Identifikation von Technologie-Features in existierenden Softwaresystemen

Masterarbeit

Tobias Fechner
M.Sc. Informatik

Matrikelnummer
6315945

Gutachter
Prof. Dr.-Ing. Matthias Riebisch
Prof. Dr. Tilo Böhmann

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
Arbeitsbereich SWK

Eingereicht zum 20. Oktober 2016

Abstrakt

Während der Softwareevolution kommt es im Zuge neuer Anforderungen und natürlicher Erosion häufig zu Anwendungsfällen, in denen Technologien ausgetauscht werden müssen. Bereits getroffene Architekturentscheidungen, insbesondere Entscheidungen für oder gegen bestimmte Technologien, spielen dabei eine entscheidende Rolle. Sie müssen revidiert oder überdacht werden, ohne die Funktionalität des Systems zu beeinträchtigen.

Schon im klassischen Entscheidungsprozess der Softwarearchitektur bilden eingesetzte Technologien wichtige Constraints bei der Suche nach möglichen Alternativen. Dieses Konzept kann verfeinert werden, indem man die Features einzelner Technologien mit einbezieht. Um auf Grundlage dieser Funktionalitäten Constraints besser einschätzen und sinnvolle Alternativen ermitteln zu können, muss bekannt sein, welche Features einer eingesetzten Technologie das betrachtete System nutzt. Diese müssen als qualifizierte Alternativen über ähnliche Funktionalitäten verfügen, um den Aufwand für Änderungen möglichst gering zu halten.

Hauptfokus liegt dabei auf der Identifikation von Merkmalen zur Extraktion und Erkennung von Technologie-Features. Im Weiteren werden ebenfalls das Finden von alternativen Technologien sowie möglicher Entscheidungen thematisiert.

Anders als viele verwandte Ansätze konzentriert sich das vorgestellte Konzept nicht auf die statische Modellierung von Softwaresystemen, sondern explizit auf bereits existierende Systeme und deren Quellcode. Die wenig abstrakte Ebene konkreter Technologie-Features wird von der Softwarearchitektur bisweilen nur selten betrachtet. Um jedoch sinnvolle alternative Technologien für bereits existierende Systeme zu bestimmen, ist eine Betrachtung dieser Ebene durchaus sinnvoll. Ein Einblick in diese niedrigere Abstraktionsebene kann als Ergebnis dieser Arbeit fundiertere Entscheidungen im Entscheidungsprozess ermöglichen.

Diese Arbeit betrachtet ein Konzept zur Klassifizierung von Technologie-Features und deren Erkennung in existierenden Softwaresystemen. Anhand eines existierenden Systems und Anwendungsfalls wird beispielhaft erläutert, welche Relevanz die Problematik nach sich zieht, und wie sie potentiell zu lösen ist. Neben der Erkennung von Technologie-Features wird ebenfalls vorgestellt, wie sich anhand dieser sinnvolle Alternativen für das betrachtete System finden lassen. Ein Prototyp zur automatisierten Erkennung und Bestimmung von Technologie-Features und Alternativen zeigt, wie sich die Konzepte in der Praxis umsetzen lassen. Im Rahmen einer Analyse wird das Konzept auf Basis weiterer Systeme evaluiert.

Danksagung

An dieser Stelle möchte ich Dank an alle aussprechen, die mich nicht nur bei der Bearbeitung dieser Arbeit, sondern während meines gesamten Studiums unterstützt und ertragen haben.
Großer Dank gebührt meinen Eltern. Ohne eure finanzielle und mentale Unterstützung und Bekräftigungen wäre ich nie so weit gekommen.
Weiterer Dank gilt meinen Betreuern des Arbeitsbereichs SWK, Sebastian Gerdes und Mohamed Soliman. Sie haben mich maßgeblich bei der Themenfindung und Bearbeitung der Arbeit unterstützt und ermutigt.
Ebenso gilt mein Dank Professor Riebisch und Professor Böhmann für ihr Gutachten dieser Arbeit.
Bedanken möchte ich mich ebenfalls bei meinen Kommilitonen und guten Freunden, David und Flo. Ohne eure Freundschaft hätte ich das Studium vermutlich schon vor Jahren aufgegeben.

Aufgabenstellung Das Ziel ist die Unterstützung des Architekten beim Treffen von Technologie-Entscheidungen während der Softwareevolution. Zur Umsetzung neuer Anforderungen oder der Behebung von Fehlern stehen einem Architekten in der Regel mehrere technologische Lösungen zur Verfügung, die je nach Anforderung entsprechende Vor- und Nachteile besitzen. Da die Menge an potentiellen Lösungen, ihren Features und Eigenschaften in der Regel nicht stets parat ist, greift der Architekt auf vorhandenes Architekturwissen, z.B. aus Online-Communities, zurück. Damit dem Architekten nun potentielle Lösungsalternativen vorgeschlagen werden können, muss zum Einen das vorliegende System analysiert und zum Anderen das Architekturwissen auf geeignete Lösungen entsprechend der neuen Anforderung durchsucht werden.
Aktuelle Arbeiten befassen sich derzeit mit der Extraktion von Architekturwissen aus Online-Communities, die die Schaffung einer strukturierten, durchsuchbaren Wissensbasis zum Ziel haben. Um den Architekten nun beim Treffen von Technologie-Entscheidungen zu unterstützen, muss das existierende System analysiert und dabei die eingesetzten Technologien und Features identifiziert werden. Mit diesem Wissen über das existierende System und dem Architekturwissen lassen sich nun entsprechende Lösungsalternativen zur Entscheidungsunterstützung vorschlagen.

Im Rahmen der Masterarbeit sollen nachfolgende Aufgaben bearbeitet werden:

1. Architekturwissen eines existierenden Systems für vorgegebene Datenstruktur erfassen und aufbereiten
    a) Technologien mit ihren vorhandenen Features identifizieren
    b) Architekturwissen mit Vor- und Nachteilen der Technologie und Features erfassen anhand gegebener Anforderungen
    c) Indikatoren zur Identifizierung von Features einer Technologie erfassen
2. Ermittlung von Lösungsalternativen (z.B. aus Online-Foren) für eine gegebene, neue Anforderung (Analog zu 1.)
3. Identifikation von Technologie-Features des existierenden Systems mit Hilfe des Architekturwissens
    a) Bestimmung der tatsächlich verwendeten Features für gesamtes System
    b) Komponenten- bzw. Paketbezogene Bestimmung der Features
    c) Vorschlag möglicher Lösungsalternativen zur Umsetzung der neuen Anforderung

In den einzelnen Schritten wird die vorgegebene Datenstruktur validiert und ggf. erweitert bzw. optimiert.
Vorbedingungen und Vorgaben:

- Existierendes System
- Neue Anforderung(en) / Change Request
- Datenstruktur / Metamodell für Architekturwissen

Inhaltsverzeichnis

Kapitel 1

Einleitung

> *„During the lifetime of a project it often happens that the state of the art evolves as new technologies emerge“*
>
> [Nowak u. a., 2010]

Eine konstante Evolution von Softwaresystemen und Technologien hat häufige Änderungen zur Folge. Anforderungen werden verfeinert, ergänzt oder können aufgrund von problematischem Verhalten nicht mehr erfüllt werden. Der Einsatz von Technologien stellt dabei im Prozess eine wichtige Erleichterung durch Nutzung vorgefertigter Bausteine dar.

Diese bilden in Folge wichtige Constraints im Prozess der Softwarearchitektur und -entwicklung. Eingesetzte Technologien haben aufgrund von Abhängigkeiten und technischen Voraussetzungen meist Einschränkungen bei der Auswahl zukünftig eingesetzter Technologien zur Folge. Hieraus ergibt sich unter Umständen eine starke Einschränkung des potentiellen Lösungsraums. Die Folgen für den Prozess der iterativen Softwareentwicklung bilden dabei den Schwerpunkt der Betrachtungen.

Genutzte Funktionalitäten eingesetzter Technologien, nachfolgend als Technologie-Features bezeichnet, werden auf dieser Ebene weitaus seltener betrachtet. Kenntnis über deren Einsatz kann den Prozess jedoch potentiell bereichern und somit zukünftige Entwurfsentscheidungen beeinflussen und deren Qualität verbessern.

Die Weiterentwicklung existierender Softwaresysteme macht häufig ein Austauschen von Technologien sowie ihrer Funktionalitäten erforderlich. Hervorzuheben ist hier eine klare Trennung zwischen den Funktionalitäten eines Softwaresystems zur Erfüllung von Anforderungen sowie den hier fokussierten technischen Bausteinen einer Technologie. Beide Ebenen sind miteinander verbunden. In Folge lässt sich anhand diesen Umstandes jedoch eine klare Trennung zweier Abstraktionsebenen feststellen.

Der konkrete Kontext ihrer Nutzung bildet dabei für den Tausch eine wichtige Komponente: Ohne die Funktionalität von Systemen zu beeinflussen müssen bestehende Technologien ausgetauscht werden. Kenntnis über den Einsatz von Funktionalitäten kann bei der Suche nach alternativen Technologien einen wichtigen Bestandteil darstellen.

Methoden des Reverse Engineering beschäftigen sich oft mit der Rekonstruktion von Softwarearchitektur und Entwurfsentscheidungen. Die Bestimmung genutzter Technologie-Features wird dabei nur oberflächlich betrachtet. Auf Grundlage von Schnittstellen und Syntax entsprechender Features lässt sich eine Formalisierung ihrer Nutzung vornehmen. Die Ausprägung charakteristischer Merkmale der Nutzung von Technologie-Features bildet die Grundlage des in dieser Arbeit vorgestellten Konzeptes. Dies ermöglicht nachfolgend eine gezielte Modellierung von Features, Technologien sowie eine anschließende Bestimmung von Alternativen. Vorteil dieser Bestimmung ist eine Reduktion des Lösungsraums und Fokussierung auf wesentliche Bestandteile bereits getroffener Entwurfsentscheidungen. Dies ermöglicht fundierte zukünftige Entscheidungen.

Der Fokus dieser Arbeit liegt auf der Formalisierung und Bestimmung von Technologie-Features und deren Nutzung. Unter Zuhilfenahme von Architekturwissen beschreibt ein Modell Eigenschaften und Beziehungen relevanter Elemente im Prozess der Softwarearchitektur. Ein technischer Ansatz zur Bestimmung auf Grundlage existierenden Quellcodes eines Anwendungssystems ist das zentrale Merkmale des Konzepts.

Diese Arbeit ist wie folgt gegliedert: Zunächst beschreibt Kapitel 2 Hintergründe und theoretische Grundlagen zur Fundierung des Konzeptes. Anschließend wird anhand von Problemstellung und einem konstruierten Anwendungsfall die Relevanz der Thematik aufgezeigt.

Kapitel 3 beschreibt das Grundkonzept zur Lösung der vorliegenden Problematik. Es werden ebenfalls von der Aufgabenstellung vorgegebene Annahmen aufgeführt, unter denen diese Arbeit zu betrachten ist. Weiterhin wird die potentielle Lösung in den Prozess der Softwarearchitektur und -evolution eingeordnet.

Den Kern der Arbeit bildet die Ausgestaltung des Konzeptes in Kapitel 4. Zentrale Formalisierungen, das exakte Vorgehen sowie Schwerpunkte werden erläutert und anhand des beschriebenen Anwendungsfalls illustriert.

Eine prototypische Implementation des Konzeptes zeigt in Kapitel 5, wie der Ansatz in der Praxis umzusetzen ist. Ebenso betrachtet wird dessen mögliche Unterstützung des Architektur-Prozesses.

Kapitel 6 evaluiert im Anschluss sowohl Konzept als auch prototypische Umsetzung. Der Ansatz findet Anwendung anhand weiterer Anwendungssysteme und validiert anhand seiner Ergebnisse das Konzept im gegebenen Rahmen. Weiterhin werden Schwächen sowie Einschränkungen des Ansatzes auf Grundlage der Aufgabenstellung herausgearbeitet. Ebenso müssen Vorteile und mögliche Einsatzbereiches des Konzeptes betrachtet werden. Potentielle aufbauende Forschungsthemen werden ebenso vorgestellt.

Kapitel 7 schließt die Arbeit konzeptionell ab und fasst wichtige Erkenntnisse von Formalisierung, Extraktion und deren Einfluss auf die Entscheidungsfindung zusammen.

Kapitel 2

Motivation

„Good software quality does not come for free“

[Nowak u. a., 2010]

Die zunehmende Komplexität von Softwaresystemen macht den Einsatz einer Softwarearchitektur mehr und mehr zu einem festen Bestandteil des Entwicklungsprozesses. Sowohl in Industrie als auch Forschung trifft das Konzept in den letzten Jahren auf große Akzeptanz [Perry u. Wolf, 1992] [Jansen u. Bosch, 2005] [Shaw u. Clements, 2006].

Ein beachtlicher Aufwand wird zwecks Gestaltung und Ausarbeitung von Softwarearchitekturen betrieben. Dieser soll langfristig die Softwarequalität verbessern und somit die Evolution der Systeme ermöglichen.

2.1 Grundlagen

Der Begriff der Softwarearchitektur selbst sowie dessen Definition haben sich im Laufe der letzten Jahre stark gewandelt, wenngleich keine allgemeingültige Definition existiert.

„There is no general agreement of what a software architecture is and what it is not“

[van der Ven u. a., 2006]

Softwarearchitektur [Garlan u. Shaw, 1994] definieren die Softwarearchitektur als die Abstraktion eines Softwaresystems, welches sich aus mehreren funktionalen Komponenten zusammensetzt. Diese Komponenten sind durch Konnektoren verbunden, welche deren Verbindungen untereinander beschreiben.
In [Bass u. a., 2012] wird die Softwarearchitektur als Menge von Strukturen, deren Beziehungen untereinander, sowie den Hintergründen dieser Beziehungen betrachtet. Dies schließt, anders als in [Garlan u. Shaw, 1994] auch explizit die Betrachtung der Motivation hinter den umgesetzten Strukturen ein.
Bezieht man die Hintergründe für den Einsatz von Architektur-Elementen und ihre Beziehungen untereinander mit ein, betrachtet [Taylor u. a., 2009] die Softwarearchitektur als eine Menge von vorrangig Entwurfsentscheidungen. Jeder Teil des Systems, jede Komponente ist bewusst ausgewählt worden, um ein Anwendungssystem

zu bilden.

Das IEEE stellt mit dem Standard 1471 [IEEE, 2000] eine Grundlage für die Beschreibung und Modellierung 'Software-intensiver Systeme' bereit. Die konsistente Modellierung und explizite Darstellung von Softwarearchitekturen birgt im Hinblick auf die langfristige Qualität des System in der Theorie viele Vorteile [Kruchten, 1995] [Hofmeister u. a., 2000]. Die Qualität von Software lässt sich etwa durch Elemente des ISO-Standards 25010 [ISO/IEC, 2011] beschreiben.

Bekannte Ansätze zur Dokumentation und Beschreibung beinhalten etwa das *4+1 Sichten*-Modell [Kruchten, 1995] oder [Clements u. a., 2002]. Einzelne Sichten fokussieren dabei auf die Beschreibung des Systems für unterschiedliche Zielgruppen, etwa Softwarearchitekten, Entwickler sowie weitere Stakeholder. Sie versuchen somit, die Komplexität der Softwarearchitektur beherrschbar zu gestalten und eine gewisse Ebene der Abstraktion einzuführen.

Die Evolution von Softwaresystemen spielt eine zentrale Rolle. Systeme werden häufiger gewartet und weiterentwickelt, als von Grund auf neu konstruiert.

> *„Because of the existing systems that are already in place [...], little or no systems are developed based on a 'greenfield' situation“*
>
> [Farenhorst, 2006]

In der Wissenschaft gehen viele Ansätze zwar vom Greenfield-Szenario aus, die Praxis weist aber häufig eine gegenteilige Situation auf. Die Modellierung und Dokumentation von Softwaresystemen soll in diesem Kontext die Wartbarkeit der Systeme erhöhen und deren langfristige Evolution ermöglichen [Hofmeister u. a., 2000].

Entwurfsentscheidungen Entwurfsentscheidungen beschreiben in diesem Kontext alle Entscheidungen, welche die Architektur eines Systems charakterisieren [Jansen u. Bosch, 2005]. Die Evolution von Softwaresystemen ist dabei stark mit den getroffenen Entscheidungen während des Prozesses verbunden [Jansen u. Bosch, 2004]. Entwurfsentscheidungen beeinflussen maßgeblich die Zukunft eines Softwaresystems und bilden Einschränkungen und Möglichkeiten für die weitere Entwicklung. [Jansen u. Bosch, 2004] sowie [Jansen u. a., 2007] gehen auf möglichen Tool-Support für die Evolution von Softwarearchitekturen unter Berücksichtigung von Entwurfsentscheidungen ein. Bewusst betrachtet wird das Konzept der Softwarearchitektur als Menge von Entwurfsentscheidungen in [Jansen u. Bosch, 2005]. Diese Entwurfsentscheidungen sind bereits implizit in der Architektur enthalten.

> *„[...] decisions are critically important to all parts and subsequent design (or even lifecycle) phases of the system“*
>
> [Shaw u. Clements, 2006]

Diese Entscheidungen können dabei die Auswahl von Technologien [LaToza u. a., 2013], Architektur- und Entwurfsmuster sowie die Anordnung von Komponenten und deren Verbindung beinhalten. Jede Entscheidung wird mit einer Begründung (*Rationale*) getroffen und lässt sich im besten Fall auf eine Anforderung zurückführen (*Traceabilty*).

> *„In creating software systems we make choices throughout the entire development process with certain intent in mind. We select objects and processes from the problem domain and exclude others because we have a certain intent as to the focus of the problem we want to solve“*
>
> [Perry u. Grisham, 2006]

Dokumentation und explizite Betrachtung von Entwurfsentscheidungen bilden einen zentralen Bestandteil der Softwarearchitektur. Sie bieten eine verbesserte Eingrenzung des Lösungsraumes, Traceabilty sowie Möglichkeiten zur Analyse der Systeme [Jansen u. Bosch, 2005]. Die Einbindung dieser Entscheidungen und deren explizite Betrachtung im Prozess der Softwarearchitektur verbessert dessen Verständlichkeit. Somit können Architektur und Hintergründe zusammengeführt sowie formalisiert werden [van der Ven u. a., 2006].

[Jansen u. Bosch, 2005] beschreiben ein Metamodell zur Formalisierung von Entwurfsentscheidungen[1]. Dieses definiert Entwurfsentscheidungen als Auswahl einer Lösung aus einer Menge möglicher Entscheidungen. Diese Auswahl dient der Lösung eines bestimmten Problems. Dieser liegen sowohl eine Ursache als auch ihre Rationale zugrunde. Getroffene Entwurfsentscheidungen resultieren in Änderungen an der Architektur eines Softwaresystems.

[Kruchten, 2004] stellt eine Klassifizierung von Entwurfsentscheidungen auf. Diese beinhaltet nicht alleine Entscheidungen für einen bestimmten Aspekt der Architektur (*Existance Decisions*), sondern auch die bewussten Entscheidungen gegen gewisse Merkmale des Systems (*Ban* oder *Non-Existance Decisions*). Ebenso beschrieben werden Entscheidungen für Eigenschaften der gewählten Elemente (*Property Decisions*) sowie prozessbezogene Entscheidungen (*Executive Decisions*).

Die alleinige Dokumentation der resultierenden Softwarearchitektur ist nicht ausreichend. Das Wissen um die getroffenen Entwurfsentscheidungen und die zugrundeliegende Rationale geht verloren [Shaw u. Clements, 2006]. Bei der Wartung und Evolution der Systeme spielen nach [Jansen u. Bosch, 2005] weitere Faktoren eine zentrale Rolle:

1. Entwurfsentscheidungen sind häufig voneinander abhängig und beeinflussen sich gegenseitig.

2. Vorgaben und Einschränkungen beim Entwurf werden verletzt.

3. Veraltete Entwurfsentscheidungen werden nicht entfernt und verbleiben im System.

Aus diesen Sachverhalten resultieren häufig Probleme in Form hoher Wartungskosten und einem erhöhten Aufwand bei Änderungen [Shaw u. Clements, 2006]. Entsprechende Systeme neigen dazu, zu erodieren [Jansen u. Bosch, 2005], auch Software-Erosion [van Gurp u. Bosch, 2002].

[1] Siehe Anhang, Abbildung A.1, Seite II

Architekturwissen Als Grundlage für viele Entwurfsentscheidungen dient potentiell eine (abstrakte) Sammlung von wiederverwendbaren Assets, das sogenannte Architekturwissen. Die Auswahl von Lösungen aus diesen Assets bildet den potentiellen Lösungsraum.

> *„In the ideal situation, each software architect would have unlimited knowledge about the problem and a complete set of architecture alternatives to choose from“*
>
> [Nowak u. a., 2010]

Entwurfsentscheidungen sind dabei wesentlicher Bestandteil des Architekturwissens [Jansen u. a., 2007]. Aussagen darüber, welche Inhalte sinnvollerweise die Grundlage des Architekturwissens bilden sollten, sind bislang nicht eindeutig. Das Konzept geht nach [Posch u. a., 2007] davon aus, dass Aufwand und Pflege der Wissensbasis inkrementell erfolgen müssen. Führt man diese Metapher des Architekturwissens als 'Werkzeugkasten' eines Handwerkers aus, müssen neue Inhalte nach erstmaliger Benutzung dokumentiert, für die Wiederverwendbarkeit nutzbar gemacht und der Wissensbasis hinzugefügt werden. Die Pflege des Architekturwissens wird dabei häufig dem Aufgabenbereich des Softwarearchitekten zugeordnet.

Obwohl Studien die möglichen Inhalte abzuschätzen versuchen [Falessi, 2007], bleibt es häufig bei einer Formalisierung. Einige Ansätze versuchen sich bereits am Aufbau und dem Befüllen entsprechender Strukturen [Liang u. Avgeriou, 2009] [Jansen u. Bosch, 2004] [Babar u. Gorton, 2007].

[Posch u. a., 2007] teilt die Inhalte des Architekturwissens in zwei Kategorien ein. Die erste Kategorie umfasst Lösungsvorlagen und -methoden, die für eine Nutzung im Rahmen von Entwurfsentscheidungen adaptiert und angepasst werden müssen. Dazu gehören Architektur- und Entwurfsmuster [Bass u. a., 2012] [Gamma u. a., 1995], Taktiken [Bass u. a., 2012] sowie Refactorings [Kerievsky, 2005]. In der zweiten Kategorie finden sich Technologien und Werkzeuge, welche in Form vorgefertigter Bausteine als Entwurfsentscheidung eingesetzt werden können. Diese umfassen neben Betriebssystemen, Programmiersprachen auch Standardbibliotheken, Commercial off-the-shelf (COTS), sowie Frameworks. Weitere Inhalte dieser Kategorie können Werkzeuge für Entwicklung und Design darstellen, etwa Tools zur Einhaltung und Rekonstruktion von Softwarearchitektur.

Um Architekturwissen zu formalisieren, existieren mehrere Ansätze in der Wissenschaft [Farenhorst u. de Boer, 2006] [de Boer u. a., 2007] [Babar u. a., 2007] [Babar u. a., 2009]. Auf Basis von Metamodellen sollen einheitliche Modellierungen den Wissenstransfer sicherstellen und somit die angestrebte Wissensbasis bereitstellen. Abbildung A.3[2] zeigt das in [de Boer u. a., 2007] vorgeschlagene Metamodell.

Existiert eine einheitliche Struktur zur Formalisierung des Architekturwissen, muss diese Struktur mit entsprechenden Assets befüllt werden, um später als Grundlage für Entwurfsentscheidungen dienen zu können. Das Befüllen des Architekturwissens ist vornehmlich ein semantisches Problem. Diese Problematik ist, zumindest in der

[2] Siehe Anhang, Seite IV

Praxis, inkrementell zu lösen [Posch u. a., 2007]. Die Wissenschaft befasst sich etwa mit Ansätzen für die automatisierte Extraktion von Architekturwissen aus existierenden Quellen und deren Kategorisierung [Soliman u. a., 2016].

Man unterscheidet bei der Verwaltung von Architekturwissen gemeinhin zwei Ansätze: Codification und Personalization [Hansen u. a., 1999]. Codification beschreibt den Ansatz, eine allgemeingültige, allen Interessenten zur Verfügung gestellte Wissensbasis zu schaffen. Im Gegensatz dazu versucht der Ansatz der Personalization den Aufbau von Architekturwissen als individuelle Aufgabe. [Hansen u. a., 1999] beschreibt diese Strategie als Individual-Aufgabe von einzelnen Unternehmen oder Softwarearchitekten für die eigene Nutzung. Zwangsläufig entstehen so nebenläufig mehrere Wissensbasen, die der Allgemeinheit nicht zur Verfügung gestellt werden. Ziel der Wissenschaft und Praxis, so [Hansen u. a., 1999], sollte der Ansatz der Codification sein. Problem ist jedoch die mangelnde Motivation der Praxis im Hinblick auf die Sinnhaftigkeit einer gemeinsamen Wissensbasis.

Zwischen den Elementen innerhalb der Wissensbasis existieren Abhängigkeiten, welche die semantischen Zusammenhänge der Elemente abbilden [Kruchten, 2004]. Einzelne Technologien können einander ausschließen, unterstützen oder den Einsatz anderer Assets hemmen. Sie bilden zwangsläufig Einschränkungen bei der Auswahl von Assets aus der Wissensbasis.

Constraints Somit sind zumeist nicht alle Bestandteile des Architekturwissens sinnvoll und einsetzbar. Einschränkungen, welche die Auswahl von Assets maßgeblich eingrenzen, werden als Constraints beschrieben.

> *„[A] constraint is a property of, or assertion about a system or one of its parts, the violation of which will render the system unacceptable to one or more stakeholders“*
>
> [Medvidovic u. Taylor, 2000] (nach [Clements, 1996])

Sie bilden, neben den Entwurfsentscheidungen, die wichtigsten Elemente bei der Evolution von Softwaresystemen [Tang u. van Vliet, 2009]. Studien zeigen die Praxisrelevanz von Constraints auf:

> *„88% of the respondents perceived design constraints as highly important and reported that they are used very often in architecture projects“*
>
> [v. d. Berg u. a., 2009]

Im Fall von Entwurfsentscheidungen beschreiben Constraints eine Einschränkung des Lösungsraumes im Bezug auf die möglichen Lösungen für eine zu treffende Entwurfsentscheidung:

> *„A limiting condition that a design concern imposes upon the outcome of a design decision“*
>
> [v. d. Berg u. a., 2009]

[Tyree u. Akerman, 2005] beschreiben Constraints als Ergebnis einer Entwurfsentscheidung mit Einfluss auf zukünftige Entwurfsentscheidungen. Folgt man der Definition nach [Kruchten, 2004], stellen Constraints eine negative Entscheidung im

Bezug auf ein System dar, bei der eine bestimmte Eigenschaft bewusst nicht ausgeprägt ist, etwa eine bestimmte Technologie nicht eingesetzt wird.

Viele Faktoren können den Ursprung von Constraints bilden und Einfluss auf Entwurfsentscheidungen haben:

> *„Design constraints can come from functional requirements, quality requirements, business and IT strategy, system goals and industry standards etc. Other constraints are project resources (budget, schedule, team size, available knowledge, etc.), contractual obligations, bargaining power, resistance to change and political pressure.“*
>
> [v. d. Berg u. a., 2009]

Somit sind nicht nur Anforderungen und Qualitätsziele, sondern auch prozessbezogene Faktoren zu betrachten. Existierende Systeme und bereits getroffene Entscheidungen bilden entsprechend ebenfalls eine Quelle. Entwurfsentscheidungen, die im aktuellen Prozess getroffen werden, bilden Constraints für zukünftige Entscheidungen. Abbildung 2.1 zeigt die Relevanz von Entwurfsentscheidungen im Prozess. Einflüsse auf ein existierendes Systeme (etwa neue Anforderungen) erfordern neue Entwurfsentscheidungen und in Folge erweiterte Dokumentation, Artefakte sowie Verantwortlichkeiten für diese Anpassungen. Beeinflusst werden Entscheidungen unter anderem auch von Stakeholdern.

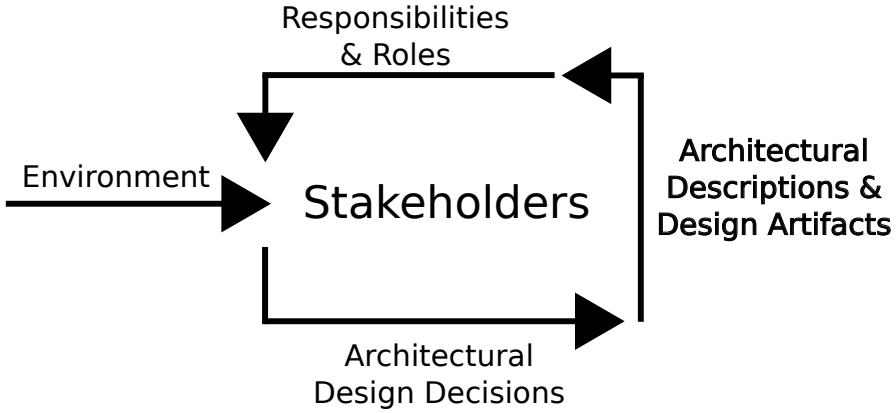


Abbildung 2.1: Beeinflussung von Entwurfsentscheidungen im Architekturprozess

[Tang u. van Vliet, 2009] schlagen vier Kategorien von Constraints vor:

Requirement Related Constraints ergeben sich aus funktionalen Anforderungen. Dies schließt beispielhaft Anforderungen nach der Offenheit von Systemen ein, bei denen eine Anbindung an ein Netzwerk erforderlich ist.

Quality Requirement Related Constraints ergeben sich aus nicht-funktionalen Anforderungen. So können etwa Bibliotheken ausgeschlossen werden, deren Berechnungen nicht in einem geforderten Zeitintervall durchgeführt werden können und bei Einsatz die Anforderungen nicht erfüllen würden.

Contextual Constraints ergeben sich aus dem Umfeld der Entwicklung sowie technologischen Einschränkungen. Dazu zählen Beispielsweise finanzielle Engpässe, aufgrund derer bestimmte sinnvolle Produkte nicht erworben werden

können. Entsprechende Produkte müssen nicht zwangsläufig bereits auf dem Markt existieren. Sie können somit nicht zugekauft werden.

Solution-related Constraints ergeben sich aus vorhandenen Softwaresystemen und vorherigen Entwurfsentscheidungen. Fällt eine Entscheidung zunächst auf eine bestimmte Technologie, können etwa Frameworks, die von anderen Technologien abhängen, nicht ohne Weiteres eingesetzt werden.

Auf abstrakter Ebene bilden Constraints eine Formalisierung aller Faktoren, die den Lösungsraum einschränken [v. d. Berg u. a., 2009]. Folge davon sind Einschränkungen bei der Auswahl von Assets aus dem Architekturwissen. Nutzt man etwa die Programmiersprache Java und ihre verwandten Technologien, schließt dies Technologien anderer Programmiersprachen zunächst aus. Constraints sind dennoch nicht unwiderruflich im Prozess verankert. Im Laufe der Evolution können Constraints wegfallen oder an Priorität verlieren. Auch getroffene Entscheidungen im Bezug auf Technologien oder sogar Programmiersprachen können revidiert und überarbeitet werden. Ebenso können Existenz- oder Eigenschaftsentscheidungen überdacht werden. Einschränkungen, die großen Einfluss auf eine Entwurfsentscheidung hatten, gelten zu einem späteren Zeitpunkt eventuell nicht mehr. Im Zuge der Evolution müssen Entwurfsentscheidung potentiell unter aktuellen Bedingungen neu evaluiert werden.

Verlieren Constraints an Priorität, können selbige potentiell aufgeweicht werden. Man spricht hierbei von einer Relaxation der Constraints [v. d. Berg u. a., 2009]. Obwohl Constraints per Definition zu erfüllen sind, das System also bei Verletzung unbrauchbar machen, können Abwägungen getroffen werden. Werden Anforderungen niedrig priorisiert, müssen Einschränkungen des Lösungsraums eventuell überdacht und neu evaluiert werden, somit potentiell bei der Auswahl von Assets ignoriert werden. Constraints schränken den Lösungsraum allerdings nicht nur negativ ein, sondern machen ihn auch durch Beschränkung von potentiell einsetzbaren Lösungen beherrschbar [Tang u. van Vliet, 2009].

2.2 Problemstellung

> *„[...] an Achilles' heel of the technology that was discovered only in its use, where there was no way for the technology to support an important use case“*
>
> [LaToza u. a., 2013]

Erst bei getroffenen Entwurfsentscheidungen und dem tatsächlichen Einsatz von Technologien können sich diese als gänzlich ungeeignet zur Erfüllung der Anforderungen herausstellen. Dies erfordert eine erneute Evaluierung von Entwurfsentscheidungen unter Betrachtung entsprechender nachteiliger Faktoren.

Als Beispiel lässt sich etwa die Technologie eines Online-Shops beschreiben, die in der Theorie alle funktionalen Anforderungen erfüllen kann, sich unter tatsächlicher Last im Kundenbetrieb jedoch nicht behaupten kann. Entsprechende Schwachstellen von Technologien sind ohne vorherigen Einsatz in der Praxis nicht vorherzusehen

[LaToza u. a., 2013]. Das Architekturwissen kann Informationen zu entsprechenden Eigenschaften von Technologien umfassen [Soliman u. a., 2015].

Für die Evolution von Softwaresystemen ist somit eine konstante Revision von Constraints und Entwurfsentscheidungen unabdingbar. Aus einer ständigen Optimierung von Prozessen, Technologien und Umfeld erwachsen bei zukünftigen Entscheidungen neue Möglichkeiten, welche Alternativen für bereits getroffene Entwurfsentscheidungen bilden können. Diese Alternativen stellen bei der Betrachtung von Entwurfsentscheidungen unter der Einschränkung durch Constraints ein zentrales Element dar.

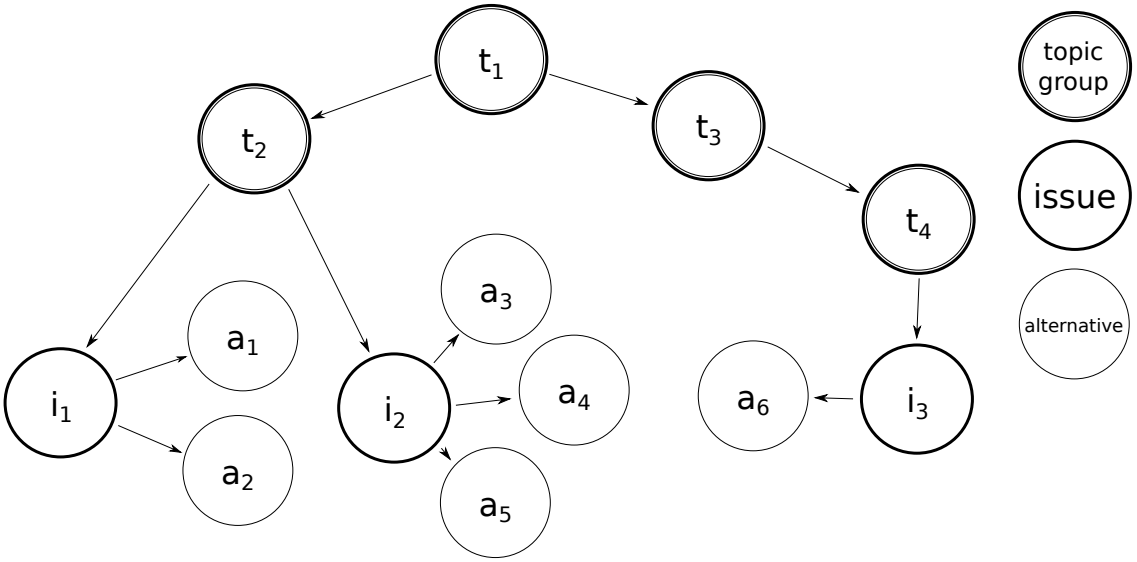
> *„Architectural knowledge may support to evaluate the impact of architectural decisions on the resulting architecture; it allows to (re-)consider alternative decisions as well“*
>
> [Clerc u. a., 2007]

[van der Ven u. a., 2006] definiert Alternativen wie folgt:

> *„Alternatives consist of a set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements“*
>
> [van der Ven u. a., 2006]

[Zimmermann u. a., 2009] betrachten Abhängigkeiten zwischen Entwurfsentscheidungen und ihre Beziehungen untereinander. Bei der Betrachtung dieser Abhängigkeiten entsteht eine Baumstruktur, bei der eine Menge von Entscheidungen (Knoten) in Beziehung zueinander stehen. [Ran u. Kuusela, 1996] bezeichnen diese Struktur als *Design Decision Tree*, also einen Baum von Entwurfsentscheidungen. Der Baum folgt dabei einer Striktordnung, das heißt die Abhängigkeiten setzen sich nur in eine Richtung fort. Es entstehen keine Zyklen. Dies hält die Komplexität überschaubar [Zimmermann u. a., 2009]. Anhand dieser Abhängigkeiten lassen sich Alternativen für Entwurfsentscheidungen bestimmen: Entscheidungen, die von derselben Entscheidung abhängen, stellen Alternativen zueinander dar, müssen sich jedoch nicht zwangsläufig ausschließen. Abbildung 2.2 zeigt exemplarisch eine entsprechende Struktur. *Topic Groups*, eine Menge semantisch verwandter Themen, für die Entscheidungen getroffen werden, führen zu konkreten Problemen (*Issues*), für die Lösungen gesucht werden müssen. Einzelne Entwurfsentscheidungen bilden die Blattknoten dieser Issues. Sie stellen mögliche Lösungsalternativen für den übergeordneten Issue bereit.
Die Blattknoten einer Entscheidung stellen zwar Alternativen zueinander dar, müssen jedoch nicht zwangsweise nur als Abhängigkeit dieser einen Entscheidung definiert sein. Die gleiche Entscheidung kann auch in der Abhängigkeit einer weiteren Entscheidung auftreten. In entsprechenden Fällen sind gleiche Entscheidungen mehrfach im Baum vorhanden, [Zimmermann u. a., 2009] als auch [Ran u. Kuusela, 1996] wählen diese Modellierung bewusst, um die Komplexität zu reduzieren.

Grundlage in [Ran u. Kuusela, 1996] ist dabei, ähnlich in [Zimmermann u. a., 2009], jedoch auch [Gerdes u. a., 2014], die Prozesssicht. Ergebnisse von Entwurfsentscheidungen werden in Folge als Teile des Baumes angefügt und werden somit selbst Teil des Architekturwissens.



Quelle: [Zimmermann u. a., 2009]

Abbildung 2.2: Beispiel Abhängigkeiten zwischen Entwurfsentscheidungen

Einzelne Entscheidungen können von anderen Entscheidungen abhängen, sie ausschließen oder begünstigen. Diese Tatsachen bilden eine Reihe kontextabhängiger Eigenschaften der Entscheidungen ab. Besondere Beachtung erfordern hierbei Technologieentscheidungen. Man spricht in diesem Rahmen auch von '*Architecturally Significant Technology Aspect*s', kurz ASTAs [Soliman u. a., 2015][3]. Diese stellen wesentliche Faktoren zur Betrachtung von möglichen Assets aus dem Architekturwissen dar. Sie bilden unter anderem Schwachstellen und ausschlaggebende Vorteile einer Technologie ab und ergeben sich zumeist aus dem Einsatz bestimmter Technologien [Soliman u. a., 2016].

ASTAs sind immer Abhängig vom Kontext ihrer Betrachtung. Dies bedeutet, dass nicht jeder Aspekt auch in jeder betrachteten Situation relevant ist. Lediglich Aspekte, die im aktuellen Anwendungskontext ausschlaggebend sind, sollten tatsächlich betrachtet werden. Sie können somit sowohl Vor- als auch Nachteile darstellen und das entsprechende Asset zu einer potentiellen Lösung machen oder ausschließen. Techniken wie die *Architecture Trade-off Analysis Method* (ATAM) versuchen sich an einer Abschätzung und Unterstützung einer Lösung unter der Betrachtung von Anforderungen und dem Einfluss von Stakeholdern [LaToza u. a., 2013].

Ein Beispiel unterstützt die Erläuterung des Sachverhaltes: Der Einsatz einer bestimmen Programmierbibliothek stellt für funktionale Anforderungen womöglich eine Kostenreduzierung dar, weil Funktionalitäten nicht neu implementiert werden müssen. Dieselbe Lösung kann sich jedoch als gänzlich ungeeignet darstellen, wenn nicht-funktionale Aspekte gefordert sind: Ist die Implementation der Bibliothek wenig optimiert, können Anforderungen an etwa Geschwindigkeit oder Sicherheit nicht befriedigt werden. Eine Neuimplementation der Funktionalitäten würde in diesem Kontext eine sinnvollere Alternative darstellen.

[3]Siehe dazu auch Abbildung A.2 im Anhang, Seite III

Technologien werden aufgrund ihrer Möglichkeiten zur Umsetzung sowohl funktionaler, als auch nicht-funktionaler Anforderungen eingesetzt. Der Einsatz bereits existierender Technologien, etwa Frameworks, COTS und Programmierbibliotheken bietet häufig Vorteile. Grundsätzliche Funktionalitäten müssen nicht selbst implementiert werden und können schlicht eingebunden werden. Auf diese Weise werden im Prozess häufig Zeit und Kosten eingespart. Entsprechende Vor- und Nachteile lassen sich durch ASTAs als Eigenschaften der Technologien modellieren.
Durch den Einsatz externer Technologien können jedoch auch Nachteile entstehen. Hohe Abhängigkeiten, Lizenz- sowie Anschaffungskosten und eine Unfähigkeit, Aspekte der Technologien anpassen zu können sind die Folge. Im Prozess und bei Entwurfsentscheidungen sind entsprechende Abwägungen vor dem Einsatz externer Technologien zu treffen.

Verändern sich Umfeld oder Anforderungen, aus deren Rahmenbedingungen sich Constraints ableiten, muss auch der Einsatz von Technologien potentiell überdacht werden. Technologien unterliegen dabei ähnlichen Rahmenbedingungen für Abhängigkeiten wie die übergeordneten Entwurfsentscheidungen.

Um ihre Anforderungen zu erfüllen, besitzen Technologien eine Reihe von Features [Soliman u. a., 2015]. Diese stellen einzelne Funktionalitäten der Technologie dar und sind dabei meist nicht einzigartig. Funktionalitäten sind in unterschiedlichen Technologien implementiert, verfügen jedoch über individuelle Eigenschaften, Vor- sowie Nachteile.

Ein Beispiel stellen häufig verwendete Mathematik-Bibliotheken dar: Je Programmiersprache existieren unter Umständen mehrere Implementationen der gleichen fachlichen Funktionalität. Diese unterscheiden sich etwa in der Genauigkeit des Ergebnisses oder der Geschwindigkeit der Berechnung. Etwa die Implementation der Logarithmus-Berechnung kann sich, je nach Umsetzung, stark unterscheiden. Die Technologien liefern zwar eine ähnliche oder die gleiche Funktionalität, sind aber an andere Rahmenbedingungen geknüpft. Auch für Technologien gelten die oben beschriebenen Eigenschaften von ASTAs. Bei der Auswahl von Technologien spielen diese Aspekte eine immense Rolle und bilden häufig den ausschlaggebenden Fakor bei der Entscheidung für oder gegen eine Technologie [Soliman u. a., 2015].

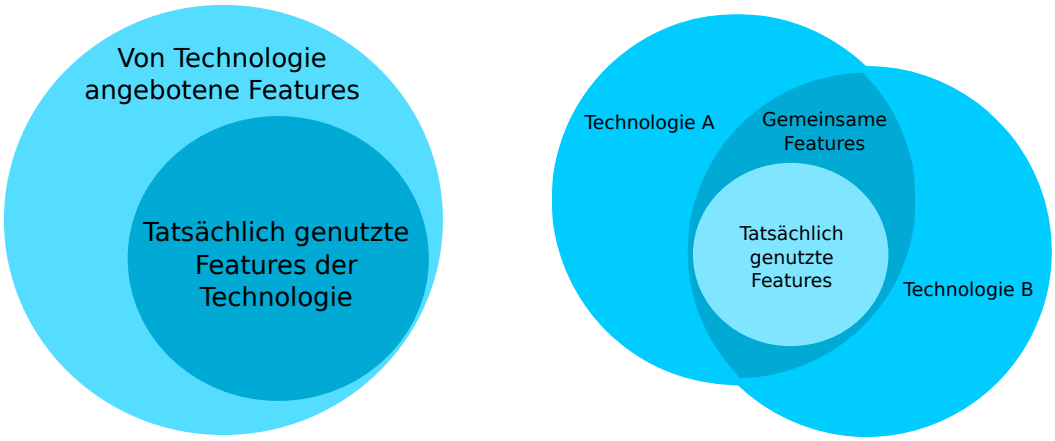
> *„To reuse architectural artifacts, we need to know the alternatives, and the rationale behind each of them“*
>
> [Jansen u. a., 2007]

Um den Nutzen und die Auswahl von Technologien zu begründen, muss die Auswahl durch die entsprechenden Rahmenbedingungen begründet werden. Neue Anforderungen und Rahmenbedingungen machen beispielsweise den Einsatz eines bestimmten Features notwendig, dass bereits eingesetzte Technologien nicht bieten. Hier muss evaluiert werden, ob Technologien ausgetauscht, oder durch neue Technologien ergänzt werden können. Die Auswahl potentieller Alternativen muss betrachtet werden. Alternativen bieten potentiell Technologien, deren Features dieselben Funktionalitäten anbieten. Dies kann sowohl durch Menge der Features einer Technologie, als auch durch eine Menge von Technologien und deren jeweiliger Features gegeben

sein.

