Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

# Diplomarbeit

## Feature Mining: Semiautomatische Transition von (Alt-)Systemen zu Software-Produktlinien

Verfasser:

Alexander Dreiling

02. Juli 2010

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake

Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme
Postfach 4120, 39106 Magdeburg, Germany

Dr.-Ing. Christian Kästner

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Hans-Meerwein Str., 35032 Marburg, Germany

**Dreiling, Alexander:**
*Feature Mining: Semiautomatische Transition von (Alt-)Systemen zu Software-Produktlinien*
Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, 2010.

# Danksagung

An dieser Stelle möchte ich mich bei all denen herzlich bedanken, die nicht nur diese Diplomarbeit, sondern auch das gesamte Studium erst möglich gemacht bzw. es prägend beeinflusst haben.

In erster Linie richtet sich mein Dank an meine Freundin Jana Munt und an meine Eltern Antonia und Juri Dreiling. Dieser engste Kreis hat mir zu jeder Zeit den notwendigen Rückhalt gegeben, sodass ich mich vollends auf das Studium konzentrieren konnte.

Insbesondere möchte ich mich zudem bei Gunter Saake für seine konstante Unterstützung bzw. Förderung über die gesamte Studienzeit bedanken. Gunter hat mir durch seine Empfehlung die Aufnahme in die Studienstiftung des deutschen Volkes ermöglicht. Darüber hinaus vermittelte er mir Anstellungen als Hilfswissenschaftler, betreute meine Studien- und Diplomarbeit und war auch stets für jegliche Anfragen ein verlässlicher Ansprechpartner.

Thomas Leich möchte ich vor allem für sein Vertrauen in meine Fähigkeiten danken. Er hat mir bei der METOP von der ersten Sekunde an das Gefühl vermittelt ein vollwertiges Mitglied des Teams zu sein. Mit anspruchsvollen Aufgaben hat er mir die Möglichkeit geboten mich weiterzuentwickeln. Zudem förderte er ohne viele Rückfragen die Weiterentwicklung meines eigenen Projekts „MyTT“. Thomas hat zudem für mich den Kontakt zu der Bayer AG hergestellt, sodass ich dort ein Auslandspraktikum absolvieren konnte. Die hieraus entstandene Studienarbeit hat er ebenso betreut.

Christian Kästner gilt mein besonderer Dank für seine ausgezeichnete Betreuung meiner Diplomarbeit. Er hat mich nicht nur an das Themenfeld herangeführt, sondern auch während der gesamten Bearbeitungszeit ein herausragendes Engagement gezeigt. Trotzt vieler anderer Verpflichtungen fand er immer die Zeit sich konstruktiv und hilfreich mit meinen Fragen und Anliegen auseinander zusetzen.

Für wertvolle Hinweise und Diskussion zu einigen Problemstellungen in dieser Diplomarbeit möchte ich zudem Thomas Thüm und Janet Feigenspan danken.

# Inhaltsverzeichnis

**Inhaltsverzeichnis** ..... v

**Abbildungsverzeichnis** ..... ix

**Tabellenverzeichnis** ..... xi

**Abkürzungsverzeichnis** ..... xiii

**1 Einleitung** ..... 1
- 1.1 Beitrag der Arbeit ..... 4
- 1.2 Gliederung der Arbeit ..... 5

**2 Software-Produktlinienentwicklung** ..... 7
- 2.1 Überblick ..... 7
- 2.2 Einführung und Betrieb ..... 10
  - 2.2.1 Vorteile und Risiken ..... 10
  - 2.2.2 Einführungsstrategien und Transitionsprozesse ..... 12
  - 2.2.3 Rahmenwerk des Betriebs ..... 15
- 2.3 Modellierung und Repräsentation von Variabilität ..... 18
  - 2.3.1 Feature-Modelle und ihre Darstellungsweisen ..... 18
  - 2.3.2 Repräsentation von Features in Quellcode-Artefakten ..... 21
- 2.4 Zusammenfassung ..... 25

**3 Feature Mining** ..... 27
- 3.1 Aufgabenstellung, Herausforderungen und Annahmen ..... 27
- 3.2 Verwandte Forschungsgebiete ..... 33
- 3.3 Zusammenfassung ..... 39

**4 Aktueller Stand der Technik** ..... 41
- 4.1 Lokalisierungstechniken ..... 42
  - 4.1.1 Statische Techniken ..... 42
    - 4.1.1.1 Textbasierte Mustersuche ..... 42
    - 4.1.1.2 Textbasierte Suche mit Information Retrieval Techniken ..... 43
    - 4.1.1.3 Quellcodestrukturbasierte Mustersuche ..... 46

4.1.1.4 Explorative Suche entlang von Strukturbeziehungen ........ 49
4.1.2 Dynamische Techniken ........ 51
4.1.2.1 Mengenoperationen auf Ausführungsspuren ........ 51
4.1.2.2 Markierung von Ausführungsspuren ........ 53
4.1.2.3 Rangordnung nach Zugehörigkeit ........ 54
4.1.3 Hybride Techniken ........ 57
4.2 Expansionstechniken ........ 59
4.2.1 Iterative Suchanfragen und Szenarien ........ 60
4.2.2 Expansion basierend auf Textinformationen ........ 60
4.2.3 Expansion basierend auf Klonen/vergleichbaren Eigenschaften ........ 62
4.2.4 Explorative Expansion entlang von Strukturbeziehungen ........ 63
4.2.5 Regelbasierte Expansion entlang von Strukturbeziehungen ........ 64
4.2.6 Relevanzwertbasierte Empfehlungen struktureller Nachbarn ........ 67
4.3 Dokumentationstechniken ........ 71
4.4 Fazit und Zusammenfassung ........ 75

**5 Lösungskonzept: LEADT ........ 77**
5.1 Strukturiertes, integriertes Konzept für das Feature Mining ........ 77
5.1.1 Funktionsweise ........ 78
5.1.2 Prototyp ........ 81
5.2 Vollständigkeitsfokussierte, automatisierte Feature-Expansion ........ 84
5.2.1 System-Modell ........ 84
5.2.2 Feature-Abhängigkeiten ........ 90
5.2.3 Expansions-Algorithmus ........ 93
5.2.3.1 Grundgerüst und Vereinigungsschema ........ 94
5.2.3.2 Vorschlagsansatz: Spezifitäts- und Bestärkungsheuristik ........ 96
5.2.3.3 Vorschlagsansatz: Textmuster-Erkennung ........ 99
5.2.3.4 Vorschlagsansatz: SPL-bewusstes Typsystem ........ 103
5.2.3.5 Vorteile der Vorschlagsansatz-Kombination ........ 107
5.3 Zusammenfassung ........ 111

**6 Evaluation ........ 113**
6.1 Evaluationsmethode ........ 113
6.2 Untersuchungsaufbau ........ 115
6.2.1 Grundsätzliches Vorgehen ........ 115
6.2.2 Wahl des Systems ........ 117
6.3 Auswertung der Fallstudie ........ 120
6.3.1 Erzielte Vollständigkeit der Features ........ 121
6.3.2 Effektivität der Element-Vorschläge ........ 123
6.3.3 Automatisierungspotential ........ 128
6.3.4 Benötigter Zeitaufwand ........ 130
6.3.5 Fazit ........ 133

6.4 Experimentelle Kritik ... 134
6.5 Zusammenfassung ... 135

**7 Zusammenfassung und Ausblick ... 137**

**A Erhobene Daten und Messwerte ... 143**
A.1 Teilmengen-Features ... 143
A.1.1 Play Music ... 144
A.1.2 SMS Transfer ... 146
A.1.3 Copy Media ... 148
A.1.4 Favourites ... 150
A.1.5 Count/Sort ... 151
A.2 Obermengen-Features ... 153
A.2.1 View Photos ... 154
A.2.2 SMS Transfer OR Copy Media ... 156

**Literaturverzeichnis ... 159**

# Abbildungsverzeichnis


Abbildung 2.1: Vom Feature bis zur Softwareproduktlinie - wesentliche Begriffe ............... 9
Abbildung 2.2: SPLE im Überblick ............................................................................................ 10
Abbildung 2.3: Rahmenwerk der SPLE im Detail ...................................................................... 16
Abbildung 2.4: Feature-Modell als Diagramm und aussagenlogischer Ausdruck.............. 20
Abbildung 2.5: Kompositionsansatz im Überblick.................................................................... 22
Abbildung 2.6: Annotationsansatz im Überblick....................................................................... 24

Abbildung 3.1: Werkzeuggestützte Realisierung einer annotationsbasierten Plattform.... 29

Abbildung 5.1: LEADT Konzept im Überblick ........................................................................... 78
Abbildung 5.2: Undisziplinierte und erweiterte disziplinierte Markierungen .................... 80
Abbildung 5.3: Screenshot des LEADT Prototypen ................................................................. 82
Abbildung 5.4: Enthält-Beziehung des System-Modells .......................................................... 87
Abbildung 5.5: Beispiele für Enthält-Beziehung....................................................................... 87
Abbildung 5.6: Referenziert-Beziehung des System-Modells.................................................. 88
Abbildung 5.7: Beispiele für Referenziert-Beziehung .............................................................. 88
Abbildung 5.8: Verwendet-Beziehung des System-Modells.................................................... 89
Abbildung 5.9: Beispiele für Verwendet-Beziehung ................................................................ 89
Abbildung 5.10: Von Feature- zu Feature-Repräsentations-Abhängigkeiten....................... 90
Abbildung 5.11: Beispiele für mögliche Äquivalenz-Abhängigkeiten ................................. 93
Abbildung 5.12: Grundgerüst des Expansions-Algorithmus................................................... 95
Abbildung 5.13: Vorschlagsansatz – Spezifitäts- und Bestärkungsheuristik........................ 97
Abbildung 5.14: Verwendet- bzw. „Wird verwendet von“-Beziehung für ein Beispiel..... 98
Abbildung 5.15.a: Vorschlagsansatz – Textmuster-Erkennung.......................................... 100
Abbildung 5.15.b: Vorschlagsansatz – Textmuster-Erkennung.......................................... 102
Abbildung 5.16: Beispiel-Register ............................................................................................ 102
Abbildung 5.17: Vorschlagsansatz – Typsystem...................................................................... 105
Abbildung 5.18: Beispiel – Bessere Abstraktion von Feature-Repräsentationen ............... 108
Abbildung 5.19: Beispiel – Aussagekräftigere, zuverlässigere Vorschlagsgenerierung ... 110

Abbildung 6.1: Feature-Diagramm von MobileMedia (7. Release)...................................... 118
Abbildung 6.2: Perfekte Element-Vorschläge in der Theorie................................................ 127
Abbildung 6.3: Element-Vorschläge durch LEADT in der Fallstudie ................................ 127

# Tabellenverzeichnis

Tabelle 2.1: Atypische Einführungsstrategien ... 13

Tabelle 3.1: Unterschiede verwandter Forschungsgebiete zum Feature Mining ... 33

Tabelle 5.1: Schlüssel-Elemente des System-Modells ... 85

Tabelle 6.1: Schlüssel-Elemente des 7. Releases vor und nach Anpassung für CIDE ... 119
Tabelle 6.2: Modifikationen des originalen MobileMedia Quellcodes ... 119
Tabelle 6.3: Erzielte Feature-Vollständigkeit ... 122
Tabelle 6.4: Maßnahmen zur Erreichung eines maximalen Vollständigkeitsgrads ... 123
Tabelle 6.5: Effektivität der Vorschläge im Vergleich zu einem Zufallsgenerator ... 125
Tabelle 6.6: Automatisierungspotential für die Feature-Expansion in MobileMedia ... 129
Tabelle 6.7: Benötigter manueller Zeitaufwand für das Feature Mining ... 132
Tabelle 6.8: Wesentliche Ergebnisse im Überblick ... 136

# Abkürzungsverzeichnis

| | |
|---|---|
| **FM** | Feature Mining |
| **IR** | Information Retrieval |
| **LEADT** | Location, Expansion and Documentation Tool |
| **SoC** | Separation of Concerns |
| **SPL** | Software-Produktlinie |
| **SPLE** | Software-Produktlinienentwicklung |

# 1 Einleitung

Seit Beginn der Computerentwicklung wurde mit zunehmender Rechenkapazität auch die ausgeführte Software komplexer. Aufgrund unzureichender Methodik und fehlender Konzepte konnten infolgedessen viele Softwareentwicklungsprojekte nicht mehr innerhalb des vereinbarten Zeit- und Kostenrahmen abgeschlossen werden. Die Qualität der Software aber auch die Wartbarkeit des Quellcodes waren mehr als ungenügend. Mit der NATO-Konferenz im Jahre 1968 wollte man durch die Etablierung des so genannten *Software-Engineering* dieser Tendenz entgegen wirken. Einem unstrukturiertem Vorgehen sollte der gezielte Einsatz von Technologien und Methoden aus den Bereichen der Informatik, des Projektmanagements sowie dem Ingenieurwesen weichen [NR69].

