlerbehandlungen integrieren	1	2	3	Sehr variabel und schwer einzuschätzen. Welche Android Geräte und Versionen müssen unterstützt werden?

Backend	Min	Real	Max	Comment
				Da es sich um eine REST API handelt sollte es hier kein extra Aufwand geben.

Overall	Min	Real	Max	Comment
	12	**16**	**21**	

Abbildung A.2: Aufwandsschätzung der ursprünglichen Entwickler für die Portierung der mobilen App *Metropole des Wissens* in Personentagen

A.7 Notwendige Änderungen zur Erweiterung der App *Metropole des Wissens*

Im Rahmen der Evaluation wurde die App *Metropole des Wissens* erweitert, um dem Nutzer die maximale Teilnehmeranzahl einer Veranstaltung anzuzeigen. Tabelle A.6 erfasst die dazu vorgenommenen Änderungen der Ausgangs und Zielimplementation. Die erste Spalte der Tabelle enthält die Bezeichner der betroffenen Typen. Lediglich in der zweiten Zeile unterscheidet sich der Bezeichner des Typs in der iOS-Implementation vom Bezeichner seiner Entsprechung in der Android-Implementation. In der zweiten Spalte ist jeweils eine kurze Beschreibung des betroffenen Typs notiert. Die dritte Spalte erfasst die notwendigen Änderungen.

Name des Typs in der Android- und iOS-Implementation	**Beschreibung des Typs**	**Beschreibung der notwendigen Änderung**
`EventModel`	Abstraktes Modell von Veranstaltungen	Hinzufügen des Attributs `maxParticipants`
`EventDetailModel`	Konkretes Modell von Veranstaltungen	Hinzufügen des Attributs `maxParticipants`
`JSONKeys.Events`(Android) `JSONKeysEvent`(iOS)	Sammlung von Konstanten, die die Attributnamen der JSON-Repräsentation von Veranstaltungen enthalten	Hinzufügen einer Konstante zur Identifikation des zusätzlichen Attributs
`EventTextCell`	User-Interface-Klasse zur Darstellung von Veranstaltungen in Listen	Darstellung der maximalen Teilnehmerzahl im entsprechenden Label
`EventTextImageCell`	User-Interface-Klasse zur Darstellung von Veranstaltungen in Listen	Darstellung der maximalen Teilnehmerzahl im entsprechenden Label
`EventImageCell`	User-Interface-Klasse zur Darstellung von Veranstaltungen in Listen	Darstellung der maximalen Teilnehmerzahl im entsprechenden Label
`EventDetailHeaderCell`	User-Interface-Klasse zur Darstellung von Informationen zu Veranstaltungen in der Detail-Ansicht einer Veranstaltung	Darstellung der maximalen Teilnehmerzahl im entsprechenden Label

Tabelle A.6: Von der Änderung betroffene Typdefinitionen

Erklärung der Urheberschaft

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Hamburg, 9. April 2020 Unterschrift