Damit Alternativen sinnvoll abgewogen werden können, muss unter gegebenen neuen Anforderungen auch der vorhandene Kontext bereits getroffener Entwurfsentscheidungen betrachtet werden. Technologien und ihre Features wurden bereits aufgrund ihrer Features ausgewählt und eingesetzt, um bestimmte Anforderungen zu befriedigen. Neue Entscheidungen müssen sich auf das Kontextwissen vorheriger Entscheidungen stützen können. Dazu ist Kenntnis darüber erforderlich, welche Features eingesetzter Technologien bereits konkret genutzt werden. Wird eine Technologie eingebunden, werden nicht zwangsläufig alle angebotenen Features genutzt.

Abbildung 2.3 (links) zeigt diesen Umstand anhand eines Venn-Diagrammes. Welche Features tatsächlich genutzt werden, gehört meist zum *Tacit Knowledge* [van der Ven u. a., 2006] und verbleibt zumeist in Verantwortlichkeit der Programmierer. Dieses Wissen wird selten dokumentiert, da es oft als weniger relevant erscheint [LaToza u. a., 2013].



Grafik erstellt durch Verfasser

Abbildung 2.3: Teilmenge tatsächlich genutzter Features einer eingesetzten Technologie

Um zu bestimmen, welche Features einer Technologie tatsächlich genutzt werden, muss deren Nutzung zunächst identifiziert werden. Existierende Ansätze befassen sich bisweilen nicht mit der Erkennung und Bestimmung eingesetzter Technologie-Features. Häufig sind, wenn überhaupt, zwar eingesetzte Technologien dokumentiert, die von diesen Technologien genutzten Features allerdings nicht. Die Kenntnis über diese kann die Suche nach Alternativen bei Änderungen allerdings stark beeinflussen. Um eine andere Technologie als voll-qualifizierte Alternative zu betrachten, müsste diese zumindest alle Features der aktuell genutzten Technologie bieten (siehe Abbildung 2.3, rechts). Da die Nutzung aller Features allerdings eher unwahrscheinlich erscheint, muss eine potentielle Alternative lediglich die tatsächlich genutzten Features ersetzen oder ergänzen. Es werden selten alle Features einer Technologie genutzt:

> *„Moreover, our intent is often directed to only a part of what the product or components provide“*
>
> [Perry u. Grisham, 2006]

Diese Tatsache erweitert den potentiellen Lösungsraum unter Umständen enorm und weicht entsprechende Constraints auf, die der Einsatz einer Technologie und aller ihrer Features mit sich bringen würde. Die Identifikation von Technologie-Features kann somit die Suche nach Alternativen verfeinern. Sie muss allerdings auch nachvollziehbare Einschränkungen zur Betrachtung sinnvoller Alternativen bereitstellen.

Einzelne Technologie-Features können dabei anhand charakteristischer Eigenschaften identifiziert werden. Die Einbindung von Technologien, Frameworks, Bibliotheken und weiteren erfolgt dabei auf rein syntaktischer Ebene: im Quellcode von existierenden Anwendungen.

Ähnliche Konzepte finden sich auch in [Mirakhorli u. a., 2014]. Auch wenn hier das Erkennen von architekturrelevanten Taktiken verfolgt wird, ist das Konzept grundsätzlich ähnlich. Einzelnen Taktiken werden jeweils charakteristische Eigenschaften, *Indikator-Terme*, zugeordnet. Diese werden nachfolgend als Indikatoren bezeichnet. Mithilfe der Sprachverarbeitung werden Kommentare und Namen im Quellcode auf bestimmte Wort-Konstellationen hin untersucht, um die eingesetzten Taktiken zu bestimmen. Ein ähnliches Konzepte ließe sich, wenn auch weniger auf der Ebene der Verarbeitung natürlicher Sprache, übertragen.

Die meisten Technologien und Bibliotheken folgen dem 'Inversion of Control'-Prinzip zur Kapselung von Funktionalität: Die eigentliche Funktionalität ist von außen nicht einsehbar, Aufrufe an die Bibliotheken stoßen die interne Verarbeitung an. Charakteristische Eigenschaften sind somit vorrangig anhand der durch die Technologien angebotenen Schnittstellen zu suchen. Können diese aus etwa Dokumentationen, Tutorials und APIs extrahiert werden, bilden sie einerseits selbst potentiell einen Teil des Architekturwissens, andererseits identifizieren sie die Nutzung eines Features der Technologie. Die Indikatorenen sind somit rein syntaktischer Natur und vorrangig statisch. Sie lassen sich zum Zeitpunkt der Kompilierung, meist aber nicht mehr zur Laufzeit einer Anwendung feststellen.

Verwandte Problembereiche In diesem Kontext mangelt es an einem Ansatz für die automatisierte Identifikation von Technologie-Features in vorhandenem Quellcode. Existierende Ansätze des Reverse Engineering versuchen jedoch, mit ähnlichem Hintergrund durch Rekonstruktion und Identifikation von Eigenschaften einer Softwarearchitektur Aussagen über diese zu treffen. Welche Fokussierungen diese Ansätze erfahren soll nachfolgend knapp betrachtet werden.

Bereits früh wird der *Design Recovery Process* beschrieben [Biggerstaff, 1989]. Dieser versucht, das Architekturverständnis zu verbessern und bei der Rekonstruktion von Design und Strukturen zu unterstützen. Weitere Ansätze schlagen etwa ein Clustering vor [Mancoridis u. a., 1999]. Werkzeuge zur *Design Recovery*, der Rekonstruktion von Informationen existieren ebenfalls [Gueheneuc u. a., 2006]. Verwandter Ansatz ist auch auch die *Traceability Recovery* [McMillan u. a., 2009], das nachträgliche Wiederherstellen von Informationen zur Traceabilty.

Ansätze des Reverse Engineering befassen sich vorrangig mit der Rekonstruktion von Features, meist zwecks Neuentwicklung [Chikofsky u. Cross, 1990]. Dabei stehen allerdings funktionale Features im Vordergrund. Entsprechende Neuentwicklungen sollen ein vorhandenes System ersetzen und meist auf Basis neuer Technologien entwickelt werden. Die Betrachtung genutzter Technologie-Features ist somit weniger zentral. Technologien werden in den meisten Fällen ohnehin ausgetauscht. Wenn auch in diesem Fall der bereits beschriebene Entscheidungsprozess Anwendung findet, bilden eher anforderungsbasierte Constraints die Grundlage. Das häufigste Problem dabei ist fehlende Dokumentation. Vorrangig Entwurfsentscheidungen und ihre Hintergründe, welche einen immensen Einfluss auf die Strukturen des Systems hatten, werden meist nicht oder nicht vollständig dokumentiert. Eine Rekonstruktion dieser Hintergründe und damit der Begründung einer Entscheidung ist ohne gegebene Dokumentation nicht ohne Weiteres möglich [van der Ven u. a., 2006].

Das Fachgebiet der *Feature Location* beschreibt die Suche nach dem Ursprung eines Features im Quellcode der entsprechenden Anwendung [Coffey u. a., 2010] [Wang u. a., 2011]. Funktionale Features haben direkte Auswirkungen auf die Funktionalität eines Systems. Wie und wo die Features im Quellcode umgesetzt sind, ist ohne hinreichende Dokumentation allerdings nicht einzusehen. Der Ansatz der *Feature Location* versucht dabei, meist zur Laufzeit, den Ursprung eines gegebenen Features zu bestimmen.
Dies steht im Gegensatz zur Identifikation von Technologie-Features: Wo sich ein Feature konkret befindet, ist im Rahmen dieser Arbeit weniger relevant. Für den Entscheidungsprozess ist vornehmlich die Nutzung eines Technologie-Features von Bedeutung. Das spätere Austauschen von Technologien erfordert hingegen die Bestimmung der Quelle und ist somit zu einem späteren Zeitpunkt relevant.

[Nie u. Zhang, 2012] unterteilen den Ansatz der *Feature Location* in drei Kategorien zur Bestimmung:

Text stellt anhand des Quelltextes Vermutungen über die Quelle von Features auf und ist in diesem Zusammengang auf der Suche nach Strukturen und Entwurfsmustern ungenau.

Statisch bestimmt auf Basis der Strukturen des Quellcodes die Beziehungen der Komponenten untereinander.

Dynamisch stellt zur Laufzeit durch Aufrufbeziehungen und Zeitverhalten Abschätzungen über den Ursprung der Features auf.

Für die Analyse von Technologie-Features kommt lediglich die erste der drei Kategorien in Frage: Eine Betrachtung auf niedrigem Abstraktionsniveau macht die Analyse von Strukturen und Aufrufbeziehungen womöglich sogar unnötig. Die Feature Location beschränkt sich vorrangig auf das dynamische Finden von Informationen zur Laufzeit. Dies geschieht durch Instrumentierung des Quellcodes und einer Protokollierung der durchlaufenen, instrumentierten Anweisungen. Zahlreiche Ansätze befassen sich bereits mit der Lokalisierung funktionaler Features [Coffey u. a., 2010] [Wang u. a., 2011] [Jordan u. a., 2015] [Ziftci u. Krüger, 2012] [Lukoit u. a., 2000] [Rubin u. Chechik, 2013] [Dit u. a., 2013], jedoch nicht mit der Identifikation von Technologie-Features.

Einsatz findet die *Feature Location* vielmehr beim Reverse Engineering zwecks Architekturverstehen als bei Architekturentscheidungen [Ziftci u. Krüger, 2012]. Eine Rekonstruktion von funktionalen Features erfordert zudem meist zusätzliches Domänenwissen, um den Kontext einer Anwendung zu verstehen und relevante Informationen über die Architektur zu extrahieren.

> *„[...] significant knowledge about the problem domain is required in order to facilitate the extraction of the system's useful architectural information“*
> [Pashov u. Riebisch, 2004]

Die Identifikation von Technologie-Features birgt dabei ein sehr viel niedrigeres Abstraktionsniveau [Harris u. a., 1995]. Gerade funktionale Features, Pattern [Seemann u. von Gudenberg, 1998] und verwandte Strukturen [Kramer u. Prechelt, 1996] sind häufig abhängig von der Struktur eines Programmes. Sie stellen wichtige Zusammenhänge für Abhängigkeiten und Aufrufbeziehungen auf.

Design und Rekonstruktion beschränken sich somit zumeist auf funktionale Features [Regli u. a., 2000]. An einer sinnvollen Identifikation von Technologie-Features zur Findung potentieller Alternativen mangelt es bisweilen.

2.3 Exemplarischer Anwendungsfall

> *„Software architecture researchers conceive of architecting as a scenario driven process“*
>
> [LaToza u. a., 2013]

Um die Relevanz der Problematik zu erläutern, soll ein exemplarischer Anwendungsfall die Hintergründe beleuchten. Folgendes Szenario nimmt die Grundsituation auf und beschreibt, an welcher Stelle im Entscheidungsprozess die Extraktion von Technologie-Features und Suche nach potentiellen Alternativen Anwendung findet.

Man nehme als Grundlage der Betrachtung das System *DecisionBuddy* aus [Gerdes u. a., 2015] an. Es handelt sich bei diesem (vergleichsweise kleinen) Softwaresystem um eine Webanwendung zur Unterstützung von Softwarearchitekten. Diese soll die Entscheidungsfindung im Softwarearchitekturprozess unterstützen. Die Anwendung baut in einem gewissen Umfang eine Basis an Architekturwissen auf, um Entscheidungen und Abwägungen vorschlagen zu können. Anhand einer Modellierung von Problemen (Die in [Zimmermann u. a., 2009] beschriebenen Issues) werden potentielle (Technologie-)Lösungen aus der Wissensbasis vorgeschlagen. Diese müssen anschließend durch den Softwarearchitekten evaluiert werden. Grundlage ist dabei ein Constraint-basierter Entscheidungsprozess [Gerdes u. a., 2015].

Technisch handelt es sich beim *DecisionBuddy* um eine in Java geschriebene Webanwendung (Abbildung 2.4). Sie basiert vorrangig auf den Technologien *Spring*-MVC, *JavaServer Pages* (JSP) sowie *Hibernate*. Die Datenbasis bildet eine MySQL-Datenbank. Aus den unterschiedlichen eingesetzten Technologien werden einzelne Funktionalitäten genutzt. Deren Einsatz bildet zusammen das System ab. Die genaue Architektur des Systems ist für die Betrachtung nicht relevant. Betrachtet werden sollen eingesetzte Technologie-Features.
Einen (konstruierten) Anwendungsfall für die zuvor genannte Problematik bildet hier folgendes Szenario:

Das System setzt auf die Nutzung einiger durch *Hibernate* implementierter Optimierungen des Datenbankzugriffes. Zu diesen gehören unter anderem die Stapelverarbeitung von Datenbankanfragen. Im Rahmen der Konfiguration von *Hibernate* lässt sich diese explizit für bestimmte Objekte nutzen. Man stellt zu Laufzeit des Systems merkliche Verzögerungen der Anwendung fest, sobald entsprechende Operationen durchgeführt werden. Bestimmte Datenobjekte werden im Arbeitsablauf der Nutzer häufiger abgefragt als erwartet. Die Optimierung scheint in diesem Anwendungsfall zur Laufzeit nicht auszureichen und sorgt für eine Unzufriedenheit der Nutzer, da Verzögerungen im Arbeitsablauf entstehen. Vor Implementation des Systems war dieses Verhalten nicht abzusehen, man hat somit zur Laufzeit eine kritische Schwachstelle der Technologie identifiziert.

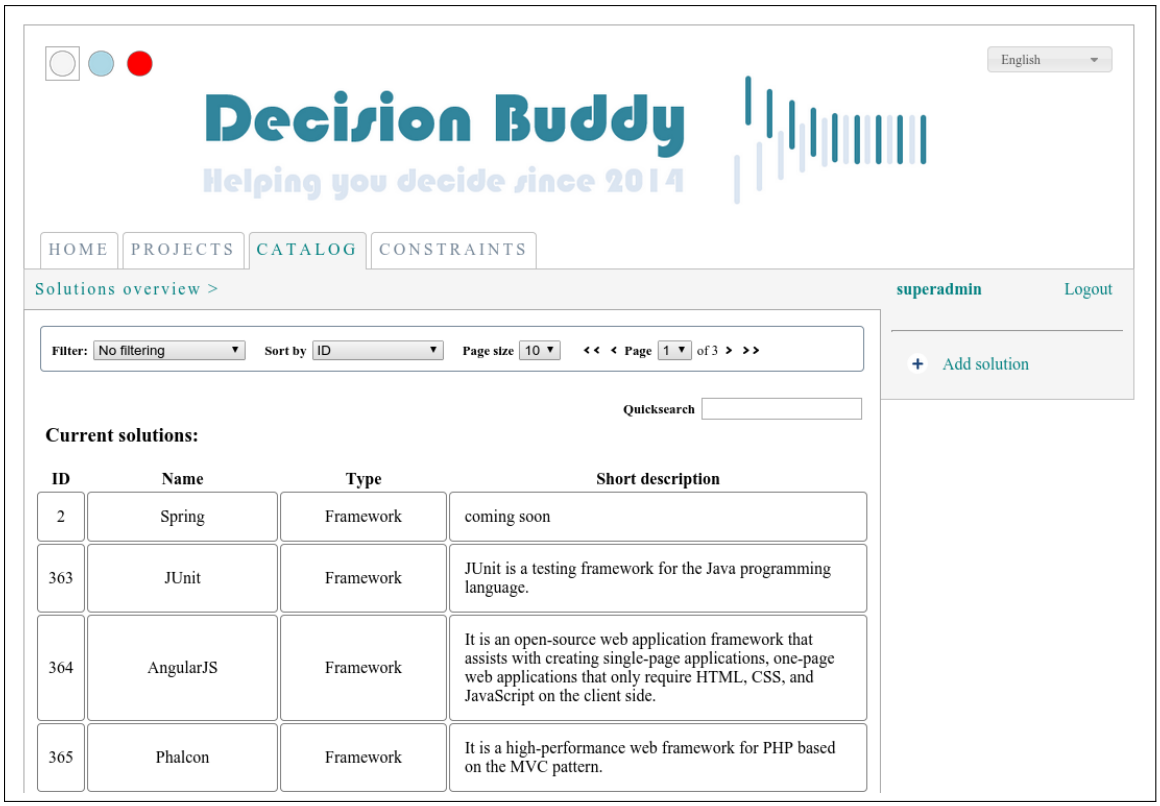


Abbildung 2.4: Weboberfläche *DecisionBuddy* zur Verwaltung von Architekturwissen

```
1 @ManyToOne(targetEntity =
    ArchitecturalPatternCategory.class, optional = false,
    fetch = FetchType.EAGER)
2 @JoinColumn(name = "ArchitecturalPatternCategoryID",
    referencedColumnName = "ID", nullable = false)
3 @BatchSize(size=25)
4 private ArchitecturalPatternCategory
    architecturalPatternCategory;
```

Quellcode-Beispiel 2.1: Einsatz von Batch Fetching (Hibernate)

Quellcode-Ausschnitt 2.1 zeigt beispielhaft den Einsatz des Features auf Implementationsebene: Neben einigen weiteren Features kommt auch die explizite Nutzung des *Batch Reading* zum Einsatz (Zeile 3). Hier wird bei der Deklaration des Feldes `architecturalPatternCategory` die Menge der Datenbank-Objekte spezifiziert, die bei Optimierung innerhalb einer Anfrage geladen werden soll. Dies zieht das zur Laufzeit nachteilige Verhalten nach sich, ist jedoch in seinen Grundzügen für die Optimierung erforderlich. Die Nutzung des Technologie-Features muss explizit spezifiziert werden.

Benötigt wird somit eine alternative Technologie, welche zwar insgesamt eine ähnliche Menge von Features umsetzt, für den konkreten Problemfall nicht-funktionale Anforderungen jedoch besser umzusetzen vermag. Diese Alternative soll möglichst wenige Änderungen am vorhandenen System zur Implementation erfordern und sich ohne grundlegende Anpassungen an der Architektur durchführen lassen.

Tabelle A.2[4] zeigt auszugsweise einen Teil der durch den *DecisionBuddy* genutzten Technologie-Features der Technologie *Hibernate*. Die genutzten Funktionalitäten beschränken sich dabei auf einige wesentliche Funktionalitäten, die für das Zusammenspiel der Technologie mit *Spring* notwendig sind. Weitere Funktionalitäten stellen grundsätzliche Eigenschaften des *Object-relational mapping*-Konzeptes (ORM) dar (etwa die Abbildung von Relationen, Vererbung, Polymorphie etc.[5]). Zumindest für diese grundsätzlichen Features, die das Konzept und die daraus resultierende Architektur begründen, finden sich alternative Lösungen auf dem Markt. Diese unterscheiden sich meist in den Optimierungen des Zugriffes und weiterer nicht-funktionaler Eigenschaften. Häufig wird Kritik an *Hibernate* geäußert, alternative Lösungen sind im Trend[6].

Weitere genutzte Features sind somit auch *Hibernate*-spezifische Optimierungen, die auf Basis der implementierten JPA-Schnittstelle vorgenommen wurden. Bei alternativen Implementationen ist das Finden eindeutiger Alternativen ähnlicher herstellerspezifischer Optimierungen schwierig. Herstellerspezifische Technologie-Features sind meist Alleinstellungsmerkmale und damit selten in allen alternativen Technologien zu finden.

Das betrachtete Technologie-Feature der Stapelverarbeitung und entsprechender Optimierung hingegen findet sich jedoch auch in alternativen JPA-Implementationen. Obwohl die implementierte Optimierung ähnlich ist, soll sich diese im Kontext effizienter Verhalten und bietet so andere Vor- und Nachteile[7]. Das Feature heißt, anders als in *Hibernate* nicht *Batch Fetching*, sondern *Batch Reading*[8] und soll das gleiche Verhalten implementieren. Die beiden Implementationen stellen damit Alternativen zueinander dar. Die umsetzende alternative JPA-Implementation *TopLink* ist somit ein potentieller Kandidat für den Austausch der Technologie.

Unter Betrachtung der vorhandenen Features und Technologien muss der Lösungsraum auf der Suche nach Alternativen eingegrenzt werden. Nicht jede auf dem Markt existierende Technologie bietet ähnliche Funktionalitäten wie die verwendete Technologie, obwohl sie prinzipiell dasselbe Konzept implementiert (etwa das ORM-Konzept). Ob die Technologien tatsächlich sinnvolle Alternativen zueinander darstellen, lässt sich anhand ihrer Abhängigkeiten alleine nicht feststellen.

Abbildung 2.5 zeigt dies exemplarisch. Die Technologie *Hibernate* (gelb) soll ausgetauscht werden. Sie hängt von der Java-Spezifikation für Object-relational mapping (ORM), der *Java Persistence API* (JPA) ab. Als direkte Alternative für *Hiberna-*

[4] Anhang, Seite VIII

[5] Siehe http://madgeek.com/Articles/ORMapping/EN/mapping.htm oder https://www.ormfoundation.org/forums/p/515/1458.aspx, zuletzt aufgerufen am 18. Oktober 2016

[6] http://blog.jhades.org/solving-orm-complexity-keep-the-o-drop-the-r-no-need-for-the-m/, zuletzt aufgerufen am 18. Oktober 2016

[7] https://community.oracle.com/thread/876139, zuletzt aufgerufen am 18. Oktober 2016

[8] http://docs.oracle.com/html/B32441_03/qrybas.htm#i1165217, zuletzt aufgerufen am 18. Oktober 2016

te erscheint hier die Technologie *TopLink*, welche ebenfalls die JPA implementiert. Da jedoch nicht allein eine Alternative gesucht ist, sondern eine Alternative, welche möglichst viele Features von *Hibernate* ebenfalls unterstützt, muss *TopLink* nicht die einzige oder sinnvollste Alternative sein. Ohne Kenntnis über den tatsächlich genutzten Funktionsumfang und semantische Beziehungen zwischen den Technologien kommt jede Technologie als potentielle Alternative in Frage (rot). Nutzt das Softwaresystem etwa nur einen kleinen Teil der Funktionalitäten der JPA-Schnittstelle, existieren unter Umständen weitere Technologien mit Unterstützung für entsprechende Funktionalitäten. Potentiell denkbar wäre auch der Einsatz der ORM-Technologie *Sormula*, die hier keine direkte Alternative darstellt.

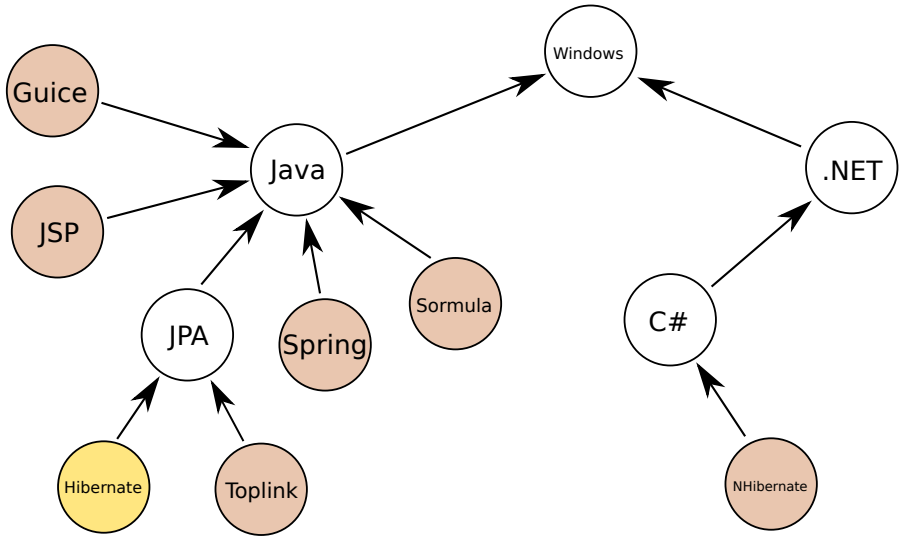


Abbildung 2.5: Potentielle Alternativen zu *Hibernate*

Zudem muss bedacht werden, dass *Hibernate* trotz seiner Abhängigkeit zu JPA nicht nur dessen Features implementiert: Hersteller-spezifische Anpassungen und Eigenheiten weist die direkte Alternative nicht zwangsweise auf. Die Suche nach alternativen Technologien muss somit auch Technologien ohne direkte Abhängigkeiten berücksichtigen. Entsprechend eingesetzte Features müssen, so weit wie möglich, auch von alternativen Technologien unterstützt werden, um den Aufwand für Änderungen gering zu halten. Im, zugegeben sehr optimistischen, Optimalfall müssten lediglich einige Bibliotheken oder Referenzen im vorhandenen Quellcode ausgetauscht werden, um die alternative Technologie einzubinden.

Gesucht sind für den betrachteten Anwendungsfall alternative Technologien, die einen möglichst großen Teil der durch den *DecisionBuddy* genutzten Features aus *Hibernate* unterstützen. Auf Ebene der Entwurfsentscheidungen sind auf Grundlage bereits getroffener Technologie-Entscheidungen und ihren Abhängigkeiten neue Alternativen in den Baum von Entscheidungen einzubringen [Zimmermann u. a., 2009]. Dies müssen, wie oben beschrieben, nicht zwangsläufig direkte Alternativen darstellen.

Damit diese Alternativen jedoch bestimmt werden können, müssen zunächst im *DecisionBuddy* genutzte Technologie-Features der Technologie *Hibernate* identifiziert werden. Die Nutzung konkreter Features ist nicht dokumentiert. Sie muss aus dem

existierenden System heraus bestimmt werden.

Um die Betrachtung der potentiellen Problemlösung zu erleichtern, wird der beschriebene Anwendungsfall im Zuge der weiteren Ausführungen referenziert. Das System *DecisionBuddy* dient im Rahmen der Aufgabestellung als Grundlage der Betrachtung. Im Folgenden soll das automatisierte Erkennen von Technologie-Features und das Finden von Alternativen zu den vorhandenen Technologien sowie möglichen weiterführenden Entscheidungsmöglichkeiten thematisiert werden. Die im Prozess zu treffenden Entscheidungen sind anschließend ohne maschinelle Hilfe im Kontext des Softwarearchitektur-Prozesses zu treffen.

Kapitel 3

Lösungsansatz

Annahmen

Bevor eine Lösung zur Identifikation von Technologie-Features ausgearbeitet werden kann, müssen zunächst einige Vorbedingungen und Ziele definiert werden. Diese leiten sich direkt aus der Aufgabenstellung ab und schränken die Arbeit in ihren Grundlagen und ihrem Umfang maßgeblich ein. Folgende Annahmen sind zu treffen:

1. **Architekturwissen** Da kein allgemeingültiger Ansatz zur Modellierung des Architekturwissens existiert, kann in diesem Rahmen nur eine Teilmenge sinnvoll betrachtet werden. Zukünftige Ansätze der Wissenschaft müssen einen entsprechend allgemeingültigen Ansatz hervorbringen [Farenhorst u. de Boer, 2009]. Eine semantisch vollständige Betrachtung des Architekturwissens ist in diesem Kontext somit nicht möglich. Das Architekturwissen muss, wo benötigt, manuell formalisiert und für den Anwendungsfall bereitgestellt werden.

2. **Umfang** Um die Analyse von Technologie-Features auf den vorliegenden Rahmen einzugrenzen, wird bewusst eine Abstraktion durchgeführt. Betrachtungen erfolgen anhand des vorliegenden Anwendungsfalls. Aus diesem lässt sich durch das Aufstellen eines Konzeptes sowie dessen Auswertung auf die Qualität der Lösung schließen. Zu diesem Zweck fällt die Analyse und Betrachtung auf das vorhandene System *DecisionBuddy*, seine Komponenten und Features sowie potentielle Alternativen. Ebenso bildet der Anwendungsfall die Grundlage für eine beispielhafte Bestimmung von Alternativen.
   Konkret wird der Umfang der Betrachtung durch den gegebenen Anwendungsfall beschränkt. Dies hat unter Umständen Einfluss auf die Allgemeingültigkeit der Lösung.

3. **Grundlage der Modellierung** Durch die Aufgabenstellung ist eine existierende Modellierung von Technologie-Features vorgegeben. Sie beinhaltet bereits die Betrachtung von Technologien, Technologie-Features sowie ASTAs. Wo notwendig, muss sie entsprechend erweitert und angepasst werden. Dies lässt später eine Eingliederung und Integration in vorhandene Ansätze zu. Der vorgestellte Lösungsansatz stellt somit eine Erweiterung bereits akzeptierter Ansätze der Forschung dar.

Bestandteile des Lösungsansatzes

Abbildung 3.1 zeigt die Bestandteile des Lösungsansatzes. Die Arbeit lässt sich folgend anhand der Aufgabenstellung in einzelne Teilaufgaben unterteilen. Weiterhin wird im Folgenden das grundlegende Vorgehen anhand der einzelnen Bestandteile von Aufgabenstellung und Lösungsansatz erläutert.

Die Implementation eines Prototypen soll anschließend zeigen, dass die technische Umsetzung, unter Betrachtung der Annahmen, eine sinnvolle Lösung des Problems darstellen kann. Nach Betrachtung des Lösungsansatzes wird eine Evaluation durchgeführt, welche die Implementation des Prototypen und damit des Konzeptes beleuchten soll.

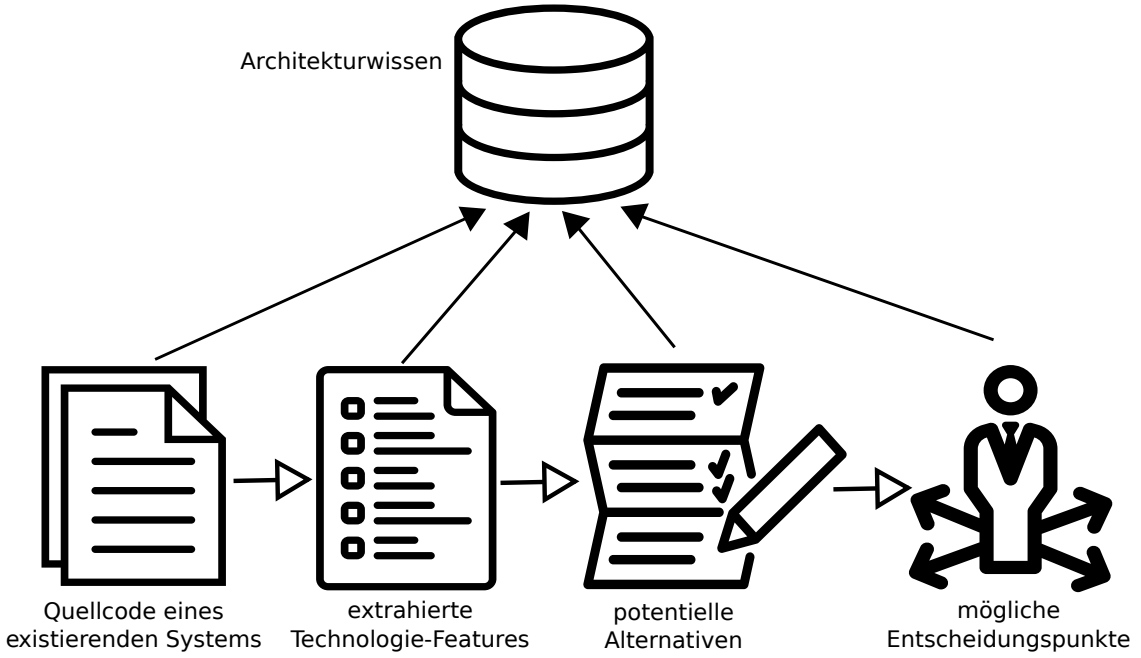


Abbildung 3.1: Bestandteile des Lösungsansatzes

Exemplarische Auswahl von Architekturwissen

Um eine sinnvolle Betrachtung von Technologie-Features zu ermöglichen, ist eine Formalisierung dieser und verwandter Konstrukte notwendig. Zu diesem Zweck muss das Architekturwissen nicht nur formalisiert werden, es müssen ebenso exemplarische Daten bereitgestellt werden. Das System *DecisionBuddy* dient unter Betrachtung der Technologie *Hibernate* und weiterer datenbankzentrierter Technologien als Grundlage im Rahmen der Aufgabenstellung.

> *„Defining templates or metamodels and referencing patterns is a good starting point towards more systematic and rigorous decision capturing“*
>
> [Zimmermann u. a., 2009]

Grundlage bildet das Modell aus [Soliman u. a., 2015][1]. Es beinhaltet bereits die Beziehungen zwischen Technologien (hier: *TechnologySolution*) und Technologie-Features (*TechnologyFeature*). Ebenso beinhaltet es die Verknüpfung von Technologie-Features und ASTAs, die hier nur exemplarisch mitgeführt wird.

Es muss ein allgemeingültiger Ansatz zur Identifikation von Technologie-Features geschaffen werden. Dieser soll im Anschluss, dynamisiert durch die Inhalte des Architekturwissens, eine Extraktion ermöglichen. Es müssen exemplarisch einzelne Features betrachtet werden, um aus diesen eine Formalisierung der charakteristischen Eigenschaften (Indikatoren) abzuleiten[2]. Zu diesem Zweck muss eine Datenstruktur aufgesetzt und gepflegt werden. Gleichzeitig dient die Vereinheitlichung der Daten der Validierung des angepassten Metamodells. Dies geschieht im Rahmen der zuvor beschriebenen Annahmen zum Umfang der Betrachtung.

Obwohl Ansätze für die automatische Extraktion und das Erfassen von Architekturwissen existieren [Soliman u. a., 2016], erfolgt die Befüllung hier manuell. Grund dafür ist die gezielte Betrachtung eines Anwendungsfalles. Eine manuelle Analyse muss die Identifikation von Technologie-Features und Indikatoren leisten. Wie, warum und durch wen diese Informationen sinnvollerweise erfasst werden sollten, muss ebenfalls betrachtet werden. Gesucht ist, anders als bei allgemeinen Ansätzen zur Betrachtung des Architekturwissens, keine vollständige Menge von Informationen.

Identifikation von Technologie-Features

Ist eine Sondierung und exemplarische Betrachtung des Architekturwissens erfolgt, können mit diesem Wissen Technologie-Features aus einem vorhandenen Softwaresystem extrahiert werden. Eingabe in diesem Schritt des Ansatzes ist das bereits identifizierte und formalisierte Architekturwissen mit seinen Technologien, Technologie-Features und deren Indikatoren. Ausgabe ist eine Menge identifizierter Features.
Ob eine eindeutige Aussage über das Vorhandensein von Technologie-Features möglich ist, muss die Untersuchung zeigen.

[1] Siehe Abbildung A.2, Anhang, Seite III
[2] Auf Grundlage von [Mirakhorli u. a., 2014], siehe dazu auch Seite 22

Der Einsatz von Technologie-Features ist anhand ihrer Syntax und damit rein textuell zu betrachteten, entgegen der statischen oder dynamischen Analyse der Struktur eines Softwaresystems. Wie in [Nie u. Zhang, 2012] beschrieben, setzt eine syntaktische Analyse vorhandener Quellcodedateien keine Kenntnis über Anwendungsdomäne oder die Architektur des Systems voraus. Zugriff und Nutzung auf Bibliotheken und Technologien erfolgt zur Kompilierzeit einer Anwendung und nicht der Laufzeit. Schätzungsweise lässt sich anhand dieser und genutzten Sprachkonstrukten ein Modell für die Extraktion aufstellen. Quellen sollen dabei Dokumentationen, Schnittstellen, Online-Foren, Tutorials sowie Beispiele zur Nutzung einzelner Features darstellen. Diese sind zwar allgemein selten als wissenschaftlich verlässliche Quellen angesehen, stellen jedoch im Kontext der Anwendung den einzig sinnvollen Ansatzpunkt dar: Aufgrund der niedrigen Abstraktionsebene konkreter Technologien finden Ansätze der Wissenschaft selten konkrete Betrachtungen entsprechender Konstrukte.

Bestimmen von Alternativen, Treffen von Entwurfsentscheidungen

Auf Basis der zuvor extrahierten, sollen, ebenso unter Verwendung des Architekturwissens, potentielle Alternativen bestimmt werden. Diese Bestimmung soll eine Reihe von Technologien liefern, welche eine Fokussierung auf potentiell sinnvolle Alternativen im Rahmen Abhängigkeiten zwischen den Technologien ermöglicht.
Die Bestimmung von Alternativen soll bewusst keine optimale Lösung vorlegen. Zu diesem Zweck wäre eine Menge an formalisiertem Domänen- und Prozesswissen notwendig. Dies ist im Rahmen der Annahmen jedoch Aufgabe anderer Ansätze des Reverse Engineering.

Um eine Bestimmung von Alternativen durchführen zu können, muss zunächst eine Definition für diese auf Grundlage der vorangegangenen Formalisierung geschaffen werden. Anders als in [Zimmermann u. a., 2009] oder [Ran u. Kuusela, 1996] Verhalten sich die Abhängigkeiten zwischen Technologien und insbesondere deren Features anders als bei Entwurfsentscheidungen. Dies liegt vorrangig der sehr viel niedrigeren Abstraktionsebene zugrunde. Um dennoch, parallel zu [Zimmermann u. a., 2009], die Komplexität zu reduzieren, muss eine einheitliche Definition geschaffen werden, welche zwei Technologie-Features als Alternativen identifizieren kann.

Ziel dieser Betrachtung ist das Vorschlagen möglicher Entscheidungspunkte. Diese berücksichtigen die Abhängigkeiten zwischen potentiellen Alternativen. Sie bieten eine weitere Abwägung bezüglich des Umfangs möglicher Entwurfsentscheidungen. Im Ermessen des Nutzers, etwa eines Softwarearchitekten liegt es, anschließend Entwurfsentscheidungen sinnvoll vornehmen zu können. Dieser muss anhand von kontextabhängigem Domänenwissen und den aktuellen Anforderungen eine Entscheidung treffen. Ebenso relevant sind Constraints. Unterstützen sollen ihn in diesem Anwendungsfall auch die im Architekturwissen hinterlegten ASTAs.

Einordnung

Um das vorgeschlagene Konzept einordnen zu können, beschreibt Abbildung 3.2 dessen Standpunkt im Gesamtbild des iterativen Softwareentwicklungsprozesses. Die Abbildung ordnet das Vorgehen grob ein und beschreibt, wann die Extraktion von Technologie-Features durchzuführen ist. Die Betrachtung des Prozesses ist dabei stark vereinfacht und fokussiert die hier relevanten Bestandteile.

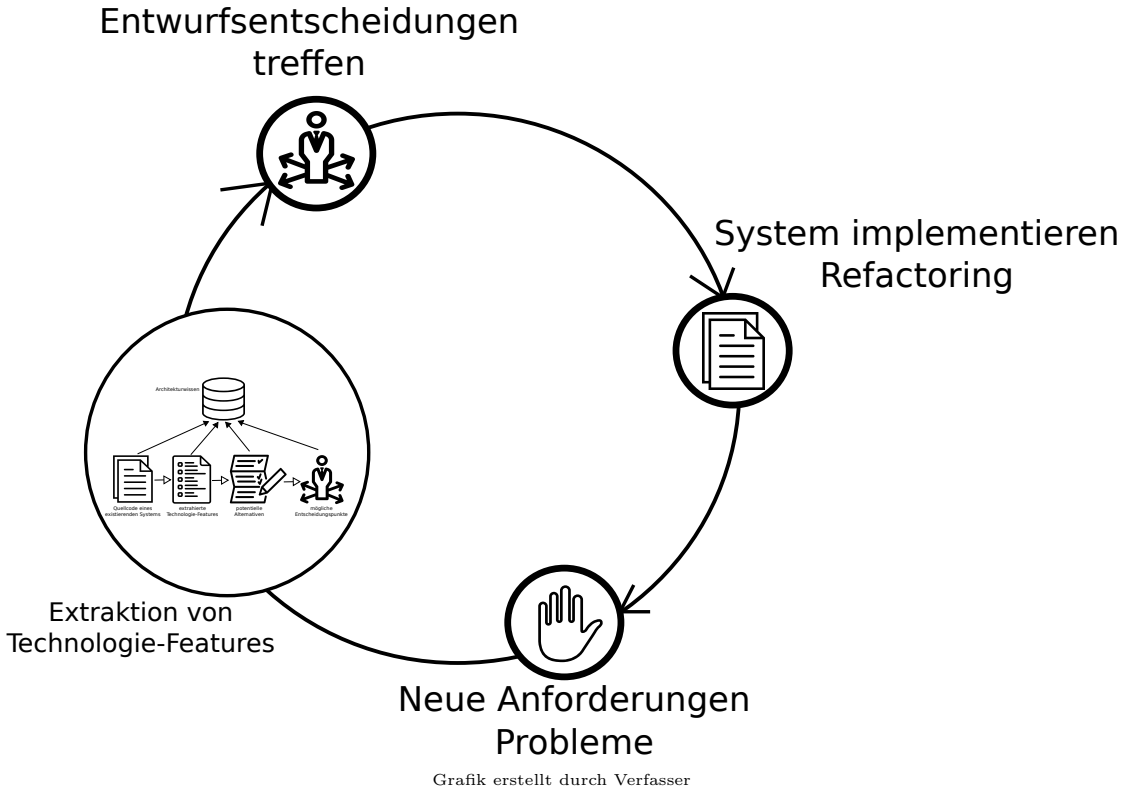


Abbildung 3.2: Einbindung des Konzeptes in den iterativen Entwicklungsprozess

Anhand getroffener Entwurfsentscheidungen wird das System in jedem Schritt der Iteration implementiert. Diese jeweiligen Schnappschüsse stellen die Grundlage der Betrachtung dar. Im Zuge der Nutzung einer konkreten Ausprägung des Systems kommt es zu den im Anwendungsfall beschriebenen Problemen oder neuen Anforderungen. Bevor im folgenden Iterationsschritt neue Entwurfsentscheidungen getroffen werden, kann die Extraktion anhand des aktuellen Zustands des Systems durchgeführt werden. Sie liefert unter Umständen wertvolle Erkenntnisse und kann den Prozess in seiner Gesamtheit unterstützen. Dies ermöglicht in Folge fundiertere Entwurfsentscheidungen.

Kapitel 4

Ausgestaltung der Problemlösung

Um die zuvor beschriebene Problematik in ihrer Gesamtheit zu adressieren, muss zunächst betrachtet werden, auf welcher Ebene der Abstraktion das folgende Konzept einzuordnen ist. Die Softwarearchitektur ist im Allgemeinen ein stark abstrahiertes Konzept der Betrachtung von Softwaresystemen. Eben dies schließt zumeist die Betrachtung der Implementation nicht ein. Wie die vorhergehende Analyse der Literatur bereits zeigt, betrachten entsprechend nur wenige Ansatz die konkrete Ebene der Technologie-Features. Bei der Betrachtung von Entwurfsentscheidungen ist diese Abstraktion im Rahmen der genannten Problematik jedoch nicht zwangsläufig von Vorteil.

Sollen Alternativen bei bereits konkreter Nutzung im Rahmen eines Softwaresystems bestimmt werden, muss der Einblick in die Implementation der vorhandenen Architektur erfolgen. Es sind somit beide Abstraktionsebenen zu berücksichtigen: Die Prozessebene sowie die Implementationsebene.
Die Prozessebene beschreibt, analog zu [Zimmermann u. a., 2009] und [Soliman u. a., 2015] die Sichtweise von Technologien als Entwurfsentscheidungen. Entscheidungen werden hier im Prozess der Softwarearchitektur getroffen und müssen, wie der beschriebene Anwendungsfall zeigt, im Laufe der Zeit überdacht und eventuell erneut getroffen werden.

Die Implementationsebene zeigt nach erfolgten Entscheidungen auf der Prozessebene ein konkretes Softwaresystem als Folge dieser Entscheidungen. Auf höherer Ebene getroffene (Technologie-)Entscheidungen beeinflussen trivialerweise das konkrete System. Ist das System pro Iterationsschritt im andauernden Prozess implementiert, hat diese konkrete Implementation jedoch ebenfalls Einfluss auf den zukünftigen Entscheidungsprozess. Während der Implementation werden weitere Entwurfsentscheidungen getroffen: Die Auswahl von Technologie-Features einer zuvor ausgewählten Technologie. Die Auswahl dieser Features wird, wenn überhaupt, noch seltener explizit dokumentiert als ein Softwaresystem in seiner Gesamtheit [Jansen u. Bosch, 2005]. Entscheidungen, die auf Basis eines existierenden Systems getroffen werden folgen zwar demselben Schema im Prozess, müssen unter Umständen jedoch wesentlich mehr Constraints berücksichtigen. Gerade die Auswahl genutzter Technologie-Features und spezielle Funktionalitäten aus zuvor ausgewählten Technologien stellen bei Änderungen Anforderungen an potentielle Alternativen.

Abbildung 4.1 zeigt die Trennung bei der Betrachtung dieser beiden Ebenen und ihrer Elemente. Um die Beziehungen zwischen zwei Technologien als potentielle Alternativen zueinander (Doppelpfeil) für ein vorhandenes Softwaresystem zu bestimmen, muss auf niedrigerer Abstraktionsebene (Rote Linie) betrachtet werden, welche Technologie-Features die Technologien gemeinsam haben. Erst wenn diese Betrachtung erfolgt ist, kann auf Ebene von Entwurfsentscheidungen auf eine entsprechende Modellierung geschlossen werden.