Trotz oder gerade durch die verbesserten Entwicklungsparadigmen nahm die Komplexität nicht ab. Im Gegenteil, seit Anfang der 1990er hat sich beispielsweise die Anzahl der Quellcodezeilen des Betriebssystems Windows verdreißigfacht. Im Bereich der eingebetteten Systeme stieg die Softwaregröße in dem gleichen Zeitraum zum Teil um das hundertfache [PBL05, S. 12ff.]. Ein wesentlicher Grund hierfür ist die zunehmende Funktionalitätsverlagerung von Hardware zu Software. Zudem kommt ein weiterer Aspekt hinzu. Software wird wichtiger Bestandteil moderner Produkte jeglicher Art. Der Wandel der Produkte selbst aber auch die Angepasstheit für verschiedene Kundengruppen und Märkte führt zu einer Explosion der spezifischen Anforderungen [BKPS04, S. 3f; LSR07, S. 3]. Gelingt es softwareentwickelnden Unternehmen nicht mit dieser Komplexität umzugehen, ist mit höheren Fehlerraten und Kosten sowie längeren Entwicklungszeiten zu rechnen. Nicht zuletzt aufgrund des weltweit immer stärker werdenden Konkurrenzdrucks können solche Folgen verheerend sein [LSR07, S. 3; PBL05, S. 12].

Die Softwareentwicklung zwingt daher Theoretiker und Praktiker, damals wie heute, zur kontinuierlichen Verbesserung der verwendeten Konzepte. In dem letzten Jahrzehnt hat sich die *Software-Produktlinienentwicklung* als einer der vielversprechendsten Antworten auf die aktuellen Herausforderungen hervorgetan [LSR07, S. 3; SPK06]. Die wesentliche Idee dieses Paradigmas besteht darin nicht mehr jedes einzelne System für sich allein zu entwickeln, sondern den Bedarf an unterschiedlicher Funktionalität (hier *Features* genannt) einer ganzen *Domäne* in einer strategisch geplanten, organisierten *Plattform* zu bündeln. Aus jener können dann durch die Selektion von benötigten Features spezifische *Produktvarianten* erzeugt werden. Mehrere solcher Varianten beschreiben zusammen eine *Software-Produktlinie* [AK09; BKPS04, S. 277ff.; CN02, S. 5ff.; LSR07, S. 314ff.; PBL05, S. 15].

Da einzelne Artefakte der Plattform, wie z.B. ausführbare Quellcode-Einheiten, bewusst entwickelt und systematisch wiederverwendet werden, können *qualitativ hochwertige* Software-Systeme mit *maßgeschneiderter Funktionalität* zu *geringen Kosten* und *kurzen Entwicklungszeiten* erzeugt werden [BKPS04, S. 3ff.; LSR07, S. 3ff.; PBL05, S. 9ff.; SEI10; SPK06].

Damit einzelne Software-Produktvarianten aus der Plattform abgeleitet werden können, müssen Features und Abhängigkeiten zwischen ihnen, welche in einem *Feature-Modell* abstrahiert sind, in den Quellcode-Artefakten repräsentiert werden. Die hierfür verwendeten Konzepte und Mechanismen müssen sicherstellen, dass Features flexibel auswählbar und komponierbar sind [CRB04].

Innerhalb der akademischen Forschung werden für die Implementierung von Software-Produktlinien vor allem *Kompositionsansätze* bevorzugt, die *Features vollständig in physisch getrennten Modulen* implementieren. Produktvarianten werden bei diesem Vorgehen generiert indem ausgewählte (Feature-)Module zur Übersetzungs- oder Startzeit, z.B. mit speziellen Compilern, zusammengesetzt werden [KAK08; Käs10]. Einige bekannte Vertreter dieser Technologiengruppe sind Frameworks [JF88], Komponentenorientierte- [HC01], Aspektorientierte- [KLM+97] sowie Featureorientierte Programmierung [Pre97; BSR04; AKL09b]. Solche Konzepte können auf lange Sicht gesehen eine hohe Softwarequalität und Architekturstabilität gewährleisten. Für die Umsetzung der Plattform sind aber „moderne“ Sprachen bzw. umfangreiche Spracherweiterungen erforderlich, welche den meisten Entwicklern noch unbekannt sein dürften [AKL09a; Käs10; KTA08].

Demgegenüber stehen *Annotationsansätze.* Sie erreichen nur bedingt die Vorteile einer physischen Modularisierung, wie beispielsweise die separate Entwicklung und Wartung einzelner Features. Allerdings bestechen sie durch einen einfachen, flexiblen und programmiersprachenunabhängigen Einsatz. Hierbei werden zu Features korrespondierende *Quellcodeelemente direkt gekennzeichnet bzw. annotiert*. Bei der Erzeugung von Produktvarianten werden alle markierten Elemente entfernt, deren zugehörige Features nicht Teil der Selektion sind. Ein solches Konzept wird beispielsweise durch *textuelle* Präprozessor-Anweisungen umgesetzt, welche in einigen Programmiersprachen, wie z.B. C und Visual Basic, zu dem Standardrepertoire gehören [KAK08]. Für übrige Sprachen wurden spezielle Werkzeuge und Erweiterungen entwickelt, wie z.B. XVCL [JBZZ03], Gears [Kru01] und pure::variants [BPS04]. CIDE ersetzt herkömmliche Präprozessor-Anweisungen durch sprachenunabhängige, *farbliche* Quellcode-Annotationen [KAK08; Käs10].

Ohne genaue Kenntnis des Marktes, der Kunden sowie detaillierten technischen Realisierungsmöglichkeiten ist die Modellierung und Umsetzung einer Software-Produktlinie nahezu unmöglich. In der Regel bietet sich dieses neue Paradigma daher vor allem für Unternehmen an, welche auf der einen Seite über entsprechende Erfahrungen durch die Entwicklung von eigenen Produkten verfügen, auf der anderen Seite aber unter gewissem

Druck stehen ihre Software-Lösungen wirtschaftlicher produzieren und warten zu müssen [BKPS04, S. 137; BOS00; Joh06; PBL05, S. 394]. Eine vollständige Neuentwicklung einer Produktlinien-Plattform, mit einem „modernen" Kompositionsansatz, kommt für diese Zielgruppe meist aus zwei entscheidenden Gründen nicht in Frage. Zum einen sind enorme Vorabinvestitionen und massive Ressourcenbindungen mit dem normalen Produktivbetrieb kaum vereinbar. Zum anderen besteht ein sehr hohes Risiko wertvolles Unternehmenswissen zu verlieren, welches über Jahre hinweg in die Software-Lösungen eingebunden wurde [BKPS04, S. 8; CK02; ES01; Joh06; Kru01; MTW93; SE02; SEI10].

Um die Idee der Software-Produktlinienentwicklung in der Praxis zu etablieren werden daher Konzepte gebraucht, welche das Transitionsrisiko und den Transistionsaufwand senken können [Kru01; CK02; SE02]. Der Einsatz von leichtgewichtigen Annotationsansätzen stellt einen ersten Schritt in diese Richtung dar, weil hierdurch das Potential geboten wird einen Großteil der vorhandenen Quellcode-Ressourcen beizubehalten [CK02]. Für die Realisierung der Plattform müssen in jenen Ressourcen allerdings alle benötigten Features so annotiert werden, dass sich hieraus gültige Produktvarianten generieren lassen [Käs10]. Da die bestehenden Systeme aber nicht zwangsläufig für die Wiederverwendung im Rahmen der Software-Produktlinienentwicklung geplant waren, ist davon auszugehen, dass implementierte Features nicht konsistent im Quelltext kommentiert oder modular repräsentiert sind [LAL+10; MR05; MRB+05, OS02; SE02; WBP+03]. Gerade für komplexe Systeme, die nicht auf einen Blick überschaut werden können, ist eine vollständig-manuelle Erfassung aller Features nicht nur sehr zeitaufwändig, sondern auch fehlerbehaftet [WR07]. Konzepte und Werkzeuge, welche die Bearbeitung jener Annotierungsaufgabe unterstützen, könnten daher einen wesentlichen Beitrag zur weiteren Verringerung der Transitionsbarriere leisten.

Das grundlegende Problem, zu einem abstrakten Konstrukt zugehörige Quellcodefragmente zu identifizieren, ist nicht neu [Rob05]. Viele unterschiedliche Themengebiete befassen sich unter anderem mit solch einer Aufgabenstellung. Reverse Engineering, Concept-, Concern-, Fault-, Feature Location, Requirements Tracing, Aspect-, Asset Mining und Impact Analysis sind einige von vielen Beispielen, die hierbei genannt werden können. Nach dem aktuellen Kenntnisstand des Autors verfolgt allerdings keiner dieser Themenbereiche das Ziel eine annotationsbasierte Software-Produktlinien-Plattform aufzubauen.

Vor jenem Hintergrund wird in dieser Arbeit unter dem Begriff *Feature Mining from Source Code*, oder abgekürzt *Feature Mining*, ein neues Forschungsgebiet aufgespannt. Jenes zielt darauf ab (semi-)automatische Lösungen zu finden, mit denen ein (Alt-)System inkrementell in eine annotationsbasierte Plattform überführt werden kann. Solche Konzepte müssen einen Anwender also dabei unterstützen zu einem definierten Feature-Modell korrespondierende Quellcodefragmente so zu lokalisieren, zu expandieren und zu dokumentie-

ren, dass im Anschluss fehlerfreie Produktvarianten für alle zulässigen Feature-Selektionen erzeugt werden können.

Damit zukünftige Forschungsanstrengungen den Feature Mining Problembereich eindeutig adressieren können, werden im Rahmen dieser Diplomarbeit wesentliche Herausforderungen, Merkmale und Anforderungen analysiert bzw. abgegrenzt. Um interessierten Forschern den Einstieg in diese Materie zu erleichtern, werden zudem aus verwandten Themengebieten zahlreiche Ansätze vorgestellt, welche für die Entwicklung bzw. Implementierung von Feature Mining Konzepten adaptiert werden könnten. Basierend auf den dargestellten Erfahrungen und Erkenntnissen wird darüber hinaus ein eigens entwickeltes, erstes Lösungskonzept vorgeschlagen, welches alle Aufgaben des Feature Mining integriert und strukturiert unterstützt. Das hierzu prototypisch implementierte *Location, Expansion and Documentation Tool - LEADT* zeigt im Rahmen einer Fallstudie, dass auf diese Weise (nahezu) vollautomatisch rund 82% der gesuchten Feature-Repräsentationen im Quellcode identifiziert werden können. Wobei die Fehlerquote für die ermittelten Quellcodeelemente unter 20% liegt. Im Vergleich hierzu hat ein Zufallsgenerator in der selben Fallstudie im Mittel eine Fehlerquote von rund 90%.

LEADT gibt mit Sicherheit nicht die letzte optimale Antwort für das Feature Mining Problem. Allerdings demonstriert jener Prototyp, dass mit bereits einfachen Methoden und Techniken der zeitliche Aufwand und das Fehlerrisiko für den Aufbau einer Produktlinien-Plattform aus (Alt-)System-Ressourcen deutlich reduziert werden kann. Es wäre für die Zukunft wünschenswert, wenn Forscher inspiriert durch diese Arbeit neue, verbesserte Sichtweisen einbringen würden, um damit die Software-Produktlinienentwicklung ein Schritt näher an die industrielle Praxis heranzuführen.

## 1.1 Beitrag der Arbeit

Im Rahmen dieser Arbeit werden vier entscheidende Beiträge geleistet, welche im Folgenden kurz zusammengefasst sind:

1. Mit dem Ziel der weiteren Senkung der Adoptionshürde der Software-Produktlinienentwicklung in der Praxis, wird in dieser Arbeit vorgeschlagen (Alt-)Systeme (semi-)automatisch in eine annotationsbasierte Plattform zu überführen. Hierfür werden wesentliche Teilaufgaben, Herausforderungen und Annahmen analysiert bzw. definiert. Es wird gezeigt, dass viele Forschungsgebiete existieren, welche eine verwandte Fragestellung adressieren, wobei aber keines von ihnen exakt die definierten Kriterien erfüllt. Aufgrund dessen wird ein neues Themen- bzw. Forschungsgebiet Namens *Feature Mining* begründet.

2. Aus Schnittmengen mit den verwandten Forschungsgebieten werden über 100 Arbeiten zusammengefasst, welche für die Entwicklung von Feature Mining Konzepten wertvolle Erfahrungen und Ergebnisse liefern könnten. Hierbei werden jeweils ähnliche Ansätze zu Konzeptkategorien gruppiert und hinsichtlich ihrer aktuellen Eignung für das Feature Mining bzw. für dessen Teilaufgaben bewertet. Diese übersichtliche Darstellung des *aktuellen Stands der Technik* soll interessierten Forschern den Einstieg in die Feature Mining Thematik erheblich erleichtern und sie somit für eigene Arbeiten animieren. Prinzipiell wird also die Intension verfolgt, dass durch diese Vorarbeit zum einen *mehr* neue Lösungen realisiert werden und dass sie vor allem *qualitativ hochwertiger* sind, weil sie auf erprobten Konzepten aufbauen.

3. Basierend auf der Analyse des aktuellen Stands der Technik wird ein erstes Lösungskonzept vorgeschlagen, welches alle Aufgaben des Feature Mining integriert und strukturiert unterstützt. In dem hierzu implementierten Prototyp Namens *Location, Expansion and Documentation Tool – LEADT* werden Werkzeuge und Techniken von anderen Forschern adaptiert und mit neu entwickelten Komponenten kombiniert.

4. Die Leistungsfähigkeit von LEADT wird anhand einer Fallstudie evaluiert. In dieser werden sieben unterschiedliche Features in einer mittelgroßen Anwendung (4600 Quellcodezeilen) annotiert. In der Ergebnisdiskussion wird festgestellt, dass LEADT die Komplexität und den Aufwand einer Transition eines Systems, hin zu einer annotierten Software-Produktlinien-Plattform, maßgeblich reduzieren kann.

## 1.2 Gliederung der Arbeit

In *Kapitel 2 (Software-Produktlinienentwicklung)* werden wesentliche Grundlagen aus dem Bereich der Software-Produktlinienentwicklung behandelt. Jene Ausführungen bilden die Basis für die Abgrenzung, Einordnung und Motivation dieser Arbeit. Sie schaffen zudem ein einheitliches Verständnis zu wesentlichen Begriffen und Konzepten, die in den folgenden Kapiteln benutzt werden.

In *Kapitel 3 (Feature Mining)* wird zur Senkung der Adoptionsbarriere von der Software-Produktlinienentwicklung vorgeschlagen Altsysteme (semi-)automatisch in eine annotationsbasierte Plattform zu überführen. Hierfür werden konkrete Aufgaben, Herausforderungen und Annahmen herausgearbeitet. Mit diesem Vorschlag wird Forschungsneuland beschritten, welches als „Feature Mining" bezeichnet wird. Um diese Feststellung zu unterlegen wird Feature Mining gegenüber anderen Forschungsgebieten, die sich mit einer ähnlichen Aufgabenstellung beschäftigen, abgegrenzt.

In *Kapitel 4 (Aktueller Stand der Technik)* werden aus Schnittmengen mit verwandten Forschungsgebieten zahlreiche Ansätze vorgestellt, welche für die Entwicklung von Feature Mining Lösungen wertvolle Erkenntnisse einbringen könnten. Hierbei werden unterschiedliche Publikationen zu Konzeptkategorien zusammengefasst, die wiederum konkreten Feature Mining Aufgaben zugeordnet werden. Diese kompakte Darstellung des aktuellen Stands der Technik versteht sich als ein möglicher Einstiegspunkt für weitere Forschungsaktivitäten in diesem Bereich.

In *Kapitel 5 (Lösungskonzept: LEADT)* wird das „Location, Expansion and Documentation Tool – LEADT" vorgeschlagen, mit dem erstmalig die Möglichkeit geboten wird alle Aufgaben des Feature Mining strukturiert und integriert zu bearbeiten. Hierbei wird einleitend ein Überblick über die Funktionsweise, Bestandteile sowie über den implementierten Prototyp gegeben. LEADT vereint diverse Werkzeuge und Techniken von anderen Forschern mit neu entwickelten Komponenten. In diesem Kapitel werden insbesondere die Letztgenannten in aller Ausführlichkeit besprochen.

In *Kapitel 6 (Evaluation)* wird anhand einer Fallstudie die Eignung bzw. das Potential von LEADT für das Feature Mining geprüft. In dieser werden mit Hilfe von LEADT insgesamt sieben implementierte Features in der MobileMedia Applikation lokalisiert, expandiert und dokumentiert. Die ermittelten Ergebnisse werden hinsichtlich verschiedener Kennzahlen bewertet und  abschließend kritisch gewürdigt.

In *Kapitel 7 (Zusammenfassung und Ausblick)* werden wesentliche Ergebnisse dieser Arbeit zusammengefasst. Darüber hinaus wird ein Ausblick zu möglichen weiteren Schritten und Aufgaben im Bereich des Feature Mining gegeben.

# 2 Software-Produktlinienentwicklung

Mit Hilfe von Konzepten wie Subroutinen in den 1960ern, Modulen in den 1970ern, Objekten in den 1980ern, Komponenten in den 1990ern und schließlich Services in den 2000ern wurde im Software-Engineering immer wieder versucht das Leistungsvermögen und die Wirtschaftlichkeit von Software zu erhöhen. Diese Entwicklung wurde nicht zuletzt durch die immer wieder ansteigende Komplexität der zu entwickelnden und zu wartenden Anwendungen angetrieben [CN02, Vorwort]. Seit der Jahrtausendwende hat die *Software-Produktlinienentwicklung* einen enormen Auftrieb erfahren und etabliert sich allmählich als nächstes bedeutendes Paradigma im Software-Engineering [BKPS04, S. 265; LSR07, S. 3]. Es wurde gezeigt, dass sich Vorteile von Produktlinien der Fertigungswirtschaft, wie beispielsweise im Automobilbereich, auch auf Software übertragen lassen [SEI10]. So wird unter anderem die Umsetzung hoher Funktionalität und Flexibilität in Softwaresystemen, zu geringeren Kosten und Entwicklungszeiten versprochen. Gleichzeitig wird zudem auch die Qualität der entwickelten Produkte verbessert [BKPS04, S. 3; SEI10; SPK06].

In diesem Kapitel werden wesentliche Grundlagen aus dem Bereich der *Software-Produktlinienentwicklung* behandelt. Jene Ausführungen bilden die Basis für die Abgrenzung, Einordnung und Motivation dieser Arbeit. Sie schaffen zudem ein einheitliches Verständnis zu wesentlichen Begriffen und Konzepten, die in den folgenden Kapiteln benutzt werden.

## 2.1 Überblick

Im Allgemeinen versteht man unter einer **Software-Produktlinie *(SPL)*** eine Menge von Softwareprodukten mit gemeinsamen *Features*, die aus einer geplanten, organisierten Menge von *Artefakten* einer *Plattform* erstellt werden, und die den Bedarf eines bestimmten Marktes, einer bestimmten *Domäne* oder einer bestimmten Aufgabe decken [BKPS04, S. 279; CN02, S. 5]. Eine **Domäne** beschreibt in diesem Kontext einen zusammenhängenden Teilbereich bzw. ein Anwendungsgebiet, welches durch charakterisierende Konzepte und Merkmale geprägt ist [BKPS04, S. 278; LSR07, S. 314]. Software für Diesel-Motoren, Autos, Flugzeuge, Schiffe, Mobiltelefone, Telekommunikationsnetzwerke, Online-Shops,

Medizingeräte, Drucker oder TV- und Hifi-Systeme sind unter anderem Beispiele für solche Domänen [Kru01; SEI10].

Je nach dem ob man SPL aus dem Blickwinkel der Modellierung oder der Realisierung betrachtet, ergeben sich für **Features** unterschiedliche Definitionen [AK09; CHS08]. Innerhalb der Modellierung beschreibt ein Feature eine abgrenzbare *Funktionalität* eines Softwaresystems. In diesem Zusammenhang helfen Features vor allem dabei die **Variabilität**, d.h. die Unterschiede der Software-Produktvarianten [BKPS04, S. 280; LSR07, S. 316], zu benennen [CE00; KCH+90]. Aus der Perspektive der Realisierung beschreibt ein Feature wie diese Funktionalität implementiert ist. Konkret also eine Struktur, welche den Aufbau eines gegebenen *Artefaktes* modifiziert, um Anforderungen einer Interessengruppe zu erfüllen sowie um eine mögliche Konfigurationsoption anzubieten [AK09]. Vor jenem Hintergrund kann man im letzteren Fall auch von einer **Feature-Repräsentation** sprechen. In dieser Arbeit werden die Begriffe *Feature* und *Feature-Repräsentation* als Synonyme verwendet. Eine Unterscheidung wird nur dann vorgenommen, wenn sich die genaue Bedeutung nicht aus dem Kontext schließen lässt. [1]

**Software-Produktvarianten** werden durch eine spezifische **Feature-Selektion** erzeugt. Hierbei werden wiederverwendbare **Artefakte**[2], welche Repräsentationen der ausgewählten Features enthalten, zu einem spezifischen Produkt gebunden. Artefakte sind hierbei jegliche Ressourcen, die während der *Software-Produktlinienentwicklung* entstehen. Unter anderem können das z.B. Dokumente, Modelle und ausführbare Einheiten der Anforderungsanalyse, des Entwurfs, der Realisierung oder des Tests sein [BKPS04, S. 277; LSR07, S. 314; PBL05, S. 23]. Die Menge aller Artefakte bildet die **Plattform** der Softwareproduktlinie [BKPS04, S. 279; CN02, S. 522; PBL05, S. 15]. *Abbildung 2.1* stellt die Beziehungen zwischen den bislang diskutierten Begriffen noch einmal in grafischer Form dar.

Die Idee der **Software-Produktlinienentwicklung** ***(SPLE)*** basiert auf der Beobachtung, dass Hersteller in der Regel ihre Softwareprodukte einmalig entwickeln und sie anschließend innerhalb einer Domäne variieren. Solche Produktvarianten haben daher meist mehr Gemeinsamkeiten als Unterschiede. Der entscheidende Punkt besteht genau darin, so viele Gemeinsamkeiten wie möglich systematisch wiederzuverwenden [BKPS04, S. 265; FK05; SPK06; SEI10]. Hierdurch wird zum einen die Arbeitslast reduziert, zum anderen aber auch der Umfang der Softwaresysteme greifbarer, weil Entwickler häufig mit gleich

---

[1] Um Missverständnissen vorzubeugen sei an dieser Stelle angemerkt, dass obwohl in einigen Publikationen *Features* mit *Belangen* gleichgesetzt werden [MLWR01], in dieser Arbeit eine andere Sichtweise vertreten wird. Es wird die Anschauung von Apel geteilt, dass Features sich auf konkrete Anforderungen von Interessengruppen zurückführen lassen und eine Konfigurationsoption anbieten. Wobei Belange diese Eigenschaft nicht zwangsweise erfüllen müssen. Demnach bilden *Features* eine Teilmenge von *Belangen* [Ape07].

[2] In englischsprachige Veröffentlichungen werden Artefakte häufig auch als *Assets* bezeichnet, wie beispielsweise in [CN02].

bleibenden Artefakten arbeiten. Der Ansatz der Komplexitätsreduktion und Produktivitätssteigerung durch Wiederverwendung ist keineswegs neu. Bereits Anfang der 1970er wurden Arbeiten mit dieser Intension veröffentlicht, wie beispielsweise von McIlroy [McI69] oder Parnas [Par76]. Allerdings haben Erfahrungen in großen Softwareprojekten der 1990ern gezeigt, dass Wiederverwendung zum Teil mehr Kosten als Nutzen bringen, wenn diese ungeplant, rein technologiegetrieben und im kleinen Maßstab für Einzelsysteme durchgeführt werden. Zahlreiche Forschungsinitiativen[3] setzten genau hier an und verwiesen auf die Notwendigkeit eines Paradigmenwechsels – hin zur *Produktlinienentwicklung*. Das heißt, Wiederverwendung muss für eine Gruppe von Softwaresystemen innerhalb einer Domäne vorausschauend *geplant* und in Übereinstimmung mit den strategischen Zielen des Unternehmens *verwaltet* werden [CN02, S. 9f.; LSR07, S. 6; PBL05, S. 394f.].