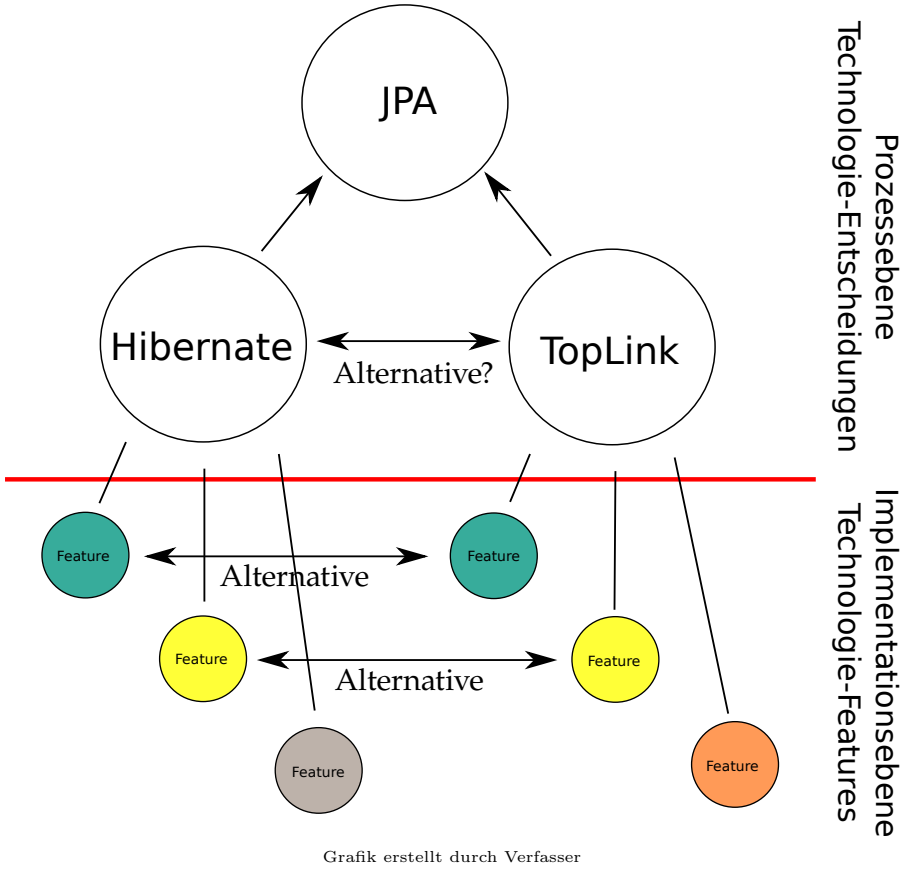


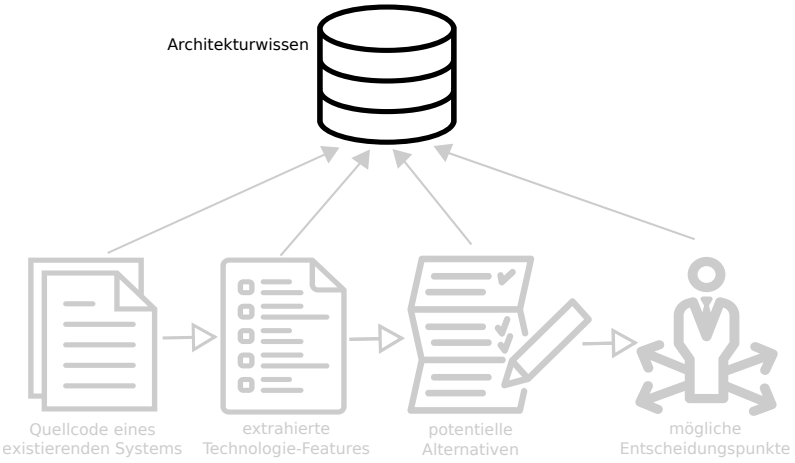
Abbildung 4.1: Abstraktionsebenen

Entsprechende Beziehungen zwischen Technologien könnten durchaus als Inhalt des Architekturwissens betrachtet werden. Jedoch hat eine statische Modellierung dieser Beziehungen untereinander einen entscheidenden Nachteil: Sie ist unflexibel. Technologien, die dem Architekturwissen hinzugefügt werden, müssen in die Struktur der Wissensbasis integriert werden. Dies erfordert initial eine Analyse aller potentieller Alternativen und gegebenenfalls eine Anpassung aller bereits hinterlegter Beziehungen. Eine Betrachtung „semantischer“ Alternativen auf dieser Ebene ist zunächst wenig zielführend: Weil unterschiedliche Technologien das Konzept des ORM implementieren, müssen sie keine sinnvollen Alternativen in Anbetracht der konkreten Nutzung darstellen.
Selbst unter Nutzung von Konzepten wie Beschreibungslogiken[1] zur Modellierung ist die Betrachtung aufwändig und hat eine hohe Komplexität zur Folge. Die Bestimmung der Technologie-Features vorausgesetzt, lassen sich die Alternativen jedoch anhand dieser bestimmen.

[1] Siehe etwa [Baader, 2003]

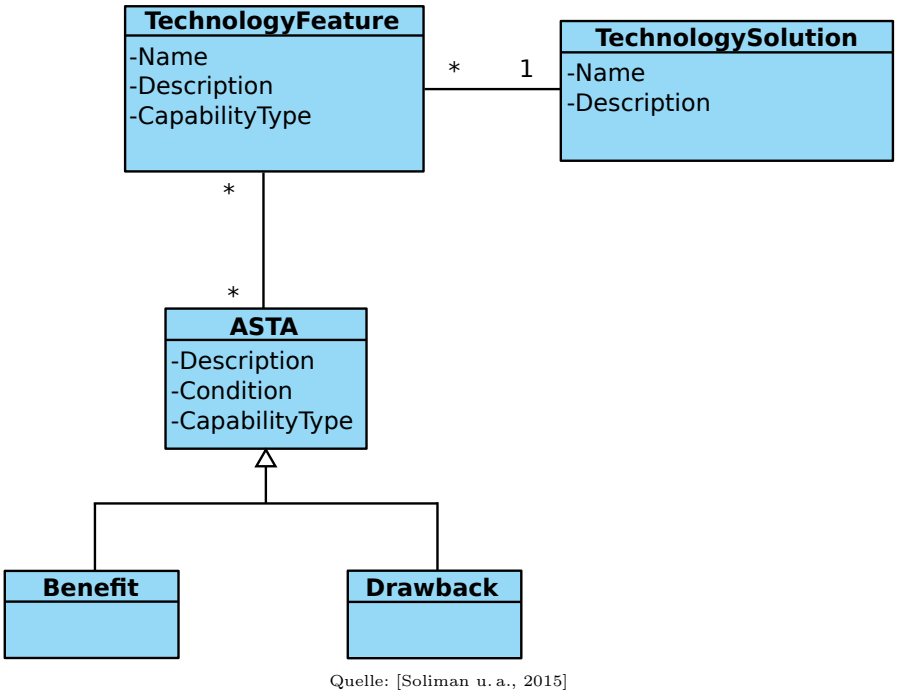
4.1 Architekturwissen

Grundlage der Betrachtung bildet eine exemplarische Menge an Architekturwissen. Dessen Modellierung und Befüllung wird im Folgenden beschrieben.



Ursprüngliches Metamodell

Grundlage der Betrachtung des Architekturwissens ist das in [Soliman u. a., 2015] vorgeschlagene Metamodell. Um nur die relevanten Aspekte des Modells zu betrachten, werden nicht benötige Elemente gekürzt. Abbildung 4.2 zeigt bereits das auf die hier relevanten Elemente und Beziehungen reduzierte Modell.



Quelle: [Soliman u. a., 2015]

Abbildung 4.2: Grundlage Modellierung

Das Modell setzt Technologien (*TechnologySolution*) und deren Features (*TechnologyFeature*) in direkte Beziehungen. Eine Technologie besitzt dabei mehrere Features.

Diese haben jeweils eine Beschreibung (*Was tut das Feature?*) und einen Fähigkeitstyp (*CapabilityType*), welcher die Funktionalität des Features grob in eine von sieben Kategorien einordnet. Abbildung 4.3 zeigt die einzelnen Typen sowie deren Beeinflussung architekturrelevanter Aspekte. Diese sind Teil des grundlegenden Modells, haben im Folgenden jedoch keine Relevanz für die Identifikation von Technologie-Features.

Capability Type	**Influenced Architectural Concerns**
Development and Configuration Capability	Development Time, Training Time, Maintainablity, Testability, Configurability, Evolvability
Behavior Capability	Performance, Reliability, Security, Accuracy, Portability, Reusability
Operational Capability	Manageability, Supportability, Availability
Interoperability Capability	Interoperability, Inter-process communication
Usability Capability	Usability
Commercial Capability	Cost of Ownership, Openness, Development Time, Training Time
Storage Capability	Data Accessability, Scalability

Quelle: [Soliman u. a., 2015]

Abbildung 4.3: Fähigkeitstypen aus [Soliman u. a., 2015]

Um das angestrebte Konzept auf Basis diesen Modells umsetzten zu können, müssen zunächst fehlende Elemente sowie Schwächen identifiziert und ergänzt werden.

Abhängigkeiten Betrachtet man Technologie-Entscheidungen als Entwurfsentscheidungen, bestehen Abhängigkeiten zwischen diesen. Ist eine Technologie A von einer weiteren Technologie B abhängig, muss zunächst eine Entscheidung für B getätigt werden, bevor A ausgewählt werden kann. Dies erfolgt Analog zu [Zimmermann u. a., 2009][2].

Betrachtet man diese Abhängigkeiten singulär (das bedeutet, eine Technologie hängt von maximal einer anderen ab), ließe sich die direkte Abhängigkeit durch eine Referenz modellieren. Abhängigkeiten zu mehreren Technologien sollten aufgrund erhöhter Komplexität bei der Modellierung vermieden werden [Zimmermann u. a., 2009].

Dem Modell aus [Soliman u. a., 2015] fehlen diese Abhängigkeiten, da sie im Kontext seines Ursprungs weniger relevant sind. Um später zur Bestimmung von Alternativen festlegen zu können, welche Technologien gegen andere Technologien ausgetauscht werden müssen, ist diese Beziehung essentiell.

Ähnliches gilt für die Betrachtung von Technologie-Features. Auch zwischen einzelnen Features können Abhängigkeiten entstehen.
Beispiel für eine solche Abhängigkeit ist etwa *Hibernate*s Unterstützung für native

[2]Siehe Abbildung 2.2, Seite 19

SQL-Queries (Abbildung 4.4). Diese Funktionalität stellt selbst bereits ein Features der Technologie dar. Die Unterstützung der Ausführung von Gespeicherte Prozeduren innerhalb dieser ist eine Erweiterung der Funktionalität. Sollen entsprechende Prozeduren in einer nativen Abfrage aufgerufen werden, muss zunächst die Unterstützung dieser vorhanden sein. Ein Einsatz ohne das übergeordnete Features ist nicht möglich oder sinnvoll.

Abbildung 4.4: Beispiel Abhängigkeiten zwischen Technologie-Features

Für die Extraktion von Technologie-Features ist diese Beziehung weniger relevant. Sie strukturiert lediglich das Architekturwissen und schafft eine klare Hierarchie unterschiedlicher Technologie-Features.

Charakteristische Eigenschaften von Technologie-Features Da das Modell keine Betrachtung der Implementationsebene durchführt, beinhaltet es keine Modellierung der Eigenschaften von Technologie-Features. Um die Erfassung dieser soweit wie möglich formalisieren zu können, müssen dem Model Indikatoren zur Erkennung hinzugefügt werden. Eine Zuordnung eines oder mehrerer Indikatoren je Technologie-Feature ermöglichen später eine Extraktion dieser. Formal bedeutet dies, dass dem Modell ein Element zur Beschreibung von Indikatoren fehlt. Das Konzept stellt in Folge entsprechende Formalisierungen von Indikatoren auf und ergänzt das Modell entsprechend.

Bestimmung von Alternativen Um Alternativen formal feststellen zu können, fehlt es an Informationen. Einzelne Technologien sind unabhängig voneinander modelliert und lassen sich im vorhandenen Modell nicht verbinden. Es fehlt eine Struktur, welche die Bestimmung von Alternativen ermöglicht.
Direkte Beziehungen zu anderen Technologien oder deren Features sind wenig zielführend. Modelliert man Beziehungen anhand einer direkten Zuordnung (etwa *„Technologie A stellt eine Alternative zu Technologie B dar“*), steigt die Menge an Beziehungen stark an. Anders als die unidirektionale Zuordnung von Technologie-Features zu Technologien müsste man bidirektionale Beziehungen in das Modell einfügen. Diese würden die Komplexität des Modells erhöhen. Dies erschwert jedoch die Betrachtung der Prozessebene [Zimmermann u. a., 2009].

Bereits eine oberflächliche Betrachtung von Technologie-Features lässt feststellen: Exakte Alternativen existieren nicht. Kein Feature einer Technologie wird in seiner Nutzung und seinem Verhalten exakt dem einer anderen Technologie entsprechen. Der in Abschnitt 2.2 beschriebene Fall der Logarithmus-Berechnung in Mathematik-Bibliotheken dient auch hier als Beispiel. Unterscheiden werden sich verschiedene Implementationen eines Features in etwa ihren nicht-funktionalen Eigenschaften,

ihren Vor- und Nachteilen (ASTAs) sowie ihrem konkreten Einsatz. Obwohl sie die gleiche Funktionalität umsetzen, sind sie nicht identisch.

Sinnvoller ist es somit, eine abstrakte Betrachtung von Features durchzuführen. Anstatt direkte Beziehungen zwischen Technologien herzustellen, lassen sich diese anhand eine Menge gemeinsamer Technologie-Features in Beziehung stellen.

Dem Modell fehlt zu diesem Zweck eine Betrachtung mehr oder weniger abstrakter Instanzen eines Features. Anstatt zwei Technologie-Features als direkte Alternativen zueinander anzusehen, lassen sie sich als Implementation eines bestimmten Konzeptes betrachten. Im Kontext der Logarithmus-Berechnung könnte dies bedeuten: Eine Funktion aus Technologie A berechnet diesen, ebenso eine Funktion aus Bibliothek B. Weil beide konkreten Funktionen die gleiche mathematische Funktion berechnen, sind ihre Technologien, zumindest im Bezug auf diese Berechnung, Alternativen zueinander. Auf welche Technologie eine Entscheidung letztendlich fällt hängt, hier jedoch auf Prozessebene, von den exakten Eigenschaften der einzelnen Technologien und ihrer Features ab.

Das Konzept ist somit wesentlich flexibler und dynamisch anzupassen. Werden neue Features oder Technologien hinzugefügt, lassen sich diese Definitionen weniger kompliziert pflegen, als Beziehungen zwischen Technologien hinterlegen zu müssen.

Erweitertes Metamodell

Um eine Grundlage für die weitere Betrachtung des Konzepts und damit des zentralen Architekturwissens zu schaffen, wird das Modell aus [Soliman u. a., 2015] gemäß der zuvor betrachteten Schwächen hin ergänzt. Abbildung 4.5 führt bereits die durchgeführten Erweiterungen auf. Im Folgenden werden diese Erweiterungen erläutert und in Beziehung gestellt.

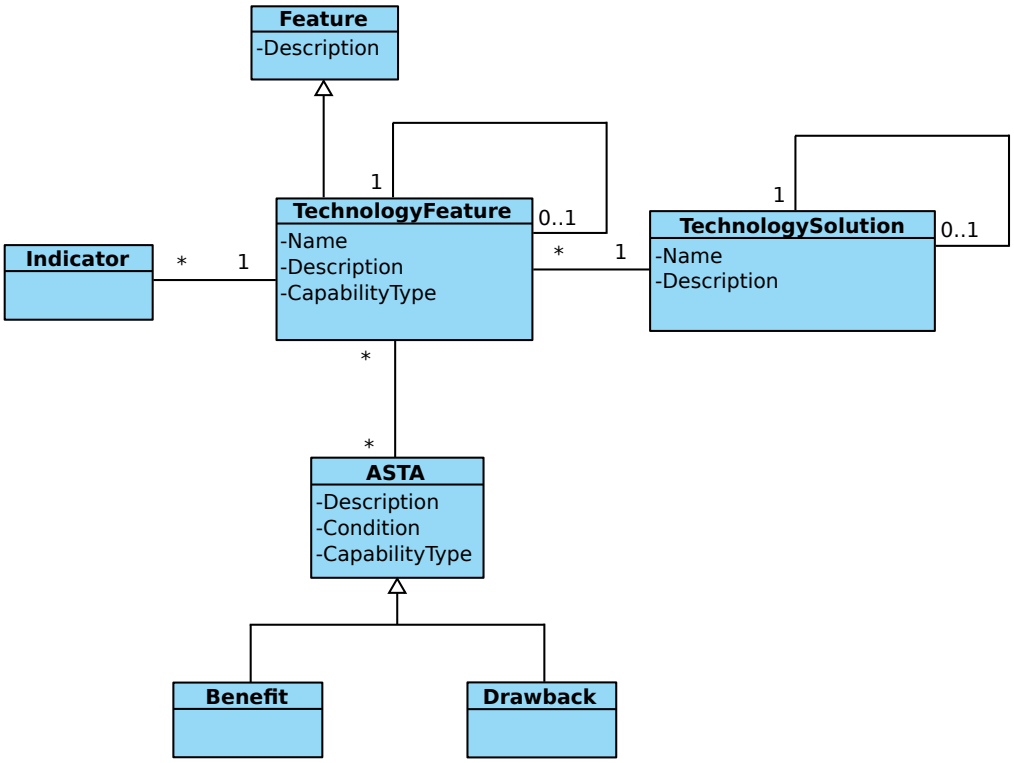


Abbildung 4.5: Erweitertes Metamodell

Auf Grundlage obiger Überlegungen wird das Modell um Abhängigkeitsbeziehungen zwischen Technologien erweitert. Eine Technologie kann dabei maximal von einer Technologie abhängen. Erscheinen Technologien in den Abhängigkeiten mehrerer Technologien, sind diese dupliziert darzustellen. Beispiel stellt hier etwa die Plattformunabhängigkeit der Programmiersprache Java dar. In diesem Fall muss die Technologie Java mehrfach, in Abhängigkeit zu der Modellierung des entsprechenden Betriebssystems aufgeführt werden. Dies gilt Analog für alle weiteren von Java abhängigen Technologien.

Auf diese Weise entsteht zwischen den Technologien ein Baum an Abhängigkeiten. Technologien ohne Abhängigkeit können ohne Vorbedingung eingesetzt werden und bilden die Wurzeln eines Baumes. Da mehrere Technologien ohne Abhängigkeiten existieren können, resultiert diese Modellierung in einer Menge von Bäumen (*Wald*) an Abhängigkeiten zwischen Technologien. Aus dieser Baumstruktur lässt sich die Parallele zu den in [Zimmermann u. a., 2009] beschriebenen Architekturentscheidungen ableiten: Wird eine Technologie (Blatt des Baumes) gewählt, müssen auch alle Technologien auf dem Pfad dahin gewählt werden. Entwurfsentscheidungen sind entlang dieser Pfade zu treffen. Diese Struktur ist essentiell, um später den Transfer

der Bestimmung von Alternativen auf die Prozess- und damit Entscheidungsebene zurückzuführen.

Ähnliches gilt für Abhängigkeiten zwischen Technologie-Features. Eine Abhängigkeitsbeziehung wird zwischen Features eingeführt. Diese können jeweils bis zu einer Abhängigkeit unterliegen. Entgegen den bereits beschriebenen Abhängigkeiten zwischen Technologien haben deren Features zumindest eine Abhängigkeit: Die Zuordnung zu ihrer entsprechenden Technologie.
Erweitert man den Baum der Abhängigkeiten zwischen Technologien um Features, bilden diese die Blätter des Baumes. Für die Betrachtung und die Extraktion ist diese Struktur, wie bereits erwähnt, weniger relevant.

Da zu diesem Zeitpunkt noch keine Betrachtung der Formalisierung von Indikatoren erfolgt ist, erfolgt deren detaillierte Formulierung im Rahmen der späteren Analyse. Indikatoren sind jeweils einem Technologie-Feature zuzuordnen. Für ein Technologie-Feature können mehrere Indikatoren erfasst werden. Sie bilden in ihrer Gesamtheit die Grundlage zur Extraktion des entsprechenden Features. Wird ein Indikator im Quellcode eines bestehenden Softwaresystems erkannt, ist dies ein Indiz dafür, dass ein bestimmtes Feature genutzt wird.

Um später Alternativen zu Technologie-Features bestimmen zu können, wurde das Modell um ein Element zur Beschreibung abstrakter Features ergänzt. Einzelne Technologie-Features sind konkrete Instanzen dieser. Sie implementieren jeweils die abstrakte Funktionalität und unterscheiden sich in den konkreten Details. Für das konkrete Modell sind somit in der Praxis, wie zuvor beschrieben, Indikatoren sowie Vor- und Nachteile (ASTAs) für jedes individuelle Feature zu erfassen.

Erfassen von Inhalten

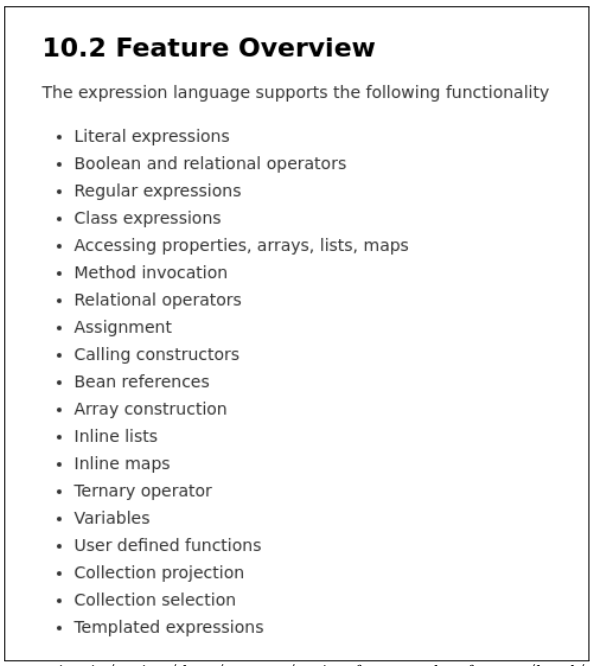
Um die beschriebene Erweiterung zu validieren, soll folgend das Erfassen von Inhalten des Architekturwissens erläutert werden. Somit lassen sich exemplarisch Struktur und Nutzen des Modells nachvollziehen.

Umfang Um das Architekturwissen exemplarisch zu befüllen, ist eine Fokussierung im Rahmen von Annahme 2 zielführend. Betrachtet man den exemplarischen Anwendungsfall in 2.3, erscheint eine Betrachtung ORM-Technologien sinnvoll. Um auf Basis des *DecisionBuddy* später Technologie-Features extrahieren zu können, müssen die von diesem eingesetzten Features erfasst werden. Da der Anwendungsfall konkret den Einsatz von *Hibernate* betrachtet, liegt der Fokus anschließend auf *Hibernate* und verwandten Datenbank-Technologien. Zu diesen lassen sich entsprechend den im Anwendungsfall beschriebenen Argumenten alternative Technologien finden.
Legt man den Fokus auf diesen zugegeben begrenzten Anwendungsfall, erscheint die Menge an potentiellen Inhalten vergleichsweise gering. Begründet liegt dieser Fokus in den Annahmen und dem hohen Aufwand zur händischen Erfassung von Inhalten. Diese Tatsache liegt zudem in Annahme 1 begründet: zukünftige Ansätze sollen automatisierte Verfahren und Strukturen zur Erfassung des Wissen ergeben, die auch dem vorliegenden Ansatz als Grundlage dienen sollen. Eine allumfassende Analyse

aller Inhalte ist somit in diesem Rahmen nicht zielführend.

Genutzte Technologie-Features aus der Anwendung heraus zu bestimmen erwies sich als aufwendig. Die Identifikation einzelner Technologie-Features, deren Zuordnung und Anreicherung mit zusätzlichen Informationen bringt einen semantischen Aufwand mit sich.

Quellen Zunächst muss bestimmt werden, wie sich Technologie-Features identifizieren lassen. Hinweise geben meist bereits die Plattformen der Anbieter: Explizite Werbung mit spezifischen Funktionalitäten, Eigenheiten und Möglichkeiten. Offizielle Quellen können dabei etwa Webseiten, Verkaufspräsentationen, APIs und Dokumentationen sein. Häufig listen Anbieter, explizit als „Features“, die Funktionalitäten einer Bibliothek oder ähnlichem auf. Exemplarisch lässt sich dies am Beispiel der *Spring*-Komponente *SpEL* (*Spring Expression Language*) betrachten (Abbildung 4.6). Jedes der aufgelisteten Features ist durch eine von der Technologie angebotenen Schnittstelle zu nutzen. Die einzelnen Technologie-Features sind dabei vorrangig sehr technisch und bilden Einheiten zur Konstruktion funktionaler und nicht-funktionaler Anforderungen.



10.2 Feature Overview

The expression language supports the following functionality

- Literal expressions
- Boolean and relational operators
- Regular expressions
- Class expressions
- Accessing properties, arrays, lists, maps
- Method invocation
- Relational operators
- Assignment
- Calling constructors
- Bean references
- Array construction
- Inline lists
- Inline maps
- Ternary operator
- Variables
- User defined functions
- Collection projection
- Collection selection
- Templated expressions

Quelle: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html, zuletzt aufgerufen am 18. Oktober 2016

Abbildung 4.6: Features *Spring*-SpEL wie vom Hersteller angegeben

Verfeinert man die Betrachtung, können auch indirekte Quellen, das heißt nicht vom Anbieter bereitgestellte Medien, konsultiert werden. Tutorials, Anwendungsbeispiele, Quick-Start-Guides oder Beiträge in Online-Foren betrachten häufig spezifische Funktionalitäten einer Software und wie sie umzusetzen sind. Vor allem aus letzterer Kategorie heraus lassen sich Indikatoren für einzelne Technologie-Features ableiten: Explizite Beispiele in Form von Quellcode-Auszügen helfen bei der Eingrenzung. Ein Beispiel bildet hier erneut die Technologie *Spring*[3].

[3]https://spring.io/guides#tutorials, zuletzt aufgerufen am 18. Oktober 2016

Anhand eines simplen Anwendungsbeispiels soll erläutert werden, wie in Folge Features erkannt und begründet werden. Zudem wird beleuchtet, wie die Modellierung von Indikatoren zu Stande kommt.

Man betrachte das Feature *Literal Expression* der Technologie *Spring-SpEL*: Der Anbieter gibt explizit an, wie das konkrete Feature zu nutzen ist[4].

```
1 import org.springframework.expression.*;
2 import org.springframework.expression.spel.support;
3 //...
4 ExpressionParser parser = new SpelExpressionParser();
5 Expression exp = parser.parseExpression("'Hello World'");
6 String message = (String) exp.getValue();
```

Quellcode-Beispiel 4.1: Einsatz von Literal Expressions (*Spring*-SpEL)

Quellcode-Beispiel 4.1 stammt vom Anbieter und lässt einige Aussagen über die Nutzung des Features zu:

Es sind die Programmpakete `org.springframework.expression` (Zeile 1) beziehungsweise `org.springframework.expression.spel.support` (Zeile 2) zu nutzen. Somit muss zur Nutzung des Features eine bestimmte Bibliothek vom Anbieter heruntergeladen, in das einsetzende System inkludiert und für die Nutzung importiert werden.

Es ist ein Objekt der Klasse `ExpressionParser` (Zeile 4) zu erstellen. Dieses stammt aus vorangehend genanntem Import.

An diesem Objekt ist eine Funktion `parseExpression(...)` (Zeile 5) mit einem in einen String eingefassten Literal aufgerufen werden. Das Literal stellt sich durch einfache Anführungszeichen dar (`'Hello World'`).

Aus diesem Beispiel lassen sich bereits wichtige Informationen zur Extraktion von Technologie-Features ableiten. Der Einsatz von Features zeichnet sich durch den expliziten Einsatz von Bibliotheken, Typen, Funktionen, Parametern und ähnlichen Sprachkonstrukten aus. Ihr Zusammenspiel klassifiziert den Einsatz eines bestimmten Features.

Aus diesen Überlegungen und Betrachtungen ergeben sich bereits erste Inhalte des Architekturwissen: *Spring-SpEL* bildet zunächst eine Technologie ab. Da der Einsatz dieser *Spring*-Komponente allerdings auch von *Spring* abhängt, muss diese Technologie ebenfalls eingefügt werden. Vorbedingung für den Einsatz von *Spring* ist der Einsatz von Java, welches wiederum erst im Kontext unterschiedlicher Betriebssysteme als Technologie eingesetzt werden kann.

Soll *Spring-SpEL* genutzt werden, müssen Entwurfsentscheidungen auch auf entsprechende Abhängigkeiten fallen. Der Einsatz von *Literal Expression* aus der Technologie stellt ein Technologie-Feature dar. Dieses besitzt keine Abhängigkeiten zu

[4] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html, zuletzt aufgerufen am 18. Oktober 2016

weiteren Features. Anderes trifft auf Features wie *Boolean and relational operators*, die als Parameter beim Aufruf der Funktion `parseExpression(...)` beim Einsatz von *Literal Expression* eingesetzt werden müssen. *Literal Expression* ist somit Vorbedingung für den Einsatz weiterer Features der Technologie. Die zuvor genannten charakteristischen Eigenschaften dieses Features bilden entsprechend das Konzept der Indikatoren ab: Sie bilden syntaktische Merkmale, welche den Einsatz des Features identifizieren können.

Im Fall des *DecisionBuddy* muss eine entsprechende Betrachtung in Eingrenzung des Umfangs aus dem existierenden System heraus erfolgen. Analog zu oben genannter Analyse von Features muss die Betrachtung anhand des existierenden Quelltextes erfolgen.

Gegensätzliches gilt entsprechend bei gänzlich unbekannten Systemen oder der Betrachtung von Systemen aus der Sicht der Softwarearchitektur. Hier werden die Implementationsebene und damit eingesetzte Features meist ignoriert. Zur Erkennung muss sich hier auf das Architekturwissen und darin hinterlegte Informationen verlassen werden. Um eine Grundlage zu schaffen, muss die Analyse in diesem Kontext auf Basis des Systems erfolgen: Kenntnisse über dessen Aufbau schaffen einen Überblick über die genutzten Features und ermöglichen es so, diese zu Erfassen und für die spätere Identifikation zu betrachten.

Werden Technologien oder Features angepasst, muss das Architekturwissen entsprechend aktualisiert werden. Die manuelle Pflege ist aufwendig, optimaler wäre das Pflegen durch den Anbieter der beinhalteten Technologien. Dieser ist sich meist im Klaren darüber, a) welche Features die Technologie umsetzt, b) welche Abhängigkeiten zwischen diesen Features und weiteren Technologien bestehen, c) welche Indikatoren ein Feature beschreiben. In der Praxis ist an dieser Stelle jedoch eine mangelnde Motivation zwecks Sinnhaftigkeit dieser Tätigkeit zu erwarten (parallel zur Codification [Hansen u. a., 1999]).

Grenzen

Die vorgestellten Erweiterungen des Modells sind wenig umfangreich. Um das folgende Konzept zur Extraktion von Technologie-Features und Bestimmung von Alternativen zu betrachtet, reicht der Detailgrad jedoch aus. Basierend auf den Annahmen (Abschnitt 3) sind jedoch einige Grenzen des Modells sowie den hier exemplarisch betrachteten Inhalten aufzuzeigen.

Fokus bei der Befüllung des Modells liegt hauptsächlich auf dem in Abschnitt 2.3 vorgestellten Anwendungsfall. Betrachtete Technologien und Features ergeben sich aus einer manuellen Analyse des *DecisionBuddy*. Dies sorgt zunächst für eine begrenzte Perspektive des Konzepts. Da allerdings nicht lediglich eine Technologie betrachtet wird, werden unterschiedliche Aspekte dieser betrachtet. Da der Umfang der Bestimmung hohen manuellen Aufwand nach sich zieht, ist die Betrachtung eines größeren (und damit eventuell realistischeren Systems) zwar unter Umständen sinnvoller, hier jedoch nicht angestrebt. Die manuelle Analyse ermöglicht, im Gegensatz zu potentiellen automatisierten Ansätzen zur Befüllung eine höhere Genauigkeit.

Formal betrachtet verringert die vorliegende Modellierung von Abhängigkeiten die Komplexität nach [Soliman u. a., 2016] drastisch. Dies lässt die Modellierung komplizierter Beziehungen zwischen Technologien nicht zu. Im Kontext des Anwendungsfalls stellt dies jedoch kein Nachteil dar. Die Analyse des vorhandenen Systems und den daraus resultierenden Technologien und Features wiesen keine Abhängigkeiten auf, die nach der händischen Analyse mit dem beschriebenen Modell nicht darzustellen waren.

Semantische Überlegungen machen die Betrachtung schwierig: In vielen Fällen lässt sich über die Zuordnung von ASTAs diskutieren. Nicht immer ist eindeutig, ob Vor- und Nachteile wirklich einzelnen Technologie-Features oder ganzen Technologien zuzuordnen sind. Gerade die Betrachtung von Vor- und Nachteilen ist häufig auch von persönlichen Faktoren geprägt [LaToza u. a., 2013]. Folgt man Online-Foren, werden vorrangig Nachteile einzelner Features einer Technologie gleich der Technologie als Nachteil zugeordnet. Eine entsprechende Betrachtung sieht das Modell nach [Soliman u. a., 2015] nicht vor. Dies gestaltet die Analyse entsprechender Fälle schwierig.

Um das Konzept auf Basis des Architekturwissen zu verifizieren, reichen aus dem *DecisionBuddy* extrahierte Inhalte alleine nicht aus. Zusätzlich müssen alternative Technologien mit ihren Features erfasst werden, um später Alternativen bestimmen zu können. Ausgangspunkt für die manuelle Suche nach Alternativen sind Online-Foren und Blog-Einträge. Diese geben einen ersten Anhaltspunkt, welche Technologien potentielle Alternativen darstellen können. Da entsprechende Quellen gemeinhin nur über eine fragwürdige Glaubwürdigkeit verfügen, wurden eingängige Untersuchungen der Technologien vorgenommen. Die spätere Suche nach Alternativen sollte somit aufzeigen, dass es sich bei den Technologien tatsächlich in einem gewissen Rahmen um Alternativen handelt. Ausschlaggebend sind dabei die in mehreren Technologien implementierten Features.

Konkret stellten sich hier folgende Alternativen für den gesuchten Anwendungsfall heraus. Als Alternative zu *Hibernate* stellen sich die JPA-Implementationen *TopLink* sowie *EclipseLink* dar. Zudem betrachtet die Analyse die alternative ORM-Technologie *Sormula* sowie die .NET-Implementation von *Hibernate*, hier *NHibernate*. Entscheidende Faktoren für die Auswahl waren die Mengen an Features, zu denen sich die Funktionalitäten der Technologien überdecken, deren Popularität sowie, im Falle von *NHibernate*, keinerlei gemeinsame Abhängigkeiten. Diese sollen später illustrieren, wo Entscheidungen zu treffen sind. Dieser Fall kann zeigen, dass Technologien mit ähnlichen Features zu den eingesetzten nicht nur bei ähnlichen Abhängigkeiten zu finden sind.

Die Suche nach Alternativen hat in diesem Rahmen jedoch ihre Grenzen. Betrachtet man Technologien auf höherer Abstraktionsebene, stellen all diese ORM-Technologien Alternativen zueinander dar, da sie im Grunde dasselbe Konzept implementieren. Die genauere Betrachtung hat jedoch gezeigt, dass dies nicht für die Mengen an Technologie-Features gilt, die im Funktionsumfang dieser liegen. Obwohl Technologien zu einem gewissen Teil zur Implementation desselben Konzeptes die gleichen Arten von Feature implementieren, ist eine komplette Überschneidung dieser Men-

gen unwahrscheinlich.

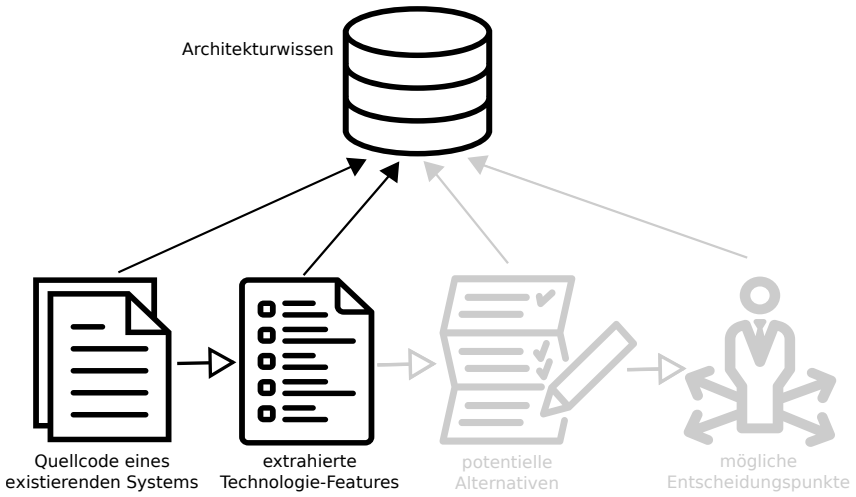
Begründet werden kann diese fehlende Abdeckung anhand herstellerspezifischer Features. Jeder Hersteller versucht, auf die eine oder andere Weise, seine Technologie zu verkaufen oder populär zu machen. Um dieses Ziel zu erreichen, setzen Hersteller häufig auf Alleinstellungsmerkmale. Dies sind meist besondere Features, die zumindest im Kontext des zugrundeliegenden Konzeptes der Technologie, in ihrer Form oder Implementation einzigartig sind. Alleinstellungsmerkmale sind meist sehr speziell auf den Kontext der Technologie zugeschnitten. Sie werden unter Umständen nicht benötigt, wenn alternative Technologien eingesetzt werden sollen. Häufig sind dies etwa Optimierungen, wie sie in *Hibernate* zu finden sind. Ein weiteres Beispiel stellen Datenbank-Dialekte dar: Jeder Hersteller implementiert zusätzlich zur Definition des Standard-SQL meist eigene Funktionen, die dessen Nutzung erleichtern oder optimieren sollen. Neben nicht-funktionalen Eigenschaften (wie etwa Antwortzeiten etc.) schränken diese trivialerweise die Portabilität der Technologie ein.

Die Darstellung von Alternativen ist, wie oben beschrieben, auf ein gemeinsames abstraktes Feature zu beziehen. Unterschiedliche Features werden sich in verschiedenen Technologien jeweils mit eigenen ASTAs ergeben. Ist das Feature tatsächlich einzigartig, lassen sich auch auf abstrakter Ebene keine Gemeinsamkeiten zu anderen Implementation finden. Dies ist insofern wichtig, als dass ein entsprechendes Feature in gewisser Weise ein Constraint bei der Suche nach Alternativen darstellen kann. Entscheidungen für Technologien können ursprünglich gerade wegen eben jener Alleinstellungsmerkmale gefallen sein. Diese sind, zusammen mit Vor- und Nachteilen häufig zentrales Entscheidungskriterium für oder gegen eine Technologie [LaToza u. a., 2013].

Finden sich für eine Technologie solche Alleinstellungsmerkmale, gibt es bei der Extraktion und Suche nach Alternativen grundsätzlich zwei Fälle zu betrachten. Im ersten Fall wird das spezielle Features nicht im analysierten System verwendet und stellt keine Problematik bei der Suche nach Alternativen dar. Es muss schlicht keine alternative Implementation gefunden werden. Wird das spezielle Feature verwendet, muss im Kontext anderer Alternativen überprüft werden, ob die Funktionalität nicht anderweitig erreicht werden kann. Ist etwa eine Optimierung unabdingbar, lässt sich die Technologie im schlimmsten Fall nicht austauschen. Lässt sich das Feature anderweitig implementieren oder in alternativen Technologien ersetzen, ist ein Constraint aufzuweichen.

Gerade letzteres ist hervorzuheben: Werden Technologien ausgetauscht, ist zu diesem Zweck meist zumindest ein Refactoring notwendig. Ist bekannt, welche Features Alternativen zueinander darstellen, lässt sich dieses gezielt vornehmen. Im Umkehrschluss bedeutet dies ebenso, dass Features, die bei Aufweichen eines Constraints nicht direkt umgesetzt werden können, bewusst Teil der Umstellung werden. Durch die explizite Betrachtung einzelner Technologie-Features kann die Umsetzung von Änderungen expliziter gestalten und so unter Umständen zur Verbesserung des Prozesses beitragen. Die explizite Betrachtung ermöglicht zudem, Risiko und Änderungsaufwand besser einzuschätzen zu können.

4.2 Extraktion von Technologie-Features



Das erweiterte Metamodell stellt die Grundlage für die Extraktion von Technologie-Features dar. Basierend auf den Überlegungen zu Quellcode-Beispiel 4.1 (Seite 46) sollten allerdings weitere Faktoren bei der Betrachtung von Technologie-Features berücksichtigt werden.

Das Beispiel zeigt bereits, dass unterschiedliche charakteristische Eigenschaften die Existenz eines bestimmten Features vermuten lassen. Diese basieren vorrangig auf der Grammatik und Syntax der analysierten Sprache. In diesem Fall gibt die Syntax der Sprache Java gewisse Sprachkonstrukte und -formate vor, anhand derer Technologien eingebunden werden können. Um die Hintergründe näher zu erläutern, wird zunächst analysiert, wie Technologien im Rahmen des Konzeptes überhaupt zu verstehen sind, und welche Informationen dies für die Extraktion liefern kann. Um weiterhin das Befüllen des Architekturwissen zu unterstützen, folgt anhand der Möglichkeiten zur Einbindung von Technologien eine Formalisierung und in Folge dessen eine exemplarische Kategorisierung der Merkmale. Dieser Abschnitt beleuchtet abschließend, wie diese Merkmale zu extrahieren sind. Ebenso betrachtet werden Schwächen und mögliche Optimierungen des Verfahrens.

Ursprung von Technologie-Features - Technologiebegriff

Bevor charakteristische Eigenschaften von Technologie-Features formalisiert werden können, muss zunächst deren Ursprung betrachtet werden. Damit einher geht die Betrachtung der diesen Technologien zugrundeliegenden Motivation.

Betrachtet man ein Softwaresystem stark abstrahiert und unabhängig von seinen Rahmenbedingungen, ist dieses auf technischer Basis lediglich ein ausführbares Programm. Es wird, je nach Programmiersprache und Plattform, aus einer Menge von Quellcode-Dateien und verwandten Bestandteilen (Konfigurations-Dateien und weitere) heraus erstellt und in unterschiedlichen Formaten ausgeführt (Binaries, Intermediate Language-Code, Skriptsprachen, ...).

Allen Varianten der Ausführung liegt dabei eine Menge von Quellcode zugrunde. Dieser Quellcode mag zwar in einigen Fällen abstrakter sein (Assembler), imperativ (Imperative Programmiersprachen) oder objektorientiert, basiert jedoch in allen Fällen auf verfasstem Quellcode. Dieser setzt das gewünschte Verhalten des Systems um. Exotische Programmiermodelle fallen eventuell nicht unter diese Betrachtungen und werden weiterhin nicht betrachtet.

Um die Arbeit an Systemen zur Erreichung dieses Verhaltens zu erleichtern, werden häufig externe Technologien eingebunden. Sie bestehen ebenfalls aus Quellcode, der jedoch durch externe Programmierer verfasst wurde. Diese Tatsache stellt eine Arbeitserleichterung dar und spart somit Zeit und Kosten. Funktionalitäten müssen nicht programmiert werden, sondern können im Rahmen der eingesetzten Programmiersprache eingebunden werden. Auf Basis der verwendeten Sprache und ihrer Möglichkeiten zur Modularisierung stellen externe Technologien ihren Quellcode über definierte Schnittstellen zur Verfügung. Zur Nutzung einer Technologie müssen diese Schnittstellen im Rahmen der Sprache eingebunden und genutzt werden. Dies sind vorrangig Programmpakete. Sie beinhalten am Beispiel objektorientierter Programmiersprachen etwa Objektstrukturen, Methoden, Klassen, Annotationen und weitere Konstrukte, welche in Folge genutzt werden können. Aufrufe an den Schnittstellen der Technologien haben einen Zweck: Implizit oder explizit eine spezifische Funktionalität der Technologie zu hervorzurufen.

Beispielhaft ist dies am Rahmen des vorangehenden Beispiels in Abschnitt 4.1 (Seite 44) erläutert. Im Rahmen von Importen kann eine Programmiersprache Programmpakete einbinden und deren Funktionalität, wie Klassen und deren Methoden nutzen, um diese Funktionalitäten in das System einzubinden. Entsprechende Sprachkonstrukte bilden die Grundlage für die Einbindung von Technologie-Features. Auch bestimmte Schlüsselworte oder Konfigurations-Dateien besitzen einen entsprechenden Informationsgehalt. Beim Aufbau von Softwaresystemen werden häufig mehrere Sprachen oder Datei-Formate eingesetzt, in denen entsprechende Hinweise auf den Einsatz von Technologie-Features existieren können. Je nach Format der einzelnen Quellcode-Dateien sind entsprechende Konstrukte zu betrachten.