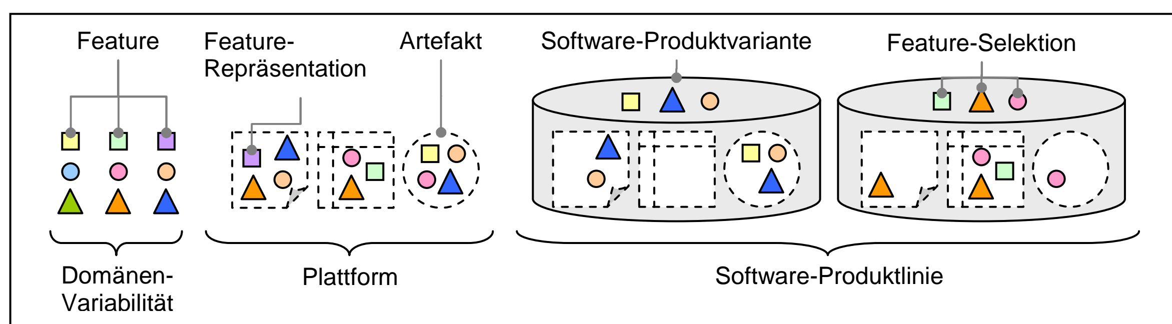



Abbildung 2.1: Vom Feature bis zur Softwareproduktlinie - wesentliche Begriffe

Die **„gemanagte“** Wiederverwendung, wie sie die SPLE fordert, lässt sich vor allem dann umsetzen, wenn der Entwicklungsprozess zweigeteilt wird. Im Rahmen des **Domain-Engineering** muss die *Variabilität*, welche sich aus den spezifischen Anforderungen der Kunden ergibt, analysiert und in der Plattform abgebildet werden. Darauf aufbauend können im **Application-Engineering** einzelne Produktvarianten effizient aus der Plattform erzeugt werden. Mit diesem Vorgehen kann eine individualisierte Massenfertigung mit qualitativhochwertigen und wirtschaftlichen Produkten erreicht werden. Weitere Vorteile, die sich durch die SPLE realisieren lassen, werden in *Kapitel 2.2.1* überblicksartig besprochen. Die Umsetzung der beschriebenen *Hauptaktivitäten* wird allerdings erst durch eine Vielzahl von unterschiedlichen Aktivitäten, Prozessen und Konzepten ermöglicht. Jene werden zu den *Methodenbereichen Software-Engineering*, *Technical*- und *Organizatio*nal-Management zusammengefasst [BKPS04, S. 3ff.; LSR07, S. 6; PBL05, S. 7, S. 394f.; SEI10].

[3] Um das Jahr 2000 wurden mit *ARES*, *Praise*, *ESAPS*, *CAFE* und *FAMILIES* umfangreiche SPLE-Forschungsprojekte innerhalb Europas durchgeführt. In den USA wurde zur gleichen Zeit die Entwicklung durch das *Carnegie Mellon Software Engineering Institute* vorangetrieben [LSR07 S.5f; SPK06].

*Abbildung 2.2* zeigt die Zuordnung zwischen den genannten Methodenbereichen und Hauptaktivitäten [SEI10].

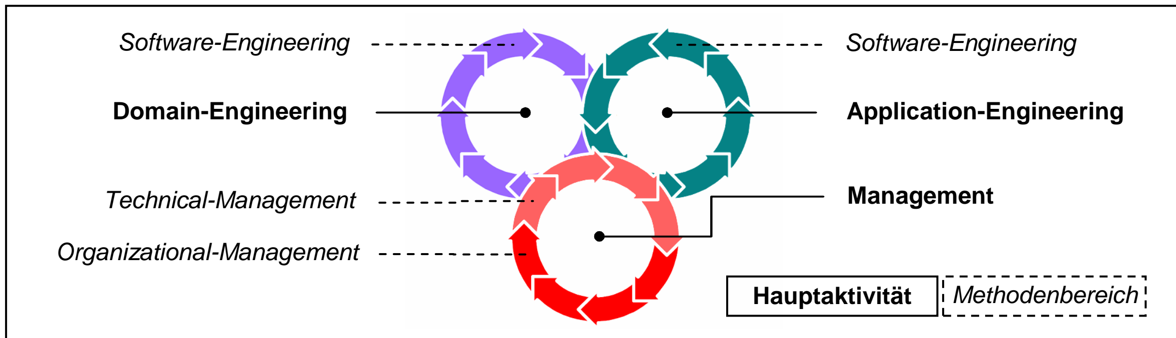



Abbildung 2.2: SPLE im Überblick

Zusammenfassend kann festgehalten werden, dass die SPLE auf zwei wichtigen Konzepten basiert – der *gemanagten Trennung von Domain- und Application Engineering* sowie der *Modellierung und Realisierung von Variabilität*. Für ein tiefergehenderes Verständnis, welches für diese Arbeit notwendig ist, werden beide Aspekte separat in den *Kapitel 2.2.3* und *2.3* ausführlich besprochen.

## 2.2 Einführung und Betrieb

In diesem Kapitel werden einige Aspekte der Einführung und des Betriebs der SPLE diskutiert. Jene Ausführungen bilden eine wesentliche Grundlage für die Motivation und Einordnung der Forschungsanstrengungen dieser Arbeit. In *Kapitel 2.2.1* werden zunächst Vorteile und Risiken besprochen, welche momentan die Einführungsentscheidung prägen. Nach dem Entschluss für dieses Entwicklungsparadigma muss die Einführung bzw. Transition organisiert werden. *Kapitel 2.2.2* stellt Strategien und Prozesse vor, die hierbei Beachtung finden können. Gefolgt von der Transition, beginnt in einem fließenden Übergang der eigentliche produktive SPLE- Betrieb. Dessen recht abstrakte Beschreibung aus dem Überblick (vgl. *Kapitel 2.1*) wird in *Kapitel 2.2.3* um konkrete Aktivitäten, Methodenbereiche und Informationsflüsse erweitert.

### 2.2.1 Vorteile und Risiken

Der Übergang von einer Einzelsystem- zu einer Produktlinienperspektive bietet zahlreiche Vorteile bei der Entwicklung, Erweiterung sowie Wartung von Software. Demgegenüber stehen allerdings auch erhebliche Risiken [PBL05, S. 14]. Im Folgenden werden beide Seiten mit ihren wichtigsten Punkten gegenübergestellt.

Eine mehrfache Nutzung von einmalig entwickelten Artefakten führt zwangsläufig zu *geringeren Entwicklungskosten pro System*. Dieser Vorteil wird aber erst durch massive Vorabinvestitionen zur Entwicklung der gemeinsamen Plattform ermöglicht. Obwohl zahlreiche Faktoren die Amortisierung dieser Fixkosten beeinflussen können, behaupten Experten, dass die SPLE bereits ab dem dritten bis vierten entwickelten System niedrigere Entwicklungskosten im Vergleich zu der Einzelsystementwicklung fordert [CN02, S. 20f.; LSR07, S. 3; PBL05, S. 9f.; WL99]. Das *Carnegie Mellon Software Engineering Institute* berichtet von Fällen, in denen die Kosten um 60% und der Mitarbeiterbedarf um 87% reduziert wurden [SEI10].

Nicht zuletzt in Bereichen wie Fahrzeug-, Flugassistenzsystemen aber auch in der Medizintechnik, kann Fehlverhalten von Software verheerende Folgen haben. Eine *höhere Qualität* der Produkte ist daher ein besonders wichtiger Vorteil, der sich aufgrund der Mehrfachnutzung der Artefakte ergibt. Die jeweiligen Softwaremodule kommen in verschiedenen Szenarien und Produkten zum Einsatz. Daher werden sie zwangsläufig öfter geprüft und getestet. Die Wahrscheinlichkeit Fehler zu entdecken und korrekt zu lokalisieren wird damit erheblich höher. In einigen Fallstudien konnte die Produktfehlerdichte um die Hälfte reduziert werden [LSR07, S. 17ff.]. Fehlerquellen werden auch dadurch gemindert, weil die Entwickler mit dem Aufbau und der Funktionsweise häufig verwendeter Artefakte besser vertraut sind. Infolgedessen wird das *Entwicklungsrisiko erheblich verringert* [BKPS04, S. 6; LSR07, S. 3ff.; PBL05, S.10].

SPLE steht vor allem für eine explizite und durchgängige Dokumentation von Variabilität in allen Artefakten der Plattform. Damit eng verbunden sind Mechanismen, die basierend auf einer Feature-Selektion, Produktvarianten effizient erzeugen bzw. generieren können. Hierdurch können an *spezifische Kundenwünsche angepasste Lösungen* mit einer bis zu 98% geringeren *Time-To-Market* bereitgestellt werden [SEI10]. Man spricht in diesem Zusammenhang auch von einer *individualisierten Massenfertigung*. In Kooperation mit gezielten Marketingaktivitäten, wie Preis- und Produktdifferenzierungsstrategien, können *verschiedenste Marktsegmente erreicht und abgeschöpft* werden [BKPS04, S. 6; CN02, S. 20ff.; Kru01; LSR07, S. 3; PBL05, S. 10ff.].

Die Nutzung einer Plattform mit effizienten Produktgenerierungsmechanismen *senkt* zudem die Kosten für *Wartung* und *Evolution* um bis zu 20% [LSR07, S. 17f.]. Zum einen können, anhand der klaren Zuordnung von Features zu Artefakten, Fehlerstellen leichter lokalisiert und Erweiterungswünsche schneller umgesetzt werden. Zum anderen müssen jene Änderungen nicht mehr in jedem Produkt, sondern nur einmalig in der Plattform durchgeführt werden. Anschließend werden die Produkte erneut generiert und an die Kunden ausgeliefert [FKBA07; LSR07, S. 4; MPG03; PBL05, S. 11f.].

Bei der SPLE profitiert nicht nur die produzierende Seite von den positiven Effekten. Kunden haben vor allem den Vorteil, dass ihnen *Lösungen perfekt abgestimmt auf ihre An-*

*forderungen, zu einem geringeren Preis,* in einer kürzeren Zeit und in einer *besseren Qualität* angeboten werden [CN02, S. 22; LSR07, S. 5; PBL05, S. 12f.].

In der Vergangenheit wurden viele Software-Produktlinien in zahlreichen Domänen erfolgreich etabliert [FSR07; PBL05; SEI10; SPLC09]. Bekanntlich hat aber jede Medaille auch eine Kehrseite. In diesem Fall müssen für die Implementierung der Plattform Chefarchitekten und -ingenieure zum Teil für Monate aus dem Produktivbetrieb freigestellt werden. Die Organisationsstruktur, Prozesse und Werkzeuge müssen auf die SPLE ausgerichtet werden. *Solche massiven Veränderungen* und *Investitionen* bringen selbstverständlich *erhebliche Risiken* mit sich, wenn sie nicht im Einklang mit der Unternehmensstrategie abgestimmt sind [BKPS04, S. 153; CN02, S. 23; Joh06; Kru01]. Zudem sind *hervorragende Domänenkenntnisse unabdingbar*. Eine SPL zahlt sich nur dann aus, wenn sie auf die (zukünftigen) Anforderungen der Kunden und Absatzmärkte abgestimmt ist. Ungenutzte Funktionalität verursacht unnötige Entwicklungs- und Wartungskosten. Fehlende Funktionalität führt hingegen zu längeren Wartezeiten für die Kunden [CN02, S. 23; BKPS04, S. 6; PBL05, S. 17f.].

Während die methodischen Grundlagen der SPLE recht klar definiert sind, ist die *Erforschung von Werkzeugen*, welche das Variabilitäts- und Konfigurationsmanagement über den gesamten Entwicklungs- und Lebenszyklus integriert unterstützen, *noch längst nicht abgeschlossen*. Bislang existieren fast nur Forschungsprototypen, die auf einzelne Aspekte ausgerichtet sind. Ganzheitliche Ansätze werden momentan nur rudimentär von spezialisierten Unternehmen angeboten. Unter diesen Umständen kann die *Entwicklung von hochwertiger Software nur mit enormen Anstrengungen* gewährleistet werden [BKPS04, S. 265; LSR07, S. 309f.; Kru01; PBL05, S. 437]. Mit der Erweiterung eines frei verfügbaren SPLE-Werkzeugs leistet diese Arbeit einen Beitrag für die Weiterentwicklung einer integrierten Gesamtlösung (vgl. *Kapitel 5*).

## 2.2.2 Einführungsstrategien und Transitionsprozesse

Entschließt sich ein Unternehmen dazu, seine Produkte als SPL zu entwickeln, so muss zu allererst geklärt werden, welche Einführungsstrategie verfolgt werden soll. Krueger unterscheidet hierbei zwischen den drei grundsätzlichen Ansätzen „*proaktiv*“, „*reaktiv*“ sowie „*extraktiv*“ und Mischformen aus diesen [Kru01]. *Tabelle 2.1* stellt die grundlegenden Strategien überblicksartig gegenüber. Die zeilenweise Einordnung gibt Aufschluss darüber wie die Plattform über eine gewisse Zeitspanne und den jeweiligen Umfang gesehen aufgebaut wird. Die Spalten grenzen die Einführungsstrategien hinsichtlich des Vorhandenseins von Altsystemen ab. Die Entscheidung welche Strategie zu wählen ist hängt nicht zuletzt von ihrem Aufwand, Risiko, ihren Kosten und Chancen ab [Kru01]. In Hinblick auf diese Kriterien werden im Folgenden die drei atypischen Strategien näher erläutert.

<table>
<tr><th></th><th>Keine existierenden Systeme</th><th>Existierende Systeme</th></tr>
<tr><td>Schlagartig</td><td>proaktiv</td><td rowspan="2">extraktiv</td></tr>
<tr><td>Inkrementell</td><td>reaktiv</td></tr>
</table>

Tabelle 2.1: Atypische Einführungsstrategien

Die **proaktive Strategie** entspricht dem Wasserfallmodell der klassischen Softwareentwicklung, angepasst an die Produktlinienperspektive. Das heißt, die Zieldomäne wird analysiert, die Variabilität modelliert, die Artefakte in Hinblick auf *alle* vorhersehbaren Produktvarianten entworfen und implementiert. Dieses Vorgehen erfordert massive Vorabinvestitionen in finanzieller, personeller sowie organisatorischer Hinsicht. Aufgrund dessen entsteht ein erhebliches Risikopotenzial (*vgl. Kapitel 2.2.1*). Bei einer optimalen Umsetzung, können aber im Anschluss die eigentlichen Produkte schnell generiert und an die Kunden gebracht werden. Die proaktive Strategie, zahlt sich vor allem dann aus, wenn die Anforderungen an die SPL genau bestimmt werden können und sich jene in der Zukunft nur kaum verändern werden [FK05; Kru01; SEI10].

Im Rahmen der **reaktiven Strategie** wird die Plattform genau dann erweitert, wenn sich während der Produktentwicklung Wiederverwendungsmöglichkeiten ergeben. Zu Beginn wird demnach *nur eine Teildomäne* des gesamten Problembereiches berücksichtigt. Über die Zeit hinweg erfolgt die inkrementelle Erweiterung der Plattform, wodurch die Anzahl der generierbaren Produktvarianten wächst. Dieses Vorgehen sollte vor allem dann gewählt werden, wenn nicht alle Features der SPL vorab bestimmt werden können. Eine solche Strategie ist insofern vorteilhaft, da zu Beginn weniger Ressourcen gebunden werden müssen und auch (Teil-)Ergebnisse schneller sichtbar sind. Allerdings können die Gesamtkosten dennoch unverhältnismäßig höher steigen, wenn die grundlegende Basisplattform, welche die Artefakte schrittweise aufnimmt, nicht adäquat entworfen wurde [FK05; Kru01].

Besitzt ein Unternehmen bereits eigene Produkte, mit vielen Gemeinsamkeiten und nur wenigen Unterschieden, so bietet sich die **extraktive Strategie** an. Hierbei werden *aus den bestehenden Produktressourcen* die Artefakte für die *Plattform abgeleitet* bzw. extrahiert. In diesem Fall kann das Vorgehen sowohl *inkrementell*, z.B. geordnet nach dem Gewinn den die Produkte erwirtschaften, als auch *für alle* Produkte *schlagartig* erfolgen. Unabhängig davon wann und welche Produkte überführt werden, müssen zwei wesentliche Herausforderungen bewältigt werden. Einerseits müssen Systeme, welche nicht mit dem Fokus der Wiederverwendung konzipiert wurden, in flexibel einsetzbare, abgeschlossene und robuste Artefakte zerlegt werden [OS02]. Andererseits muss meist mit bestehenden Ressourcen parallel zu der Einführung der vorhandene Produktivbetrieb des Unternehmens erhalten werden [BKPS04, S. 153]. Aufgrund dessen sind *Werkzeuge und Konzepte*, welche eine zuverlässige und schnelle Transition ermöglichen, von außerordentlicher Bedeutung. Wenn die vorhandene Expertise des Unternehmens effektiv eingesetzt wird, kann die

Transition zur SPLE mit geringem Aufwand, Risiko und mit verhältnismäßig geringen Vorabinvestitionen vollzogen werden [FK05; Kru01].

Obwohl aus theoretischer Sicht die proaktive und reaktive Einführungsstrategie durchaus sinnvolle Optionen darstellen, ist für die *industrielle Praxis* eher das *extraktive Vorgehen* interessant [BOS00]. Der Grund dafür ist, dass ohne genaue Kenntnis des Marktes, der Kunden sowie detaillierten technischen Realisierungsmöglichkeiten die Modellierung und Umsetzung einer SPL nahezu unmöglich ist. In der Regel bietet sich dieses neue Entwicklungsparadigma daher vor allem für Unternehmen an, welche auf der einen Seite über entsprechende Erfahrungen durch die Entwicklung von eigenen Produkten verfügen, auf der anderen Seite aber unter gewissem Druck stehen ihre Software-Lösungen wirtschaftlicher produzieren und warten zu müssen [BKPS04, S. 137; BOS00; Joh06; PBL05, S. 394]. Ein erneuter Beginn auf der „grünen Wiese" kommt für diese Zielgruppe meist aus zwei entscheidenden Gründen nicht in Frage. Zum einen sind enorme Vorabinvestitionen und massive Ressourcenbindungen mit dem normalen Produktivbetrieb kaum vereinbar. Zum anderen besteht ein sehr hohes Risiko wertvolles Unternehmenswissen zu verlieren, welches über Jahre hinweg in die Software-Lösungen eingebunden wurde [BKPS04, S. 8; CK02; ES01; Joh06; Kru01; MTW93; SE02; SEI10]. Im Rahmen der extraktiven Einführung ist es daher ratsam vorhandene Altsystem-Ressorcen *nahezu vollständig* aber gleichzeitig mit *möglichst geringem Aufwand* für die Plattform zu *übernehmen* [BKPS04, S. 10; Joh06; SEI10]. Nach einer solchen extraktiven Einführung könnte sich dann aber ein reaktives Vorgehen anschließen [Kru01].

Nachdem sich ein Unternehmen, basierend auf einer Kosten-Nutzen-Risiko-Rechnung, bewusst für eine SPLE-Einführungsstrategie entschließt, muss der genaue Transitionsprozess beschrieben werden. Ergebnis dieses Arbeitsschrittes sollte ein Plan sein, der detailliert vorgibt, wie eine Organisation von ihrem aktuellen Zustand der Produktentwicklung zu einer Produktlinienentwicklung überführt werden soll [BKPS04, S. 139; SEI10]. Leider gibt es aufgrund der Vielzahl möglicher Faktoren die auf ein Unternehmen einwirken keinen allgemeinanwendbaren Transitionsplan [BKPS04, S. 139; LSR07, S. 139]. Allerdings geben einige Autoren ein grobes, generisches Muster vor, welches sich für die spezielle Situation eines Unternehmens anpassen lässt.

Schreiber [BKPS04, S. 140f.] spannt beispielsweise einen Raum von Schritten und inhaltlichen Schwerpunkten auf. Die vier Hauptschritte *„Analyse & Assessment"*, *„Planung & Improvement"*, *„Implementierung"* sowie *„Institutionalisierung"* werden jeweils unter dem Blickwinkel der inhaltlichen Schwerpunkte *„Strategie & Scope"*, *„Architektur"* sowie *„Prozess & Organisation"* verfeinert. Das Carnegie Mellon Software Engineering Institute schlägt hingegen das Technologiewechselmodell *„IDEAL"* vor, welches für die fünf Hauptphasen *„Initiating"*, *„Diagnosing"*, *„Establishing"*, *„Acting"* und *„Learning"* steht. Dieses Modell beschreibt einen iterativen Prozess, bestehend aus insgesamt vierzehn Aktivitäten, mit dem Technik jeglicher Art in eine Organisation eingeführt werden kann

[McF96]. Spezielle SPL-bezogene *Produkt-*, *Organisation-* und *Prozessaspekte* werden zusätzlich durch eigene Aktivitäten und Methodenbereichen behandelt, welche in dem *„Adoption Factory Pattern"* definiert sind [CN02, S. 393ff.; SEI10]. Hinsichtlich der Darstellung von bislang vorgeschlagenen Transitionsprozess-Mustern wird keineswegs der Anspruch auf Vollständigkeit erhoben. Die beiden genannten Ansätze sollen lediglich darauf hinweisen, dass eine Einführung der SPLE auf mehreren Ebenen geplant, organisiert und unterstützt werden muss.

In Hinblick auf die praktische Relevanz wird in dieser Arbeit ein Konzept erarbeitet und vorgeschlagen, welches die extraktive Einführungsstrategie (semi-)automatisch unterstützt. Hinsichtlich der vorgestellten Transitionsprozess-Muster ordnet sich jenes Konzept auf der Architektur- bzw. der Produktentwicklungsebene ein (vgl. *Kapitel 4*).

### 2.2.3 Rahmenwerk des Betriebs

Der Wechsel von der traditionellen Einzelsystem- zu der Produktlinienentwicklung impliziert auch tiefgehende Veränderungen hinsichtlich der Ausrichtung des gesamten Unternehmens (vgl. *Kapitel 2.2.1*; *2.2.2*). Während die bisherigen Ansätze vor allem von einer opportunistischen Wiederverwendung für einzelne Systeme bestimmt waren, verlangt die SPLE ein Vorgehen mit organisierter Variabilität und gemanagter Wiederverwendung für eine Gruppe von Softwaresystemen. Dieses neue, strategische Paradigma basiert vor allem auf der Unterscheidung der Hauptaktivitäten *Domain Engineering* (Entwicklung für Wiederverwendung), *Application Engineering* (Entwicklung mit Wiederverwendung) und *Management* (Steuerung der Wiederverwendung). Umgesetzt werden diese Aktivitäten durch detaillierte Methoden und Prozesse aus dem Bereich des *Software-Engineering* sowie dem *Technical-* und *Organizational-Management* (vgl. *Kapitel 2.1*). Die Methodenbereiche und Hauptaktivitäten bilden zusammen das Rahmenwerk der Software-Produktlinienentwicklung [BKPS04; SEI10]. Im Folgenden wird für ein genaueres Verständnis der Aktivitäten, Artefakte und Informationsflüsse innerhalb der SPLE, das Rahmenwerk näher beschrieben und in *Abbildung 2.3* visualisiert.

Das **Domain Engineering** wird durch das **Product Management** angestoßen. Im Hinblick auf ökonomische Aspekte und vorhandene Kundenanforderungen werden Produktvarianten geplant und zu einer *Roadmap* zusammengefasst. Die hierbei identifizierten gemeinsamen und unterschiedlichen Features der Produkte definieren die Variabilität der zu entwickelnden SPL. Aufbauend auf der Roadmap werden im **Domain Requirements Engineering** die konkreten Anforderungen an die Produkte analysiert und in Text- oder Modellartefakten festgehalten. Ein weiteres wesentliches Ergebnis dieses Teilprozesses ist das *Variabilitätsmodell*. In jenem wird die bereits vorhandene Feature-Liste um Abhängigkeiten und Beschränkungen erweitert. In allen erzeugten Artefakten der folgenden Phasen muss dann diese definierte Variabilität entsprechend umgesetzt werden.