Die Syntax der Sprache und die durch diese begrenzten Möglichkeiten, Quellcode in eine Anwendung einzubinden, hat einen entscheidenden Vorteil: Entsprechende Sprachkonstrukte lassen sich präzise identifizieren. Vorgegebene Schlüsselworte erlauben bei der Suche nach Technologie-Features bereits, etwa Importe einzugrenzen und somit relativ eindeutig zu identifizieren. Da die konkrete Nutzung dieser Sprachkonstrukte zur Einbindung von Features bereits eine Dynamisierung der Sprache darstellen, lassen sich entsprechende Merkmale auf Grundlage von Syntax und Grammatik leicht formalisieren.

Je Sprachkonstrukt lassen sich Regeln definieren, welche dessen Nutzung identifizieren können. Um diese jedoch umsetzen zu können, müssen folgend entsprechende Sprachkonstrukte erfasst und beispielhaft berücksichtigt werden.

Betrachtet man unterschiedliche Programmiersprachen, unterscheiden sich diese Konstrukte trivialerweise. Jede Sprache bringt ihre eigene Syntax mit sich, bietet so-

mit unterschiedliche syntaktische Konstrukte zur Einbindung von Funktionalitäten. Wenn auch im Rahmen dieses Konzeptes Java betrachtet wird, lässt sich das Grundprinzip übertragen. Etwa die Sprache C++ bietet auf Grundlage ihrer Syntax Konstrukte wie Pointer, Makros, oder den Einsatz von eines Präprozessors an. Auch diese lassen in einem gewissen Rahmen die Einbindung von Funktionalitäten zu. Andere Sprachkonstrukte, wie die Einbindung von Dateien und Programmpaketen existieren ebenfalls, unterscheiden sich jedoch anhand der konkreten Schlüsselworte (etwa `include` anstatt `import`). Entsprechend müssen für diese sowie weitere Sprachen (auch Skript-Sprachen, Datenbank-Dialekte) ebenfalls korrespondierende Konstrukte erfasst und deren Erkennung definiert werden.

Die Einbindung von Technologien erfolgt somit ausschließlich statisch. Welche Technologien somit wie oben beschrieben eingebunden werden, entscheidet sich vor der Kompilierung. Selbst dynamische Algorithmen oder Entwurfsmuster stellen hier keine Ausnahme dar:
Betrachtet man etwa das Strategy-Pattern, wird erst zur Laufzeit auszuführender Quellcode ausgewählt. Wird auf einem der Zweige ein bestimmtes Feature genutzt, ist dieses auch Teil des Systems, selbst wenn dieser Zweig praktisch nie durchlaufen wird. Das Feature wurde trotzdem explizit eingebunden und ist Teil des Systems, da dessen Einsatz im Quellcode des Systems definiert wurde. Der Einsatz von Technologie-Features ist somit eine bewusste Entscheidungen, die im Regelfall unabhängig von der Laufzeit getroffen wird.

Die Struktur des Systems selbst hat für den Einsatz von Features jedoch keine Bedeutung. Anders als die Einbindung von Sprachkonstrukten, dienen Architektur und Beziehungen zwischen Klassen sowie weitere Beziehungen lediglich der Strukturierung einer Anwendung. Sie erlauben keine Aussage über den Einsatz von Technologie-Features, sondern lediglich über Verhalten und Funktionalität der Anwendung. Werden externe Komponenten genutzt, ist deren Nutzung von Technologie-Features ebenfalls zu untersuchen. Architektur sowie deren Anforderungen und semantischer Hintergrund sind bei der Analyse von technischen Schnittstellen und Sprachkonstrukten nicht zielführend. Sie haben lediglich Einfluss auf das Verhalten der Anwendung zur Laufzeit. Woher in diesem Zusammenhang eine bestimmte Funktionalität stammt, ist auf fachlicher Ebene irrelevant.

Finale Erweiterung des Metamodells

Auf Basis der vorangegangenen Analyse von Technologien lässt sich schließlich eine finale Erweiterung des Metamodells vornehmen (Abbildung 4.7). Diese umfasst die bisher offenen Eigenschaften zur Beschreibung von Indikatoren. Eine Menge dieser soll später ein Technologie-Feature identifizieren. Folgende Parameter sind für die Betrachtung von Indikatoren zu betrachten. Anschließend wird anhand von Beispielen im Rahmen der Programmiersprache Java eine Menge von Kategorien erläutert und begründet. Aus diesen ergibt sich die Motivation für die nachfolgend genannten Parameter.

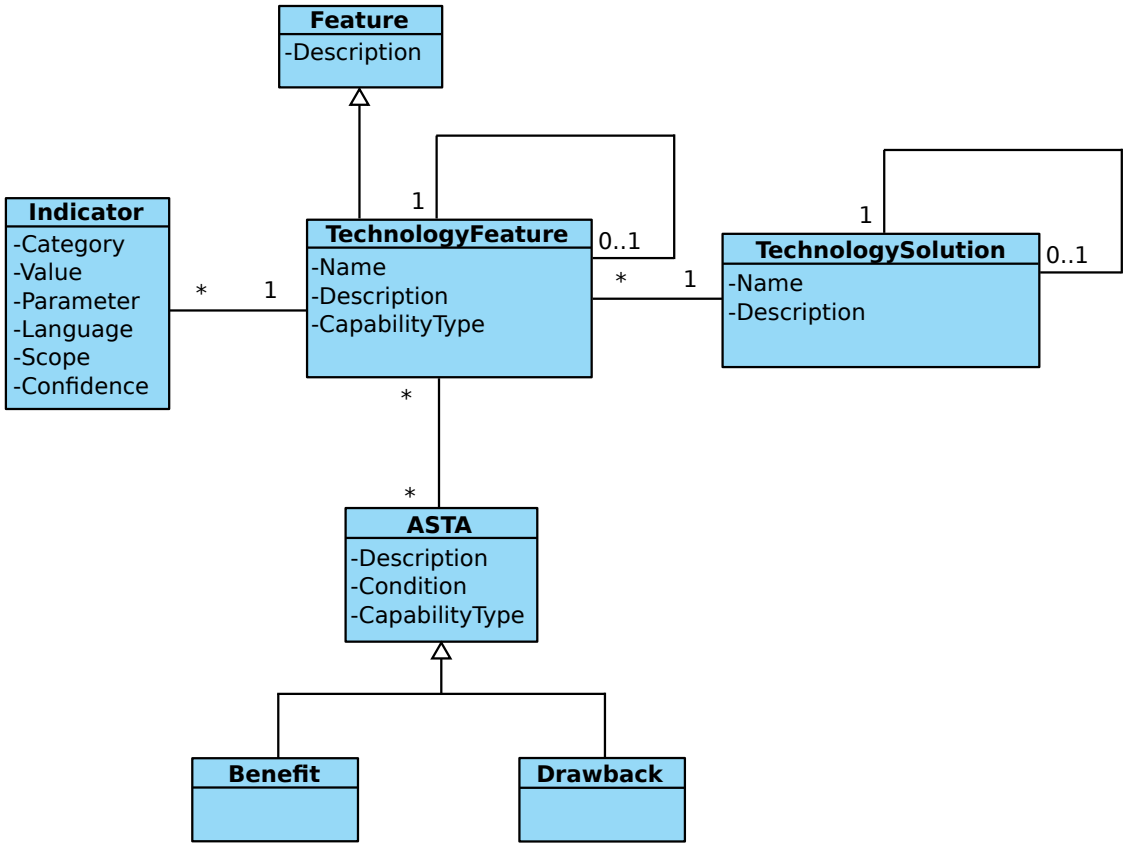


Abbildung 4.7: Finale Erweiterung des Metamodells

Category Um unterschiedliche Arten von Sprachkonstrukten und Syntax zur Einbindung von Technologien zu betrachten, erfolgt eine Kategorisierung anhand dieser. Eine Kategorisierung hilft ebenfalls, für jede Art von Konstrukt entsprechende Regeln zur maschinellen Verarbeitung festzulegen. Eine Kategorie gibt lediglich an, welche Art von Schlüsselworten und Sprachkonstrukten zu untersuchen ist. Beispiele sind hier Importe, Methodenaufrufe, Vererbung und verwandte Konstrukte. Eine Kategorie kann mit ihren Sprachkonstrukten potentiell in mehreren Programmiersprachen auftreten. In diesem Fall unterscheiden sich die entsprechenden Schlüsselworte.

Value Um die Erkennung basierend auf einzelnen Kategorien zu parametrisieren, muss ein Wert festgelegt werden. Je nach Kategorie muss dieser Wert in einem

bestimmten Kontext gesucht werden. Kategorien geben feste Schlüsselworte (Etwa `import`) vor, der Wert individualisiert diese Kategorie.
Als Beispiel dient der Wert „`org.springframework.expression`“, wird dieser der Kategorie eines Imports zugeordnet, ist konkret nach einer Zeile mit dem Inhalt ähnlich „`import org.springframework.expression;`“ zu suchen. Dies ist ein Hinweis darauf, dass eine bestimmte Schnittstelle genutzt wird, in diesem Fall ein durch das Framework *Spring*, hier als Technologie betrachtet, bereitgestelltes Programmpaket.

Parameter Sprachkonstrukte können neben ihrer Dynamisierung weitere Parameter erfordern. Begründet liegt dies etwa im Rahmen von Quellcode-Beispiel 4.1 (Seite 46): Betrachtet man den Einsatz von Literalen, muss neben dem Aufruf einer Methode (*Category*) und dem Namen der Methode (*Value*) auch betrachtet werden, welche Parameter beim Aufruf der Methode übergeben werden. Für diese muss hier konkret gelten, das die übergebenen Parameter einem bestimmten Format entsprechen. Diese Betrachtung gilt eventuell nicht für alle Kategorien und ist in diesem Fall optional.

Scope Die von Programmiersprachen und weiteren Datei-Typen beschriebenen Merkmale können in ihrer Ausprägung meist nicht beliebig verwendet werden. Sie können unter Umständen nur in bestimmten Datei-Typen, in Klassen oder Methoden auftreten. Bei der Extraktion kann somit der Suchraum für bestimmte Kategorien und einzelne Indikatoren definiert werden. Dieser kann für die spätere Optimierung der Extraktion genutzt werden, gibt jedoch in jedem Fall Aufschluss darüber, in welchem Rahmen der Indikator zu finden ist.

Language Um die korrekte Implementation einer Regel anwenden zu können ist es notwendig, die zugrundeliegende Programmiersprache zu hinterlegen. Jedem Indikator ist zusätzlich zu seiner Kategorie eine Programmiersprache zuzuordnen. Somit kann sichergestellt werden, dass relevante Schlüsselworte sowie Syntax der entsprechenden Programmiersprache betrachtet werden.

Confidence Wie die spätere Betrachtung einzelner Kategorien zeigen wird, haben nicht jeder Indikator und jede Kategorie die gleiche Aussagekraft. In einigen Fällen sind Kategorien im direkten Vergleich eher ein Indiz dafür, dass ein bestimmtes Feature eingesetzt wird, als andere. Im Rahmen eines Wertes zur Betrachtung dieses Sachverhaltes kann bei der Analyse der Ergebnisse abgeschätzt werden, wie wahrscheinlich die Nutzung des entsprechenden Features tatsächlich ist. Weiterhin wird die Betrachtung zeigen, dass nicht jeder Indikator ein eindeutiges Indiz für die Nutzung eines Features sein muss.

Exemplarische Kategorisierung von Indikatoren

Auf Grundlage des finalisierten Metamodells lassen sich Sprachkonstrukte und Syntax der Sprache Java exemplarisch zwecks Identifikation untersuchen. Entsprechende Konzepte lassen sich auch auf andere Formate und Sprachen übertragen. Anzumerken ist hierbei, dass anders als bei Ansätzen der Sprachverarbeitung [Mirakhorli u. a., 2014] größtenteils lediglich die Syntax der Sprache ausschlaggebend ist. Die Betrachtung wird zeigen, wie weit sich die Identifikation an Schlüsselworten der

Sprache und einer Dynamisierung des Konzepts durch entsprechend vorgegebene Textbausteine aus dem Architekturwissen verlassen kann.

Die einzelnen Indikator-Typen unterscheiden sich dabei in folgenden Gesichtspunkten: a) dem zu betrachtenden Sprachkonstrukt b) dem Kontext ihrer Anwendung sowie der zugrundeliegenden Motivation c) ihrer relativen Aussagekraft sowie d) Möglichkeiten zur Identifikation im Quelltext eines existierenden Systems.

A. Importe

Grundlegendes Sprachkonstrukt zur Einbindung von Technologien bilden in Java Importe. Sie weisen auf die Nutzung von Programmpaketen hin. Diese können zwar auch aus eigens verfasstem Quellcode stammen, werden jedoch ebenfalls verwendet, um Bibliotheken und ähnliche verwandte Technologien einzubinden. Der Import eines Paketes ermöglicht im Rahmen einer Klasse oder ähnlichem die Nutzung der Schnittstellen des entsprechenden Programmpakets. In den meisten Fällen ermöglicht ein Import die Nutzung weiterer Sprachkonstrukte, ist jedoch nicht verbindlich. Was dies genau bedeutet, ist später bei den beinhalteten Sprachkonstrukten zu betrachten. Die Nutzung von Programmpaketen kann auch ohne explizite Deklaration eines Imports erfolgen.

```
1  import javax.persistence.Column;
2  import javax.persistence.Entity;
3  import javax.persistence.FetchType;
4  import javax.persistence.JoinColumn;
5  import javax.persistence.ManyToOne;
6  import javax.persistence.PrimaryKeyJoinColumn;
7  import javax.persistence.Table;
8
9  import org.hibernate.annotations.BatchSize;
10
11 import de.uni_hamburg.informatik.swk.masterprojekt.
12                          decisionbuddy.util.ColumnLength;
```

Quellcode-Beispiel 4.2: Typische Importe zu Beginn einer Java-Datei

Quellcode-Beispiel 4.2 zeigt einen typischen Block von Importen zu Beginn einer Java-Datei. Aus diesem leitet sich am Beispiel des *DecisionBuddy* die Nutzungen mehrerer Programmpakete ab. Neben einigen Importen aus JPA (`javax.persistence`), greift die Datei ebenfalls auf ein *Hibernate*-Paket zu. Dieses beinhaltet die nötige Konfiguration für die im Anwendungsfall beschriebene Optimierung *Batch Reading* (`org.hibernate.annotations.BatchSize`). Zudem wird der Import eines Programmpaketes des Systems selbst definiert. Dieser ist für die Bestimmung nicht relevant. Tatsächlich relevante Importe sind im Architekturwissen zu hinterlegen.

```
1  import javax.persistence.*;
```

Quellcode-Beispiel 4.3: Beispiel Import unter Nutzung einer Wildcard

Die Deklaration von Importen lässt sich ebenfalls verkürzen. Durch Einsatz einer Wildcard können Teile der Paket-Hierarchie ausgelassen werden. Quellcode-Beispiel

4.3 zeigt die Verkürzung im Bezug auf vorangegangenes Beispiel. Anstelle der Spezifikation zur Nutzung jedes Typen aus dem Paket `javax.persistence` mithilfe eines individuellen Imports wird hier die Nutzung aller Unterpakete spezifiziert.

Wird für einen Indikator ein Import deklariert, muss der vollständige Name des zu importierenden Paketes angegeben werden. Im Fall von Beispiel 4.2, Zeile 2 ist dies der Wert „`javax.persistence.Entity`“. Wird dieser Import genutzt, ist dies ein Hinweis darauf, dass ein *Hibernate*-Feature zur expliziten Spezifikation von Datenbankobjekten genutzt wird. Zu beachten ist, dass auch auch die Nutzung aller übergeordneten Programmpakete mit einer Wildcard das gesuchte Paket einschließt.

```
1 @org.hibernate.annotations.BatchSize(size=10)
2 private ArchitecturalPatternCategory
     architecturalPatternCategory;
```

Quellcode-Beispiel 4.4: Beispiel auspezifizierte Deklaration eines Namensraumes

Bei der Nutzung entsprechender Konstrukte kann der Namensraum auch explizit angegeben werden. Beispiel 4.4 zeigt diesen Sachverhalt am Beispiel einer Annotation. Anstelle eines Imports zu Beginn der Datei wird bei Nutzung der Annotation der Namensraum explizit vorgegeben (Zeile 1).

Die Deklaration von Importen ist jedoch wenig aussagekräftig. Anders als eine konkrete Nutzungen anderer Sprachkonstrukte, stellen Importe lediglich eine Möglichkeit zur besseren Strukturierung und Übersicht dar. Da sie nicht verbindlich sind, muss die Nutzung eines Sprachkonstruktes aus diesem Programmpaket nicht dessen Import voraussetzen. Der Einsatz eines Imports kann somit ein Hinweis auf die Nutzung eines bestimmtes Features sein, sollten die Schnittstellen des Programmpakets später genutzt werden. Innerhalb der Schnittstelle eines Programmpakets können mehrere Features zu nutzen sein, was dazu führt, dass ein Import mehreren Technologie-Features zugeordnet sein kann. Ist dies jedoch ebenfalls nicht der Fall, hat der Import keine Aussagekraft. Er ist entweder fälschlicherweise deklariert worden, wurde nach einem Refactoring nicht entfernt oder überschrieben. Obwohl moderne Entwicklungsumgebungen diesen Zustand zu vermeiden wissen, ist dieser Fall dennoch nicht auszuschließen und bei der Bewertung zu berücksichtigen.
Aus einem Indikator der Kategorie Import lässt sich somit lediglich ein grober Hinweis auf die Nutzung eines oder mehrerer Technologie-Features ableiten.

Um einen Indikator dieser Kategorie für die Programmiersprache Java zu erkennen, muss lediglich das Schlüsselwort `import` gefolgt vom entsprechenden Paketnamen betrachtet werden. Dieses wird dem Indikator als *Value* zugeordnet. Aus diesem lässt sich ableiten, welche Pakete auch bei Nutzung einer Wildcard beachtet werden müssen. *Scope* ist in diesem Fall eine Java-Datei.

Tabelle 4.1 zeigt, welche Werte für die Identifikation als Element im Architekturwissen hinterlegt werden müssen.

B. Nutzung eines Datentyps

Innerhalb der importierten oder genutzten Programmpakete finden sich Klassen, Interfaces und weitere Datenstrukturen als Schnittstelle einer eingebundenen Tech-

Category	**Import**
Value	Vollständiger Name des gesuchten Programmpakets
Scope	Java-Datei
Parameter	-
Confidence	Sehr gering

Tabelle 4.1: Erfassung von Indikatoren des Typs Import

nologie. Die Nutzung bestimmter Datentypen kann, wie folgend ein Beispiel zeigen wird, bestimmte Funktionalitäten einer Technologie ausweisen.

```
1 import org.hibernate.collection.internal.PersistentBag;
2 // ...
3 PersistentBag bag = (PersistentBag) object;
4 String name = bag.get(idx);
```

Quellcode-Beispiel 4.5: Beispiel Nutzung eines speziellen *Hibernate*-Datentyps

Anhand von Quellcode-Beispiel 4.5 ist die Nutzung einer speziellen Listen-Implementation in *Hibernate* zu erkennen. Neben einem qualifizierenden Import (Zeile 1) wird ein Objekt der Klasse `PersistentBag` deklariert (Zeile 3) und anschließend verwendet. Bereits die Deklaration einer Variablen oder ähnlichem reicht aus, um die Nutzung der zugrundeliegenden Funktionalität zu identifizieren. In diesem Fall, unabhängig von der Nutzung der deklarierten Variable, wird eben diese spezielle Implementation einer Liste genutzt. Der `PersistentBag` ist im Kontext von *Hibernate* ein Feature zur Implementation einer in Java nicht vorhandenen Listen-Variante[5].

Betrachtet man den Informationsgehalt des Beispiels ist festzustellen, dass die Nutzung eines konkreten Datentyps vergleichsweise hoch ist. Verglichen mit der Betrachtung der Importe, lässt die Nutzung des Typen wenig Zweifel an der Nutzung zugrundeliegender Features. Lediglich strukturelle Missstände sind problematisch: Nicht verwendete Methoden oder unbenutzte Variablen weisen auf keine funktionale Nutzung des Features hin. Da diese Tatsache allerdings erst zur Laufzeit zu betrachten ist, muss trotzdem auf den Einsatz des Features geschlossen werden: Um die Anwendung auszuführen ist die Abhängigkeit entsprechender Technologien nichtsdestotrotz vorhanden. Dies gilt analog für definierte und nicht verwendete Variablen. Die Technologie wird für die Ausführung des Systems auf technischer, nicht jedoch auf Ebene der Softwarearchitektur benötigt und ist somit relevant. Auch hier kann davon ausgegangen werden, dass Entwicklungsumgebungen diesen inkonsistenten Zustand vermeiden sollten.

```
1 org.hibernate.collection.internal.PersistentBag bag =
    (org.hibernate.collection.internal.PersistentBag)
    object;
2 String name = bag.get(idx);
```

Quellcode-Beispiel 4.6: Beispiel vollqualifizierte Nutzung eines `PersistentBag`

Zu beachten ist dabei, dass bei Nutzung eines Typs nicht zwangsweise zuvor ein Import deklariert worden sein muss (Siehe Indikator-Typ Import). Die Nutzung

[5] https://docs.jboss.org/hibernate/orm/3.2/api/org/hibernate/collection/PersistentBag.html, zuletzt aufgerufen am 18. Oktober 2016

des Datentyps kann vollständig spezifiziert sein. Dies zeigt Quellcode-Beispiel 4.6, anstellte des Imports aus vorherigem Beispiel reicht hier die vollqualifizierte Deklaration des Datentyps aus.

Category	**TypeUsage**
Value	Name des Datentyps (Klassenname)
Scope	Der Namensraum des Typs innerhalb einer Java-Klasse
Parameter	-
Confidence	Hoch

Tabelle 4.2: Erfassung von Indikatoren des Typs Datentyp-Nutzung

C. Methodenaufruf

Lässt sich die Nutzung bestimmter Datentypen identifizieren, ermöglicht dies in Folge auch die Identifikation des Aufrufs von Methoden. Durch den Aufruf von Methoden an Objekten aus eingebundenen Technologien lassen sich explizit Funktionalitäten nutzen.

```
1 Session session = sessionFactory.openSession();
2 Transaction tx = session.beginTransaction();
3 // ...
4 tx.commit();
5 session.close();
```

Quellcode-Beispiel 4.7: Beispiel Aufruf einer Methode zum Starten einer Transaktion

Anhand von Quellcode-Beispiel 4.7 lassen sich die Hintergründe dieser Kategorie erläutern. Um ein Objekt des Datentyps `Transaction` zu initialisieren wird die spezielle Methode `beginTransaction()` aufgerufen (Zeile 2). Diese macht die Nutzung einer Transaktion explizit. Die Deklaration des Datentyps ist weniger aussagekräftig als der tatsächliche Beginn der Transaktion, indiziert durch die Methode. Das Beenden der Transaktion ist ebenfalls dieser Kategorie zuzuordnen (Zeile 5). Ähnliches gilt für das Öffnen und Schließen einer Datenbank-Verbindung (Zeile 1). Diese Menge von Informationen veranlasst zu der Annahme, dass das dem Beispiel zugrundeliegende System Datenbank-Transaktionen nutzt.

Eine Besonderheit ist dennoch zu beachten: Methoden sind häufig mit bestimmten Parametern aufzurufen. Je nach Auswahl dieser kann der Aufruf auf die Ausprägung unterschiedlicher Features hinweisen.

```
1 Expression exp = parser.parseExpression("'Hello World'");
```

Quellcode-Beispiel 4.8: Beispiel Aufruf einer Methode mit qualifizierenden Parametern

Als Beispiel dient hier die *Spring*-SpEL aus Beispiel 4.1 (Seite 46). Quellcode-Beispiel 4.7 betrachtet die Methode `parseExpression`. Deren Aufruf weist bereits auf die Nutzung der SpEL hin. Anhand des Formats des übergebenen Parameters lässt sich jedoch die Nutzung eines weiteren abhängigen Features identifizieren.

Das Literal `'Hello World'` innerhalb des Parameters weist darauf hin, dass auch

die ausgewiesene Unterstützung von Literalen genutzt wird. Dies ist insofern relevant, als das Technologien bei der Suche nach Alternativen eben auch diese spezielle Form der Verarbeitung unterstützen müssen.

Verglichen mit Importen und der Nutzung von Datentypen ist die Aussagekraft eines Methodenaufrufes sehr hoch. Anders als Deklarationen oder ähnliche Sprachkonstrukte, führen imperative Methodenaufrufe mitunter zu konkreten Ergebnissen. Sie generieren Rückgabewerte, stoßen die Verarbeitung an und implementieren im Allgemeinen konkrete Funktionalitäten. Die Nutzung dieser ist somit, in Kombination mit einem korrespondierenden Datentyp, ein deutliches Indiz für die Nutzung eines bestimmten Features.

Anhand der Formalisierung lassen sich mit den bereits vorgestellten Indikatoren entsprechende Sachverhalte abbilden. Die Identifikation dieser anhand der Schlüsselworte ist auf Basis des Architekturwissens ebenfalls simpel. Aus dem Namen der Methode sowie potentiellen Paramentern lässt sich, gemeinsam mit weiteren Indikatoren zur Identifikation der Nutzung eines Datentyps, auch die Nutzung eines Technologie-Feature identifizieren.

Category	**MethodCall**
Value	Name der Methode
Scope	Objekt der Klasse
Parameter	Aufrufparameter
Confidence	Sehr hoch

Tabelle 4.3: Erfassung von Indikatoren des Typs Methodenaufruf

D. Vererbung

Unter Betrachtung des Inversion of Control-Prinzips können Schnittstellen von Technologien ebenfalls durch Vererbung genutzt werden. Um Funktionalitäten aus Technologien zu überschreiben oder eigene Datentypen in die Architektur von Bibliotheken einzureihen, kann von diesen Komponenten geerbt werden. Diese Tatsache nutzt die Polymorphie der Sprache aus, und ermöglicht es so dem Nutzer der Technologie, eigene Implementation in etwaige Funktionalitäten einzubinden oder zu konkretisieren.

```
1 import org.springframework.context.Lifecycle;
2
3 public interface LifecycleProcessor extends Lifecycle
4 {
5   // ...
6 }
```

Quellcode-Beispiel 4.9: Beispiel Vererbung eines `Lifecycle`

Als Beispiel dient der Lebenszyklus der von *Spring* verwalteten Objekte[6] (Quellcode-Beispiel 4.9). Wie das definierte Interface zeigt, erbt der `LifecycleProcessor` von

[6] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-factory-lifecycle-processor, zuletzt aufgerufen am 18. Oktober 2016

`Lifecycle`. Durch diese Tatsache definiert sich der Prozessor als ein von *Spring* verwaltetes Objekt. Durch die Einbindung in den *Spring*-Lebenszyklus werden dessen Features genutzt.

Im Bezug auf ihre Aussagekraft liegt für diese Kategorie der Vergleich mit der Nutzung eines Typen nahe. Beide Kategorien berufen sich auf die rein deklarative Natur der zugrundeliegenden Sprachkonstrukte. Anders als Methodenaufrufe werden durch die Vererbung allerdings keine expliziten Funktionalitäten hervorgerufen. Die Vererbung sorgt lediglich dafür, dass an anderer Stelle, etwa durch Aufruf einer Methode, Funktionalitäten genutzt werden können. Die Vererbung kann dennoch ein Indiz dafür sein, dass eben diese Funktionalität an anderer Stelle genutzt wird.

Um diese Kategorie zu identifizieren, kann das Java-Schlüsselwort `extends`, gefolgt von dem Namen der zu erbenden Klasse, genutzt werden. Die Extraktion ist entsprechend trivial. Tabelle 4.4 beschreibt die entsprechende Formalisierung zur Befüllung des Architekturwissens.

Category	**Extends**
Value	Name der zu erbenden Klasse
Scope	Java-Datei
Parameter	Der Namensraum der zu erbenden Klasse
Confidence	Neutral

Tabelle 4.4: Erfassung von Indikatoren des Typs Vererbung

E. Implementation von Interfaces

Ähnlich der Vererbung bietet auch die Implementation von Interfaces eine Möglichkeit zur Einbindung eigener Funktionalitäten in externe Technologien. Eine entsprechende Implementation nutzt gegebene Schnittstellen, um eigene Funktionalitäten an eine gegebene Struktur anzupassen.

```
1  public class ArchitecturalPatternValidator
2      extends SolutionValidator implements Validator
3  {
4    @Override
5    public boolean supports(Class<?> object)
6    {
7      // ...
8    }
9
10   @Override
11   public void validate(Object object, Errors errors)
12   {
13     // ...
14   }
15 }
```

Quellcode-Beispiel 4.10: Beispiel Implementation eines Interfaces

Betrachtet werden kann dies am Beispiel der Implementation des *Spring*-Interfaces `Validator`[7] (Quellcode-Beispiel 4.10). Die Implementation dieses Interfaces erfordert lediglich die individuelle Implementation der Methoden `supports(...)` sowie `validate(...)`. Vorteil bei der Nutzung eines solchen Interfaces ist zumeist, dass entsprechende implementierende Klassen in weitere Funktionalitäten der Technologie eingebunden werden können. Anders als im Fall der Vererbung werden jedoch keine Funktionalitäten aus der Oberklasse übernommen. Im Fall des im *Decision-Buddy* eingesetzten *Spring*-MVC bedeutet dies konkret, dass für alle dieses Interface implementierenden Klassen zur Laufzeit eine Validierung vorgenommen werden kann. Hieraus resultiert die Nutzung eines Technologie-Features zur Nutzung von Validation zur Laufzeit. Dieses Feature wäre ohne Nutzung der Technologie individuell zu implementieren.

Die Aussagekraft dieses Indikators entspricht etwa der der Vererbung. Ähnlich wie diese hat die Implementation eines Interfaces ebenfalls einen rein deklarativen Hintergrund. Die Implementation weist zwar auf eine Nutzung hin, zeigt jedoch nicht direkt auf, an welcher Stelle das konkrete Feature genutzt wird. Im Kontext des Beispiels bedeutet dies, dass statisch nicht zu bestimmen ist, ob die *Spring*-interne Implementation zur Laufzeit durchgeführt wird. Es kann dennoch davon ausgegangen werden, dass das entsprechende Feature genutzt wird.

Um Indikatoren dieses Typs erkennen zu können, liegt der Fokus auf dem Java-Schlüsselwort `implements`. Analog zur Vererbung stellt dieses ein eindeutiges Indiz für die Nutzung einer Implementation dar. Es wird um den Namen des zu implementierenden Interfaces ergänzt und ist lediglich zu Beginn einer Klassen-Deklaration und verwandter Konstrukte zu finden. Tabelle 4.5 zeigt die Formalisierung diesen Typs.

Category	**Implements**
Value	Name des zu implementierenden Interfaces
Scope	Java-Datei
Parameter	Der Namensraum des zu implementierenden Interfaces
Confidence	Neutral

Tabelle 4.5: Erfassung von Indikatoren des Typs Implementation

F. Annotationen

Die Dokumentation der Sprache Java definiert Annotationen als:

> „*Annotations*, a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.“[8]

[7] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html#validator, zuletzt aufgerufen am 18. Oktober 2016

[8] https://docs.oracle.com/javase/tutorial/java/annotations/, zuletzt aufgerufen am 18. Oktober 2016

Durch ihre ebenfalls deklarative Natur weisen auch Annotationen auf die Nutzung bestimmter Features hin. Sie weisen den Compiler zur Generierung zusätzlichen Quellcodes an. Aus dieser Tatsache lassen sich weitere Features ableiten. Annotationen stellen somit ein indirektes Sprachkonstrukt dar, welches zur Einbindung von Funktionalitäten aus Technologien und damit zur Identifikation derer Features genutzt werden kann.

```
1 @BatchSize(size=25)
2 private ArchitecturalPatternCategory
     architecturalPatternCategory;
```

Quellcode-Beispiel 4.11: Beispiel einfache Annotation zur Nutzung von Batch Reading

Quellcode-Beispiel 4.11 zeigt anhand des Features *Batch Reading* die Nutzung einer Annotation. Die im Anwendungsfall betrachtete Optimierung des *DecisionBuddy* findet faktisch zur Laufzeit statt und ist damit prinzipiell dynamisch. Durch die Definition der Annotation im Quellcode wird die Optimierung jedoch statisch konfiguriert und der Laufzeitumgebung vorgegeben.

Das Beispiel erläutert einen weiteren Sonderfall der Annotation: Diese können über weitere Parameter konfiguriert werden. Im konkreten Fall gibt diese Konfiguration die Menge der pro Anfrage abzurufenden Datenbank-Objekte fest. Hier besitzt der Parameter für die Identifikation keine zusätzliche Aussagekraft.

```
1 @Entity
2 @org.hibernate.annotations.Proxy(lazy = false)
3 @Table(name = "ArchitecturalPattern")
4 @PrimaryKeyJoinColumn(name = "SolutionID")
5 public class ArchitecturalPattern extends Solution
```

Quellcode-Beispiel 4.12: Vollqualifizierte Annotation, mit Parameter

Dies ist jedoch in Quellcode-Beispiel 4.12 der Fall: Neben der Nutzung der Optimierung mittels eines Proxys (Zeile 2) gibt die Annotation anhand des Parameters `lazy = false` ebenfalls vor, welche spezielle Form dieser Optimierung beim Laden von Datenbank-Objekten durchgeführt werden soll. Je nach Ausprägung der Parameter können sich unterschiedliche Formen ergeben. Wird als Parameter etwa `lazy = true` angegeben, ändert sich das Laufzeitverhalten erheblich.

Die Annotation `@Entity` (Zeile 1) kommt ohne Parameter aus. Sie identifiziert explizit die Konfiguration eines Datenbank-Objektes. Auch diese Konfiguration lässt sich bereits als Technologie-Feature identifizieren: *Hibernate* unterstützt die explizite Definition von Datenbank-Objekten im Quellcode.

Eine weitere Besonderheit findet sich ebenfalls in Zeile 2: Die Annotation ist, ähnlich der Nutzung eines Datentypen, vollqualifiziert. Annotationen stammen für gewöhnlich ebenfalls aus Programmpaketen. Diese werden entsprechend über einen Import eingebunden oder, wie im Beispiel dargestellt, vollqualifiziert. Das Programmpaket wird somit spezifiziert. Der Hintergrund ist trivialerweise derselbe, wie bei der Betrachtung von Datentypen.

Da Annotationen per Definition keinen direkten Einfluss auf den Programmablauf haben, ist deren Aussagekraft stark kontextabhängig. Erst der konkrete Einsatz bestimmter Technologien, etwa *Spring* oder *Hibernate*, interpretiert Annotationen und resultiert folglich in der Nutzung daraus abgeleiteter Features. Die reine Definition von Annotationen auf Grundlage der Syntax ist dennoch ein deutlicher Hinweis auf die Nutzung bestimmter Features. Sie sind weniger aussagekräftig als die Nutzung von Datentypen und Methodenaufrufen. Aufgrund ihrer deklarativen Natur erscheinen sie jedoch ähnlich aussagekräftig wie Vererbung oder Implementation.

Die Extraktion von Annotationen erfolgt ebenfalls auf Basis der durch Java bereitgestellten Syntax. Diese gibt dem Compiler durch das Zeichen `@` gefolgt vom Namen der Annotation entsprechende Anweisungen. Zusätzlich sind, wie oben beschrieben, Parameter sowie vollqualifizierte Deklarationen zu beachten.

Category	**Annotation**
Value	Name der Annotation
Scope	Programmpaket, welches die Annotation bereitstellt
Parameter	Optional: Parameter der Annotation
Confidence	Neutral

Tabelle 4.6: Erfassung von Indikatoren des Typs Annotation

G. Schlüsselwort

Neben syntaktischen Sprachkonstrukten können weitere Indikatoren bei der Extraktion von Technologie-Features helfen. Diese setzen sich, ähnlich zu Ansätzen der Sprachverarbeitung, aus fest definierten Schlüsselworten oder einer Kette festgelegter Zeichen zusammen und können einen Hinweis auf die Nutzung eines Features geben.

Anders als syntaktische Konstrukte sind diese Schlüsselworte nicht an feste Dateiformate gebunden. Quellcode-Beispiel 4.13 zeigt einen Ausschnitt aus der Datenbank-Konfigurationsdatei des *DecisionBuddy*. Um festzulegen, welcher Datenbank-Treiber zur Laufzeit geladen werden soll, ist bei Angabe der Ziel-Datenbank durch den Bestandteil `jdbc:mysql://` die Nutzung einer MySQL-Datenbank spezifiziert. Eine explizite Auswahl des Treibers (Zeile 1) anhand `com.mysql.jdbc.Driver` ist ein weiteres Indiz für die Nutzung einer MySQL-Datenbank. Im Java-Quellcode des *DecisionBuddy* findet sich dagegen kein Hinweis auf die konkret genutzte Datenbank-Technologie.

```
1 db.driver= com.mysql.jdbc.Driver
2 db.url=jdbc:mysql://localhost:3306/DecisionBuddy
```

Quellcode-Beispiel 4.13: Beispiel Schlüsselwort zur Auswahl eines Datenbank-Dialektes

Auffällig ist entsprechend, dass die Nutzung dieses Features hier durch kein anderes Konstrukt der Sprache zu identifizieren ist. Die Nutzung der konkreten Datenbank-Technologie ist über das gesamte System hinweg lediglich durch die Parametrisierung

der Konfiguration festgelegt. Das Text-Format zur Ausweisung eines Datenbank-Treibers ist allein durch diese spezifische Komponente der entsprechenden Textzeile auszumachen.

Anhand von Konfigurationsdateien lässt sich auf wichtige Features schließen. Der Einsatz der konkreten Datenbank-Technologie hat unter Umständen einen starken Einfluss bei der Suche nach Alternativen. Die genutzte Datenbank-Technologie bildet ein zentrales Constraint bei der Suche, da eine Datenmigration häufig ein immenses Risiko und einen hohen Kostenaufwand mit sich bringen kann [Morris, 2012]. Alternative Technologien sollten somit die bereits eingesetzte Datenbank-Technologie ebenfalls unterstützten, um den Aufwand zu reduzieren.

Sind essentielle Features an keinem syntaktischen Merkmal des Systems festzumachen, müssen entsprechende Sonderfälle berücksichtigt werden. Aufgrund eines Mangels an zugrundeliegenden, festen Sprachkonstrukten der Syntax ist ihre Aussagekraft allerdings leidlich gering. Die Betrachtung von Zeichenketten ohne reservierte Schlüsselworte verringert deren Eindeutigkeit im Hinblick auf die Verwendung eines Features erheblich. Dies kann bei der Extraktion zu falschen positiven Ergebnissen führen und verringert im Mittel die Qualität dieser.

Verfeinert werden könnte der Ansatz entsprechend, sollten für die Syntax von Konfigurationsdateien entsprechende Konstrukte und Regeln definiert sein. Da dies aufgrund der Menge an unterschiedlichen Formaten zur Konfiguration einen hohen Aufwand darstellt, beschränkt sich die Betrachtung hier auf die Analyse entsprechender Schlüsselworte. Klassische Ansätze der Sprachverarbeitung könnten das Konzept verfeinern (Als Beispiel siehe [Mirakhorli u. a., 2014], [Soliman u. a., 2016]).

Category	**Keyword**
Value	Schlüsselwort
Scope	Name bzw. Format der zu betrachtenden Datei
Parameter	-
Confidence	Gering

Tabelle 4.7: Erfassung von Indikatoren des Typs Schlüsselwort

H. Regulärer Ausdruck

Parallel zur Suche nach spezifischen Schlüsselworten und Zeichenketten ist auch der Textvergleich anhand regulärer Ausdrücke geeignet, um auf den Einsatz von Features zu schließen. Die Motivation hinter dieser Betrachtung ergibt sich analog zum Indikator-Typ Schlüsselwort.

Anstelle eine feste Folge von Zeichen in den statischen Texten des Quelldateien des Systems zu suchen, kann hiermit eine Menge von ähnlichen oder verwandten Zeichenketten abgedeckt werden. Genutzt werden kann dies etwa, um Groß- und Kleinschreibung zu ignorieren, Wertebereiche abzudecken, oder die Formatierung von Texten zu ignorieren. Weiterhin sind durch diese Kategorie bestimmte Zeilen einer Konfigurationsdatei in ihrem Wertebereich einzugrenzen.

Auch in diesem Fall ist die Aussagekraft entsprechender Indikatoren gering. Aufgrund fehlender Regeln zur Betrachtung der Syntax in Konfigurationsdateien muss die Betrachtung über die Qualität des regulären Ausdrucks gesteuert werden. Ähnlich wie die Extraktion von Indikatoren für tatsächliche Sprachkonstrukte lässt ein regulärer Ausdruck allerdings eine feinere Steuerung der Granularität zu. Dies sorgt unter Umständen für weniger falsche positive Ergebnisse.

Category	**Regulärer Ausdruck**
Value	Regulärer Ausdruck
Scope	Name bzw. Format der zu betrachtenden Datei
Parameter	-
Confidence	Gering

Tabelle 4.8: Erfassung von Indikatoren des Typs Regulärer Ausdruck

I. Implizite Technologie-Features

Ohne konkrete Betrachtung im Rahmen des Anwendungsfalls lassen sich Überlegungen zu einer weiteren Kategorie anstellen. Die Betrachtung potentieller Alternativen hat im Zusammenhang eine Reihe expliziter Features ergeben, die sich nicht anhand syntaktischer Merkmale oder ihrer Struktur und somit dem Quelltext einer Anwendung identifizieren ließen.

Fragwürdig ist, ob sich alle Features konkret durch die Nutzung von Sprachkonstrukten im Quelltext identifizieren lassen. Die leichtgewichtige ORM-Technologie *Sormula* wirbt etwa mit einer Fähigkeit, Funktionalitäten ohne Konfiguration nutzen zu können. Dies bedeutet konkret, dass Funktionalitäten des ORM-Prinzips, etwa das Abrufen oder Speichern von Datenbank-Objekten und deren Zuordnung zu Java-Objekten ohne Konfiguration möglich ist. Dies unterscheidet die Technologie von *Hibernate* oder ähnlichen Technologien, welche diese Konfiguration explizit vornehmen.

Weiterhin schließt dies optionale Komponenten von Technologien mit ein. Viele Technologien legen ohne nähere Konfiguration bei der Nutzung bestimmter Funktionalitäten einen Standard-Wert fest. Wird eine manuelle Konfiguration vorgenommen, ändert sich unter Umständen das Verhalten der Technologie. Dieses geänderte Verhalten lässt sich als eigenes Features verstehen, entsprechende Funktionalitäten werden meist explizit angeboten und haben häufig Einfluss auf funktionale sowie nicht-funktionale Eigenschaften eines Systems.

Diese Kategorie wird ohne konkretes Beispiel im Rahmen des Anwendungsfalls folgend nicht weiter betrachtet.

Extraktion von Technologie-Features

Anhand vorangegangener Kategorisierung relevanter Sprachkonstrukte lässt sich eine automatisierte Extraktion dieser betrachten. Die vorrangig fest definierte Syntax anhand definierter Schlüsselworte macht die Extraktion potentiell verlässlich und lässt eine hohe Genauigkeit erwarten. Eine Menge fester Regeln für Syntax und die Betrachtung relevanter Sprachkonstrukte ist Grundlage für die Bestimmung eingesetzter Technologie-Features.

Feature:	*Batch Reading* (*Hibernate*)
Category	Import
Value	`org.hibernate.annotations.BatchSize`
Scope	Java-Datei
Parameter	-
Category	Annotation
Value	`BatchSize`
Scope	`org.hibernate.annotations`
Parameter	-

Tabelle 4.9: Indikatoren zur Identifikation des Technologie-Features *Batch Reading*

Ein Beispiel in Tabelle 4.9 illustriert die Modellierung von Indikatoren anhand des im Anwendungsfall beschriebenen Technologie-Features *Batch Reading*. Neben einem (optionalen) Import des Programmpakets `org.hibernate.annotations.BatchSize` wird der Einsatz vorrangig durch die Annotation `BatchSize` explizit. Sind diese Textbausteine in einer Quellcode-Datei der Sprache Java im Rahmen des untersuchten Systems vorhanden, kann auf die Nutzung des Features geschlossen werden. Die Tabelle zeigt, welche Merkmale der Indikatoren im Architekturwissen hinterlegt werden müssen.

Es kann somit bestimmt werden, welche Einheit des Systems welches Technologie-Feature nutzt. Diese Betrachtung lässt sich auf Ebene des Programmpakete aggregieren und ermöglicht somit eine komponentenorientierte Analyse des Systems.

Für jeden zuvor beschriebenen Indikator-Typ sind individuelle Regeln zu verfassen. Diese stellen anhand einer textbasierten Analyse die Existenz der jeweiligen Sprachkonstrukte fest. Es ist dabei für jedes Statement zu überprüfen, ob die entsprechende Zeile im Text ein entsprechendes Sprachkonstrukt in der jeweiligen Ausprägung enthält.

```
1 for sourceFile in ExistingSystem
2   for technologyFeature in AK
3     for indicator in technologyFeature.indicators
4       check(sourceFile, indicator)
5       calculateConfidence
6     end for
7   end for
8 end for
```

Quellcode-Beispiel 4.14: Pseudocode Extraktionsalgorithmus

Quellcode-Beispiel 4.14 beschreibt im Pseudocode das Vorgehen bei der Suche nach Technologie-Features. Zu beachten ist hierbei, dass es sich bei der Suche um das grundsätzliche Vorgehen zur Analyse handelt. Es muss, analog zu obigen Überlegungen, jede aussagekräftige Datei des Zielsystems betrachtet werden (Zeile 1).

Dies schließt alle Dateien ein, die eine Auswirkung auf Verhalten und Struktur der Anwendung haben. Neben Java-Dateien ist somit auf Konfigurations- und Mapping-Dateien zu untersuchen. Log-Dateien und Meta-Daten enthalten meist keine Informationen, die auf den Einsatz eines Technologie-Features schließen lassen. Sie können bei der Extraktion ignoriert werden.

Einzelne Dateien gliedern sich in eine Menge einzelner Textzeilen. Annahme ist hier, dass jede Zeile eine vollständige Anweisung enthält, eine Anweisung somit nicht umgebrochen oder formatiert wurde. Für jedes betrachtete Technologie-Feature müssen jeweils die für dieses hinterlegten Indikatoren untersucht werden. Pro Zeile (Anweisung) jeder Datei muss diese prinzipiell auf alle Indikatoren hin untersucht werden.

Jeder Indikator gehört einer einzigen Kategorie an. Diese bestimmt, wie die Inhalte des Architekturwissen für diesen Indikator zur Extraktion genutzt werden müssen. Praktisch beeinflusst dies, auf welche Schlüsselworte eine Textzeile hin untersucht werden muss.

Anhand der gefundenen Indikatoren lässt sich die Aussagekraft des Ergebnisses für die betrachtete Datei errechnen (Zeile 5). Das genaue Vorgehen wird nachfolgend betrachtet. Prinzipiell ist dabei zu berücksichtigen, welche Indikatoren für das jeweilige Technologie-Feature gefunden wurden und welche Aussagekraft sie jeweils besitzen. Anhand dieser Informationen lässt sich eine relative Aussage über die Sicherheit treffen, mit der das betreffende Feature tatsächlich genutzt wird. Da es sich bei dem vorgeschlagenen Konzept um eine textbasierte Analyse handelt, ist eine fehlerfreie Bestimmung der eingesetzten Features nicht in jedem Fall möglich. Die Betrachtung einer Konfidenz je extrahiertem Ergebnis kann später bei der Betrachtung Alternativen zur Aufweichung von Constraints genutzt werden.

Ansätze zur Reduktion der Komplexität

Das Vorgehen resultiert in einer hohen Komplexität. Diese liegt in $O(n^m)$, wobei n der Anzahl der zu untersuchenden Textzeilen entspricht und m die Gesamtzahl aller Indikatoren im Architekturwissen beschreibt. Ein gewisses Optimierungspotential existiert dennoch. Dieses wird anschließend exemplarisch anhand der Programmiersprache Java betrachtet.

Die Sprache Java definiert neben den für die Indikatoren berücksichtigten Sprachkonstrukten weitere Regeln ihrer Syntax. Nicht jeder Textbaustein kann beliebig innerhalb einer Quellcode-Datei auftreten. Wird dieser Umstand bei der Extraktion berücksichtigt, kann die vorangehend beschriebene Komplexität drastisch reduziert werden.

Für Konfigurationsdateien und weitere nicht Java-verwandte Dateitypen gilt dies unter Umständen nicht oder entsprechend ihrer eigenen Syntax. Wie bereits bei der Betrachtung des Indikator-Typs Schlüsselwort eingehend erwähnt, erfordert dies eine grundsätzliche Erweiterung des Konzepts im Hinblick auf weitere Sprachen und Formate. Diese Optimierung wird in Folge nicht näher betrachtet. Zur Bestimmung von relevanten Merkmalen in entsprechenden Formaten wird weiterhin die Suche nach Schlüsselworten angenommen.

Der Prozess der Extraktion von Technologie-Features wird grundsätzlich nicht als zeitkritisch angesehen. Es existieren keine Echtzeitanforderungen an das Verfahren. Somit ist in Frage zu stellen, warum Optimierungen an sich überhaupt betrachtet werden sollten. Der Hintergrund ist die Genauigkeit des Verfahrens. Wird jede Zeile des Zielsystems auf jeden Indikator hin untersucht, besteht eine Wahrscheinlichkeit, entsprechende Textzeilen auch an einer nicht relevanten Stelle zu finden. Auskommentierter Quellcode, Kommentare oder Freitexte können prinzipiell ebenfalls die entsprechenden Schlüsselworte enthalten und somit falsche positive Ergebnisse erzielen.

Um diesen Umstand zu vermeiden, kann die Suche je Sprachkonstrukt auf die relevanten Bestandteile einer Quellcode-Datei reduziert werden. Nachfolgend sollen einige grundsätzliche Betrachtungen zwecks Maßnahmen zur Verfeinerung des Grundkonzeptes vorgestellt werden. Anschließend werden grundsätzliche Problematiken beleuchtet, welche die automatisierte Extraktion von Technologie-Features erschweren können.

Importe Der typische Block an Importen zu Beginn einer Java-Datei ist ein fest definiertes syntaktisches Konstrukt. Er kann lediglich zu Beginn der Datei, nach dem Namen des aktuellen Programmpaket und vor etwaigen Definitionen platziert sein[9].
Mit diesem Wissen kann sichergestellt werden, dass eine Datei bei Verarbeitung zuerst lediglich auf Importe hin untersucht wird. Dies reduziert die Komplexität im Folgenden ungemein: Die Menge an Indikatoren des Typs Import müssen unterhalb der Importe nicht weiter berücksichtigt werden.

[9]Siehe https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html, zuletzt aufgerufen am 18. Oktober 2016

Methodenaufrufe Methodenaufrufe erfolgen zumeist an konkreten Instanzen bestimmter Objekte. Entsprechende Indikatoren sind somit nur in einem begrenzten Kontext zu identifizieren. Um den Kontext eines Methodenaufrufes zu identifizieren ist die Kombination mit Ansätzen zur strukturellen Analyse der zugrundeliegenden Dateien notwendig.

Vererbung und Implementation Vererbung und Implementation sind in diesem Zusammenhang parallel zu betrachten. Ihre Deklaration kann ausschließlich im Rahmen einer Klassen- oder Interface-Deklaration erfolgen. Dies bedeutet, dass auf Indikatoren dieser Typen lediglich in einer reduzierten Menge an Zeilen einer Java-Datei geprüft werden muss.

Zeilen ohne Informationsgehalt Nicht jede Zeile einer Datei trägt tatsächlich relevante Informationen für die Identifikation von Features. Zu diesen gehören etwa leere Zeilen oder Kommentare. Klammern, Semikola, Punkte und weitere Zeichen sind ohne konkrete Anweisung nicht zu interpretieren.
Um die Menge an zu untersuchenden Zeilen zu reduzieren, ist der Text zu bereinigen. Entsprechende Zeilen ohne Informationsgehalt können entfernt werden. Das Grundkonzept überprüft auch für diese Zeilen jeden Indikator, wird jedoch in keinem Fall ein positives Ergebnis erzielen. Entsprechende Zeilen können bei der Analyse einer Datei seiteneffektfrei ignoriert werden.

Scope Der bereits definierte *Scope* ist bereits ein Mittel zur Eingrenzung der Komplexität. Er gibt unter anderem an, in welchem Datei-Typ ein bestimmtes Merkmal zu finden ist. Andere Datei-Typen müssen nicht auf den entsprechenden Indikator hin untersucht werden. Dies reduziert die Menge der zu überprüfenden Indikatoren pro Textzeile erheblich.

Schwächen

Obwohl die genannten Möglichkeiten das Verfahren verfeinern können, sind zusätzlich einige Schwächen der alleinigen Betrachtung der Syntax zu beachten. Inwieweit sich diese Schwächen auf die Ergebnisse auswirken, zeigt später die Betrachtung und Evaluation der prototypischen Implementation.

Dateiformate Gerade unter Betrachtung des *Scope* spielt die Benennung der zu analysierenden Dateien ein große Rolle. Sind Dateien nicht entsprechend ihres tatsächlichen Formates benannt, kann dies keine korrekte Analyse sicherstellen. Die Dateien können frei benannt und dennoch verarbeitet werden. Bei der Analyse sind jedoch potentiell weitere Bestimmungen notwendig. Bei der Sondierung der Quelldateien muss das tatsächliche Format für jede Datei festgestellt werden. Dies erhöht die Komplexität der Analyse.

Formatierung Unterschiedliche Quellcode-Konventionen machen eine einheitliche Formatierung der zu analysierenden Dateien unwahrscheinlich. Dies kann unter Umständen für umgebrochene Anweisungen, unkonventionelle Klammersetzung und weitere Schwierigkeiten sorgen. Um diesem Umstand vorzubeugen, muss der Quelltext vor der Analyse entsprechend einheitlich formatiert werden. In Anbetracht des vorangegangenen Konzepts ist eine Anweisung pro Textzeile sinnvoll. Diese bildet eine sinnvolle Einheit zur Identifikation charakteristischer Eigenschaften.

Überdeckung (Shadowing) Eine strukturelle Schwäche weist das Konzept im Hinblick auf die Überdeckung von Methoden, Klassen und Programmpaketen auf. Da es sich im Rahmen des Konzepts lediglich um Textvergleiche handelt, muss die potentielle semantische Mehrdeutigkeit von Bezeichnern erwähnt werden. Die Sprache Java erlaubt es, Bezeichner und dergleichen frei zu wählen. Dies schließt Klassennamen, Variablen-Bezeichner und weitere ein. Da dies allerdings exakt die Merkmale sind, die sich die Extraktion von Technologie-Features zu Nutze macht, muss ihnen eine besondere Betrachtung zuteil werden. Die Wahl von Schlüsselworten und Bezeichnern kennzeichnet Bibliotheken und Programmpakete. Entsprechende Bezeichner wurden vom Verfasser dieser Bibliotheken gewählt. Andere Implementationen können, unfreiwillig oder nicht, dieselbe Benennung erhalten. Dies sorgt dafür, dass Technologien unter Umständen nicht eindeutig identifiziert werden können.
Es liegt jedoch die Vermutung nahe, dass dieser Fall in der Menge der Bezeichner sehr unwahrscheinlich ist und deshalb in diesem Rahmen vernachlässigt werden kann. Geht man auch hier von Quellcode-Konventionen aus[10], sind Programmpakete mit dem Namen der verfassenden Organisation und somit relativ eindeutigen Merkmalen zu versehen. Sie unterscheiden sich somit von anderweitig implementierten Programmpaketen in der nutzenden Organisation. Ist dies der Fall, lässt sich im Rahmen von Importen und vollqualifizierten Bezeichnern eindeutig eine bestimmte Technologie als Ursprung einer Benennung ausmachen.

Gewichtung und Unschärfe extrahierter Technologie-Features

Die Ergebnisse der vorangegangenen Extraktion bedürfen weiterer Betrachtung. Wie bereits bei der Kategorisierung der Indikatoren angegeben, besitzen einzelne Typen eine unterschiedliche Aussagekraft. Selbiges gilt unter Umständen auch für die Formalisierung einzelner Features innerhalb einer Technologie.

Auf Grundlage dieser Überlegungen erscheint es wenig sinnvoll, die relativ simple Texterkennung alleine als hinreichend genau anzusehen. Um die Aussagekraft der Extraktion selbst geeignet zu betrachten, muss die Bewertung der Ergebnisse vorherige Faktoren einbeziehen.

Folge dieser Tatsache ist eine Unschärfe bei der Bestimmung der extrahierten Features. Selbst wenn entsprechend der Indikatoren definierte Textmerkmale gefunden werden, ist die tatsächliche Nutzung eines Features nicht eindeutig zu bestimmen. Diesen Umstand berücksichtigt folgend eine Gewichtung der Ergebnisse. Sie kann im Anschluss genutzt werden, um die Ergebnisse der Extraktion einzuordnen. Ebenfalls kann sie genutzt werden, um später Constraints aufzuweichen. Um die Unschärfe der einzelnen Ergebnisse zu berücksichtigen, wird jedem extrahierten Technologie-Feature eine Gewichtung zugeordnet. Diese bezieht relevante Faktoren ein und versucht, gefundene Indikatoren voneinander abzugrenzen.

Anhand eines Beispiels kann der Sachverhalt erläutert werden: Man nehme an, eine

[10] Siehe https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html, zuletzt aufgerufen am 18. Oktober 2016

Quellcode-Datei enthält Indikatoren mehrerer Features A und B, jedoch nicht alle Indikatoren des Features B. Beide Features besitzen jeweils einen Indikator des Typs Import sowie Nutzung eines Datentypen. Für Feature A werden beide Indikatoren identifiziert. Der untersuchte Quelltext weist die ausgewiesenen Schlüsselworte, den entsprechenden Datentypen sowie Import auf. Für Feature B kann lediglich der Import identifiziert werden. Die Nutzung von Feature A ist somit deutlich wahrscheinlicher als die von Feature B.

Betrachtet man den Fall aus einer weiteren Perspektive, wird für Feature B zwar ein Datentyp identifiziert, jedoch kein Import. Die Nutzung des Features ist hier nach wie vor wahrscheinlicher, als bei Existenz ein Importes allein. Je nach Typ unterscheidet sich die Aussagekraft und damit die Konfidenz mit der das Verfahren zur Extraktion die Nutzung eines Features identifizieren kann.

Für Features, deren Indikatoren vollständig identifiziert werden, weist die Extraktion eine hohe Konfidenz auf. Wird ein Indikator sogar mehrfach identifiziert, steigt diese weiter an. In Anbetracht der genannten Probleme der Textanalyse und entsprechenden Schwächen lässt sich somit jedoch nicht zweifelsfrei sicherstellen, dass ein Feature tatsächlich korrekt identifiziert wurde.

Die Gewichtung stellt pro Technologie-Feature und Quelldatei die Konfidenz der Extraktion an dieser Stelle fest. Sie sagt aus, wie sicher sich das Verfahren der Extraktion ist, die Nutzung des betreffenden Features tatsächlich identifiziert zu haben. Anhand unterschiedlicher Faktoren wird die Konfidenz für dieses Feature in der entsprechenden Datei berechnet. Dies bedeutet, Features können in mehreren Dateien gefunden werden. Sie werden entsprechend mehrfach eingesetzt, erhalten jedoch mitunter unterschiedliche Gewichtungen.

Für jedes betrachtete Technologie-Feature werden pro untersuchter Quelldatei nachfolgende Faktoren berücksichtigt.

I: Gewichtung Indikator-Typ Es gilt $I \in \mathbb{R}$ mit $\{0 \leq I \leq 1\}$
Die Gewichtung des Indikator-Typs selbst leitet sich aus dessen Aussagekraft ab. Wie zuvor beschrieben, haben einzelne Indikatoren relativ zueinander einen eigenen Informationsgehalt. Je niedriger dieser Wert, desto geringer ist der Informationsgehalt. Wird etwa ein Import gefunden, ist dieser Wert niedriger, als bei Nutzung eines Datentypen.

C: Gewichtung des Indikators Es gilt $C \in \mathbb{R}$ mit $\{ 0 \leq C \leq 1\}$
Je nach Modellierung eine Features können sich einzelne Indikatoren für dieses als unterschiedlich aussagekräftig herausstellen. Durch diesen Faktor kann bei Formalisierung der Features im Architekturwissen einzelne Indikatoren in ihrer Aussagekraft abgeschwächt werden.

n: Gesamtzahl der identifizierten Indikatoren für dieses Feature Die Menge der Indikatoren, die per Quelldatei identifiziert werden können beschränken sich nicht auf die unterschiedlichen Indikatoren pro Feature. Ein Indikator kann mehrfach identifiziert werden. Etwa die Mehrfache Nutzung von Annotationen oder Datentypen ist möglich. Je häufiger ein Indikator gefunden wird, desto wahrscheinlicher ist die Nutzung eines Features.

v: Anzahl der identifizierten unterschiedlichen Indikatoren Es gilt $v \leq n$
Zudem muss berücksichtigt werden, dass einem Feature nur eine begrenzte Menge unterschiedlicher Indikatoren zugeordnet sind. Werden alle unterschiedlichen Indikatoren je mindestens einmal gefunden, ändert sich die Gewichtung nicht. Wird dagegen ein Indikator nicht gefunden, schwächt dies die Konfidenz ab.

i: Anzahl der für dieses Feature definierten einzigartigen Indikator Für jedes Technologie-Feature wird eine Menge von Indikatoren definiert. Diese Anzahl ist von den im Architekturwissen hinterlegten Indikatoren abhängig.

Betrachtete beispielhafte Berechnungen ergaben folgende Formel:

$$\frac{\sum\left(\frac{2I+C}{3}\right)}{n} * \frac{v}{i}$$

Tabelle 4.10 zeigt mögliche Gewichtung der einzelnen Indikator-Typen gemäß vorheriger Überlegungen. Ihre konkreten Ausprägungen sind von einer Implementation abhängig. Diese richtet sich nach ihrer relativen Aussagekraft (Siehe Seite 54).
Die Berechnung der Gewichtung lässt sich anhand eines Beispiels erläutern: Man nehme an, für ein Technologie-Feature werden drei verschiedene und insgesamt fünf Ausprägungen der für dieses definierten Indikatoren identifiziert.

Tabelle 4.11 zeigt die Ergebnisse der Identifikation für das betreffende Feature, dessen tatsächliche Funktionsweise hier nicht relevant ist. Jeder identifizierte Indikator erhält zunächst seine eigene Gewichtung. Diese setzt sich aus der Gewichtung seines

Typ	I
Import	0.30
Typnutzung	0.75
Methodenaufruf	0.95
Vererbung	0.50
Implementation	0.50
Annotation	0.50
Schlüsselwort	0.25
Regulärer Ausdruck	0.25

Tabelle 4.10: Mögliche Gewichtung der Indikator-Typen

Indikator	**Anzahl**	C	**Gewichtung**(I)
Import	1	1	$\frac{2*0.3+1}{3} = \frac{8}{15}$
Typnutzung	2	1	$\frac{2*0.75+1}{3} = \frac{5}{6}$
Annotation	2	0.5	$\frac{2*0.5+1}{3} = \frac{3}{4}$

Tabelle 4.11: Beispiel Berechnung Gewichtung bei vollständig gefundenen Indikatoren

jeweiligen Typs sowie dem individuellen Faktor für dieses Feature zusammen (C). Aus den identifizierten Instanzen der Indikatoren ergeben sich $n = 5$ sowie $v = 3$, da insgesamt fünf und davon drei einzigartige Indikatoren identifiziert werden konnten. Jeweils zwei Indikatoren wurden mehrfach identifiziert. Die folgende Gleichung summiert diese entsprechend auf.

$$\frac{\frac{8}{15} + \frac{5}{6} + 2 * \frac{3}{4}}{5} + 2 * \frac{3}{3} = \frac{37}{50} * \frac{3}{3} = 0.74$$

Für das beschriebene Beispiel ergibt sich eine Konfidenz von 0.74. Es ist somit relativ wahrscheinlich, dass das identifizierte Feature auch tatsächlich genutzt wird. Ein weiteres Beispiel zeigt einen verwandten Fall, dieser verwendet ähnliche Indikatoren, bei der Extraktion ist allerdings nur einer dieser Indikatoren identifiziert worden. Tabelle 4.12 zeigt die beispielhafte Berechnung

Indikator	**Anzahl**	C	**Gewichtung**(I)
Import	1	1	$\frac{2*0.3+1}{3} = \frac{8}{15}$
Typnutzung	0	1	$\frac{2*0.75+1}{3} = \frac{5}{6}$

Tabelle 4.12: Beispiel Berechnung Gewichtung bei nicht gefundenen Indikatoren

Trotz gleicher Gewichtungen für die Indikatoren ergibt sich eine unterschiedliche Summe. Diese zeigt im Beispiel auf, wie unwahrscheinlich die Nutzung eines Features lediglich bei Existenz eines Imports erscheint.

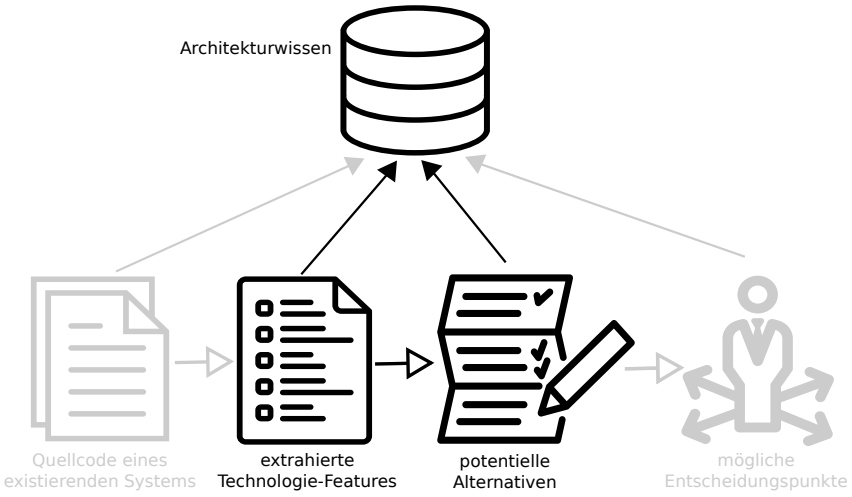
$$\frac{8}{15} * \frac{1}{3} = 0.17\overline{8}$$

Die Unschärfe dieser Ergebnisse macht ein besonderes Augenmerk bei der Betrachtung notwendig: Wie weitere Ansätze zur Unterstützung des Softwarearchitektur-

Prozesses (Etwa der *DecisionBuddy*, siehe [Gerdes u. a., 2015]) ist auch die Extraktion eine Unterstützung zur Eingrenzung des Lösungsraumes. Um die Qualität dieser Eingrenzung besser abschätzen zu können, dient die berechnete Konfidenz als Grundlage für Betrachtungen und Abschätzungen im Prozess. Sie sollen lediglich die angesprochene Unterstützung zur Eingrenzung des Lösungsraumes erhalten.

Um anhand dieser Gewichtung eine Aufweichung von Constraints zu erreichen, kann auf dessen Basis ein Schwellwert für die weitere Betrachtung genutzt werden. Dieser berücksichtigt im Weiteren lediglich extrahierte Features, deren Gewichtung über einem definierten Wertebereich liegt. Schon anhand eines einfachen Beispiels ergibt sich die Sinnhaftigkeit einer entsprechenden Aufweichung: Wird eine Technologie genutzt, weist diese meist mehrere Programmpakete auf. Wird ein Technologie-Feature aus diesem Paket genutzt, identifiziert das Verfahren zunächst Indikatoren für mehrere Features. Diese stammen aus einem gemeinsamen Programmpaket und nutzen denselben Import. Prinzipiell ist somit auf die Nutzung dieser Features zu schließen. Diese werden allerdings, aufgrund ihrer Indikator-Kategorie und dem Fehlen weiterer Indikatoren für das entsprechende Feature, mit einer niedrigen Gewichtung bestimmt. Faktisch sind sie nicht relevant, können somit vernachlässigt werden. Hebt man den Schwellwert, unterhalb dem extrahierte Features ignoriert werden können, lassen sich entsprechend falsche Extraktionsergebnisse ausschließen.

4.3 Bestimmen von Alternativen zu eingesetzten Technologien



Um im Prozess der Softwarearchitektur sinnvolle Alternativen berücksichtigen zu können, müssen anhand extrahierter Technologie-Features sinnvolle alternative Technologien bestimmt werden. Eine Betrachtung von alternativen Technologie-Features alleine ist zwar ebenfalls denkbar, erscheint im Bezug auf den Anwendungsfall allerdings weniger relevant und zu feingranular.

Wie das Metamodell in Abschnitt 4.1 (Seite 43) bereits festlegt, können alternative Technologie-Features anhand der gemeinsamen Implementation eines abstrakten Features identifiziert werden. Diese Modellierung reicht prinzipiell bereits aus um alternative Technologie-Features zu bestimmen. Um den Lösungsraum sinnvoll zu reduzieren, ist dies allerdings wenig zielführend: Für jedes extrahierte Technologie-Feature existiert unter Umständen eine Menge von Alternativen. Es entstehen Redundanzen; Features können unter Umständen mehrfach identifiziert werden. Einzig zielführend zur Reduzierung des Suchraums ist somit eine Betrachtung auf Technologie-Ebene.

Sollen nicht nur eingesetzte Technologien ausgetauscht werden, sondern auch neue Anforderungen umgesetzt werden, müssen neue Features ergänzt werden. Durch ein manuelles Anreichern der Extraktionsergebnisse können diese geforderten Features bei der Bestimmung von Alternativen ebenfalls berücksichtigt werden.

Gruppieren der extrahierten Technologie-Features

Die Extraktion betrachtet Technologie-Features unabhängig ihrer Abhängigkeiten. Um alternative Technologien zu identifizieren zu können, müssen zunächst tatsächlich genutzte Technologien aus der Menge extrahierter Features bestimmt werden.

Durch ihre Modellierung geben einzelne identifizierte Technologie-Features Aufschluss über ihre Abhängigkeiten und somit der ihnen zugeordneten Technologie. Die Betrachtung kann auf Komponenten-Ebene erfolgen. Diese Tatsache ist für den Prozess relevant: Werden bestimmte Technologien mit kritischen Features nur in

wenigen Programmpaketen eingesetzt, muss die weitere Betrachtung nur in diesem Rahmen erfolgen. Somit kann sich der Aufwand für die Analyse, Refactoring und weitere Anpassungen verringern. Durch die bewusste Beschränkung auf eine Komponente kann sich das Risiko, welches beim Austauschen einer Technologie entsteht, eingrenzen.

Aus eine Menge extrahierter Technologie-Features wird somit eine Menge von genutzten Technologien. Jeder dieser Technologien ist eine Menge genutzter abstrakter Features zugeordnet. Die Implementation in der entsprechenden Technologie ist an dieser Stelle nicht weiter relevant: Um alternativen zu den eingesetzten Features zu bestimmen muss diese nicht betrachtet werden.

Das der Einsatz bestimmter Technologie-Features unter Umständen mehrfach pro Komponente identifiziert wird, kann vernachlässigt werden. Durch das mehrfache Nutzen eines Features ändert sich die Grundproblematik nicht: Zu dem entsprechenden Feature besteht eine Abhängigkeit, für die es eine Alternative zu bestimmen gilt.

Bestimmen von Alternativen

Damit aus den angestrebten Datenstrukturen heraus überhaupt sinnvolle Alternativen bestimmt werden können, müssen zunächst einige Eigenheiten betrachtet werden. Nicht jede Technologie eignet sich als potentielle Alternative. Ihre Abhängigkeiten spielen eine zentrale Rolle.

Betrachtet man die Abhängigkeiten zwischen einzelnen Technologien, ergibt sich wie bereits erläutert ein Baum (Abbildung 4.8). Anhand diesen Beispiels lassen sich relevante Eigenschaften der Struktur betrachteten. Dies ergibt sich parallel zur Betrachtung der Abstraktionsebenen: Erst nach Bestimmung tatsächlich genutzter Technologie-Features ist eine fundierte Aussage über potentielle Alternativen möglich.

Obwohl zwei Technologien eine ähnliche Menge von Features implementieren, sind sie keineswegs ohne Weiteres als Alternativen zu betrachteten. Ausschlaggebend ist dabei ihre relative Lage zueinander im Pfad ihrer Abhängigkeiten.

Im Beispiel zeigt sich dies konkret an den Technologien JPA und *Hibernate*. Die Menge der von *Hibernate* implementierten Features beinhaltet die von JPA bereitgestellten Features. *Hibernate* implementiert die JPA-Spezifikation. Entsprechend wäre auch JPA als Alternative zu *Hibernate* zu betrachten, da beide Technologien eine gemeinsame Teilmenge an Features besitzen. Da jedoch beide Technologien auf demselben Pfad des Baumes liegen, ist ihre Beziehung als Alternative auszuschließen.

Eine konkrete Technologie durch ihre Spezifikation auszutauschen ist nicht zielführend. Selbiges gilt im Beispiel für *TopLink*. Hieraus lässt sich ableiten, dass Technologien, die im Pfad der Abhängigkeiten direkt über der betrachteten Technologie liegen nicht als Alternativen genutzt werden können. Sie sind somit bei der Bestimmung zu ignorieren.

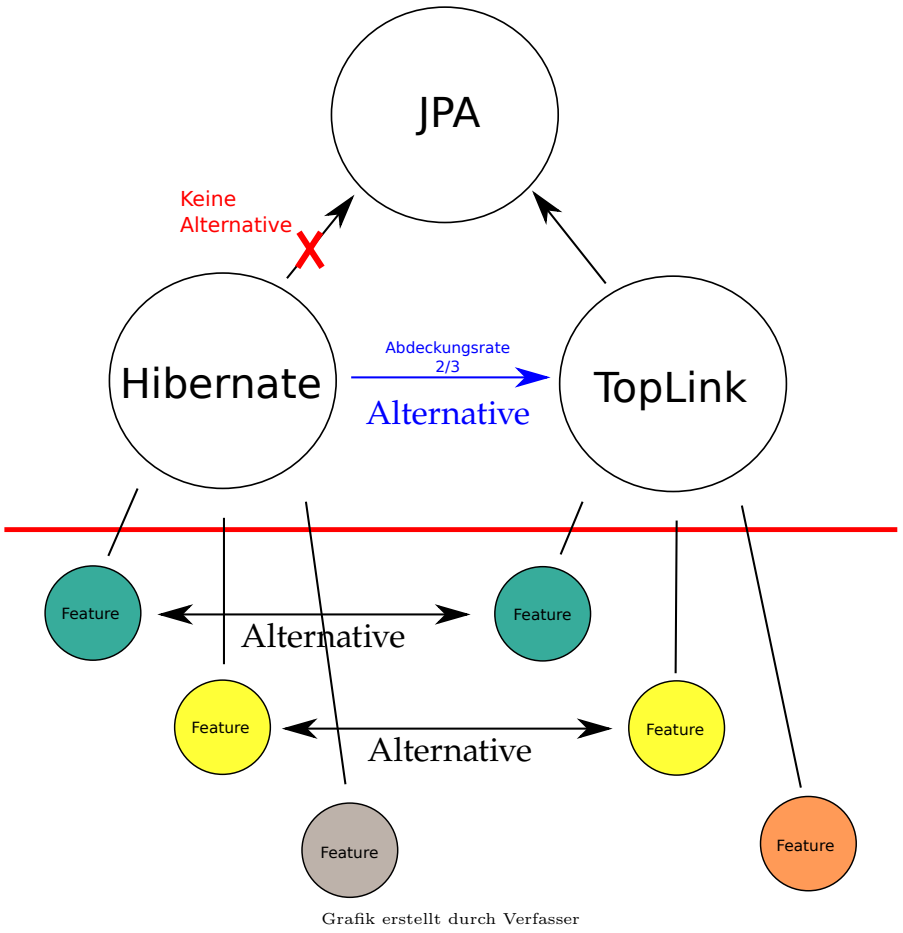


Abbildung 4.8: Beispiel zur Bestimmung von Alternativen

Aus dieser Abhängigkeit lässt sich jedoch eine weitere Eigenschaft ableiten: Entwurfsentscheidungen resultieren ebenfalls in einem Baum von Alternativen, dieser ergibt sich analog zum ausgewiesenen Beispiel. Konkrete Entwurfsentscheidungen zwischen Alternativen fallen auf ein Blatt des Baumes. In diesem Fall etwa die Technologie *Hibernate* mit ihren Abhängigkeiten JPA und Java. Betrachtet man diesen Pfad als untrennbare Eigenschaft einer Entwurfsentscheidung, lassen sich alle durch diesen gewählten Technologien auf dem kompletten Pfad betrachten. Anstatt jede Technologie für sich zu betrachten muss ebenfalls berücksichtigt werden, wie diese Auswahl zu Stande gekommen ist.

Man kann dies grob als Vererbung von Technologie-Features bezeichnen. Das Blatt des Pfades ist eine Ausprägung aller zuvor gewählten Abhängigkeiten. Abbildung 4.9 zeigt dies an einem Beispiel. Jede Technologie besitzt ihre eigenen Features. Technologien erben jedoch für die Betrachtung die Features ihrer Abhängigkeiten. Um ein Blatt zu berücksichtigen sind somit dessen Abhängigkeiten zu genutzten Features ebenfalls zu betrachten. Eine Alternative stellt ebenfalls ein Blatt dar, jedoch mit einem unterschiedlichen Verlauf des Pfades. Es stehen sich im Beispiel potentiell die Pfade *Hibernate* - JPA - Java und *TopLink* - JPA - Java gegenüber.

Beide Knoten unterhalb der *Java Persistence API*-Spezifikation sind somit poten-

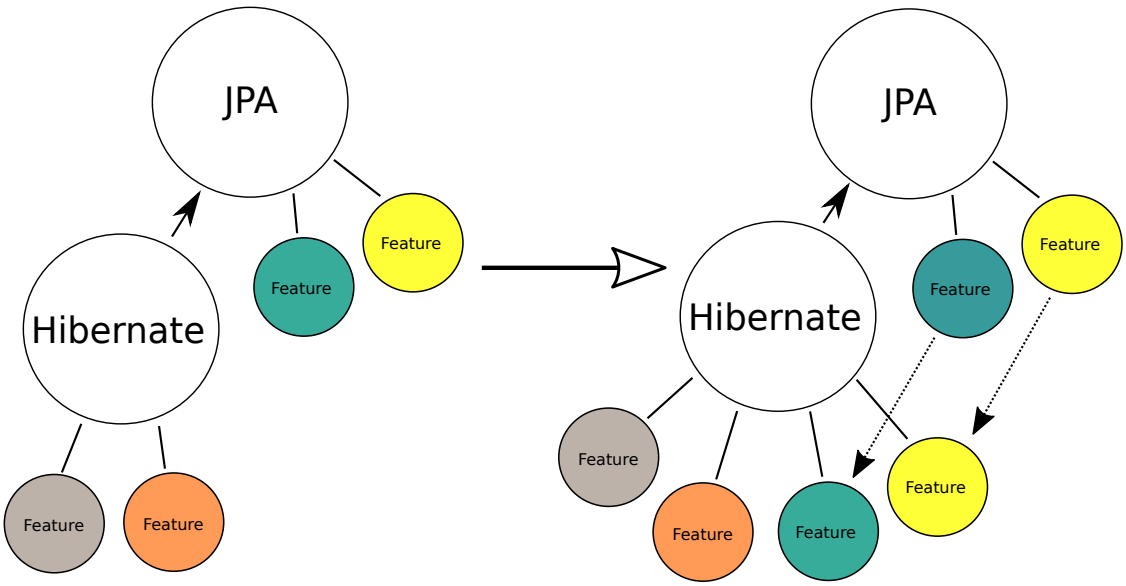


Abbildung 4.9: „Vererbung“ von Features anhand ihrer Abhängigkeiten

tielle Alternativen zueinander. Hier ist zu berücksichtigen, welche Features beide Technologien implementieren. Im dargestellten Beispiel besitzen die Technologien eine gemeinsame Menge an Features (gleichfarbige Blattknoten). Diese können sowohl von der konkreten Technologie selbst stammen als auch von dessen Abhängigkeiten. Relevant ist die Betrachtung der Technologie als Entwurfsentscheidung.

Abhängigkeiten der konkreten Technologie-Features untereinander sind an dieser Stelle nicht weiter relevant. Lediglich die Nutzung bestimmter Features ist zu betrachten. Zudem ist davon auszugehen, dass bei Abhängigkeiten von Features untereinander alle Abhängigkeiten ebenfalls erkannt werden. Dies ist auf die zugrundeliegende Syntax zurückzuführen: Sind Technologie-Features von anderen abhängig, sind ihre charakteristischen Merkmale spezifischere Ausprägungen der charakteristischen Eigenschaften ihrer Abhängigkeiten.

Es wird konkret keine Alternative zu einer Technologie gesucht, sondern zu der Entwurfsentscheidung, deren Blattknoten die konkrete Technologie gewesen ist. Die Extraktion bestimmt nicht nur die Features der konkreten Technologie, sondern auch von dessen Abhängigkeiten. Berücksichtigt werden muss damit nicht die formelle Entwurfsentscheidung, sondern die tatsächlich implementierte. Diese lässt sich aus dem existierenden System durch Nutzung von Technologie-Features ableiten.

Abdeckungsrate

Technologien stellen sich gerade bei Betrachtung eines existierenden Systems nicht zwangsläufig als eindeutige Alternativen zueinander dar. Dies bedeutet, dass zwei Technologien nur in seltenen Fällen eine vollständig deckungsgleiche Menge an Features implementieren. Wäre dies der Fall, bieten die Technologien den gleichen Funktionsumfang, unterscheiden sich jedoch in den ASTAs der einzelnen Features.

Geht man davon aus, dass Anbieter einzelner Technologien auf die Implementation herstellerspezifischer Alleinstellungsmerkmale setzen, ist eine eindeutige Abdeckung unwahrscheinlich. In Anbetracht existierender Systeme und der Nutzung einer Teilmenge dieser Features ist eine teilweise Abdeckung jedoch nicht unwahrscheinlich: Etwa im Fall der JPA-Implementationen können sich implementierte Features durchaus überdecken. Werden im existierenden System lediglich Features der JPA-Spezifikation genutzt, sollten prinzipiell alle Technologien, welche diese implementieren, auch deren Gesamtheit an Features unterstützen. Dies ist etwa bei den Implementationen *Hibernate* und *TopLink* der Fall. Beide Technologien implementieren herstellerspezifische Optimierungen. Werden diese jedoch nicht genutzt, muss für sie somit keine Alternative bestimmt werden. Alle genutzten Features lassen sich in diesem Fall entsprechend ihrer Funktionalität ersetzen.

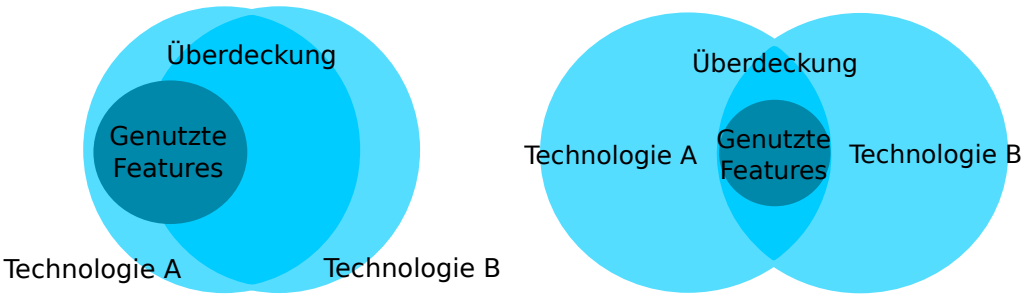
Um diesen Umstand zu berücksichtigen muss bestimmt werden, wie gut sich eine Technologie als Alternative eignet. Zwecks Bereitstellung eines einheitlichen und vergleichbaren Maßes zu Bestimmung der Qualität von Alternativen wird hier die sogenannte *Abdeckungsrate* betrachtet. Dieser Wert gibt für je zwei Technologien an, wie gut sie sich als Alternative füreinander eignen. Entscheidend ist dabei die Menge der von der potentiellen Alternative ebenfalls implementierter Features.

Ein Beispiel erläutert diesen Faktor: Man nehme an, ein Anwendungssystem nutzt Technologie A und setzt neun konkrete Technologie-Features dieser Technologie ein. Technologien B, C und D implementieren je eine ähnliche Menge an Features. Technologie B setzt sieben Features ebenfalls um, die Abdeckungsrate beträgt in diesem Fall 7/9. Technologie C hingegen setzt lediglich zwei der Features um und resultiert in einer Abdeckungsrate von 2/9, ist somit weniger als Alternative geeignet. Technologie D ist am besten als Alternative geeignet: Sie implementiert alle aktuell genutzten Features ebenfalls und wartet entsprechend mit einer Abdeckungsrate von 1 auf. Die Abdeckungsrate gibt somit lediglich den Grad an, zu dem potentielle Alternativen ähnliche Features implementieren. Sind die Mengen implementierter Features zweier Technologien teilerfremd, liegt die Abdeckungsrate bei null. Die Technologien implementieren keine verwandten Features und sind somit gänzlich als Alternativen zueinander ungeeignet.

Parallel zum Schwellwert der Konfidenz bei der Bestimmung von Technologie-Features kann auch die Abdeckungsrate zur Aufweichung von Constraints genutzt werden. Finden sich keine eindeutigen Alternativen, müssen Features selbst implementiert oder auf diese verzichtet werden. Die Suche nach Alternativen soll effektiv den Suchraum reduzieren. Zu diesem Zweck erfolgt die Suche zunächst nach Alternativen mit einer hohen Abdeckungsrate. Die aktuell genutzten Features bilden ein Constraint für die Suche: Ihre Funktionalität muss nach wie vor gewährleistet werden. Existieren keine geeigneten Technologien, muss der Schwellwert gesenkt werden, Technologien mit niedrigerer Abdeckungsrate müssen betrachtet werden. Fehlende Features müssen anderweitig umgesetzt werden.

Man kann die Bestimmung der Abdeckungsrate in zwei Kategorien unterteilen. Sie lässt sich statisch ebenso wie dynamisch bestimmen. Die statische Bestimmung beschreibt den Überdeckungsgrad aller Features zweier Technologien. Sie berücksich-

tigt alle Features einer Technologie und ist nicht bivalent. Bei der dynamischen Bestimmung werden lediglich die tatsächlich genutzten Features eines Anwendungssystems für die Überdeckung berücksichtigt. Die Betrachtung erfolgt nur aus Perspektive der Technologie, für die Alternativen zu bestimmen sind.



Grafik erstellt durch Verfasser

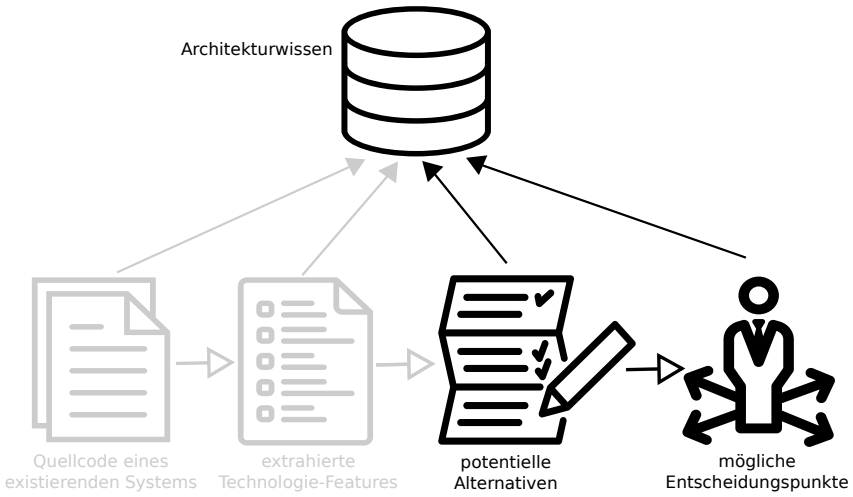
Abbildung 4.10: Betrachtung dynamischer (links) und statischer Bestimmung der Abdeckungsrate

Das einführende Beispiel zeigt jedoch auf, warum diese Bestimmung nicht zielführend ist: herstellerspezifische Features sorgen dafür, dass die schätzungsweise Abdeckung nicht eindeutig sein wird. Somit müssten Alternativen anhand der statischen Werte betrachtet werden. Dies grenzt den Lösungsraum bei der Suche nach Alternativen jedoch nur unzureichend genau ein: Technologien mit einer niedrigeren statischen Abdeckungsrate können prinzipiell die bessere Alternativen darstellen.

Abbildung 4.10 (links) zeigt die Nutzung der Features zweier Technologien. Die Technologien besitzen statisch eine hohe Abdeckungsrate, die gemeinsame Teilmenge implementierter Features. Bei der Betrachtung der Schnittmenge tatsächlich genutzter Features lässt sich jedoch feststellen, das die im konkreten Fall benötigten Features nicht implementiert werden. Abbildung 4.10 (rechts) zeigt dagegen die Betrachtung des dynamischen Falls. Ihnen fehlen in diesem Kontext zwar bestimmte Features, sie implementieren dennoch die aktuell genutzten und auszutauschenden Funktionalitäten. Die dynamische Bestimmung von Alternativen und die entsprechende Berücksichtigung der Abdeckungsrate bezieht diesen Umstand mit ein.

Das Ergebnis der Analyse und Bestimmung von Alternativen stellt für jede eingesetzte Technologie eine Reihe von Technologien als Alternativen bereit. Dabei werden tatsächlich eingesetzte Features berücksichtigt. Für jede Alternative wird die jeweilige Abdeckungsrate berechnet. Anhand dieser kann die Qualität der potentiellen Lösung abgeschätzt werden.

4.4 Bestimmen von Entscheidungspunkten



Sind potentielle Alternativen zu eingesetzten Technologien bestimmt, muss ein weiterer Faktor berücksichtigt werden. Das Austauschen oder Hinzufügen einer neuen Technologie birgt unter Umständen neue Abhängigkeiten oder muss existierende berücksichtigen. Da diese Teil des Architektur-Prozesses und damit der Entscheidungsfindung sind, kommt ihnen eine gewisse Relevanz zu. Je mehr Abhängigkeiten hinzugefügt oder verändert werden, desto mehr Änderungen bedeutet dies für den Entwicklungsprozess und die Anpassung. Dies erhöht das Risiko der Umstellung und stellt potentiell einen Zeit- und Kostenfaktor dar.

Betrachtet man Technologieentscheidungen als Entwurfsentscheidungen, sind Alternativen als Kinder eines Knoten im Baum der Entscheidungen zu betrachten [Zimmermann u. a., 2009], [Soliman u. a., 2015]. Konkret bedeutet dies, dass zwei alternative Technologien (Entscheidungen) eine gemeinsame Abhängigkeit besitzen. Liegen beide Entscheidungen in unterschiedlichen Bäumen des Architekturwissens, ist ein gemeinsamer Wurzelknoten zu konstruieren.