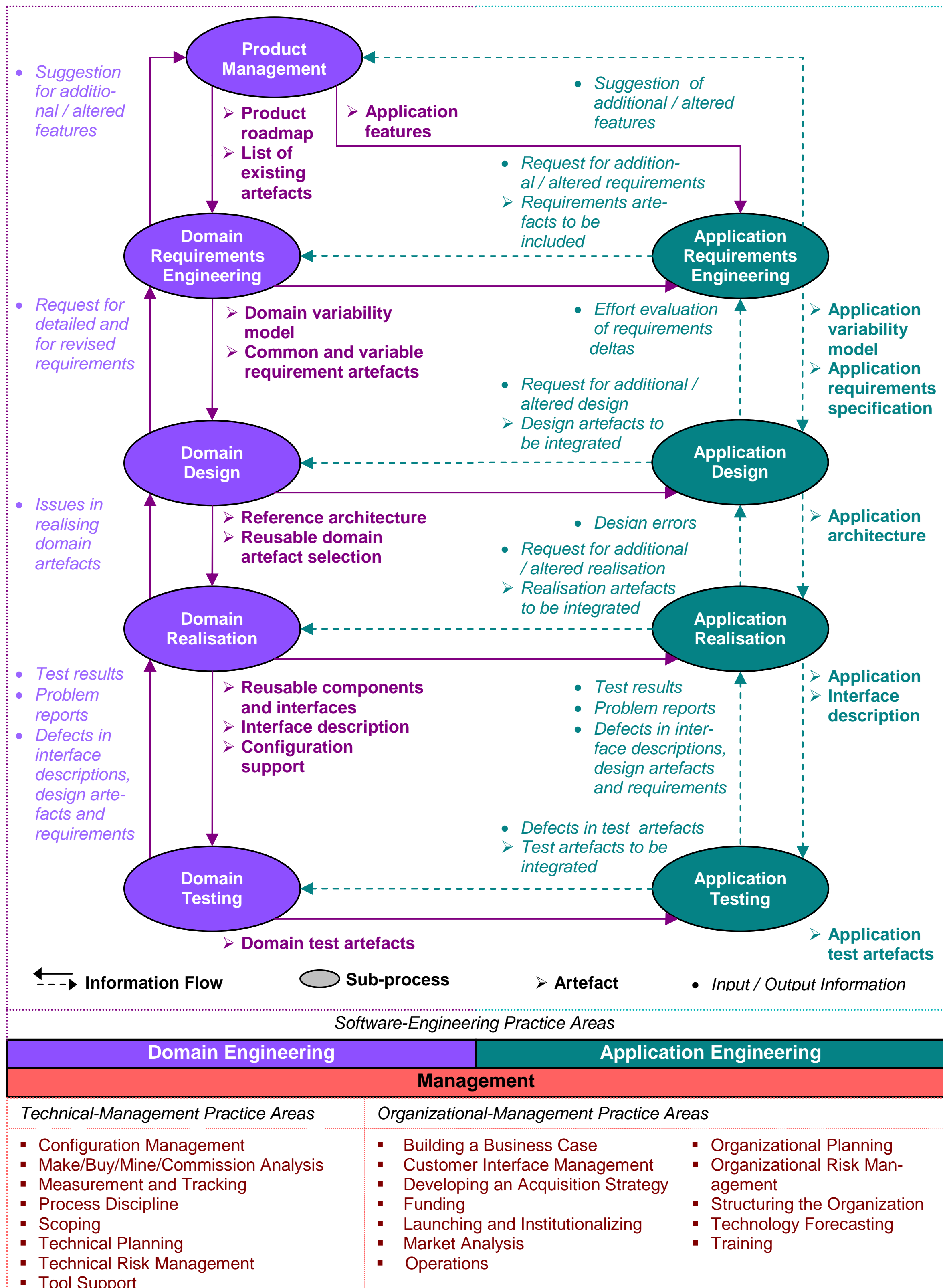



Abbildung 2.3: Rahmenwerk der SPLE im Detail, konsolidiert aus [BKPS04] und [SEI10]

In dem **Domain Design** wird die *Referenzarchitektur* mit Hilfe von Struktur- und Verhaltensdiagrammen für alle Produktvarianten entworfen. Weiterhin werden die *Selektions- bzw. Konfigurationsmechanismen* sowie die *zu implementierenden Softwareartefakte* bestimmt. Die eigentliche Bereitstellung der *Produktgenerierungsmechanismen* sowie der *ausführbaren Quellcode-Artefakte* erfolgt während des **Domain Realisation** Teilprozesses. Die variabilitätsumsetzenden Strukturen können hierbei selbstständig implementiert oder aus bestehenden Anwendungen extrahiert werden. Während des extraktiven Einführungsprozesses, der meist fließend in den normalen Produktivbetrieb übergeht, spielt gerade der zuletzt genannte Fall eine entscheidende Rolle. Mögliche Optionen sind aber auch, der Einkauf von Standardkomponenten oder die Auslagerung der Realisierungsaktivitäten an externe Dienstleister. Das **Domain Testing** zielt darauf ab die generischen, wiederverwendbaren Artefakte zu validieren und entsprechende Testpläne bereitzustellen (vgl. *Abbildung 2.3*) [BKPS04, S. 278; LSR07, S. 15; PBL05, S. 24ff.].

Im Rahmen des **Application Engineering** werden die konkreten Produkte entwickelt bzw. konfiguriert. Anhand einer Feature-Selektion werden alle notwendigen Artefakte zur Wiederverwendung bereitgestellt. In der Regel deckt die Plattform schon den größten Anteil des finalen Produktes ab, sodass bereits ein erster Prototyp erzeugt werden kann. Der fehlende Teil wird, analog zum Domain Engineering, in den Phasen **Application Requirements Engineering**, **Application Design** und **Application Realisation** entwickelt und in dem **Application Testing** validiert. Die hierbei entstandenen Artefakte werden in die Plattform eingegliedert und stehen dann für die Entwicklung weiterer Systeme zur Verfügung. Während der Entwicklung der einzelnen Produkte werden darüber hinaus alle Verbindungen zwischen den Artefakten der Plattform und den genutzten Instanzen dokumentiert. In der Wartung und Evolution können aufgrund dessen Veränderungen der Plattform effizient propagiert werden. (vgl. *Abbildung 2.3*) [BKPS04, S. 277; LSR07, S. 15f.; PBL05, S. 30ff.].

Die Software-Produktlinienentwicklung umfasst nicht nur die bereits beschriebenen Seiten des **Software-Engineering**, sondern vor allem auch das **Technical-** und **Organizational-Management**. Auch diese managementbezogene Ausrichtung unterscheidet die SPLE von klassischen ad-hoc Wiederverwendungsansätzen [BKPS04, S. 279]. Durch Methoden, Prozesse und Aktivitäten des *Organizational-Management* werden die gesamten Entwicklungsanstrengungen erst ermöglicht und orchestriert. Es werden hierbei nicht zuletzt die Produkt- und Finanzierungsstrategie erarbeitet, die Organisationsstruktur bestimmt und die notwendigen Ressourcen festgelegt. Weiterhin erfolgen innerhalb dieses Aufgabenfeldes die Koordination wesentlicher technischer Aktivitäten sowie die Kommunikation mit Kunden und Lieferanten. Aufgrund der Bedeutung der bearbeiteten Aufgaben kann festgehalten werden, dass dieser Managementbereich für den Erfolg oder das Scheitern des gesamten Vorhabens maßgeblich verantwortlich ist. Durch das *Technical-Management* wird hingegen das *Software-Engineering*, d.h. also die Entwicklung und Evolution aller Artefakte und Produkte, organisiert und überwacht. Zudem werden Entwicklungsme-

thoden festgelegt und alle projektmanagement-bezogenen Aufgaben übernommen (vgl. *Abbildung 2.3*) [CN02, S. 55, S. 151, S. 219; SEI10].

Der Fokus dieser Arbeit liegt vor allem auf dem Gebiet des Software-Engineering. Daher wird an dieser Stelle darauf verzichtet, die jeweiligen Management-Bereiche genauer zu besprechen. Der interessierte Leser sei hierfür auf das *„Framework for Software Product Line Practice, Version 5.0"* [4] des Carnegie Mellon Software Engineering Institute verwiesen.

## 2.3 Modellierung und Repräsentation von Variabilität

Die Software-Produktlinienentwicklung unterstützt die Entwicklung und Wartung von einer Gruppe von Softwareprodukten. Jene Produkte sind für verschiedene Kunden oder sogar unterschiedliche Marktsegmente bestimmt. Aufgrund von abweichenden Anforderungen, Gesetzgebungen oder technischen Einschränkungen ergeben sich Unterschiede hinsichtlich der Features der Systeme. Diese Abweichungen innerhalb der SPL determinieren die *Variabilität* (vgl. *Kapitel 2.1*). Jene muss exakt definiert, modelliert, innerhalb aller Artefakte repräsentiert und über die Zeit hinweg erweitert werden [LSR07, S. 8ff.]. Aufgrund der zentralen Bedeutung von Variabilität innerhalb der SPLE [BKPS04, S. 13], werden wichtige Grundlagen an dieser Stelle genauer diskutiert. In *Kapitel 2.3.1* wird sich der Modellierung von Features gewidmet, wohingegen *Kapitel 2.3.2* konkrete Konzepte zur Repräsentation von Features in Quellcode-Artefakten beschreibt. Diese zwei Bereiche entsprechen nur einem ausgesuchten Ausschnitt aus dem gesamten Variabilitätsmanagement, für das Verständnis dieser Arbeit stellt diese Auswahl jedoch einen hinreichenden Einblick in die Gesamtthematik dar.[5]

### 2.3.1 Feature-Modelle und ihre Darstellungsweisen

Während des Domain Engineering wird die Variabilität der geplanten Software-Produktlinie erfasst und in einem Variabilitätsmodell festgehalten. Dieses Modell bildet die zentrale Kommunikationsgrundlage für die Umsetzung der Plattform. Gleichzeitig dient es aber auch im Application Engineering als Vorschrift für alle generierbaren Produktvarianten (vgl. *Kapitel 2.2.3*) [BKPS04, S. 13; PBL05, S. 58f.]. In der Vergangenheit wurden zahlreiche Variabilitätsmodelle vorgestellt [CBA09; LSR07, S. 9ff.; PBL05, S. 73ff.].

---

[4] http://www.sei.cmu.edu/productlines/frame_report/index.html

[5] Als Einstieg für eine ausführliche Betrachtung der Variabilität innerhalb der SPLE ist das Buch *„Software Product Line Engineering – Foundations, Principles, and Techniques"* von Pohl, Böckle und van der Linden [PBL05] zu empfehlen.

Hiervon haben sich in der Forschung und Praxis die so genannten **Feature-Modelle** als De-facto-Standard etabliert [AK09; Joh06; LSR07, S. 8ff.]. Sie wurden erstmalig von Kang et al. [KCH+90] in dem *„Feature-Oriented Domain Analysis"*-Vorgehensmodell eingeführt. Zur Erhöhung ihrer Ausdrucksfähigkeit wurden sie mehrfach in anderen Publikationen erweitert [AK09; CBA09]. In dieser Arbeit wird in Anlehnung an Czarnecki und Eisenecker [CE00, S. 38f.] ein Feature-Modell definiert als eine Menge von Features einer Domäne und ihren Abhängigkeiten, sowie optionalen detaillierten Erläuterungen zu allen Elementen. Die Auswahl einer Teilmenge der Features beschreibt dabei eine *Variante*. Erfolgt die Auswahl der Features konform zu allen definierten Abhängigkeiten und Beschränkungen, so bezeichnet man die Variante als *gültig* [Käs10].

Feature-Modelle werden graphisch repräsentiert durch **Feature-Diagramme** [KCH+90]. Die hierarchische Ordnung aller Features innerhalb dieser Diagramme lässt sich folgendermaßen interpretieren. Wenn ein Kind-Feature in einer Variante enthalten ist, so muss sein Eltern-Feature ebenfalls Teil der Variante sein. Grundsätzlich gibt es noch drei weitere Abhängigkeitstypen zwischen einem Elternknoten und seinen Kindern. Bei der Auswahl des Elternknotens in einer

*Und*-Gruppe, müssen alle *obligatorischen* Kinder auch ausgewählt werden. *Optionale* Kinder hingegen können ausgewählt werden.

*Alternativ*-Gruppe, muss *genau ein* Kind ausgewählt werden.

*Oder*-Gruppe, muss *mindestens ein* Kind ausgewählt werden.

*Notwendigkeits* (ein Feature benötigt ein anderes)- oder *Ausschlussbeziehungen* (ein Feature darf nicht mit einem anderen auftreten), die unabhängig von der Baumstruktur sind, können darüber hinaus durch gestrichelte Querverbindungen oder aussagenlogische Ausdrücke abgebildet werden [CE00, S. 87ff.; Käs10; TBK09].

*Abbildung 2.4* stellt ein Feature-Diagramm mit verschiedenen Abhängigkeitstypen an einem kleinen Beispiel dar.[6] Gezeigt wird eine Chat-Anwendung bestehend aus neun Features. Laut dem Modell muss jede gültige Variante die Features CHAT, BASIS und AUSGABE enthalten. Darüber hinaus muss genau eine Ausgabeart definiert werden, d.h. entweder GUI oder KONSOLE. Die Auswahl einer BENUTZERVERWALTUNG ist optional, wobei eine von der Baumstruktur unabhängige Ausschlussbeziehung eingrenzt, dass jenes nur dann gilt, wenn auch eine GUI zur Verfügung steht. Varianten die fakultativ eine VERSCHLUESSELUNG von Textnachrichten unterstützen, müssen zudem mindestens ein spezifisches Verfahren implementieren. Hierbei stehen die Features CAESAR oder RUECK-

[6] Die in dieser Arbeit verwendeten Feature-Diagramme wurden mit dem Eclipse-Plugin *FeatureIDE* [KTS+09] erstellt. Die Syntax orientiert sich dabei an den Vorgaben von Kang et al. [KCH+90] sowie von Czarnecki und Eisenecker [CE00, S. 87ff.].

WAERTS zur Auswahl. Insgesamt können aus diesem Feature-Modell bzw. Feature-Diagramm zwölf gültige Chat-Varianten erzeugt werden.

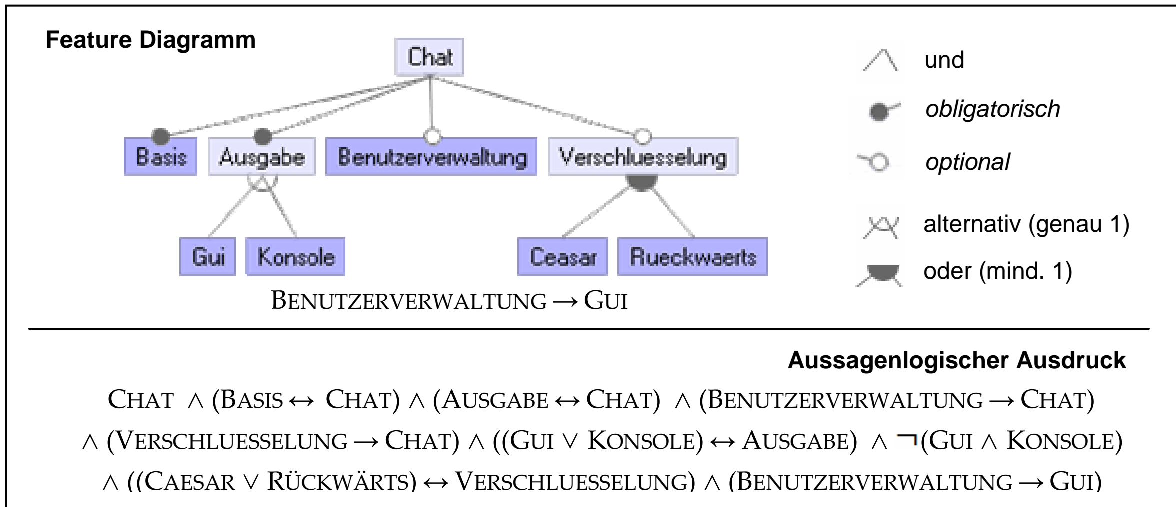



Abbildung 2.4: Feature-Modell als Diagramm und aussagenlogischer Ausdruck

In der Praxis können Feature-Modelle hunderte oder tausende Features enthalten [Käs10; TBK09]. Für eine automatisierte Unterstützung der Auswahl von gültigen Varianten, wie z.B. in einem Produktkonfigurator, ist es hilfreich Feature-Modelle bzw. -Diagramme in **aussagenlogische Ausdrücke** zu überführen. *Abbildung 2.4* zeigt zu dem beschriebenen Chat-Beispiel exemplarisch einen korrespondierenden aussagenlogischen Ausdruck [Bat05; Man02].[7] Für jedes Feature wird hierbei eine Boolsche-Variable definiert. Jene wird mit *wahr* belegt, wenn das Feature selektiert ist, andernfalls wird ihr der Wert *falsch* zugewiesen. Für gültige Feature-Selektionen nimmt der gesamte Ausdruck immer den Wert *wahr* an. Während der Konfiguration einer Variante kann diese Eigenschaft genutzt werden, um basierend auf der aktuellen Selektion automatisch zu ermitteln, ob die bisherige Auswahl gültig ist, welche weiteren Features noch selektiert werden können bzw. müssen oder auch nicht mehr selektiert werden dürfen.[8] Hierbei kann für eine effiziente Bearbeitung auf bereits verfügbare und erprobte Werkzeuge zur Lösung des Erfüllbarkeitsproblems in der Aussagenlogik (sog. *SAT Solvers*) zurückgegriffen werden [Bat05]. In dieser Arbeit wird eine solche Funktionsweise von Produktkonfiguratoren adaptiert, um aus einem beliebigen Feature-Diagramm Notwendigkeits- oder Ausschlussbeziehungen, die zwischen Features bestehen, auszulesen (vgl. *Kapitel 5.2.2).*

[7] Überführungsregelen von Feature-Diagrammen, über Grammatiken, zu aussagenlogischen Ausdrücken werden beispielsweise von Batory [Bat05] oder Thüm et al. [TBK09] diskutiert.

[8] Darüber hinaus können mit Hilfe von aussagenlogischen Ausdrücken noch zahlreiche andere Fragestellungen beantwortet werden – vgl. hierzu Auflistung in [AK09] oder [Käs10].

### 2.3.2 Repräsentation von Features in Quellcode-Artefakten

Damit einzelne Produktvarianten aus der Plattform abgeleitet werden können, muss die modellierte Variabilität (vgl. *Kapitel 2.3.1*) in den Quellcode-Artefakten repräsentiert werden. Die hierfür verwendeten Konzepte und Mechanismen müssen sicherstellen, dass alle Features flexibel auswählbar und komponierbar sind [CRB04]. In erster Annäherung zu diesem Ziel, ist es erforderlich den Quellcode so zu strukturieren, dass Fragmente der gemeinsamen Basis sowie die jeweiligen Features von einander abgegrenzt sind [Pre97]. Erst darauf aufbauend kann ein konzeptuelles Feature zu seiner Repräsentation zugeordnet und separat behandelt werden.

Der Wunsch Software so in handhabbare Einheiten bzw. **Module**[9] zerlegen zu können, dass sie sich zu **semantisch zusammenhängenden Belangen**[10] zuordnen lassen, ist fast so alt wie das Software-Engineering selbst. Schließlich ließe sich mit der vollständigen Umsetzung dieses so genannten *Seperation of Concerns (SoC)* Ansatzes [Par72; Dij76] die Verständlichkeit, Wartbarkeit, Wiederverwendbarkeit und nicht zuletzt die Maßschneiderbarkeit von Software verbessern [Ape07; IBM09]. Allerdings besagt die *Tyrannei der dominanten Dekomposition*, dass in traditionellen Paradigmen, wie z.B. auch in der Objektorientierten Programmierung, eine vollständige Modularisierung aller Belange zur gleichen Zeit unmöglich ist. Grund dafür ist nicht zwangsläufig ein inadäquater Entwurf, sondern meist die gegebene Dekompositionsbeschränktheit der jeweiligen Konzepte [TOHS99]. Modulare Einheiten wie Funktionen und Klassen werden stets von anderen Belangen, wie z.B. Logging, Fehlerbehandlung oder Synchronisierung, quer geschnitten. Solche **querschneidenden Belange** sind typischerweise über die gesamte Anwendung *verstreut* (engl. code scattering) – zum Teil *auch als Kopien (*engl. code replication*)*. Außerdem *vermischen* sie sich in verschiedenen Modulen mit anderen Belangen (engl. code tangling). Für Softwareentwickler ist es unter diesen Umständen sehr schwierig einen Überblick über die Zuordnung zwischen Belangen und den entsprechenden Implementierungseinheiten zu behalten [KLM+97; TOHS99]. Für die effiziente Dekomposition und Komposition von Features in Quellcode-Artefakten werden daher neue Konzepte, Methoden, Formalismen und Werkzeuge benötigt [Ape07].

Innerhalb der akademischen Forschung werden für Software-Produktlinien vor allem solche **Kompositionsansätze** bevorzugt, die *Features vollständig in physisch getrennten Mo-*

---

[9] Als ein Modul bezeichnet man einen abgeschlossenen Programmbaustein, welcher Informationen hinter einer Schnittstelle verbirgt und verwandte Konstrukte gruppiert. Daher lassen sich Module auch separat entwickeln und kompilieren [Ape07; Par72].

[10] Semantisch zusammenhängende Belange sind z.B. Features, Anforderungen, Designentscheidungen oder Datenstrukturen. Grundsätzlich können aber hierzu jegliche Fragestellungen gezählt werden, die innerhalb eines Problembereichs von Interesse sind [Ape07].

*dulen* implementieren. Produktvarianten werden bei diesem Vorgehen generiert indem ausgewählte (Feature-)Module zur Übersetzungs- oder Startzeit, z.B. mit speziellen Compilern, zusammengesetzt werden [KAK08; Käs10]. Einige bekannte Vertreter dieser Technologiengruppe sind beispielsweise Frameworks [JF88], Aspektorientierte Programmierung [KLM+97], Featureorientierte Programmierung [Pre97; BSR04; AKL09b], Aspektuelle-Feature-Module [ALS08], Komponentenorientierte Programmierung [HC01], Themenorientierte Programmierung [HO93] sowie Hypermodule [TOHS99; TO01].

Kompositionsansätze spielen ihre Stärken besonders dann aus, wenn neue Software mit großen, wiederverwendbaren Codeeinheiten proaktiv oder reaktiv (vgl. *Kapitel 2.2.2*) entwickelt werden muss. Durch die physische Umsetzung des SoC-Ansatzes können Features ihren Quellcode-Repräsentationen direkt zugeordnet werden. Hierdurch sind die Features in Quellcode-Artefakten besonders gut umsetzbar, nachvollziehbar und wartbar. Problematisch sind solche Ansätze hingegen dann, wenn Variabilität feingranular abzubilden ist, d.h. auf der Ebene von Parametern, Anweisungen und lokalen Variablen. Bei der extraktiven Überführung von bestehenden Altsystemen zu einer SPL-Plattform (vgl. *Kapitel 2.2.2*), muss bei Kompositionsansätzen die Architektur und somit auch der Quellcode sehr stark abgeändert werden. Zudem müssen die Entwickler neue Sprachen bzw. Spracherweiterungen lernen, für die es zudem momentan kaum ausgereifte Werkzeugunterstützungen gibt. Damit ist neben enormen Investitionen vor allem auch ein erhöhtes Fehlerrisiko in der Entwicklung verbunden (vgl. *Kapitel 2.2.1*). Aufgrund dessen finden jene Ansätze in der industriellen Praxis, die vom extraktiven Vorgehen geprägt ist (vgl. *Kapitel 2.2.2*), nur bedingt Anwendung [AKL09a; Käs10; KTA08].

*Abbildung 2.5* visualisiert noch einmal schematisch das Vorgehen der Produktvariantengenerierung mit Hilfe von Kompositionsansätzen und fasst stichpunktartig die daraus resultierenden Vor- und Nachteile zusammen.

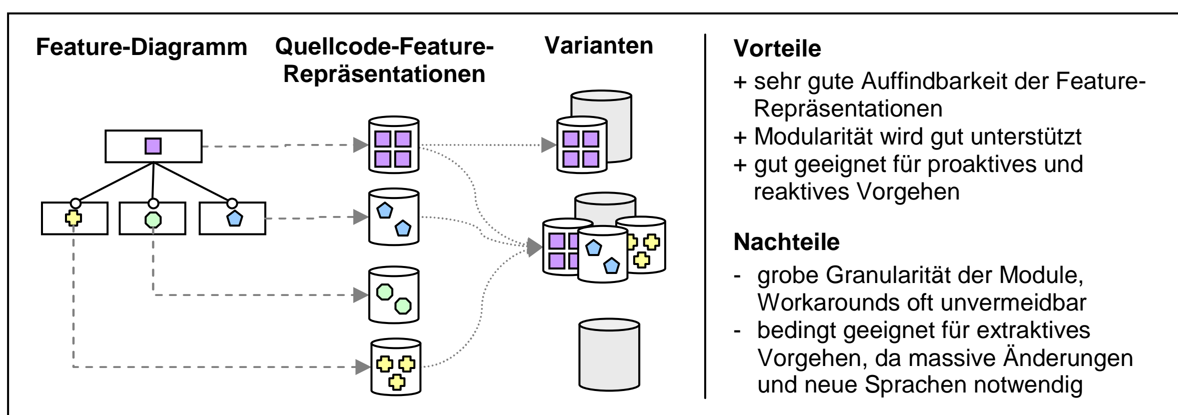



Abbildung 2.5: Kompositionsansatz im Überblick

In **Annotationsansätzen** werden Feature-Repräsentationen innerhalb einer Codebasis mit Hilfe von textuellen oder graphischen Markierungen ausgezeichnet. Bei der Erzeugung von Produktvarianten werden alle annotierten Quellcodeelemente entfernt, deren korrespondierende Features nicht Teil der Selektion sind. In dem aktuellen industriellen Umfeld werden meist Präprozessor-Anweisungen zur textuellen Markierung der Quellcodebasis genutzt. Solche Anweisungen gehören in einigen Sprachen, wie z.B. C, Fortran, Pascal oder Visual Basic, zu dem Standardrepertoire. Für Programmiersprachen, die keine Präprozessoren enthalten, wurden spezielle Tools und Erweiterungen entwickelt [KAK08; Käs10]. Antenna[11] und Munge[12] sind hierbei Beispiele für Java. Eine sprachenunabhängige Lösung bieten unter anderem die Erweiterung GGP[13] oder XVCL [JBZZ03] sowie die kommerziellen SPL-Werkzeuge Gears von BigLever [Kru01] und pure::variants von pure::systems [BPS04].