Diese gemeinsamen Wurzelknoten zweier Technologien werden in Folge als Entscheidungspunkte bezeichnet. Sie geben an, wo eine Entscheidung getroffen beziehungsweise revidiert werden muss, um eine alternative Technologie einzubinden. Eine wichtige Rolle spielt dabei, wie weit die betreffenden Technologien sowie der Entscheidungspunkt voneinander entfernt sind. Entscheidungen, die unterhalb des Entscheidungspunktes liegen, müssen berücksichtigt werden. Je mehr Abhängigkeiten unterhalb des Entscheidungspunktes liegen, desto höher ist der Aufwand.

Abbildung 4.11 zeigt einen simplen Entscheidungspunkt. Die Technologien A und B besitzen die gleiche Abhängigkeit. Diese bildet somit einen Entscheidungspunkt. Im ursprünglichen Entscheidungsprozess fiel die Wahl etwa auf Technologie A. Um nun Technologie B einzusetzen, kann diese aufgrund der gemeinsamen Abhängigkeit entsprechend ohne große Konsequenzen ausgetauscht werden.

Hängen von dieser jedoch weitere Entscheidungen und Technologien ab, müssen diese ebenfalls überprüft und eventuell revidiert werden. Die Betrachtung von Entscheidungspunkten hat keinen direkten Einfluss auf den Prozess der Entscheidungs-

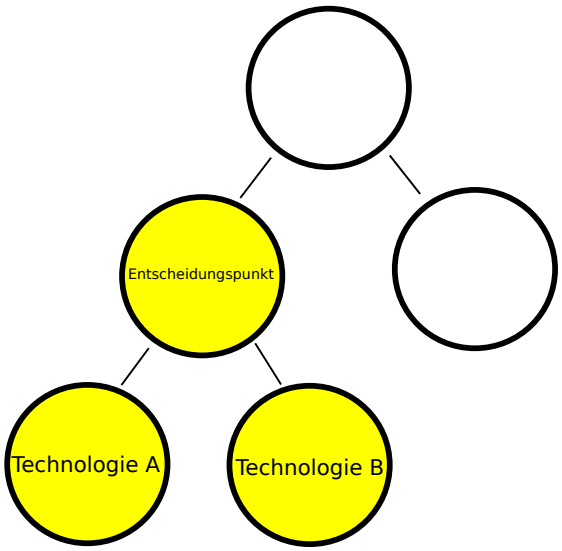


Abbildung 4.11: Beispiel simpler Entscheidungspunkt

findung. Sie macht lediglich auf relevante Abhängigkeiten im Baum der Entwurfsentscheidungen aufmerksam, die für die Anpassungen der Technologien relevant sind. Dies grenzt den Suchraum gerade bei simplen Entscheidungspunkten drastisch ein.

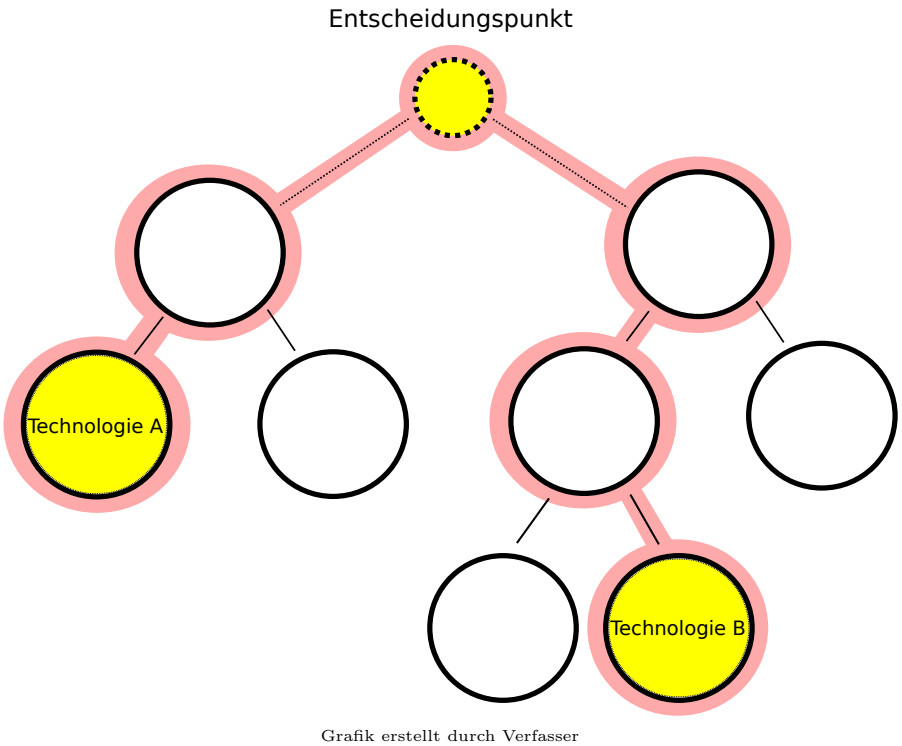


Abbildung 4.12: Beispiel komplexer Entscheidungspunkte

Das Hinzufügen eines konstruierten Wurzelknotens zeigt Abbildung 4.12. Die Technologien A und B besitzen keine gemeinsamen Abhängigkeiten. Die Wurzeln der Abhängigkeiten beider Technologien werden mithilfe eines gemeinsamen Knotens verbunden. Dieser dient als Entscheidungspunkt. Konkret kann dieser Fall auftreten, wenn Technologien etwa auf unterschiedlichen Plattformen betrachtet werden.

Wird die Programmiersprache Java innerhalb einer Linux-Umgebung eingesetzt, findet sich beispielsweise eine Alternative im Rahmen des .NET-Frameworks. Dieses hat als Wurzelknoten jedoch lediglich eine Abhängigkeit zur Plattform Windows. Beide Technologien lassen sich nicht anhand gemeinsamer Abhängigkeiten austauschen, es muss ein Plattformwechsel erfolgen.

Um Entscheidungspunkte zu bestimmen, muss anhand der Abhängigkeiten beider Technologien (`TechnologieA` sowie `TechnologieB`) die gemeinsame Wurzel identifiziert werden. Pseudocode-Beispiel 4.15 zeigt das Vorgehen exemplarisch. Die Iteratoren `t1` und `t2` (Zeile 7 und 8) betrachten die Kette der Abhängigkeiten beider Technologien und prüfen für jedes Paar, ob ein gemeinsamer Knoten festzustellen ist (Zeile 16). Ausgangspunkt sind hier die unmittelbaren Abhängigkeiten beider Technologien. Das Ergebnis (`decisionPoint`, Zeile 17) ist die als Entscheidungspunkt zu betrachtende Technologie als gemeinsamer Wurzelknoten.

```
1  if TechnologieA == TechnologieB
2    return null
3
4  pathLeft := 1
5  pathRight := 1
6
7  t1 := TechnologieA.dependsOn
8  t2 := TechnologieB.dependsOn
9
10 while t1.dependsOn != null
11   pathLeft = 0
12
13   while t2.dependsOn != null
14     pathLeft += 1
15
16     if t1 == t2
17       decisionPoint := t1
18       break
19
20       t2 = t2.dependsOn
21   end while
22
23   pathRight += 1
24   t1 = t1.dependsOn
25 end while
26
27 pathLength := pathLeft = pathRight
```

Quellcode-Beispiel 4.15: Pseudocode zur Bestimmung von Entscheidungspunkten

Anhand der Kette an Abhängigkeiten wird der kleinste gemeinsame Wurzelknoten bestimmt. Eine Suche oberhalb dieses Knotens ist nicht notwendig. Beide Technologien besitzen oberhalb dieses Knotens die gleiche Kette an Abhängigkeiten. Wird für beide Technologien kein gemeinsamer Knoten ermittelt, wird die Konstruktion eines künstlichen Knotens angenommen.

Dieser verbindet immer die letzten Glieder in der Kette der Abhängigkeiten. Zudem muss die Entfernung zwischen den beiden betrachteten Technologien berücksichtigt werden. Je größer diese ist, desto mehr Abhängigkeiten liegen zwischen den beiden Technologien. Diese können, wie bereits beschrieben, die Komplexität und den Suchraum vergrößern. Die Länge des Pfades (`pathLength`) berücksichtigt zu diesem Zweck den gesamten Pfad, der auf der Suche nach dem Entscheidungspunkt passiert wird (Zeile 27). Betrachtet werden die Abhängigkeiten der beiden Technologien auf dem jeweiligen Pfad (`pathLeft` sowie `pathRight`).

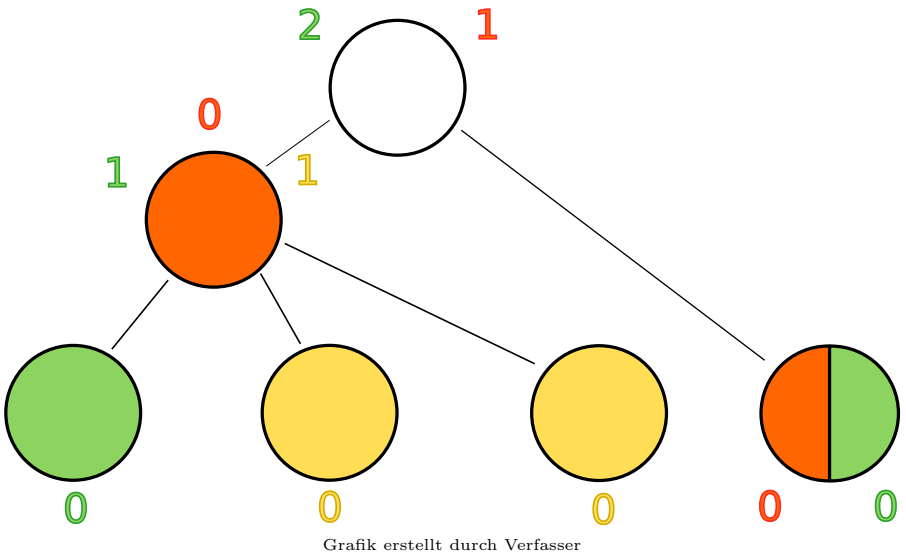


Abbildung 4.13: Beispiel Berechnung der Pfadlänge

Abbildung 4.13 erläutert die beschriebene Berechnung der Pfad-Länge. Die Farbcodierung der einzelnen Knoten gibt Alternativen an. Alle Technologien haben den weißen beziehungsweise orangen Knoten als Entscheidungspunkt. Die grünen Technologien ergeben eine Pfadlänge von 2. Die roten Technologien dagegen ein Länge von 1. Selbiges gilt für die gelben Technologien, obwohl sie im Baum weiter unten positioniert sind.

Ähnlich zur Konfidenz der Extraktion sowie der Abdeckungsrate kann die Länge des Pfades zur Aufweichung von Constraints genutzt werden. Grundsätzlich ist zunächst eine geringe Pfadlänge anzustreben. Diese reduziert die für Änderungen auszutauschenden Abhängigkeiten auf ein Minimum und schränkt den Suchraum für die Betrachtung signifikant ein. Sind in diesem beschränkten Raum keine geeigneten Alternativen zu finden, müssen auch Alternativen mit längeren Abhängigkeitspfaden berücksichtigt werden. Dies bedeutet letztendlich, dass mehr Abhängigkeiten berücksichtigt und potentiell mehr Entwurfsentscheidungen revidiert werden müssen. Kürzere Pfade sind zu bevorzugen. Im Optimalfall sind dies direkte Alternativen (Abbildung 4.11).

Aus den vorgeschlagenen Entscheidungspunkten muss schlussendlich die für den aktuellen Anwendungsfall passende Alternative gewählt werden. Die Menge an Entscheidungspunkten ist somit eine Menge potentieller Lösungen.

Kapitel 5

Prototypische Implementation

Um das vorangegangene Konzept zu validieren, setzt eine prototypische Implementation die Grundzüge des Ansatzes um[1]. Teil dieser Implementation sind eine exemplarische Auswahl von Architekturwissen sowie eine automatisierte Extraktion von Technologie-Features. Zudem bietet der Prototyp weitere Unterstützung im Prozess der Softwarearchitektur, etwa durch ein Vorschlagen potentieller Alternativen und möglicher Entwurfsentscheidungen. Diese basieren auf dem exemplarischen Architekturwissen sowie den extrahierten Ergebnissen. Von der rein technischen Extraktion wird somit der Transfer in die abstrahierte Ebene der Softwarearchitektur geschaffen.

Im Folgenden sollen die umgesetzten Bestandteile des Konzepts, implementierte Architektur und Funktionsweise, sowie Möglichkeiten zur weiteren Verarbeitung und Einbindung der Ergebnisse diskutiert werden. Zudem werden Vorteile und Schwächen beleuchtet.

5.1 Umsetzung des Konzepts

Das Tool *FeatureExtractor* setzt die wesentlichen Bestandteile des in Abschnitt 4 dargestellten Konzeptes um. Der beschriebene Prototyp stellt ein Tool zur Unterstützung des Softwarearchitektur-Prozesses dar und grenzt im Mittel den Lösungsraum für die sinnvolle Entscheidungsfindung ein. Zu den implementierten Funktionalitäten zählen dabei unter anderen:

- Ein Datenmodell zur Umsetzung des in 4.1 beschriebenen Metamodells inklusive Verwaltung zur Konsistenzsicherung der Datenbasis, sowie deren Visualisierung
- Eingabe eines aktuellen Schnappschusses eines existierenden Softwaresystems (Quellcode-Dateien) zwecks Extraktion ohne manuellen Selektionsaufwand
- Eine Menge von Regeln zur automatisierten Erkennung der in 4.2 beschriebenen Kategorien von Technologie-Features
- Die Aggregation extrahierter Features auf Ebene der Programmpakete zwecks Übersichtlichkeit und Einschätzung von Umfang und Auswirkung potentieller Änderungen

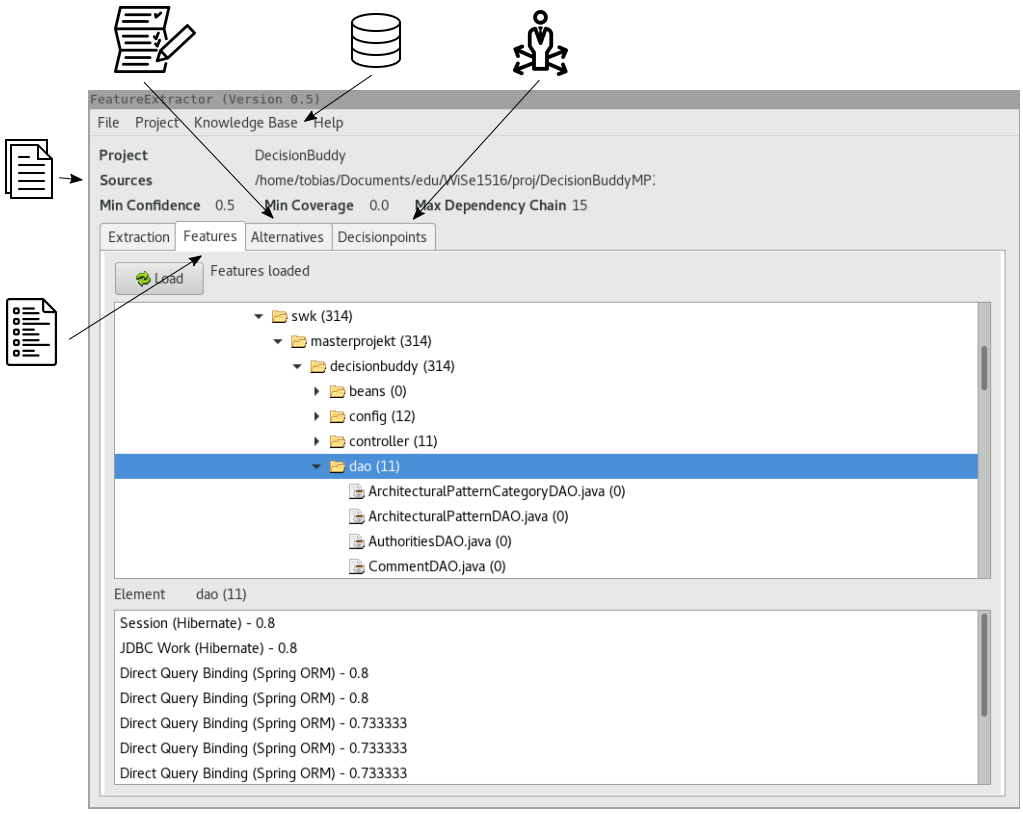
[1] Das Tool ist in ausführbarer Form auf dem beigelegten Medium und auf https://github.com/1fechner/FeatureExtractor zu finden

- Eine Implementation des Algorithmus zur automatisierten Erkennung von Technologie-Features
- Die Bestimmung von potentiellen Alternativen basierend auf den extrahierten Features
- Die Darstellung einer Matrix zur Visualisierung der Qualität potentieller Alternativen mit Hinweisen auf Vor- und Nachteile (ASTAs) relevanter Technologie-Features
- Das Vorschlagen potentieller Entscheidungspunkte als Ergebnis und Grundlage für den weiteren Softwarearchitektur-Prozess

5.2 Architektur

Die Softwarearchitektur des Prototypen *FeatureExtractor* selbst orientiert sich an den Grundzügen des Konzeptes. Abbildung 5.1 zeigt die Zuordnung der einzelnen Elemente des Konzeptes. Sie können mehr oder weniger unabhängig voneinander ausgeführt und betrachtet werden.

Abbildung A.4 zeigt die Architektur des Prototypen. Das aus dem finalen Metamodell abgeleitete Datenmodell findet sich in Abbildung A.5[2]



Grafik erstellt durch Verfasser

Abbildung 5.1: *FeatureExtractor*: Umsetzung des Grundkonzeptes und seiner Elemente

[2] Anhang, Seiten V sowie VI

Auf Grundlage einer gemeinsamen Datenbasis (Architekturwissen) ermöglicht der Prototyp die Unterstützung der drei wesentlichen Arbeitsschritte:

1. Die Extraktion von Technologie-Features
2. Die Bestimmung von Alternativen
3. Das Vorschlagen von Entscheidungspunkten

Einzelne Teile des Konzeptes sind prinzipiell zwar unabhängig voneinander auszuführen, hängen jedoch von den Ergebnissen des vorherigen Arbeitsschrittes ab. Damit Alternativen bestimmt werden können, müssen zunächst Features extrahiert worden sein. Ähnliches gilt für Entscheidungspunkte und Alternativen. Diese Trennung ist insofern sinnvoll, als das innerhalb der einzelnen Schritte Parameter zur Aufweichung von Constraints modifiziert werden können. Diese beeinflussen sowohl die Ergebnisse des aktuellen Arbeitsschrittes als auch die Ergebnisse der weiteren Verarbeitung. Konkret ist dies unter Nutzung der im Konzept vorgestellten Parameter implementiert. Essentiell ist hier die Festlegung einer minimalen Konfidenz bei der Extraktion, einer minimalen Abdeckungsrate für Alternativen sowie einer maximalen Länge des Abhängigkeitspfades bei der Bestimmung von Entscheidungspunkten.

Das dem Prototypen zugrundeliegende Datenmodell ist eine direkte Abbildung des in 4.1 definierten Metamodells. In einer Datenbank wird das genutzte Architekturwissen abgelegt und verwaltet. Diese Art der Speicherung orientiert sich an verwandten Ansätzen zur Verwaltung von Architekturwissen [Gerdes u. a., 2015] [Babar u. Gorton, 2007] [Liang u. Avgeriou, 2009]. Alle Arbeitsschritte des Prototypen nutzen dieselben Datenstrukturen, um eine gemeinsame Wissensbasis zu implementieren.

Die genutzte Wissensbasis orientiert sich an dem in Abschnitt 2.2 beschriebenen Anwendungsfall. Die aus diesem extrahierte Menge an Architekturwissen wurde formalisiert und strukturiert abgelegt. Sie dient der Analyse des *DecisionBuddy* und der Extraktion der von diesem genutzten Features. Weitere Features sind zwar ebenfalls hinterlegt, einen Anspruch auf Vollständigkeit erhebt die Wissensbasis dennoch nicht. Dies wird unterstützt durch die Annahmen 1 sowie 2.

Um die einzelnen Arbeitsschritte strukturell voneinander deutlich abzugrenzen, ist jeder einzelne Schritt unabhängig voneinander implementiert.

Die Extraktion von Technologie-Features findet im `ExtractionService` statt. Dieser wiederum zergliedert sich in drei Komponenten, welche sukzessive das zu analysierende System betrachten. Eine Art Vorverarbeitung analysiert zunächst die Ordner- und damit Paketstruktur des Systems. Optional können Verzeichnisse ohne Informationsgehalt oder bestimmte Dateitypen ignoriert werden. Zu diesen zählen etwa Verzeichnisse mit Abbildungen, Log-Dateien oder im Projektverzeichnis abgelegte Konfigurationsdateien von Entwicklungsumgebungen. Alle relevanten Elemente werden indiziert, ihr Datei-Typ wird bestimmt. Die Implementation des vorgestellten Extraktionsalgorithmus analysiert anschließend jede einzelne relevante Datei unabhängig von ihrem Ablageort. In diesem Schritt können Optimierungen, etwa eine

Vorformatierung der Quellcodedatei eingebunden werden, um die Genauigkeit der Extraktion zu verbessern. Das System durchsucht dabei jede Zeile einer Datei auf die im Architekturwissen hinterlegten Indikatoren. Je nach Typ des pro Feature hinterlegen Indikators wird dabei die jeweilige Implementation einer Regel (`Rule`), abhängig von der gewählten Programmiersprache, angewendet. Einzelne Implementationen beschränken die Suche unter Nutzung von Parameter und Scope jeweils auf die für diesen Typen relevanten syntaktischen Konstrukte. Die Anwendung der jeweiligen Regeln liefert abstrahiert lediglich Informationen darüber, ob ein Indikator identifiziert werden konnte. Eine Nachverarbeitung aggregiert anschließend die Ergebnisse und fasst Technologie-Features auch auf Ebene der jeweiligen Programmpakete zusammen.

Die Ergebnisse stehen im Anschluss für die weiteren Verarbeitungsschritte zur Verfügung. Eine Bestimmung von Alternativen nutzt diese im Rahmen des `AlternativeService`. Dieser gruppiert zunächst die extrahierten Features und bestimmt genutzte Technologien. Anhand dieser Gruppierungen wird eine Menge von Alternativen bestimmt, welche sich aus den im Architekturwissen hinterlegten Technologien ergibt. Ebenfalls wird die Abdeckungsrate der Alternativen berechnet. Die Visualisierung in Form einer Matrix bietet einen Überblick über potentielle Alternativen sowie deren Qualität. Auf einen Blick sind implementierte beziehungsweise von Alternativen nicht implementierte Features einzusehen. Aus dieser Informationen und der kontextbezogenen Relevanz eingesetzter Features können Constraints abgeleitet werden. Alternativen, die essentielle Features nicht unterstützen, kommen nicht als Lösung in Frage.

Identifizierte Alternativen können anschließend durch den `DecisionpointService` verarbeitet werden. Für jedes Technologie-Paar aus den Alternativen wird der Pfad ihrer gemeinsamen Abhängigkeiten bestimmt. Ergebnis des gesamten Prozesses ist eine Menge potentieller Entscheidungen. Aus diesen kann die für den aktuellen Anwendungskontext sinnvollste Alternative gewählt werden. Die Bestimmung des Kontextes und damit der optimalen Lösung ist im Rahmen der Annahmen bewusst nicht implementiert.

Der Prototyp soll den Nutzer beim Finden möglicher Lösungen unterstützen, indem Extraktion und Bestimmung den Lösungsraum eingrenzen. Durch die gezielte Betrachtung von ASTAs kann jedoch ebenfalls die Betrachtung des Kontextes zielgerichtet gesteuert werden. Betrachtet man die Menge an potentielle Lösungen im Architekturwissen, grenzt bereits der Prototyp die vorliegende Wissensbasis sinnvoll ein und ermöglicht somit eine Fokussierung auf den eigentlichen Kontext des Anwendungsfalls.

5.3 Technische Implementation und Hintergründe

Allgemeines

Den technischen Kern der Implementation bildet eine MySQL-Datenbank zur Verwaltung des Architekturwissens. Hintergrund sind hier, wie eingangs erwähnt, verwandte Ansätze zur Verwaltung von Architekturwissen. Potentiell fällt für die Analyse eine große Datenmenge an. Dies ist der potentielle Lösungsraum aller Entwurfsentscheidungen.

Die verwendete Datenbasis hat in ihrem Umfang ohne Implementation der genannten Optimierungen (Siehe Abschnitt 4.2) einen deutlichen Einfluss auf die Laufzeit der Extraktion. Diese wächst bei zunehmender Menge an Architekturwissen exponentiell an. Es muss zunächst angenommen werden, dass Merkmale jedes Features genutzten werden können. Weitere Optimierungen müssen diesen Sachverhalt näher betrachten und den Suchraum vorweg reduzieren.

Der Prototyp ist in der Sprache Java verfasst. Grundlage ist die Abbildung des Metamodell auf eine Datenbasis. Weiterhin erfolgt der Zugriff auf die Datenbasis mittels der Technologie *Hibernate*. Diese bildet eine Schicht für den Datenbankzugriff und ermöglicht den zentralen Zugriff auf die benötigten Datenstrukturen. Zudem sorgt dies für eine lose Kopplung bei hoher Kohärenz der einzelnen Arbeitsschritte im Prozess.

Die genutzten Technologien und Programmiersprache sind bewusst nah am Anwendungsfall des *DecisionBuddy* und dem Konzept selbst gehalten: Aus der manuellen Extraktion gewonnene Informationen und Features können am Prototypen selbst verifiziert werden. Dies bedeutet im Umkehrschluss, dass sich der Quellcode des Prototypen durch diesen selbst untersuchen lässt. Aufgrund der verwandten Menge hinterlegter Features erzielt das System verlässliche Ergebnisse auf Basis seiner eigenen Quellen.

Die Implementation des Prototypen sieht eine Reihe einzelner Arbeitsschritte aus zwei Gründen vor: Einerseits können pro Schritt Grenzwerte angepasst und somit Ergebnisse feiner oder gröber granuliert werden. Andererseits spielen Laufzeiten und Atomarität der Ergebnisse eine Rolle. Gerade die Extraktion kann aufgrund einer großen Datenbasis einen hohen Aufwand bedeuteten.

In letzterem Fall können Ergebnisse eines Schrittes auch ohne weitere Verarbeitung genutzt werden. Extrahierte Technologie-Features etwa können in Dokumentation nachgepflegt werden. Ebenso kann das Ergebnis manuell nachvollzogen und bewertet werden. Erst im Anschluss sollte eine Bestimmung von Alternativen erfolgen. Diese ist maßgeblich auf ein fundiertes Kontextwissen und eine sinnvolle Auswertung angewiesen. Ähnliches gilt im Anschluss für die Bestimmung von Entscheidungspunkten.

Für den Prozess ist eine niedrige Laufzeit der Extraktion nicht zwangsweise erforderlich: Eine Extraktion wird pro Iterationsschritt im Regelfall ein einziges Mal durchgeführt, eine sinnvolle Konfiguration der Grenzwerte vorausgesetzt. Der Einsatz stellt somit, anderes als Maßnahmen des Refactoring oder ähnlichem, keinen

zeitkritischen Bestandteil des Prozesses dar. Es bestehen keine Echtzeitanforderungen.

Regeln

Implementation des Interfaces `Rule` betrachten je nach betrachtetem Sprachkonstrukt eine Menge regulärer Ausdrücke. Konkret setzt der Prototyp die definierten Typen von Indikatoren für die Sprache Java um. Die Typen *Schlüsselwort* sowie *Regulärer Ausdruck* sind Sprachunabhängig. Zusätzlich bietet der Prototyp ebenfalls die Möglichkeit, XML-basierte Formate zu erkennen. Dazu gehören beispielsweise XML-Tags sowie deren Parameter. Dieser Fall deckt etwaige von *Hibernate* für ein ORM-Mapping genutzte Konfigurations-Dateien ab. Die allgemeinen Regeln dienen als Absicherung, sollten für eine bestimmte Sprache beziehungsweise Sprachkonstrukt keine Regel definiert sein. Weitere Sprachen lassen sich als Erweiterungen des Regelwerkes umsetzen. Sprachen mit ähnlichen syntaktischen Konstrukten (Etwa C#) können dieselbe oder eine ähnliche Menge von Indikator-Typen nutzen. In diesem Fall unterscheiden sich lediglich die spezifischen Schlüsselworte sowie die Syntax. Pro Sprache und relevantem Indikator-Typ muss eine entsprechende Implementation ergänzt werden. Dies setzt eine entsprechende Kategorisierung und Formalisierung von Sprachkonstrukten voraus.

Die resultierende Menge implementierter Regeln beschränkt sich auf die in Abbildung 5.2 gezeigten Typen.

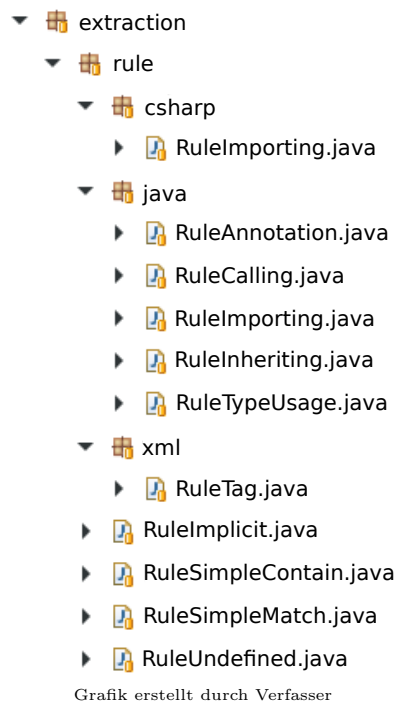


Abbildung 5.2: Implementierte Regeln im Rahmen des *FeatureExtractor*

Kapitel 6

Evaluation

Das folgende Kapitel behandelt abschließend die Evaluation des vorangegangenen Konzeptes. Der Fokus liegt hierbei auf der Extraktion von Technologie-Features. Eine Betrachtung von Alternativen sowie Entscheidungspunkten erfolgt im Rahmen der Annahmen aus Abschnitt 3 nur exemplarisch. Die eingeschränkte Wissensbasis eignet sich lediglich zur Erläuterung der grundsätzlichen Funktionsweise dieser Konzepte sowie deren potentiellen Ergebnissen.

Ebenso werden mögliche Vorteile im Rahmen des Reverse Engineering beleuchtet. Im diesem Zuge sind einige Grenzen der Allgemeingültigkeit des Konzeptes hervorzuheben. Diese ergeben sich aus der Aufgabenstellung und den daraus resultierenden Annahmen.

Auswertung

Die Evaluation soll unter Berücksichtigung der genutzten Wissensbasis die Genauigkeit der Ergebnisse bewerten. Dies bedeutet im Umkehrschluss die Allgemeingültigkeit der Extraktion im Rahmen der Aufgabenstellung.
Da sowohl der für die Ausgestaltung des Konzeptes genutzte *DecisionBuddy* als auch der *FeatureExtractor* selbst in ihrem Umfang über eine geringe Praxisrelevanz verfügen, sind deren Extraktionsergebnisse weniger repräsentativ. Wenngleich auch für diese beiden Systeme die Betrachtung der Ergebnisse sinnvoll ist, sollen im Folgenden zusätzliche Systeme aus der Praxis als Grundlage der Analyse dienen. Eingeschränkt durch die für den *DecisionBuddy* befüllte Wissensbasis soll die Genauigkeit der Extraktionsergebnisse und damit deren Qualität betrachtet werden.

Die Extraktion wird unter Nutzung des *FeatureExtractor* durchgeführt. Sukzessive Durchläufe des Extraktionsprozesses erhöhen stückweise die Konfidenz und liefern entsprechend feinere Ergebnisse. Welche Schlussfolgerungen sich aus diesem Verhalten ziehen lassen, soll die Evaluation im Folgenden zeigen.

Die Qualität der Extraktion ist wie folgt zu bewerten: Identifiziert werden konkrete Instanzen der Nutzung eines Technologie-Features. Dies bedeutet den Einsatz eines Features in einer relevanten Datei. Features können dabei pro System mehrfach erkannt werden. Entsprechendes gilt pro Programmpaket. Je Datei wird für erkannte Features die Konfidenz (Abschnitt 4.2) berechnet. Wird ein Feature pro

Datei mehrfach verwendet, wird es nicht mehrfach erkannt. In diesem Fall steigt lediglich die zugeordnete Konfidenz.

Auf Grundlage der Extraktionsergebnisse sollen für die betrachteten Systeme Precision sowie Recall nach [Powers, 2007] errechnet werden. Relevant sind hierbei die Menge korrekt erkannter Nutzungen (True Positives - TP), fälschlicherweise erkannter Nutzungen (False Positives - FP) sowie fälschlicherweise nicht erkannter Nutzungen (False Negatives - FN). Zentral zur Bewertung der Qualität sind dabei die Betrachtung der Fehlerquote des Systems sowie der Anteil korrekt erkannter Nutzungen.

Wird die Nutzung eines Features in einer Quelldatei erkannt ist zu prüfen, ob entsprechende syntaktische Konstrukte (Indikatoren) tatsächlich vorliegen. Dies erfolgt analog zur händischen Erfassung der Features zuvor. Ist dies der Fall, wurde die Nutzung des Technologie-Feature korrekt identifiziert. Fälschlicherweise erkannte Nutzungen lassen sich zumeist auf die in Abschnitt 4.2 genanten Problematiken zurückführen. Fälschlicherweise nicht erkannte Nutzungen lassen sich ähnlich identifizieren: Eine Volltextsuche über alle Dateien resultiert in einer Menge für dieses Konstrukt relevanter Dateien. Ein händischer Abgleich bestimmt, ob die Nutzung korrekt erkannt oder bei der Extraktion nicht berücksichtigt wurde.

Tabelle A.1[1] zeigt die vollständigen Ergebnisse der Extraktion der besseren Übersicht halber gebündelt. Pro Durchlauf wurde der Schwellwert für die Konfidenz der Ergebnisse sukzessive erhöht. Je Zeile findet sich ein Durchlauf der Extraktion unter Verwendung des jeweiligen Schwellwertes. Pro Durchlauf wurden die Ergebnisse analysiert und die korrekte Erkennung von Features im Quellcode verifiziert. Rechts gibt die Tabelle dabei relevante Werte für Precision sowie Recall an.

FeatureExtractor Der *FeatureExtractor* setzt eine Teilmenge der von *Hibernate* und JPA angebotenen Features um. Die Extraktion liefert in diesem Rahmen eine hohe Genauigkeit: In den relevanten 135 Quellcode- und Konfigurationsdateien werden 13 Nutzungen 6 unterschiedlicher Features korrekt erkannt (Tabelle 6.1). Das System hat einen Umfang von 10500 Lines of Code (LOC). Eine manuelle Analyse zeigt keine fälschlicherweise nicht erkannten Nutzungen, ebenso kein fälschlicherweise erkannten Nutzungen.

Identifizierte Instanzen der Nutzung weisen eine Konfidenz von 60 bis 85 Prozent auf. Unterschiede ergeben sich hier aus den jeweiligen Gewichtungen der relevanten Indikator-Typen. Erste Vermutungen lassen darauf schließen, dass die Extraktionsergebnisse bei Überschreitung einer gewissen Konfidenz solide Ergebnisse erzielen.

DecisionBuddy Die Evaluation der Ergebnisse des *DecisionBuddy* liefert ebenfalls solide Ergebnisse. Um die extrahierten Technologie-Features jedoch sinnvoll auswerten zu können, müssen sie detailliert betrachtet werden.

Aus den 382 relevanten Quelldateien (41400 LOC) werden 457 konkrete Instanzen

[1] Siehe Anhang, Seite VII

Min. Konfidenz	Instanzen	Dateien	Features	TP	FP	FN	Precision	Recall
0.00	13	135	6	13	0	0	1.000	1.000
0.25	13	135	6	13	0	0	1.000	1.000
0.50	13	135	6	13	0	0	1.000	1.000
0.75	4	135	2	4	0	9	1.000	0.308
1.00	0	135	0	0	0	13	0.000	0.000

Tabelle 6.1: Ergebnisse Extraktion *FeatureExtractor*

extrahiert (Tabelle 6.2). Es wird die Nutzung von bis zu 50 unterschiedlichen Features identifiziert. Wie schon im Fall des *FeatureExtractor*, ergeben sich bei ähnlichen Schwellwerten zur Konfidenz belastbare Ergebnisse. Obwohl der *FeatureExtractor* auf Basis des *DecisionBuddy* entworfen wurde, sind keine eindeutigen Ergebnisse zu erwarten. Es ist nicht damit zu rechnen, dass lediglich tatsächlich genutzte Features identifiziert werden. Die Unschärfe bei der Erkennung der Technologie-Features ist die Ursache für fälschlicherweise erkannte Features. Diese weisen jedoch im Vergleich zu tatsächlich genutzten Features eine wesentlich geringere Konfidenz auf. Eine Justierung des Schwellwertes erzielt das gewünscht Ergebnis. Aufgrund des kontextfreien Textvergleiches kommt es wie in Abschnitt 4.2 beschreiben zu fälschlich positiv erkannten Ergebnissen. Auch in diesem Fall unterschlägt die Extraktion kein im Architekturwissen hinterlegtes Features. Falsche positive Ergebnisse treten größtenteils aufgrund von Importen überlappender Programmpakete auf.

Min. Konfidenz	Instanzen	Dateien	Features	TP	FP	FN	Precision	Recall
0.00	469	382	50	386	83	0	0.823	1.000
0.25	469	382	50	386	83	0	0.823	1.000
0.27	401	382	42	386	15	0	0.962	1.000
0.48	351	382	39	351	0	35	1.000	0.912
0.50	351	382	39	351	0	35	1.000	0.912
0.75	141	382	26	141	0	245	1.000	0.365
1.00	0	382	0	0	0	386	0.000	0.000

Tabelle 6.2: Ergebnisse Extraktion *DecisionBuddy*

Praxisrelevante Anwendungssysteme Um die Evaluation um praxisnahe Anwendungsfälle zu erweitern, wurden einige offene Systeme für die weitere Betrachtung ausgewählt. Kriterien bei der Auswahl waren offene Verfügbarkeit des Quellcodes, bekannte Nutzung von Features aus *Hibernate* sowie JPA und ein entsprechender Umfang (Menge/Komplexität der Quellcode-Dateien).

Die Auswahl fiel auf die freien Systeme *BroadLeaf*[2], *Plazma*[3] sowie *PowerStone*[4]. Der tatsächliche Funktionsumfang der Systeme ist in diesem Zusammenhang weniger relevant. Eine Betrachtung soll zeigen, dass im Architekturwissen hinterlegte Features allgemeingültig anhand der definierten Regeln zu erkennen sind. Die Systeme sollen, ähnlich der Analyse des *DecisionBuddy*, solide Ergebnisse liefern. Die

[2]Siehe https://github.com/BroadleafCommerce/BroadleafCommerce,
zuletzt aufgerufen am 18. Oktober 2016

[3]Siehe http://plazma.sourceforge.net/, zuletzt aufgerufen am 18. Oktober 2016

[4]Siehe https://sourceforge.net/projects/powerstone/, zuletzt aufgerufen am 18. Oktober 2016

eingehende Analyse zeigt eine angebrachte Nutzung der im Architekturwissen hinterlegten Technologien. Dies bedeutet, dass die betreffenden Systeme mehrere Features unterschiedlicher Technologien nutzen.

Weitere der durch die Systeme genutzten Features auch der bereits hinterlegter Technologien wurden nicht modelliert und werden im Rahmen der Annahmen entsprechend nicht erkannt. Betrachtet wird hier speziell die bereits genutzte Wissensbasis. Nicht hinterlegte Technologie-Features lassen sich jedoch auf Grundlage des Konzeptes formalisieren und hinterlegen. Sie werden deshalb im Folgenden nicht als falsche positive Ergebnisse gewertet.

BroadLeaf Die Kernkomponente des Systems *BroadLeaf* umfasst insgesamt 3861 relevante Dateien. Die Extraktion bestimmt die Nutzung 3139 konkreter Instanzen innerhalb des gesamten Systems. Da sich in diesem Fall die händische Analyse tatsächlich korrekt erkannter Features sehr umfangreich darstellt, wird eine Teilmenge des Systems repräsentativ betrachtet. Entsprechend erfolgt die weitere Evaluation auf Grundlage der Kernkomponente des von *BroadLeaf* bereitgestellten Anwendungsframeworks (*BroadLeaf* Core). Diese umfasst etwa ein Drittel des Umfanges des gesamten Frameworks.

Innerhalb dieser Komponenten können 376 konkrete Instanzen in 735 Dateien identifiziert werden (Tabelle 6.3). Das System umfasst etwa 130000 Zeilen Quellcode (LOC). Trotz unterschiedlicher Quellcode-Konventionen und einem gänzlich anderen Architekturansatz liefert der *FeatureExtractor* auch für dieses Framework solide Ergebnisse. Aufgrund des Umfanges ohne Schwellwert sind etwa zwei Drittel der identifizierten Nutzungen als falsche positive Ergebnisse einzuordnen. Diese sind vorrangig überlappenden Importe zuzuschreiben. Weitere falsche positive Ergebnisse erzielt das System aufgrund von Textvergleichen, die Schlüsselworte und Quellcodebeispiel in Kommentaren und Dokumentation aufweisen. Für diese wird jedoch wie erhofft eine niedrige Konfidenz bestimmt. Eine Erhöhung des Schwellwertes erlaubt es somit, diese fälschlicherweise identifizierten Features auszuschließen.

Min. Konfidenz	Instanzen	Dateien	Features	TP	FP	FN	Precision	Recall
0.00	376	735	33	239	137	0	0.636	1.000
0.25	376	735	33	239	137	0	0.636	1.000
0.27	296	735	30	239	57	0	0.806	1.000
0.50	236	735	27	236	0	3	1.000	0.986
0.75	66	735	11	66	0	173	1.000	0.275
1.00	0	735	0	0	0	235	0.000	0.000

Tabelle 6.3: Ergebnisse Extraktion *BroadLeaf*

Plazma Eine ähnliche Betrachtung wird anhand des Systems *Plazma* durchgeführt. Das System nutzt ebenfalls einen Teil der im Architekturwissen hinterlegten Features umfasst etwa 3600 relevante Quellcode-Dateien (Tabelle 6.4). Das System umfasst, ähnlich der betrachteten Komponente aus *BroadLeaf*, etwa 130000 LOC. Die Extraktion liefert dabei etwa 400 identifizierte Instanzen. Es ergeben sich ähnliche Ergebnisse wie für den *DecisionBuddy* als auch *BroadLeaf*. Wenngleich das System

umfangreicher ist, wird nur ein geringer Teil der hinterlegten Features genutzt. Dieser Umstand lässt sich auf die Kernmotivation zurückführen: Nicht jedes Technologie--Feature einer eingesetzten Technologie wird verwendet. Unterschiedliche Systeme nutzen eine eigene Menge von Funktionalitäten. Auffällig ist dies aufgrund der fehlenden Modellierung von Features tatsächlich genutzter Technologien. Diese werden auf Grundlage der Annahmen nicht erkannt. Bei entsprechender Modellierung sind jedoch vergleichbare Ergebnisse zu erwarten.

Min. Konfidenz	Instanzen	Dateien	Features	TP	FP	FN	Precision	Recall
0.00	403	3593	9	106	296	0	0.264	1.000
0.25	403	3593	9	106	296	0	0.264	1.000
0.50	103	3593	8	103	0	3	1.000	0.972
0.75	103	3593	8	103	0	3	1.000	0.972
1.00	0	3593	0	0	0	235	0.000	0.000

Tabelle 6.4: Ergebnisse Extraktion *Plazma*

PowerStone Mit *PowerStone* erfolgt aufgrund des geringen Umfanges eine weitere detaillierte Analyse, ähnlich zu Betrachtung des *FeatureExtractor*. Obgleich seines geringen Umfangs liefert die Extraktion auch hier ähnlich präzise Analyseergebnisse wie für umfangreiche Systeme. Aufgrund seines geringen Umfangs ermöglicht das System eine vollständige händische Analyse aller Quellcode-Dateien. Diese identifiziert in 35 Dateien (ca. 1300 LOC) insgesamt 28 Instanzen bis zu neun unterschiedlicher Features (Tabelle 6.5). Die Quote falscher positiver Ergebnisse liegt zunächst extrem hoch, eine Justierung des Schwellwertes schließt allerdings auch in diesem Fall entsprechende Ergebnisse aus und verbessert so die Qualität der Extraktion. Tatsächlich lässt sich für kleine System eine sehr viel genauere Adjustierung des Schwellwertes vornehmen: Ähnlich zu den Extraktionsergebnissen des *FeatureExtractor* ist dieser so zu adjustieren, dass keine falschen positiven Ergebnisse auftreten, alle zu identifizierenden Features jedoch trotzdem erkannt werden. Precision sowie Recall liegen in diesem Fall bei eins. Dies ist bei komplexen Systemen nicht zu beobachten.

Min. Konfidenz	Instanzen	Dateien	Features	TP	FP	FN	Precision	Recall
0.00	28	35	9	4	24	0	0.143	1.000
0.25	28	35	9	4	24	0	0.143	1.000
0.50	4	35	3	4	0	0	1.000	1.000
0.75	2	35	2	2	0	2	1.000	0.500
1.00	0	35	0	0	0	4	0.000	0.000

Tabelle 6.5: Ergebnisse Extraktion *PowerStone*

Interpretation der Beobachtungen Inwieweit die betrachteten Systeme und ihre Extraktionsergebnisse statistisch signifikant erscheinen, hängt stark vom Umfang des genutzten Architekturwissens ab. Unter den eingeschränkten Kriterien zur Auswahl der System beschränkt sich deren Diversität nach wie vor auf die Annahmen zur Beschränkung des Architekturwissens. Die Betrachtung weiterer Technologien sollte in ihren Grundzügen jedoch ähnliche Ergebnisse liefern, die Modellierung von Technologien und Features vorausgesetzt. Unter den Annahmen der Aufgabenstellung stellen die betrachteten Systeme somit eine sinnvolle Menge zu Betrachtung

dar, da sie a) eine ähnliche Menge von Technologien einsetzen, b) größtenteils einen praxisnahen Umfang im Bezug auf Architektur und Quellcode besitzen c) nicht aus derselben Quelle stammt. Entsprechend unterscheiden sich Formatierung, Programmierstil, Dokumentation in Kommentaren sowie weitere Merkmale, welche die syntaktische Analyse beeinflussen können.

Auf Grundlage der beschriebenen Evaluation lassen sich somit einige Aussagen über die Qualität und einen potentiellen Einsatz des Konzeptes zur Extraktion treffen.

Auffällig ist zunächst die hohe Trefferquote bei der Extraktion der Features. Ohne Konfiguration eines Schwellwertes fällt kein Features durch das Raster. Dies bedeutet im Umkehrschluss, dass im Architekturwissen hinterlegte Features mit hoher Wahrscheinlichkeit erkannt werden. Betrachtet man die Grundlage der Extraktion, ist dies nicht verwunderlich: Die Extraktion beschränkt sich auf eine durch die jeweilige Sprache fest definierter Schlüsselworte. Deren Einsatz und Struktur ist vorgegeben und lässt sich bei gültigen Quellcodedateien nur schwer umgehen. Da Indikatoren eben diesen Schlüsselworten zugeordnet sind, ist die Identifikation anhand simpler Textvergleiche möglich.

Die hohe Menge falscher positiver Ergebnisse lässt sich auf die generelle Nutzung von Technologien zurückführen. Viele Technologie-Features teilen sich dasselbe Programmpaket. Datentypen, Konstanten, Methoden und weitere Konstrukte stammen aus demselben Namensraum. Diese Namensräume bilden einen Indikator des jeweiligen Features ab. Für konkret genutzte Features können weitere Indikatoren identifiziert werden, nicht genutzte Features weisen nur den genutzten Namensraum auf. Das Konzept berücksichtigt diesen Umstand jedoch bereits: Für tatsächlich nicht genutzte Features wird trotz genutztem Programmpaket eine niedrige Konfidenz berechnet. Dies hat zu Folge, dass durch Justierung des Schwellwertes entsprechend falsche Ergebnisse ausgeschlossen werden können. Technologie-Features sollten zumindest einen Indikator ohne Bezug auf ein genutztes Programmpaket besitzen. So ist sichergestellt, dass entsprechende Fälle berücksichtigt werden können.

Die sukzessive Justierung des Schwellwertes lässt eine steigende Precision bei minimalen Änderungen des Recalls beobachten. Durch ein Erhöhen des Wertes lassen sich bereits beschriebene falsche negative Ergebnisse herausfiltern. Auffällig ist, dass tatsächlich genutzte Features, aufsteigend sortiert, allesamt eine hohe Konfidenz erhalten. Die entsprechende Berechnung durch den *FeatureExtractor* kann somit als qualitatives Indiz der Extraktion betrachtet werden.

Alle relevanten Ergebnisse werden erkannt. In seltenen Fällen liegen in dieser Ordnung falsche Positive zwischen den tatsächlichen positiven Ergebnissen. In diesem Fall sinkt bei Steigerung des Schwellwertes der Recall. Ein Teil der relevanten Ergebnisse wird nicht erfasst. Die Justierung des Schwellwertes ist somit ein wichtiges Maß zur Steuerung der Qualität der Extraktion. Die Betrachtung der analysierten Systeme weist jedoch keinen fixen Schwellwert auf: Welcher Grenzwert für das betrachtete System sinnvoll ist, muss im jeweiligen Kontext betrachtet werden. Um diesen Umstand zu beherrschen, ist unter Umständen ein manuelles Überprüfen von Resultaten mit niedriger Konfidenz notwendig.

Anders als eine händische Analyse bietet die Extraktion dennoch eine signifikante Einschränkung des Lösungsraumes. Nach einer händischen Prüfung kann der Schwellwert entsprechend neu gesetzt werden, um tatsächlich nicht verwendete Ergebnisse auszuschließen. Für den weiteren Prozess bedeutet dies eine höhere Qualität der Ergebnisse. Je nach Anwendungssystem, so zeigen die beispielhaften Berechnungen, kann sich dieser Schwellwert in seiner Ausprägung unterscheiden.

6.1 Nutzen

Geht man von einer gewissen Signifikanz obiger Beobachtungen aus, lassen sich einige Vorteile des Konzeptes betrachten. Im Kontext des Reverse Engineering ist der Ansatz zunächst als eine Rekonstruktion implementierter Technologie-Features zu verstehen. Aus diesen Informationen können Aussagen über den weiteren Prozess und das betrachtete System gewonnen werden.

Zunächst muss betrachtet werden, inwieweit sich das vorgeschlagene Konzept von anderen Ansätzen des Reverse Engineering unterscheidet. Anders als Ansätze zur Rekonstruktion der Architektur [Gueheneuc u. a., 2006] [McMillan u. a., 2009] beschränkt sich das vorgeschlagene Konzept rein auf die syntaktischen Eigenschaften des analysierten Systems. Verwandte Ansätze berufen sich zumeist auf die Struktur und Architektur der analysierten Elemente (etwa Beziehungen zwischen Klassen), um Entwurfsmuster und weitere Strukturen zu erkennen. Entsprechende Strukturen sind unabhängig von extern betrachtetem Architekturwissen vorhanden und können analysiert werden. Der Ansatz zur Extraktion von Technologie-Features erfordert jedoch eine semantische Menge von Architekturwissen zur Anreicherung. Dies erfordert zwar einen hohen Aufwand, erhöht die Präzision der Ergebnisse jedoch wesentlich. Aus den gewonnen Informationen sind jedoch keine direkten Aussagen über die Softwarearchitektur eines Systems möglich. Potentielle Ableitungen müssen geschlussfolgert werden.

Aus den gewonnen Informationen über genutzte Technologie-Features lassen sich, eine konsequente Softwarearchitektur vorausgesetzt, Ableitungen über Schichten und Kapselungen treffen: Eine Konzentrierung bestimmter Technologien oder Features in einem oder mehreren Programmpaketen kann Aussagen über die Architektur erlauben. Werden in einer Schicht etwa ausschließlich Features für den Zugriff auf Datenbanken verwendet, kann davon ausgegangen werden, dass es sich hierbei um eine Schicht für den Datenbankzugriff handelt. Im gegensätzlichen Fall lassen sich parallele Aussagen treffen: Sind Features nicht gruppiert (etwa Datenbankzugriff aus allen Komponenten des Systems heraus), kann dies auf eine inkonsequente Architektur hinweisen.

Information über tatsächlich genutzte Technologie-Features können den Prozess weiterhin bereichern. Die Dokumentation des Systems kann gepflegt oder ergänzt werden. Eine explizite Betrachtung tatsächlich genutzter Features kann Unklarheiten beseitigen. Tacit Knowledge kann ausgeräumt werden [Kruchten, 1995] [Hofmeister u. a., 2000].

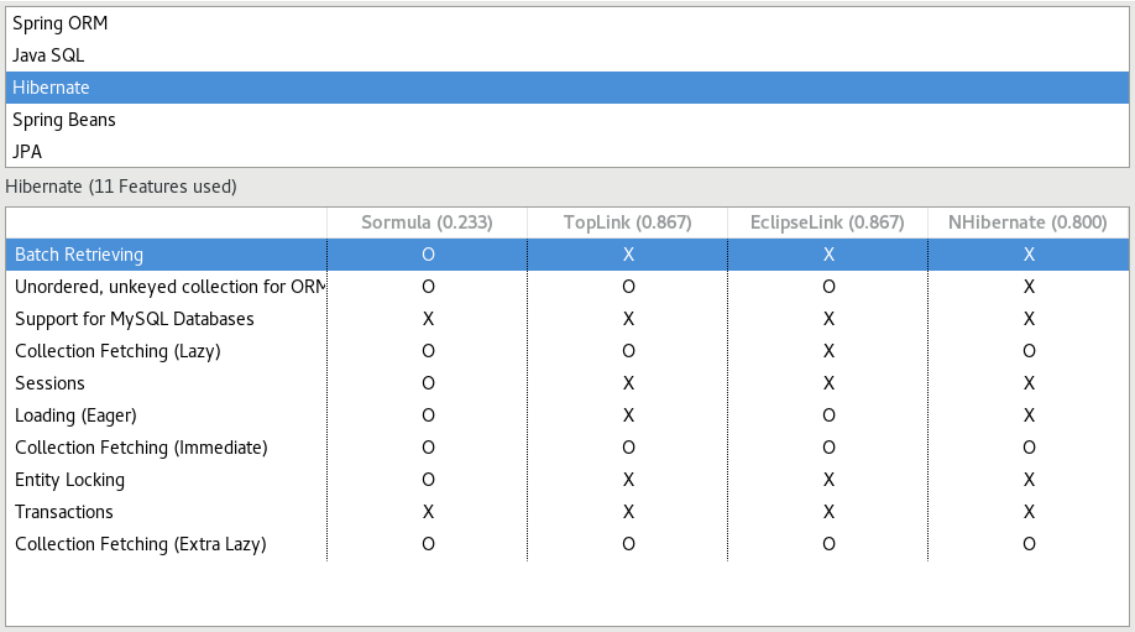
Bei der Betrachtung der Nutzung kann zudem in gewissen Aspekten eine Kluft zwischen Architektur und Implementation festgestellt werden. Dies geschieht unter Zuhilfenahme der Erkenntnisse der vorherigen Betrachtung der Architektur. Die Implementation des Systems nutzt unter Umständen Technologien, die von der Softwarearchitektur des Systems und seinen Entwurfsentscheidungen nicht vorgesehen sind. Diese Form der technischen Schuld wird durch die Extraktion tatsächlich genutzter Features explizit.
Die Extraktion genutzter Technologie-Features kann zudem Aufschluss über die externen Abhängigkeiten des Systems geben: Es kann bestimmt werden, welche Technologien verwendet werden. Ebenso kann eine Aussage darüber getroffen werden, welche Teile des Systems bei Änderungen an der Nutzung dieser Technologien betroffen sind.

Die manuelle Suche nach alternativen Technologien umfasst zumeist eine große Menge potentieller Kandidaten. Durch eine Fokussierung auf eingesetzte Technologie-Features schlägt das Konzept gezielt eine Menge sinnvoller Alternativen vor. Dies reduziert den Suchraum teils drastisch. Statt einer Menge lose zusammenhängender Lösungen aus dem Architekturwissen müssen lediglich einige wenige Lösungen auf ihre Tauglichkeit hin evaluiert werden. Abbildung 6.1 zeigt die vom *FeatureExtractor* vorgeschlagenen Alternativen auf Grundlage extrahierter Features. In Form einer Abdeckungsmatrix kann übersichtlich aufgeführt werden, welche alternative Technologie welches der geforderten Features implementiert. Somit kann bei der Auswahl anhand der jeweiligen Abdeckungsrate priorisiert werden.

Ebenso können von Alternativen zu implementierende Features priorisiert werden: Kritische Features können bei der Auswahl ein fixes Constraint darstellen, das nicht aufgeweicht werden kann. Alternative Implementationen weiterer Features können sich entsprechend als weniger relevant darstellen. Somit können anhand der implementierten Features Technologien schnell ausgeschlossen werden.

Weiterhin ist auf dieser Grundlage eine Fokussierung auf relevante Aspekte der Technologien und ihrer Features möglich. Die im Metamodell eingebundenen ASTAs aus [Soliman u. a., 2015] finden ihre Anwendung in diesem Aspekt: Um aus den potentiellen Alternativen einen Kandidaten auszuwählen, sind diese mit entsprechendem Kontextwissen zu vergleichen und abzuwägen. Die im exemplarischen Anwendungsfall beschriebene Problematik wird hier abgebildet. Abbildung 6.2 zeigt entsprechend die Aspekte des relevanten Features im direkten Vergleich. Anhand der Vor- und Nachteile der Features der jeweiligen Technologien können fundiertere Entscheidungen getroffen werden.

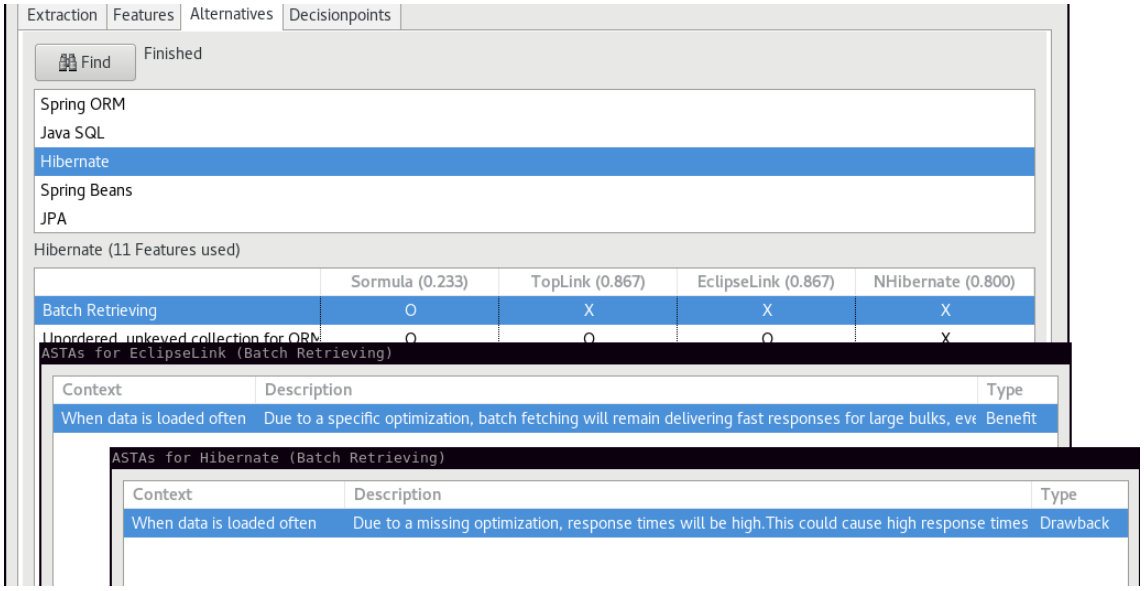
Ähnliches gilt für das nachfolgende Treffen konkreter Entscheidungen. Wenngleich diese ebenfalls stark vom hier nicht näher betrachteten Kontext der Anwendung abhängen, sind die vorgeschlagenen Entscheidungspunkte eine weitere Möglichkeit zur Eingrenzung des Lösungsraumes. Sie geben vor, an welchen Stellen im Entscheidungsbaum Abhängigkeiten beachtet werden müssen und Entscheidungen potentiell revidiert werden müssen. Abbildung 6.3 zeigt eine entsprechende Ausgabe des Prototypen für den Anwendungsfall. Für jede potentielle Alternative kann bestimmt werden, welche abhängigen Technologien beachtet werden müssen. Hintergrund dieser Betrachtung ist die in der Softwarearchitektur häufig angewandte Abstraktion:



	Sormula (0.233)	TopLink (0.867)	EclipseLink (0.867)	NHibernate (0.800)
Batch Retrieving	O	X	X	X
Unordered, unkeyed collection for ORM	O	O	O	X
Support for MySQL Databases	X	X	X	X
Collection Fetching (Lazy)	O	O	X	O
Sessions	O	X	X	X
Loading (Eager)	O	X	O	X
Collection Fetching (Immediate)	O	O	O	O
Entity Locking	O	X	X	X
Transactions	X	X	X	X
Collection Fetching (Extra Lazy)	O	O	O	O

Grafik erstellt durch Verfasser

Abbildung 6.1: *FeatureExtractor*, Abdeckungsmatrix: Alternativen im Rahmen des Anwendungsfalls *DecisionBuddy*



Grafik erstellt durch Verfasser

Abbildung 6.2: *FeatureExtractor*, Betrachtung relevanter ASTA pro Feature im Rahmen des Anwendungsfalls *DecisionBuddy*

Werden tatsächlich verwendete Features und deren Abhängigkeiten betrachtet, kann ein Fokus auf deren Abhängigkeiten die Komplexität des zu betrachtenden Entscheidungsbaumes ausblenden.

Eine weitere maschinelle Verarbeitung der Ergebnisse soll hier nur marginal betrachtet. Denkbar wären Einbindungen in weitere Tools zur Unterstützung der Softwarearchitektur oder automatisierten Generierung von Dokumentationen. Sowohl die Formalisierung von Indikatoren als auch die Ablage der Ergebnisse und deren Aussa-

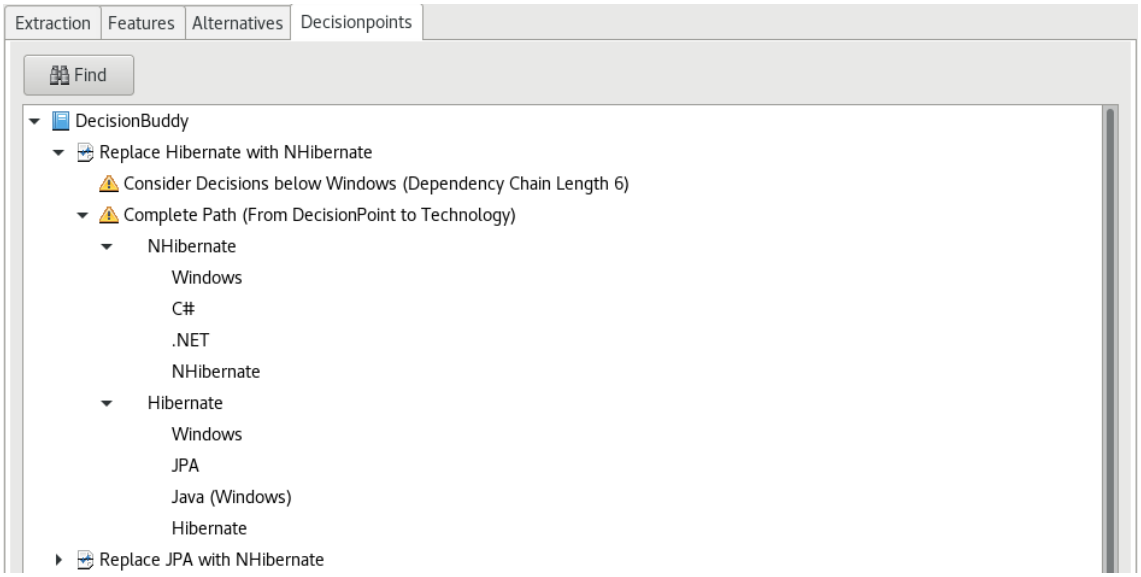


Abbildung 6.3: *FeatureExtractor*, Bestimmung von Entscheidungspunkten

ge sind bewusst simpel gehalten, um eine leichte Erweiterbarkeit zu ermöglichen. Die Einbindung in verwandte Konzepte der Softwarearchitektur und dementsprechende Ansätze des Architekturwissen sind für eine potentielle Nutzung mittelfristig anzustreben.

6.2 Grenzen der Allgemeingültigkeit

Die Ergebnisse der Evaluation versprechen zwar einige Vorteile, ihre Allgemeingültigkeit muss jedoch gesondert betrachtet werden. Grund hierfür sind vorrangig Einschränkungen, die bei der Konzeptionierung sowie Betrachtung der Annahmen berücksichtigt wurden. Im Folgenden sollen zentrale Elemente dieser Grenzen beleuchtet und eingeordnet werden.

Das dem *FeatureExtractor* zugrundeliegende Konzept berücksichtigt nach [Nie u. Zhang, 2012] lediglich die statische Ebene der Analyse von Anwendungssystemen. Der Kontext der Nutzung wird bewusst nicht mit einbezogen. Verwandte Ansätze der Softwarearchitektur beziehen diesen ein, um Aussagen über die Struktur der Systeme zu ziehen. Anders als der Ansatz zur Extraktion kommen verwandte Ansätze des Reverse Engineering auch ohne Architekturwissen aus. Etwa Beziehungen und Struktur von Quellcode-Elementen untereinander lassen sich auch ohne zusätzliches Wissen bestimmen. Da der vorgeschlagene Ansatz jedoch unabhängig von diesen Strukturen umgesetzt werden soll, ist zusätzliches Wissen erforderlich. Dieses Wissen reichert den Prozess jedoch nur um syntaktische Informationen an, bezieht somit den Anwendungskontext nicht ein. Die Nutzung bestimmter Technologie-Features hat zwar im Prozess der Entscheidungsfindung ihre Hintergründe, diese sind zur Bestimmung der tatsächlichen Nutzung im Umkehrschluss jedoch nicht erforderlich. Wie eingangs erwähnt, beziehen sich Technologien selbst auf eine Menge angebotener Schnittstellen. Diese sind aufgrund technischer Gegebenheiten nunmehr lediglich auf syntaktischer Ebene zu erkennen. Eine tatsächliche Nutzung der Features zur

Laufzeit kann ohne Kontextwissen nicht bestimmt werden. Diese erscheint jedoch in dem betrachteten Kontext wenig relevant: Die tatsächliche Nutzung ist eher eine Frage der Architektur. Abhängigkeiten zu Technologien ergeben sich im Quellcode und sind somit eine technische Voraussetzung für eine fehlerfrei Kompilation sowie Installation. Lediglich diese Ebene wird entsprechend betrachtet.

Das genutzte Architekturwissen ist sehr beschränkt. Im Rahmen der Annahmen wurde es manuell befüllt, um dem Konzept beispielhaft als Grundlage zu dienen. Allgemeine Aussagen lassen sich auf Grundlage der gewählten Modellierung somit nicht zwangsläufig treffen. Dennoch stützt sich das Konzept auf bekannte Ansätze zur Betrachtung von Architekturwissen und erweitert diese. Die Wissenschaft befasst sich sowohl mit dem Aufbau eines Ansatzes zur Erfassung von Architekturwissen [Hansen u. a., 1999] als auch dessen automatisierter Befüllung [Soliman u. a., 2016] [Gorton u. a., 2015].

Ein Fortführen entsprechender Ansätze ermöglicht die Einbindung des vorliegenden Konzeptes in vorhandene Strukturen sowie deren Befüllen. Das Erfassen von Indikatoren bildet eine zusätzliche Maßnahme, die besonderer Betrachtung bedarf: Deren Erfassung und Formalisierung kann potentiell auf ähnlicher Grundlage aus Dokumentation, Tutorials sowie verwandten Dokumenten extrahiert werden. Eine entsprechende Erfassung setzt das Konzept im Rahmen der Annahmen mitunter händisch um.

Die gewählte exemplarische Wissensbasis fokussiert sich entsprechend der Aufgabenstellung auf ein konkretes System und wenige Technologien. Bereits oberflächliche Betrachtungen weiterer Technologien zeigten jedoch, dass die beschriebene Formalisierung in ihren Grundzügen valide bleibt.

Grundlage des exemplarischen Architekturwissens sind vorrangig Dokumentation, Tutorials sowie Online-Foren. Deren wissenschaftliche Relevanz lässt sich zumeist anzweifeln. Ihre hohe Praxisrelevanz ermöglicht dennoch solide Aussagen über den Einsatz sowie die Nutzung der beschriebenen Technologie-Features. Auch Ansätze der Wissenschaft befassen sich mit der Extraktion von Architekturwissen aus Online-Quellen [Soliman u. a., 2016].

Die bereits in Abschnitt 4.2 beschriebenen Möglichkeiten zur Beschränkung der Komplexität zeigen weitere marginale Schwächen auf. Eine hohe Komplexität der Extraktion sowie weitere syntaktische Schwächen machen den Ansatz in der Praxis nicht unfehlbar. Potentiell hohe Laufzeiten und falsche positive Ergebnisse sind die Folge. Um die Laufzeit des Ansatzes zu verbessern lassen sich auf Grundlage der Syntax einer betrachteten Sprache weitere Einschränkungen vornehmen. Diese können zur Folge haben, dass die zu betrachtenden Sprachkonstrukte pro Zeile reduziert werden können (indem nur auf Konstrukte geprüft wird, die laut Syntax an dieser Stelle überhaupt korrekt sind). Um den Ansatz in seinem Kern möglichst allgemein und übertragbar zu halten, wurde auf diese Optimierungen verzichtet. Andere Programmiersprachen machen unter Umständen gröbere oder weniger restriktive Einschränkungen. Eine entsprechende Umsetzung bleibt offen.

Um die Problematik falscher positiver Ergebnisse zu adressieren, beinhaltet das Konzept bereits die Berücksichtigung der Konfidenz von Extraktionsergebnissen. Wie die Evaluation zeigt, lässt sich diese tatsächlich sinnvoll verwenden, um die Qualität im Prozess der Extraktion zu verbessern.

Eine Evaluation von Konzept und Prototyp im konkreten Anwendungskontext ist unter den genannten Annahmen bisweilen nicht möglich. Um diese sinnvoll durchführen zu können, ist eine Menge von Architekturwissen im einem praxisrelevanten Umfang notwendig. Dies bedeutet etwa, eine komplette Menge von Features einer Technologie zu hinterlegen. Ebenso sind zumindest relevante Alternativen zu hinterlegen. Die Auswirkungen der Ergebnisse, ebenso vorgeschlagene Alternativen, Entscheidungspunkte sowie ASTAs sind jedoch nur im Kontext der Softwarearchitektur abzuschätzen. Sinnvollerweise sind Studien über die Sinnhaftigkeit der tatsächlichen Unterstützung im Prozess durchzuführen. Konkret ist die Qualität der tatsächlichen Unterstützung für den nutzenden Softwarearchitekten zu bewerten. Eine entsprechende Auswertung erscheint im Rahmen dieser Arbeit unter Anbetracht der Einschränkungen wenig sinnvoll.

6.3 Ausblick

Bevor abschließend eine Schlussfolgerung die Ergebnisse der Arbeit zusammenfasst, sollen potentielle zukünftige Forschungsthemen beleuchtet werden. Die Folgende Auflistung potentieller Schwerpunkte zeigt auf, inwieweit das vorangegangene Konzept zu Erweitern und weiterhin zu evaluieren ist.

Unter den Annahmen der Aufgabenstellung ist weitere Arbeit bei der Formalisierung und dem Befüllen des Architekturwissens zentral. Mit diesem und verwandten Ansätzen steigt die potentielle Praxistauglichkeit des vorgestellten Konzeptes. Zukünftige Ansätze können die Formalisierung von Technologie-Features und ihren Indikator berücksichtigen.

Das Befüllen des Architekturwissens mit Indikatoren stellt ein weiteres potentielles Forschungsthema dar. Potentiell ist, analog zu [Soliman u. a., 2016] auch eine Extraktion dieser aus der Dokumentation der Hersteller, Tutorials oder weitere Quellen denkbar. Durch ständige Anpassungen und Erweiterungen von Technologien muss die Datenbasis in Folge aktuell gehalten werden. Zu Verfolgen ist hier, ähnlich dem Grundgedanken des Architekturwissens in vielerlei Hinsicht, der Ansatz der Codification. Der hohe Aufwand der händischen Identifikation von Indikatoren hat gezeigt, dass ein individueller Ansatz auf lange Sicht keine umfangreichen Ergebnisse erzielen kann. Weitere Konzepte könnten eine Versionierung von Features sowie Technologie und deren Indikatoren einbeziehen. Ebenso sinnvoll ist in Folge eine tiefgreifende Analyse existierender Systeme unter Zuhilfenahme einer vollständigen Datenbasis. Nur durch diese kann die Qualität der Ergebnisse statistisch signifikant erwiesen werden.

Das Konzept stellt grundlegende Indikator-Typen und Regeln für die Programmiersprache Java vor. Eine Erweiterung dieses Konzeptes für weitere Sprachen und Dateiformate kann den Ansatz verfeinern und die Analyse weiterer Systeme ermöglichen.

Aufgrund der hohen Formalisierung von Technologie-Features sowie deren Indikatoren ist eine Erweiterung ohne weiteres möglich.

Die Folgen des Konzeptes für die Praxis der iterativen Softwareentwicklung konnten unter den Annahmen nicht evaluiert werden. Studien können, eine entsprechende Menge praxisrelevanten Architekturwissens vorausgesetzt, deren Qualität auswerten. Diese sind schätzungsweise auf Grundlage realer Anwendungsfälle denkbar. Die Auswertung der Ergebnisse durch Softwarearchitekten sollte deren tatsächliche Qualität zeigen.

Kapitel 7

Fazit

Das vorangegangene Konzept zeigt, dass eine Formalisierung und Extraktion von Technologie-Features durchaus sinnvoll möglich ist. Die exemplarische Betrachtung der Sprache Java liefert einen grundsätzlichen Ansatz zur Extraktion dieser anhand syntaktischer Merkmale und Sprachkonstrukte. Technisch bildet das Konzept in Form von Entwurfsentscheidungen eine Schnittstelle zwischen konkreter Programmierung (Implementation) und Softwarearchitektur eines existierenden Systems.

Der implementierte Prototyp zeigt, dass die strikte Syntax von Programmiersprachen und Konfigurationsdateien eine hohe Genauigkeit der Identifikation ermöglicht. Wenngleich der Ansatz im genannten Anwendungsfall sehr speziell betrachtet wird, liegt seine Erweiterbarkeit auf der Hand: Sprachen und Formate mit Informationsgehalt zwecks Bestimmung lassen sich ähnlich identifizieren und formalisieren.

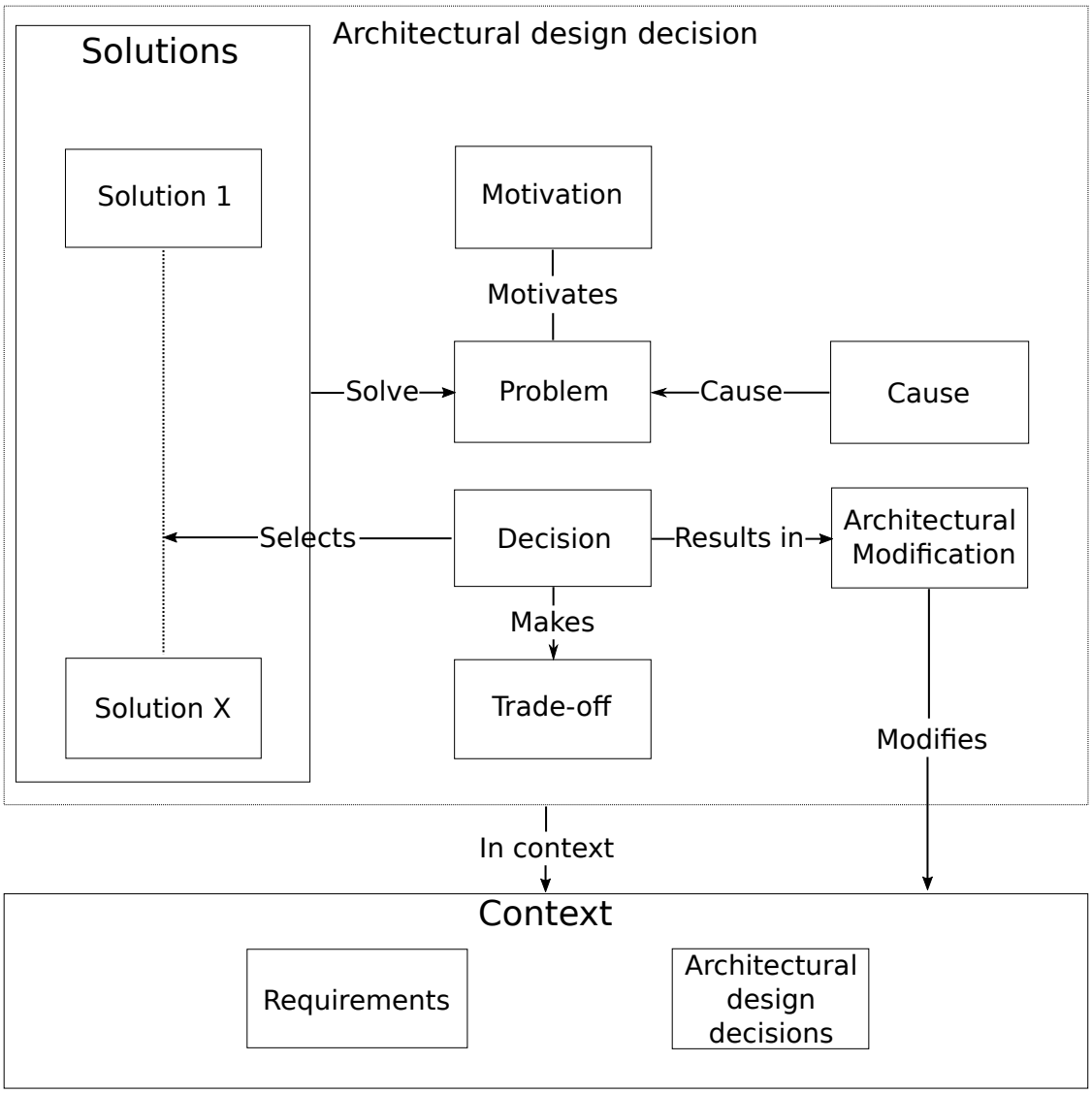
Einbindungen in etablierte Ansätze der Softwarearchitektur erfordern jedoch eine semantisch sinnvolle Menge an strukturiertem Architekturwissen. Aktuelle Ansätze der Wissenschaft betrachten zwar Möglichkeiten zur Schaffung dieser, eine konkrete Implementation einer Wissensbasis ist jedoch Grundlage für die Praxistauglichkeit des Konzeptes. Das ausgearbeitete Modell hängt in seiner Gesamtheit jedoch von zukünftiger Forschung ab.

Nichtsdestotrotz können Erkenntnisse im Prozess der Entscheidungsfindung eine Unterstützung für Softwarearchitekten bilden. Sie können den Lösungsraum der Entwurfsentscheidungen sinnvoll eingrenzen.

Durch das Einbeziehen zweier Abstraktionsebenen (Implementation und Entscheidungsfindung) kann eine semiformale Berücksichtigung von Technologie-Features helfen, den Lösungsraum von Entwurfsentscheidungen und der Alternativenfindung zu unterstützten.
Ob der Prozess der Entscheidungsfindung und damit der Softwarearchitektur durch diese Erkenntnisse sinnvoll unterstützt werden kann, bleibt unter den genannten Annahmen jedoch offen.

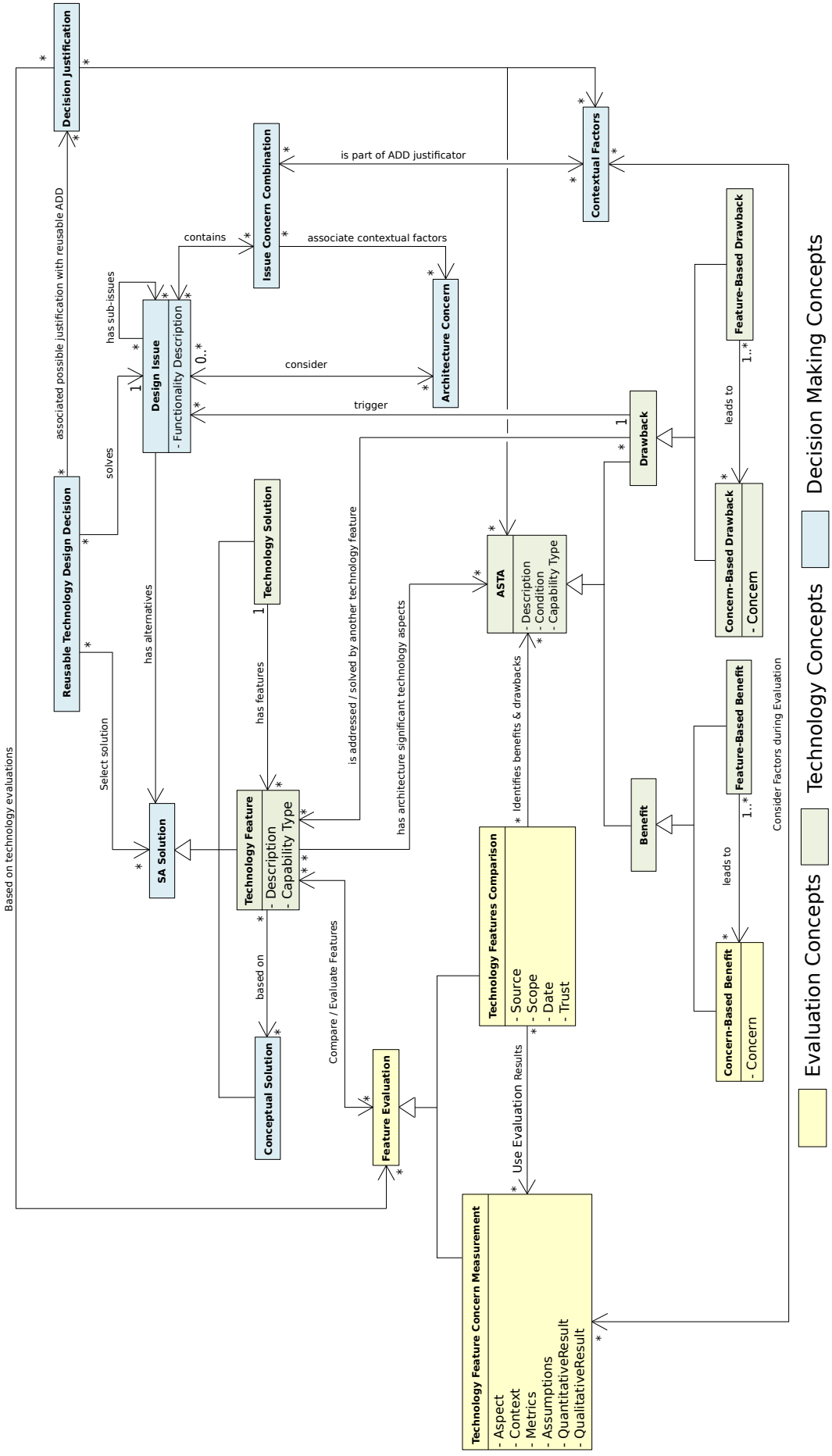
Anhang A

Zusätzliche Abbildungen



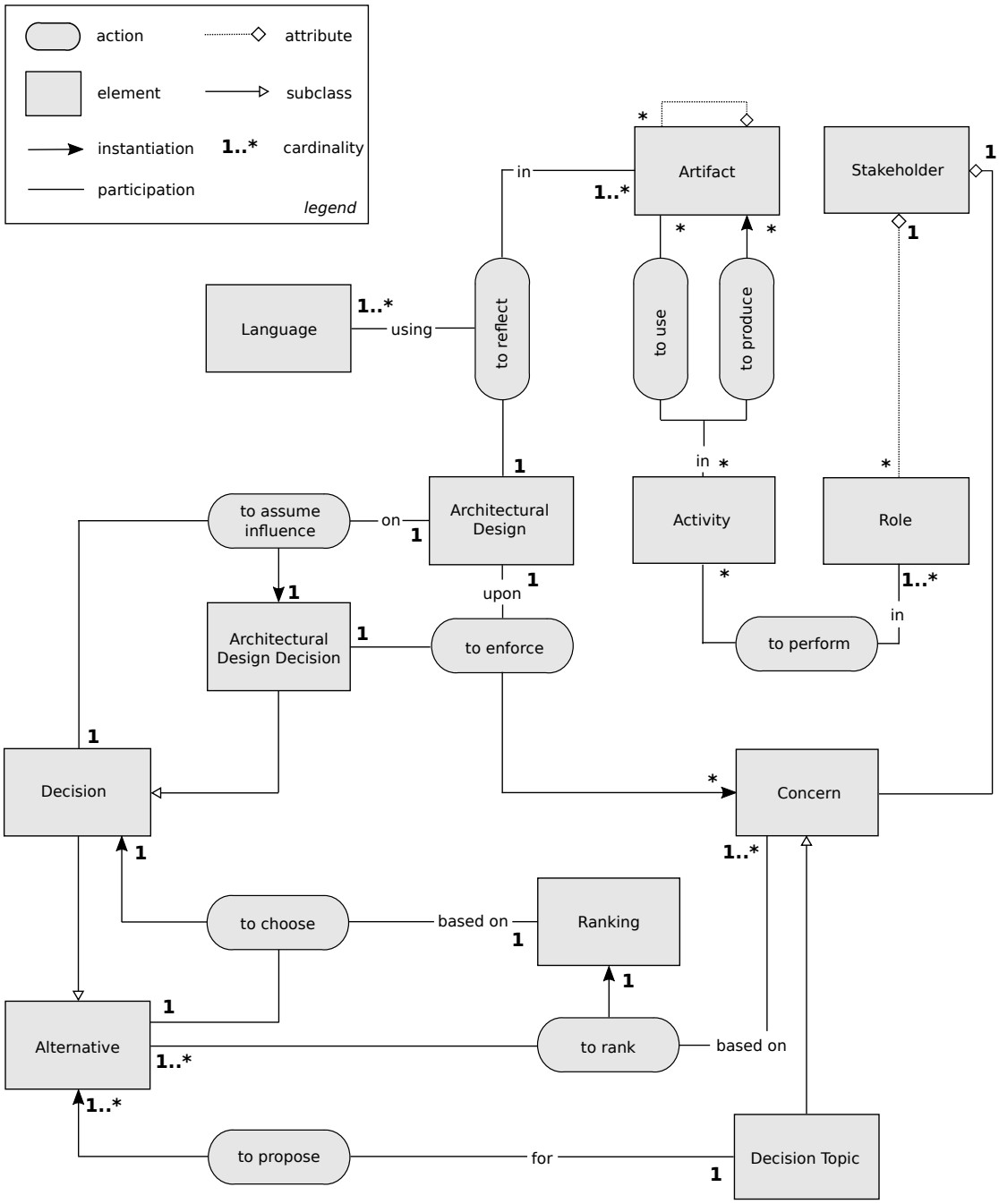
Quelle: [Jansen u. Bosch, 2005]

Abbildung A.1: Meta-Model zur Modellierung von Entwurfsentscheidungen



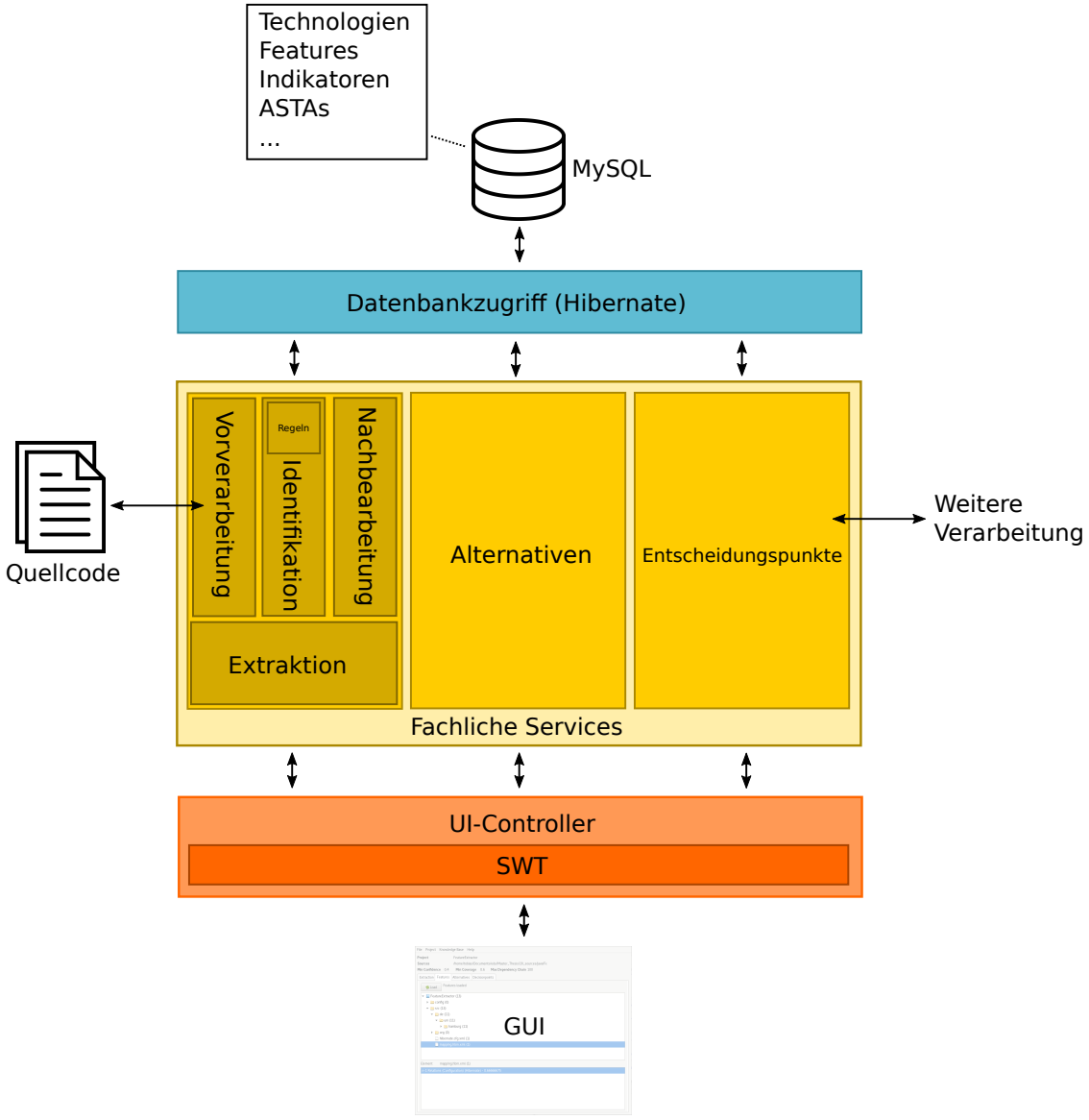
Quelle: [Soliman u. a., 2015]

Abbildung A.2: Meta-Model Architekturwissen



Quelle: [de Boer u. a., 2007]

Abbildung A.3: Grundlegendes Metamodell Architekturwissen



Grafik erstellt durch Verfasser

Abbildung A.4: Architektur des *FeatureExtractor*

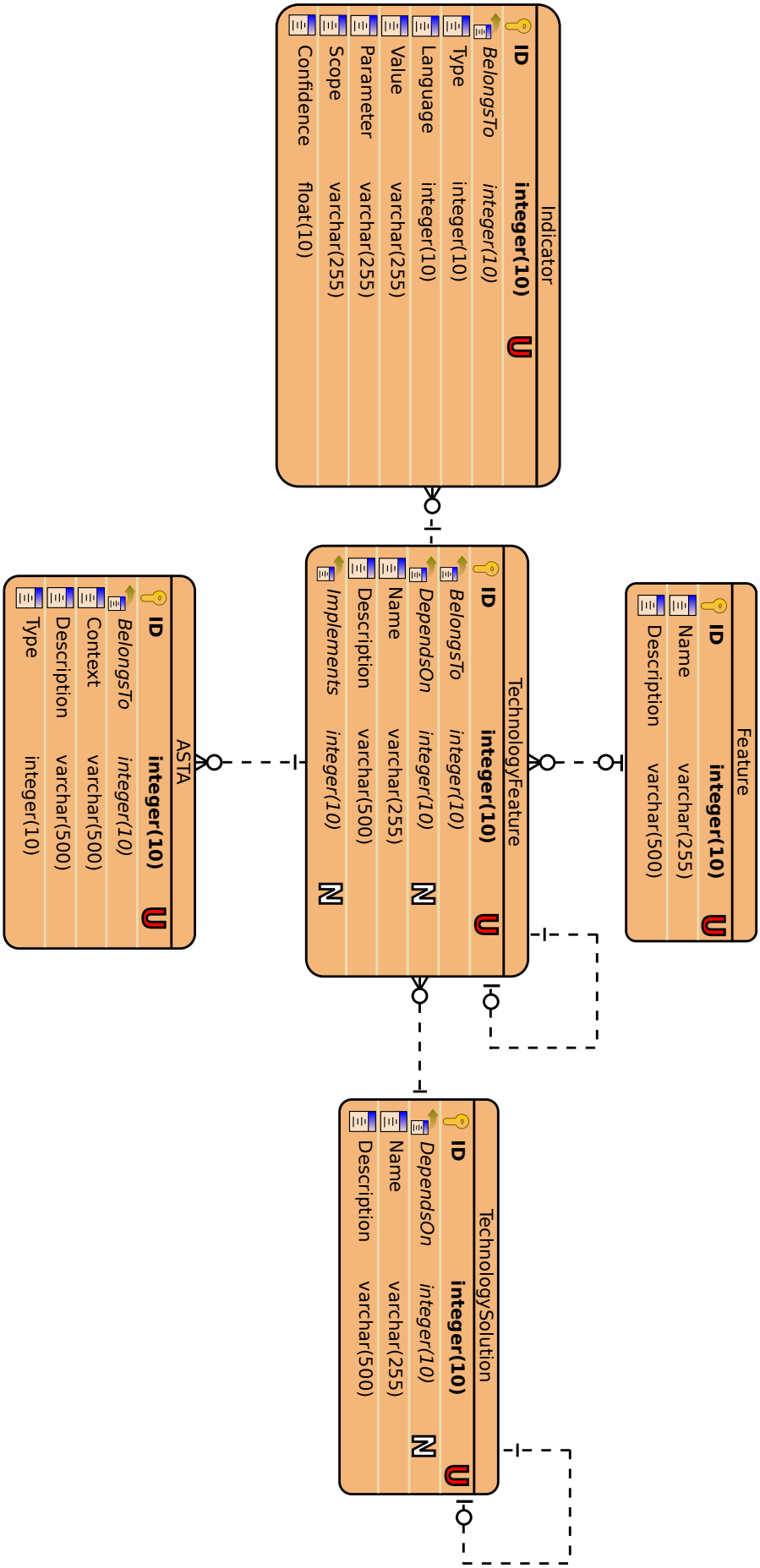


Abbildung A.5: Datenmodell *FeatureExtractor*

***FeatureExtractor* (ca. 10500 LOC)**

Min. Konfidenz	Instanzen	Dateien	Features	TP	FP	FN	Precision	Recall
0.00	13	135	6	13	0	0	1.000	1.000
0.25	13	135	6	13	0	0	1.000	1.000
0.50	13	135	6	13	0	0	1.000	1.000
0.75	4	135	2	4	0	9	1.000	0.308
1.00	0	135	0	0	0	13	0.000	0.000

***DecisionBuddy* (ca. 41400 LOC)**

Min. Konfidenz	Instanzen	Dateien	Features	TP	FP	FN	Precision	Recall
0.00	469	382	50	386	83	0	0.823	1.000
0.25	469	382	50	386	83	0	0.823	1.000
0.27	401	382	42	386	15	0	0.962	1.000
0.48	351	382	39	351	0	35	1.000	0.912
0.50	351	382	39	351	0	35	1.000	0.912
0.75	141	382	26	141	0	245	1.000	0.365
1.00	0	382	0	0	0	386	0.000	0.000

***BroadLeaf* (Programmpaket Common) (ca. 130000 LOC)**

Min. Konfidenz	Instanzen	Dateien	Features	TP	FP	FN	Precision	Recall
0.00	376	735	33	239	137	0	0.636	1.000
0.25	376	735	33	239	137	0	0.636	1.000
0.27	296	735	30	239	57	0	0.806	1.000
0.50	236	735	27	236	0	3	1.000	0.986
0.75	66	735	11	66	0	173	1.000	0.275
1.00	0	735	0	0	0	235	0.000	0.000

***Plazma* (ca. 132200 LOC)**

Min. Konfidenz	Instanzen	Dateien	Features	TP	FP	FN	Precision	Recall
0.00	403	3593	9	106	296	0	0.264	1.000
0.25	403	3593	9	106	296	0	0.264	1.000
0.50	103	3593	8	103	0	3	1.000	0.972
0.75	103	3593	8	103	0	3	1.000	0.972
1.00	0	3593	0	0	0	235	0.000	0.000

***PowerStone* (ca. 1300 LOC)**

Min. Konfidenz	Instanzen	Dateien	Features	TP	FP	FN	Precision	Recall
0.00	28	35	9	4	24	0	0.143	1.000
0.25	28	35	9	4	24	0	0.143	1.000
0.50	4	35	3	4	0	0	1.000	1.000
0.75	2	35	2	2	0	2	1.000	0.500
1.00	0	35	0	0	0	4	0.000	0.000

Tabelle A.1: Evaluation der Extraktion anhand unterschiedlicher Systeme

Grundlage der Analyse bilden die Quellcodedateien der jeweiligen Systeme in der zum Zeitpunkt der Analyse (Mitte August 2016) aktuellen Version. Die finale Version des *FeatureExtractor* (Version 1.0) wurde verwendet. Als Grundlage für die Analyse des *DecisionBuddy* dient die letzte Version des Masterprojekts aus dem Wintersemester 2015/2016, erweitert um die Nutzung einiger zusätzlicher Technologie-Features.

Technologie	**Technologie-Feature**	**Kurzbeschreibung**
JPA	Entity Manager	Verwaltung von Entitäten
	Associations	Beziehungen zwischen Entitäten
	m-n Associations	m-n Beziehungen
	n-1 Associations	n-1 Beziehungen
	Mapping Enities	Zuordnung von Datenbanktabellen zu Java-Objekten
	Entities	Definition von Entitäten
	Mapping Inheritance	Vererbung von Entitäten
	Relations Fetching	Optimierung für das Laden von Entitäten
	Eager Relations Fetching	Sofortiges Laden von Relationen
	Lazy Relations Fetching	Opportunistisches Laden von Relationen
	...	
Hibernate	Batch Reading	Lesen von mehreren Objekten pro Anfrage
	Sessions	Datenbank-Sessions
	Entity Locking	Locken von Entitäten
	Database Support	Unterstützt Datenbanken
	MySQL-Support	Unterstützt MySQL
	Persistent Bag	Optimierte Listenform
	Collection Fetching	Optimierung für das Laden von Listen
	Lazy Collection Fetching	Opportunistisches Laden
	Extra-Lazy Collection Fetching	Sehr opportunistisches Laden
	Immediate Collection Fetching	Sofortiges Laden
	...	
EclipseLink	Batch Fetching	Lesen von mehreren Objekten pro Anfrage
	EclipseLink Sessions (ELUG)	Datenbank-Sessions
	EclipseLink Transactions (ELUG)	Datenbank-Transaktionen
	Database Support	Unterstützt Datenbanken
	MySQL-Support	Unterstützt MySQL
	...	
TopLink	Batch Reading	Lesen von mehreren Objekten pro Anfrage
	Sessions	Datenbank-Sessions
	Transactions	Datenbank-Transaktionen
	Entity Locking	Locken von Entitäten
	Database Support	Unterstützt Datenbanken
	MySQL-Support	Unterstützt MySQL
	...	
...	...	

Tabelle A.2: Exemplarische Wissensbasis: Technologien und Features (Auszug)

Eingerückte Features stellen abhängige Funktionalitäten dar.
Die vollständige Wissensbasis sowie die Beschreibung der Technologien sowie Features findet sich in Form eines Datenbankabbildes auf dem beigefügten Datenträger. Dies beinhaltet ebenso die zugeordneten Indikatoren.

Anhang B

Abbildungsverzeichnis

Abbildungen 4.2, 4.5, 4.7 sowie A.5 wurden mit Visual Paradigm 12.1 (Akademische Lizenz, Universität Hamburg) erstellt

Weitere Abbildungen nach Fig. 3.1, Seiten 32, 39, 50, 75 sowie 81 basieren auf Grafiken von Madebyoliver via www.flaticon.com unter Creative Commons teCC 3.0 BY

Anhang C

Tabellenverzeichnis

Anhang D

Quellcodeverzeichnis

Anhang E

Literaturverzeichnis

[Baader 2003] Baader, Franz: *The description logic handbook: Theory, implementation and applications.* Cambridge University Press, 2003

[Babar u. a. 2007] Babar, M. A. ; de Boer, Remco C. ; Dingsoyr, T. ; Farenhorst, Rik: Architectural Knowlege Management Strategies: Approaches in Research and Industry. In: *Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent, 2007. SHARK/ADI '07: ICSE Workshops 2007. Second Workshop on*, 2007, S. 2–2

[Babar u. Gorton 2007] Babar, M. A. ; Gorton, I.: A Tool for Managing Software Architecture Knowledge. In: *Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent, 2007. SHARK/ADI '07: ICSE Workshops 2007. Second Workshop on*, 2007, S. 11–11

[Babar u. a. 2009] Babar, Muhammad A. ; Dingsøyr, Torgeir ; Lago, Patricia ; Vliet, Hans van: *Software architecture knowledge management.* Springer, 2009

[Bass u. a. 2012] Bass, Len ; Clements, Paul ; Kazman, Rick: *Software Architecture in Practice.* 3. Auflage. Addison-Wesley Professional, 2012

[v. d. Berg u. a. 2009] Berg, M. v. d. ; Tang, A. ; Farenhorst, R.: A Constraint-Oriented Approach to Software Architecture Design. In: *2009 Ninth International Conference on Quality Software*, 2009, S. 396–405

[Biggerstaff 1989] Biggerstaff, T. J.: Design recovery for maintenance and reuse. In: *Computer* 22 (1989), July, Nr. 7, S. 36–49

[Chikofsky u. Cross 1990] Chikofsky, E. J. ; Cross, J. H.: Reverse engineering and design recovery: a taxonomy. In: *IEEE Software* 7 (1990), Jan, Nr. 1, S. 13–17

[Clements u. a. 2002] Clements, Paul ; Garlan, David ; Bass, Len ; Stafford, Judith ; Nord, Robert ; Ivers, James ; Little, Reed: *Documenting software architectures: views and beyond.* Pearson Education, 2002

[Clements 1996] Clements, PC: *Succeedings of the Constraints Subgroup of the EDCS Architecture and Generation Cluster.* 1996

[Clerc u. a. 2007] Clerc, Viktor ; Lago, Patricia ; Vliet, Hans van: The Architect's Mindset. In: *Proceedings of the Quality of Software Architectures 3rd International*

Conference on Software Architectures, Components, and Applications. Berlin, Heidelberg : Springer-Verlag, 2007 (QoSA'07), S. 231–249

[Coffey u. a. 2010] Coffey, J. ; White, L. ; Wilde, N. ; Simmons, S.: Locating Software Features in a SOA Composite Application. In: *Web Services (ECOWS), 2010 IEEE 8th European Conference on*, 2010, S. 99–106

[de Boer u. a. 2007] de Boer, Remco C. ; Farenhorst, Rik ; Lago, Patricia ; Vliet, Hans van ; Clerc, Viktor ; Jansen, Anton: Architectural Knowledge: Getting to the Core. In: *Proceedings of the Quality of Software Architectures 3rd International Conference on Software Architectures, Components, and Applications*. Berlin, Heidelberg : Springer-Verlag, 2007 (QoSA'07), S. 197–214

[Dit u. a. 2013] Dit, Bogdan ; Revelle, Meghan ; Gethers, Malcom ; Poshyvanyk, Denys: Feature location in source code: a taxonomy and survey. In: *Journal of Software: Evolution and Process* 25 (2013), Nr. 1, S. 53–95

[Falessi 2007] Falessi, Davide: *A Toolbox for Software Architecture Design*, UniversitÃ degli Studi di Roma Tor Vergata, Dissertation, 2007

[Farenhorst 2006] Farenhorst, Rik: Tailoring Knowledge Sharing to the Architecting Process. In: *SIGSOFT Softw. Eng. Notes* 31 (2006), September, Nr. 5. – ISSN 0163–5948

[Farenhorst u. de Boer 2006] Farenhorst, Rik ; Boer, Remco C.: Core Concepts of an Ontology of Architectural Design Decisions / Vrije Universiteit. 2006 (IR-IMSE-002). – Forschungsbericht

[Farenhorst u. de Boer 2009] Farenhorst, Rik ; Boer, Remco C.: Knowledge management in software architecture: State of the art. In: *Software Architecture Knowledge Management*. Springer, 2009, S. 21–38

[Gamma u. a. 1995] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995

[Garlan u. Shaw 1994] Garlan, David ; Shaw, Mary: An Introduction to Software Architecture. Pittsburgh, PA, USA : Carnegie Mellon University, 1994. – Forschungsbericht

[Gerdes u. a. 2014] Gerdes, Sebastian ; Lehnert, Steffen ; Riebisch, Matthias: Combining architectural design decisions and legacy system evolution. In: *Proceedings of the 8th European Conference on Software Architecture*. Springer, 2014, S. 50–57

[Gerdes u. a. 2015] Gerdes, Sebastian ; Soliman, Mohamed ; Riebisch, Matthias: Decision Buddy: Tool Support for Constraint-Based Design Decisions During System Evolution. In: *Proceedings of the 1st International Workshop on Future of Software Architecture Design Assistants*, ACM, 2015 (FoSADA '15), S. 13–18

[Gorton u. a. 2015] Gorton, I. ; Klein, J. ; Nurgaliev, A.: Architecture Knowledge for Evaluating Scalable Databases. In: *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, 2015, S. 95–104

[Gueheneuc u. a. 2006] Gueheneuc, Y. G. ; Mens, K. ; Wuyts, R.: A comparative framework for design recovery tools. In: *Conference on Software Maintenance and Reengineering (CSMR'06)*, 2006, S. 10 pp.–134

[Hansen u. a. 1999] Hansen, Morten T. ; Nohria, Nitin ; Tierney, Thomas: What's your strategy for managing knowledge? In: *The Knowledge Management Yearbook 2000–2001* (1999)

[Harris u. a. 1995] Harris, D. R. ; Reubenstein, H. B. ; Yeh, A. S.: Reverse Engineering to the Architectural Level. In: *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, 1995, S. 186–186

[Hofmeister u. a. 2000] Hofmeister, Christine ; Nord, Robert ; Soni, Dilip: *Applied Software Architecture*. 1. Auflage. Addison-Wesley Professional, 2000

[IEEE 2000] IEEE: *Standard zur Architekturbeschreibung von Software-Systemen (ANSI/IEEE 1471-2000)*. 2000. – `http://www.iso-architecture.org/ieee-1471/` zuletzt aufgerufen am 18. Oktober 2016

[ISO/IEC 2011] ISO/IEC: *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models (ISO/IEC 25010:2011)*. 2011. – `http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733` zuletzt aufgerufen am 18. Oktober 2016

[Jansen u. Bosch 2004] Jansen, A. ; Bosch, J.: Evaluation of tool support for architectural evolution. In: *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, 2004. – ISSN 1938–4300, S. 375–378

[Jansen u. a. 2007] Jansen, A. ; Ven, J. V. D. ; Avgeriou, P. ; Hammer, D. K.: Tool Support for Architectural Decisions. In: *Software Architecture, 2007. WICSA '07. The Working IEEE/IFIP Conference on*, 2007, S. 4–4

[Jansen u. Bosch 2005] Jansen, Anton ; Bosch, Jan: Software Architecture As a Set of Architectural Design Decisions. In: *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, IEEE Computer Society, 2005 (WICSA '05), S. 109–120

[Jordan u. a. 2015] Jordan, H. ; Rosik, J. ; Herold, S. ; Botterweck, G. ; Buckley, J.: Manually Locating Features in Industrial Source Code: The Search Actions of Software Nomads. In: *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*, 2015, S. 174–177

[Kazman u. a. 2000] Kazman, Rick ; Klein, Mark ; Clements, Paul: ATAM: Method for architecture evaluation / DTIC Document. 2000. – Forschungsbericht

[Kerievsky 2005] Kerievsky, Joshua: *Refactoring to patterns*. Pearson Deutschland GmbH, 2005

[Kramer u. Prechelt 1996] Kramer, C. ; Prechelt, L.: Design recovery by automated search for structural design patterns in object-oriented software. In: *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*, 1996, S. 208–215

[Kruchten 1995] Kruchten, Philippe: The 4+1 View Model of Architecture. In: *IEEE Softw.* 12 (1995), November, Nr. 6, S. 42–50

[Kruchten 2004] Kruchten, Philippe: An ontology of architectural design decisions in software intensive systems. In: *2nd Groningen workshop on software variability* Citeseer, 2004, S. 54–61

[LaToza u. a. 2013] LaToza, T. D. ; Shabani, E. ; Hoek, A. van d.: A study of architectural decision practices. In: *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*, 2013, S. 77–80

[Liang u. Avgeriou 2009] Liang, Peng ; Avgeriou, Paris: Tools and technologies for architecture knowledge management. In: *Software Architecture Knowledge Management*. Springer, 2009, S. 91–111

[Lukoit u. a. 2000] Lukoit, K. ; Wilde, N. ; Stowell, S. ; Hennessey, T.: TraceGraph: immediate visual location of software features. In: *Software Maintenance, 2000. Proceedings. International Conference on*, 2000, S. 33–39

[Mancoridis u. a. 1999] Mancoridis, S. ; Mitchell, B. S. ; Chen, Y. ; Gansner, E. R.: Bunch: a clustering tool for the recovery and maintenance of software system structures. In: *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, 1999, S. 50–59

[McMillan u. a. 2009] McMillan, Collin ; Poshyvanyk, Denys ; Revelle, Meghan: Combining Textual and Structural Analysis of Software Artifacts for Traceability Link Recovery. In: *Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, IEEE Computer Society, 2009 (TEFSE '09), S. 41–48

[Medvidovic u. Taylor 2000] Medvidovic, N. ; Taylor, R. N.: A classification and comparison framework for software architecture description languages. In: *IEEE Transactions on Software Engineering* 26 (2000), Jan, Nr. 1, S. 70–93

[Mirakhorli u. a. 2014] Mirakhorli, Mehdi ; Fakhry, Ahmed ; Grechko, Artem ; Wieloch, Matteusz ; Cleland-Huang, Jane: Archie: A Tool for Detecting, Monitoring, and Preserving Architecturally Significant Code. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA : ACM, 2014 (FSE 2014). – ISBN 978–1–4503–3056–5, S. 739–742

[Morris 2012] Morris, Johny: *Practical Data Migration*. 1. Auflage. British Informatics Society Ltd., 2012

[Nie u. Zhang 2012] Nie, K. ; Zhang, L.: Software Feature Location Based on Topic Models. In: *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific* Bd. 1, 2012, S. 547–552

[Nowak u. a. 2010] Nowak, Marcin ; Pautasso, Cesare ; Zimmermann, Olaf: Architectural Decision Modeling with Reuse: Challenges and Opportunities. In: *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge*. New York, NY, USA : ACM, 2010 (SHARK '10), S. 13–20

[Pashov u. Riebisch 2004] Pashov, I. ; Riebisch, M.: Using feature modeling for program comprehension and software architecture recovery. In: *Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop on the*, 2004, S. 406–417

[Perry u. Grisham 2006] Perry, Dewayne E. ; Grisham, Paul S.: Architecture and design intent in component & COTS based systems. In: *Commercial-off-the-Shelf (COTS)-Based Software Systems, 2006. Fifth International Conference on* IEEE, 2006, S. 10–pp

[Perry u. Wolf 1992] Perry, Dewayne E. ; Wolf, Alexander L.: Foundations for the Study of Software Architecture. In: *SIGSOFT Softw. Eng. Notes* 17 (1992), Oktober, Nr. 4, S. 40–52

[Posch u. a. 2007] Posch, Torsten ; Birken, Klaus ; Gerdom, Michael: *Basiswissen Softwarearchitektur: verstehen, entwerfen, wiederverwenden*. 2. Auflage. dpunkt-Verlag, 2007

[Powers 2007] Powers, D: Evaluation: From Precision, Recall and F Factor to ROC, Informedness, Markedness & Correaltion. In: *Sch. Informatics Eng. Flinders* (2007)

[Ran u. Kuusela 1996] Ran, Alexander ; Kuusela, Juha: Design Decision Trees. In: *Proceedings of the 8th International Workshop on Software Specification and Design*, IEEE Computer Society, 1996 (IWSSD '96), S. 172–

[Regli u. a. 2000] Regli, William C. ; Hu, Xiaochun ; Atwood, Michael ; Sun, Wei: A survey of design rationale systems: approaches, representation, capture and retrieval. In: *Engineering with computers* 16 (2000), Nr. 3-4, S. 209–235

[Rubin u. Chechik 2013] Rubin, Julia ; Chechik, Marsha: A survey of feature location techniques. In: *Domain Engineering*. Springer, 2013, S. 29–58

[Seemann u. von Gudenberg 1998] Seemann, Jochen ; Gudenberg, Jürgen W.: Pattern-based Design Recovery of Java Software. In: *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 1998 (SIGSOFT '98/FSE-6), S. 10–16

[Shaw u. Clements 2006] Shaw, M. ; Clements, P.: The golden age of software architecture. In: *IEEE Software* 23 (2006), März, Nr. 2, S. 31–39

[Soliman u. a. 2016] Soliman, Mohamed ; Galster, Matthias ; Salama, Amr R. ; Riebisch, Matthias: *Architectural Knowledge for Technology Decisions in Developer Communities*. 2016

[Soliman u. a. 2015] Soliman, Mohamed ; Riebisch, Matthias ; Zdun, Uwe: Enriching Architecture Knowledge with Technology Design Decisions. In: *12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2015, S. 135–144

[Stachowiak 1973] Stachowiak, Herbert: *Allgemeine Modelltheorie*. Springer-Verlag, Wien, 1973

[Tang u. van Vliet 2009] Tang, A. ; Vliet, H. van: Modeling constraints improves software architecture design reasoning. In: *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, 2009, S. 253–256

[Taylor u. a. 2009] Taylor, Richard N. ; Medvidovic, Nenad ; Dashofy, Eric M.: *Software architecture: foundations, theory, and practice.* Wiley Publishing, 2009

[Tyree u. Akerman 2005] Tyree, J. ; Akerman, A.: Architecture decisions: demystifying architecture. In: *IEEE Software* 22 (2005), March, Nr. 2, S. 19–27

[van der Ven u. a. 2006] van der Ven, Jan S. ; Jansen, Anton G. ; Nijhuis, Jos A. ; Bosch, Jan: Design decisions: The bridge between rationale and architecture. In: *Rationale management in software engineering.* Springer, 2006, S. 329–348

[van Gurp u. Bosch 2002] van Gurp, Jilles ; Bosch, Jan: Design Erosion: Problems and Causes. In: *J. Syst. Softw.* 61 (2002), Nr. 2, S. 105–119. – ISSN 0164–1212

[Wang u. a. 2011] Wang, J. ; Peng, X. ; Xing, Z. ; Zhao, W.: An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In: *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, 2011, S. 213–222

[Ziftci u. Krüger 2012] Ziftci, C. ; Krüger, I.: Feature Location Using Data Mining on Existing Test-Cases. In: *Reverse Engineering (WCRE), 2012 19th Working Conference on*, 2012, S. 155–164

[Zimmermann u. a. 2009] Zimmermann, Olaf ; Koehler, Jana ; Leymann, Frank ; Polley, Ronny ; Schuster, Nelly: Managing architectural decision models with dependency relations, integrity constraints, and production rules. In: *Journal of Systems and Software* 82 (2009), Nr. 8, S. 1249 – 1267

Anhang F

Glossar

.NET Framework der Firma Microsoft zur Entwicklung von Windows-Applikationen, Siehe www.microsoft.com, zuletzt aufgerufen am 18. Oktober 2016 XX, XXII, XXIII, 42, 76

Abdeckungsrate Gibt den Grad an, zu dem zwei Technologien verwandte Funktionalitäten implementieren. Lässt sich statisch oder anhand einer Menge tatsächlich genutzter Features bestimmen IX, 71–73, 77, 81, 82, 92

Alleinstellungsmerkmal Merkmal oder Feature einer Technologie, dass in seiner Ausprägung einzigartig ist und einen potentiellen Kunden vom Nutzen des Produktes überzeugen soll 42, 43

Annotation Syntaktische Anweisung für Compiler oder Präprozessor. Wird meist verwendet, um Quellcode oder zusätzliche Funktionalitäten einzubinden X, 45, 50, 56, 57, 60, 65, 66

Anwendungsfall Szenario zur Beschreibung der Betrachtung eines Anwendungssystems zwecks Erreichung eines bestimmten fachlichen Ziels IX, 2, 5, 7, 21, 22, 24, 25, 27–29, 41, 49, 56, 59, 60, 68, 78, 81–83, 87, 92, 93, 96, 97

Application Programming Interface Schnittstelle für die Anwendungsprogrammierung. Stellt Funktionalitäten einer Technologie anhand bestimmter syntaktischer Sprachkonstrukte bereit XXVI

Architecturally Significant Technology Aspect Aspekt eines Technologie-Features mit Einfluss auf die Softwarearchitektur eines Systems. Kann sich als Vor- oder Nachteil darstellen und ist an einen Anwendungskontext gebunden. Siehe [Soliman u. a., 2015] XXVI, 16, 28, 42

Architecture Trade-off Analysis Method Methode zur Auswertung von Softwarearchitekturen bezüglich deren Qualitätsattribute. Siehe [Kazman u. a., 2000] XXVI

Architekturwissen Auch *Architectural Knowledge*. Sammlung von Entwurfs- und Architekturmustern, Technologien sowie weiteren wiederverwendbaren Assets III, IV, IX, XX, XXIII, XXIV, 6, 10–14, 16, 19, 21, 25–28, 31, 33, 35–38, 40, 41, 44, 49–51, 53, 54, 60, 61, 65, 66, 74, 79, 81–83, 87–97

Assembler Programm zur Übersetzung des Quelltextes einer Anwendung in ein maschinenlesbares Format 45

Asset Wiederverwendbare Einheit des Architekturwissens. Besteht meist aus Technologien, Entwurfsmusters, Werkzeugen oder ähnlichem XIX, XX, 10–14, 16

Betriebssystem Verwaltet Hardware-Ressourcen und stellt eine Laufzeitumgebung für Anwendungssysteme bereit 11, 37

Binary Auch *Executable*. Maschinenlesbare Ausführung eines Anwendungssystems. Kann meist in einer Laufzeitumgebung ausgeführt werden 44

BroadLeaf Eine Enterprise eCommerce-Lösung. Siehe www.broadleafcommerce.com, zuletzt aufgerufen am 18. Oktober 2016 VII, X, 87, 88

C++ objektorientierte, plattformunabhängige Programmiersprache 45

C# objektorientierte Programmiersprache unter Verwendung des .NET-Frameworks XXIII, 84

Clustering Eine Ansammlung von Objekten mit ähnlichen Eigenschaften 19

Codification Strategie zum Aufbau von Architekturwissen. Ziel ist eine allgemeingültige und kontextunabhängige Darstellung des Architekturwissens 12, 41, 96

Commercial off-the-shelf Standardkomponente für den Einsatz in Softwaresystemen XXVI

Compiler Programm zur Übersetzung von Quellcode in eine maschinenlesbare Sprache. Nimmt zumeist eine Syntaxprüfung vor und stellt somit die Konsistenz des Quellcodes sicher XIX, 56, 57

Constraint Einschränkung des Lösungsraums. Grenzt die Auswahl möglicher Assets ein. Hängt häufig von Anforderungen, Technologien, Umfeld und vorherigen Entwurfsentscheidungen ab XXII, XXIV, 2, 6, 12–14, 16, 18, 19, 21, 28, 30, 42, 43, 58, 61, 64, 67, 72, 77, 81, 82, 92

Datenbankmanagementsystem System zur Steuerung und Verwaltung von Datenbank-Installationen XXI, XXIII

DecisionBuddy Softwaresystem zur Unterstützung von Softwarearchitekturentscheidungen im Zuge der Softwareevolution auf Basis von [Gerdes u. a., 2015] VII, IX, X, 21, 22, 24, 25, 27, 38, 40, 41, 49, 55–57, 67, 81, 83, 85–88, 92, 93

EclipseLink Implementation der JPA, siehe www.eclipse.org, zuletzt aufgerufen am 18. Oktober 2016 VIII, 42

Entscheidungspunkt Technologie, unterhalb derer im Abhängigkeitsbaum Entscheidungen berücksichtigt werden müssen. Relevant, wenn Technologien in Form von Blattknoten ausgetauscht werden sollen IX, XI, 5, 28, 74–78, 80, 81, 83, 85, 93–95

Entwicklungsumgebung Meist eine Menge von Tools zur Bearbeitung, Ausführung sowie Verwaltung von Quellcode. Bietet häufig umfangreiche Möglichkeiten für das Refactoring 50, 51, 81

Entwurfsentscheidung (Explizite) Dokumentation einer Entscheidung im Entwurfsprozess. Diese kann unter anderen die Auswahl von Architektur- oder Entwurfsmustern sowie die Auswahl von Technologien beinhalten. Ist im Optimalfall mit der zugrundeliegenden Begründung im Prozess zu dokumentieren II, IX, XX, XXII, XXIV, XXV, 6, 8–17, 19, 24, 28–31, 34, 52, 70, 71, 74, 77, 79, 82, 92, 97

Entwurfsmuster Abstrahierte Lösung für ein wiederkehrendes Problem der Softwareentwicklung. Siehe [Gamma u. a., 1995] XX, XXV, 9, 11, 20, 46, 91

Extensible Markup Language Erweiterbare, maschienen- sowie menschenlesbare Auszeichungssprache XXVI

Feature Eigenschaft einer Software-Lösung. Bietet eine bestimmte Funktionalität zur Umsetzung von Anforderungen an. Unterscheidet sich in seiner Granularität von Technologie-Features XXI, 17–20, 28

Feature Location Ansatz des Reverse Engineering um den Ursprung der Umsetzung funktionaler Features im Quellcode einer Anwendung zu bestimmen. Meist genutzt, um existierende Funktionalitäten anzupassen 19, 20

Framework Abstrakte Implementation einer Softwarekomponente, welche durch individuelle Implementation an eigene Bedürfnisse angepasst werden kann 11, 13, 16, 18, 76, 88

Gespeicherte Prozedur Eine Folge von Anweisungen, welche direkt im Datenbankmanagementsystem ausgeführt wird 35

Greenfield-Szenario Auch *Neuentwicklung*. Entwicklung eines Softwaresystems ohne Berücksichtigung vorhandener Systeme 9

Hibernate Populäres ORM-Framework für die Programmiersprache Java. Siehe www.hibernate.org, zuletzt aufgerufen am 18. Oktober 2016 VIII, IX, XI, XXIII, 21–24, 27, 34, 38, 42, 49–51, 56, 57, 59, 60, 69, 70, 72, 83, 84, 86, 87

Imperative Programmiersprache Programmiermodell. Führt einzelne Anweisungen nacheinander aus 45

Import Explizite Inklusion eines Programmpakets. Dient in seiner Grundform vorrangig der besseren Lesbarkeit von Quellcode, im Gegensatz zur vollqualifizierten Nutzung von Konstrukten aus anderen Namensräumen X, XI, 45, 49–53, 57, 60, 62, 64–67, 87, 88

Indikator Charakteristisches Merkmal eines Technologie-Features. Weist auf Grundlage von Syntax und Sprachkonstrukten einer Programmiersprache oder ähnlichem auf dessen Nutzung im Rahmen eines existierenden Softwaresystems hin VIII, X, 18, 19, 27, 28, 35, 37–40, 46, 48–55, 57–67, 82–84, 86, 90, 93, 95, 96

Institute of Electrical and Electronics Engineers Organisationsverband, hauptsächlich bestehend aus Informatikern und Elektrotechnikern. Siehe www.ieee.org, zuletzt aufgerufen am 18. Oktober 2016 XXVI

Interface Schnittstelle zwischen Systemkomponenten. Im Kontext der objektorientierten Programmierung eine Schnittstellendefinition, die von Klassen implementiert werden kann XI, 51, 54, 55, 62, 83

Intermediate Language Objektorientierte Programmiersprache als Konnektor zwischen statischer und dynamischer Kompilierung der .NET-Sprachfamilien 44

Inversion of Control Auch: *Dependency Injection*. Umkehrung des Kontrollflusses 18, 53

Issue Problem einer Softwarearchitektur, dass meist durch Entwurfsentscheidungen adressiert werden kann. Siehe [Zimmermann u. a., 2009] XXII, 15, 21

Iterator Objekt zur Traversierung von Datenstrukturen 76

Java objektorientierte, plattformunabhängige Programmiersprache VIII, XI, XXI, XXIII, 14, 21, 37, 40, 44–46, 49–52, 54–57, 59–63, 70, 75, 83, 96, 97

Java Persistence API Spezifikation für Persistierung von Datenbankobjekten. Siehe www.oracle.com, zuletzt aufgerufen am 18. Oktober 2016 XXVI, 23, 42, 70

JavaServer Pages Software-Framework zur Erstellung dynamischer Webseiten. Siehe www.oracle.com, zuletzt aufgerufen am 18. Oktober 2016 XXVI, 21

Klasse Konzept objektorientierter Programmiersprachen als abstraktes Modell oder Bauplan für eine Menge von Objekten XXII, XXIII, 49, 51–55, 63

Komponente Teil eines Softwaresystems. Wirkt mit anderen Komponenten zusammen XXII, 8, 68, 69

Konnektor Schnittstelle zwischen Komponenten 8

Lösungsraum Hier: Eine Menge potentieller Problemlösungen und Entwurfsentscheidungen zur Adressierung eines Issues. Muss durch Constraints und weitere Einschränkungen begrenzt werden, um sinnvolle Lösungen auswählen zu können XX, XXIV, 6, 10, 12–14, 18, 82, 91, 97

Laufzeitumgebung Umgebung, in der Computerprogramme aufgeführt werden. Stellt zumeist eine Abstraktionsschicht zwischen Hard- und Software dar XX, 56

Lines of Code Metrik der Softwarearchitektur. Gibt die Gesamtzahl an Zeilen im Quellcode eines Anwendungssystems an XXVI, 86

Linux Freies Mehrbenutzer-Betriebssystem 75

Makro Anweisung für einen Präprozessor. Ersetzt das Makro durch eine definierte Menge von Anweisungen XXIII, 45

Mapping Hier: Bidirektionale Assoziation zweier Datensätze im Sinne des ORM-Konzeptes 61

Metamodell Abbildung eines Gegenstandes oder Sachverhaltes. Entgegen der üblichen Definition nach [Stachowiak, 1973] hier allerdings auf einer höheren Abstraktionsebene IV, IX, 10, 11, 27, 33, 36, 44, 46, 47, 49, 68, 79–81, 93

Model View Controller Softwarearchitektur-Muster. Sieht eine Entkoppelung von Datenmodell (*Model*), Benutzeroberfläche (*View*) und Vermittlungsschicht (*Controller*) vor XXVI

MySQL Relationales Datenbankmanagementsystem. Siehe www.mysql.com, zuletzt aufgerufen am 18. Oktober 2016 21, 57, 82

NHibernate ORM-Implementation und .NET-Pendant zu *Hibernate*.
Siehe www.nhibernate.info, zuletzt aufgerufen am 18. Oktober 2016 42

Object-relational mapping Programmiermodell objektorientierter Sprachen. Stellt zumeist eine direkte Zuordnung zwischen Datenbank-Tabellen oder -Relationen und Klassen-Objekten der jeweiligen Sprache dar. Entsprechende Objekte können bearbeitet und in der Datenbank persistiert werden XXIII, XXIV, XXVI, 22, 23, 31, 42

Objektorientiert Methodik zur Beschreibung von Anwendungssystemen als Menge von Objekten als Abbildung von Funktionalitäten XX, XXII, 45

Personalization Strategie zum Aufbau von Architekturwissen. Ziel ist eine personalisierte und kontextabhängige Darstellung des Architekturwissens 12

Plazma Eine benutzerfreundliche Anwendung zur Ressourcenplanung und Pflege von Kundendaten für klein- und mittelständische Unternehmen.
Siehe www.plazma.sourceforge.net, zuletzt aufgerufen am 18. Oktober 2016 X, 87–89

Pointer Datenstruktur hardwarenaher Programmiersprachen. Zeigt auf eine Adresse im physischen Speicher und ermöglicht somit einen direkten Zugriff auf diesen 45

Polymorphie Konzept objektorientierter Programmiersprachen bei dem ein Bezeichner unterschiedliche Datentypen annehmen kann 53

PowerStone Ein Opensource Workflow Management System in Java. Siehe www.sourceforge.net, zuletzt aufgerufen am 18. Oktober 2016 X, 87, 89

Präprozessor Programm zur Vorverarbeitung von Quellcode vor der eigentlichen Kompilierung. Ersetzt meist Makros und ähnliche Anweisungen durch konkreten Quellcode XIX, XXII, 45

Precision In der statistischen Analyse ein Faktor zur Betrachtung einer Menge relevanter positiv bestimmter Datensätze 86, 89, 90

Programmpaket Abgeschlossene Einheit eines Programmes. Meist eine Menge funktional zusammengehörigen Quellcodes. Dient der Kapselung VII, XXI, 39, 45–47, 49–51, 57, 60, 62–64, 67, 69, 79, 82, 85, 87, 90, 91

Proxy Stellvertreter. In der Programmierung häufig verwendet, um Funktionalitäten zu kapseln. 56

Quellcode-Konvention Menge von Richtlinien zur Erstellung von Quellcode. Definiert zumeist Formatierung oder die Benennung von Bezeichnern explizit 64

Rationale Begründung, die einer Entwurfsentscheidung zugrunde liegt. Ist häufig auf Anforderungen zurückzuführen 9, 10

Recall In der statistischen Analyse ein Faktor zur Betrachtung der anteiligen Menge relevanter Datensätze 86, 89, 90

Refactoring Neustrukturierung vorhandenen Quellcodes. Siehe auch [Kerievsky, 2005] XXI, 11, 50, 69, 83

Reverse Engineering Dekonstruktion eines Softwareproduktes im Hinblick auf dessen Funktionsweise und Struktur. Meist mit dem Ziel einer Neuentwicklung oder Anpassung XXI, 6, 19, 85, 91, 94

Tacit Knowledge Eine Art von Wissen, die mündlich oder schriftlich nur schwer zu übermitteln ist und deshalb häufig in Vergessenheit gerät. Siehe [van der Ven u. a., 2006] 17, 91

Skriptsprache Programmiermodell. Interpretiert Quellcode zur Laufzeit, anstatt die Anwendung statisch in maschinenlesbare Befehle zu übersetzen 44

Software-Erosion Verfall eines Software-Systems. Kommt häufig aufgrund fehlender Wartung zu Stande. Folgen sind etwa abnehmende Leistung oder auftretende Fehler. Siehe [van Gurp u. Bosch, 2002] 10

Softwarearchitekt Entwirft Softwaresysteme und stellt Vorgaben für deren Umsetzung bereit. Ebenso fällt das Treffen von Entwurfsentscheidungen in den Aufgabenbereich des Architekten. Das Architekturwissen stellt eine Art Werkzeugkasten des Architekten bereit und bildet dessen Lösungsraum ab. Dieser muss anhand von Constraints eingeschränkt werden XXV, 9, 11, 21, 28, 96, 97

Softwarearchitektur Nach [Jansen u. Bosch, 2005] eine Menge von Designentscheidungen, welche im Bezug auf die Struktur und Implementation eines Softwaresystems getroffen wurden XIX, XXII, XXV, 2, 6–11, 19, 24, 30, 40, 51, 67, 68, 79, 80, 91–94, 96, 97

Sormula Simple Object-relational mapping-Implementation mit geringem Funktionsumfang, siehe www.sormula.org, zuletzt aufgerufen am 18. Oktober 2016 23, 42, 59

Sprachverarbeitung Ansatz der Computerlinguistik. Versucht, natürliche Sprache in gesprochener oder geschriebener Form maschinell zu verarbeiten 18, 49, 57, 58

Spring Ein verständliches Programmier- und Konfigurationsmodell für moderne Java-Systeme. Siehe www.projects.spring.io, zuletzt aufgerufen am 18. Oktober 2016 IX, 21, 22, 38–40, 47, 53–55, 57

Stakeholder Person oder Entität mit einem begründeten Interesse am oder Teilen eines Softwaresystems. Dazu gehören unter anderen Kunden, Softwareentwickler und Softwarearchitekten 9, 13, 16

Stapelverarbeitung Maschinelle Verarbeitung mehrerer konsekutiver Datensätze, im Gegensatz zum 'Eingabe-Verarbeitung-Ausgabe'-Prinzip 22, 23

Strategy-Pattern Entwurfsmuster, welches die Auswahl einer konkreten Implementation zur Laufzeit ermöglicht 46

Taktik Muster der Softwarearchitektur als abstrakte Lösung eines potentiell wiederkehrenden Architekturproblems. Siehe [Bass u. a., 2012] 18

Technologie-Feature Funktionale Einheit einer Technologie. Stellt bestimme technische Funktionalitäten bereit VIII–X, XIX, XXI, 2, 5, 6, 17–31, 34–38, 40–46, 50, 53, 55–57, 59–62, 64–72, 79–83, 85–92, 94–97

Topic Group Menge architekturrelevanter Problematiken, die einer Entwurfsentscheidung bedürfen. Siehe [Zimmermann u. a., 2009] 15

TopLink Implementation der JPA, siehe www.oracle.com, zuletzt aufgerufen am 18. Oktober 2016 VIII, 23, 42, 69, 70, 72

Traceabilty Rückverfolgbarkeit. Ermöglicht das Verknüpfen von Anforderungen mit der Umsetzung eines Systems 9, 10, 19

Transaktion Einheit der Atomarität innerhalb einer Datenbankoperation 52

Wildcard Bestimmtes Zeichen zur Repräsentation einer Menge von Zeichen XI, 50

Windows Mehrbenutzer-Betriebssystem der Firma Microsoft 76

Anhang G

Abkürzungen

API Application Programming Interface 19, 38

ASTA Architecturally Significant Technology Aspect IX, 16, 17, 26, 27, 35, 37, 41, 71, 80, 82, 93, 96

ATAM *Architecture Trade-off Analysis Method* 16

COTS Commercial off-the-shelf 11, 16

IEEE Institute of Electrical and Electronics Engineers 8

ISO International Organization for Standardization 9

JPA *Java Persistence API* VIII, XX, XXV, 23, 49, 69, 70, 72, 86, 87

JSP *JavaServer Pages* 21

LOC Lines of Code VII, 86, 88, 89

MVC Model View Controller 21, 55

ORM Object-relational mapping XXI, XXIII, 23, 38, 59, 84

XML Extensible Markup Language 84

Erklärung gemäß §59 Abs. 3 HmbHG

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen - benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Tobias Fechner

Hamburg, den 20. Oktober 2016