Bei dem Einsatz von Präprozessoren findet keine physische Modularisierung statt, wie sie das klassische SoC fordert. Ein unübersichtlicher Quellcode mit schwer zu wartenden querschneidenden Belangen und Features ist üblicherweise die Folge. Kästner [Käs10] schlägt daher eine SPLE mit einer neuen Generation von Präprozessoren vor, welche eine *virtuelle Trennung von Belangen* ermöglichen. Die wesentliche Erweiterung begründet sich darin, dass mit Hilfe von Werkzeugunterstützung zusätzliche Sichten auf einzelne Feature oder Varianten verwaltet werden. Durch die Verwendung von disziplinierten Annotierungen und einem produktlinienbewussten Typsystem wird zudem die Konsistenz aller Feature-Repräsentationen sichergestellt. In dem Forschungsprototyp *CIDE*[14] ist dieser neue Ansatz bereits programmiersprachenunabhängig umgesetzt wurden. CIDE verwendet darüber hinaus statt textuellen Annotationen, farbliche Markierungen, um die Struktur des Quellcodes nicht zu verändern und somit die Lesbarkeit zu erhöhen [KAK08; KTA08].

Annotationsansätze können nicht zuletzt aufgrund der virtuellen Trennung von Belangen gleichwertige Alternativen zu Kompositionsansätzen darstellen. Bei entsprechender Umsetzung können sie durch einen einfachen, flexiblen, feingranularen und programmiersprachenunabhängigen Einsatz bestechen. Vorteilhaft ist weiterhin, dass Altsysteme durch geringfügige Modifikationen zu einer SPL-Plattform überführt werden können. Aufgrund dessen eignen sich Annotationsansätze insbesondere für extraktive Transitionsvorhaben (vgl. *Kapitel 2.2.2*). Obwohl durch Werkzeugunterstützung ebenfalls eine gute Verfolgbarkeit von Features in Quellcode-Artefakten realisiert werden kann, erreicht

[11] http://antenna.sourceforge.net/

[12] http://weblogs.java.net/blog/tball/archive/2006/09/munge_swings_se.html

[13] http://en.nothingisreal.com/wiki/GPP

[14] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/

die emulierte Modularität nicht alle Vorteile ihres physischen Pendants. Beispielweise können Features in der Regel nicht separat kompiliert, getestet, parallel entwickelt oder als Black-Box-Konstrukte wiederverwendet werden [Käs10].

In *Abbildung 2.6* werden wesentliche diskutierte Aspekte des Annotationsansatzes erneut graphisch aufgegriffen und überblicksartig dargestellt. Hierbei wird speziell hervorgehoben, dass in sich geschlossene Module durch querschneidende Feature-Repräsentationen ersetzt werden, welche durch Werkzeugunterstützung adressiert werden können.

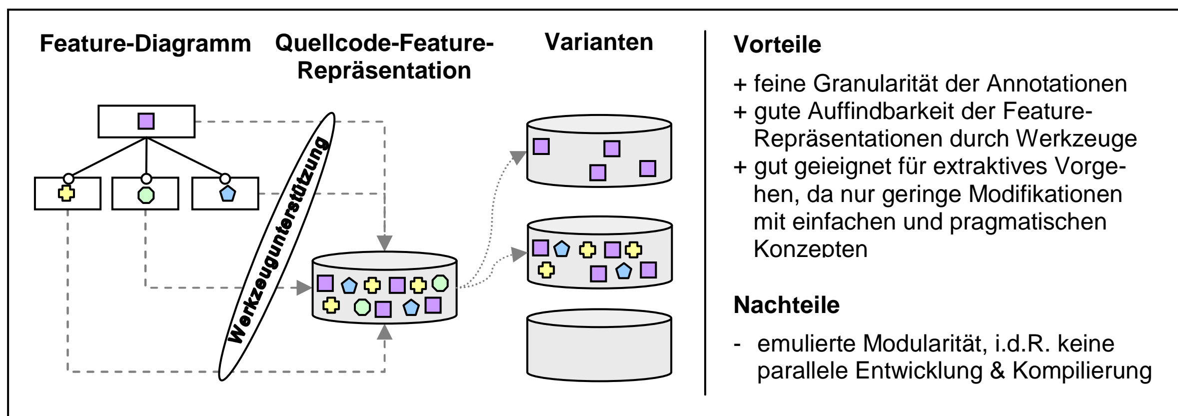



Abbildung 2.6: Annotationsansatz im Überblick

Grundsätzlich kann nicht entschieden werden, welcher der beiden Ansatzformen besser geeignet ist, da sie viele Gemeinsamkeiten und komplementäre Vorteile aufweisen. Eine Integration zu einem kombinierten Ansatz, während der Etablierung einer SPLE in einem Unternehmen, wäre daher die logische Folge. Hierbei könnten durch den pragmatischen Annotationsansatz bestehende Altsysteme mit geringem Aufwand und minimalen Veränderungen initial in eine SPL-Plattform überführt werden [AKL09a]. Langfristig ist eine inkrementelle (automatisierte) Transformation der annotierten Features in physische Module denkbar [KAK09]. Ein solches kombiniertes Vorgehen würde die Vorabinvestitionen und Einführungsrisiken mindern (vgl. *Kapitel 2.2.1*), auf lange Sicht gesehen aber dennoch eine hohe Softwarequalität und Architekturstabilität gewährleisten [Käs10].[15] In Einklang mit jener Erkenntnis wird in dieser Arbeit ein Konzept erarbeitet, welches das annotationsbasierte Werkzeug CIDE so erweitert, dass der Transitionsaufwand während eines extraktiven Einführungsvorhabens weiter gemindert werden kann (vgl. *Kapitel 5*).

[15] Kästner [Käs10] gibt in seiner Dissertation einen umfassenden Überblick und Vergleich von Annotations- und Kompositionsansätzen, sowie Konzepten die sich nicht eindeutig in eine der beiden Kategorien einordnen lassen und kommt dabei unter anderem zu dem beschriebenen Ergebnis.

## 2.4 Zusammenfassung

In diesem Kapitel wurden wesentliche Grundlagen aus dem Bereich der Software-Produktlinienentwicklung behandelt. Einleitend wurden Begriffe, welche im Kontext der SPLE gebraucht werden und für diese Arbeit relevant sind, definiert – wie unter anderem: Artefakt, Feature, Feature-Repräsentation, Feature-Selektion, Domäne, Plattform, Software-Produktlinienentwicklung, Software-Produktlinie, Software-Produktvariante sowie Variabilität.

Im Anschluss darauf wurden Aspekte der Einführungsentscheidung einer SPLE diskutiert. Hierbei wurde festgestellt, dass obwohl der Übergang von einer Einzelsystem- zu einer Produktlinienperspektive zahlreiche Vorteile bietet, gerade mangelhafte Domänenkenntnis und die aktuell ungenügende integrierte Werkzeugunterstützung ein erhebliches Risikopotential darstellen. Ein weiteres Ergebnis jener Diskussion ist die Feststellung, dass sich eine SPLE vor allem für Unternehmen anbietet, die bereits über Kenntnisse des Marktes verfügen und schon eigene Softwareprodukte besitzen. Obwohl sich aus theoretischer Sicht ein proaktives oder reaktives Vorgehen während der Einführung anbietet, kommt für solche Unternehmen eine erneute Softwareentwicklung beginnend auf einer „grünen Wiese“ meist nicht in Frage. Aufgrund der geringeren Vorabinvestitionen und Risiken ist daher in der Regel eine extraktive Einführungsstrategie empfehlenswert, bei der mit möglichst geringem Aufwand, soviel wie nur möglich der vorhandenen Ressourcen der Altsysteme in die Plattform überführt werden.

Gefolgt von der Transition, beginnt in einem fließenden Übergang die eigentliche SPLE. Dieses neue, strategische Paradigma basiert vor allem auf der Unterscheidung der Hauptaktivitäten *Domain Engineering* (Entwicklung für Wiederverwendung), *Application Engineering* (Entwicklung mit Wiederverwendung) und *Management* (Steuerung der Wiederverwendung). Dieses Kapitel beschreibt in einem Rahmenwerk die Methodenbereiche, Prozesse und Informationsflüsse innerhalb jener Hauptaktivitäten, um das Verständnis für wesentliche Zusammenhänge des produktiven Betriebs zu vertiefen.

Im Rahmen der SPLE muss die Entwicklung und Wartung von einer Gruppe von Softwareprodukten unterstützt werden. Aufgrund abweichender Anforderungen, Gesetzgebungen oder technischen Einschränkungen ergeben sich Unterschiede hinsichtlich der Features der einzelnen Produkte. Diese Variabilität muss zunächst exakt modelliert werden. Dieses Kapitel zeigt wie Feature-Modelle hierbei Beachtung finden können, die je nach Anwendungszweck auch als Feature-Diagramme oder aussagenlogische Ausdrücke realisiert werden können.

Zu der organisierten Variabilität gehört in der Folge auch die Repräsentation der Features in allen Quellcode-Artefakten. Grundsätzlich können hierfür Kompositions- oder Annotationsansätze genutzt werden. Beide weisen neben vielen Gemeinsamkeiten vor allem

komplementäre Vorteile auf, welche in diesem Kapitel überblicksartig besprochen wurden. Darauf basierend wurde sich der Anschauung angeschlossen, dass durch eine Kombination beider Konzepte die SPLE in einem Unternehmen bestmöglich etabliert werden kann. Demnach sollten Annotationsansätze während der extraktiven Einführung favorisiert werden, um die Vorabinvestitionen und Einführungsrisiken zu mindern. Auf lange Sicht gesehen, ist aber eine Transformation hin zu Kompositionsansätzen empfehlenswert, damit eine hohe Softwarequalität und Architekturstabilität gewährleistet werden kann.

# 3 Feature Mining

Aus dem vorangegangenen Kapitel lässt sich festhalten, dass sich in der Regel die Software-Produktlinienentwicklung für Unternehmen anbietet, welche auf der einen Seite über entsprechende Erfahrungen durch die Entwicklung und Wartung von eigenen Produkten verfügen, auf der anderen Seite aber unter gewissem Druck stehen ihre Software-Lösungen wirtschaftlicher produzieren und warten zu müssen. Aufgrund der geringeren Vorabinvestitionen und Risiken empfiehlt sich meist für diese Zielgruppe eine extraktive Einführungsstrategie, bei der mit möglichst geringem Aufwand, soviel wie nur möglich der vorhandenen Ressourcen der Altsysteme in die Plattform überführt werden. Für den Aufbau der wiederverwendbaren Produktlinien-Plattform kommen hierbei vor allem Annotationsansätze in Frage, da diese mit ihren Eigenschaften die Einführungsziele sehr gut unterstützen.

Auch wenn die Überführung der bestehenden Systeme zu einer annotationsbasierten Plattform eine geringe Adoptionsbarriere verspricht, bleibt sie dennoch komplex und aufwändig. Gerade für umfangreiche Systeme, mit tausenden Quellcodezeilen sowie verstreuten und vermischten Feature-Repräsentationen, ist eine vollständig-manuelle Erfassung und Annotierung der Features nicht nur sehr zeitaufwändig, sondern auch fehlerbehaftet [WR07]. Zur weiteren Senkung des Transitionsaufwands und -risikos wird in diesem Kapitel daher vorgeschlagen jene Aufgabe werkzeuggestützt bzw. semiautomatisch zu lösen. Um die damit verbundene Problemstellung zu präzisieren werden nachfolgend Teilaufgaben, Herausforderungen und Annahmen herausgearbeitet. Die anschließende Untersuchung von verwandten Forschungsbereichen unterstreicht, dass jenes Problem bislang in der Forschung nicht adressiert wird. Aufgrund dessen wird für dieses Themengebiet der neue Begriff *Feature Mining* eingeführt.

## 3.1 Aufgabenstellung, Herausforderungen und Annahmen

Um die Software-Produktlinienentwicklung in der Praxis zu etablieren werden Konzepte gebraucht, welche das Transitionsrisiko und den Transistionsaufwand senken können [Kru01; CK02; SE02]. Der Einsatz von leichtgewichtigen Annotationsansätzen stellt einen ersten Schritt in diese Richtung dar, weil hierdurch das Potential geboten wird einen Großteil der vorhandenen Quellcode-Ressourcen beizubehalten [CK02]. Da die bestehen-

den Systeme aber nicht zwangsläufig für die Wiederverwendung im Rahmen der Software-Produktlinienentwicklung geplant waren, ist davon auszugehen, dass implementierte Features nicht konsistent im Quelltext kommentiert oder modular repräsentiert sind [LAL+10; MR05; MRB+05, OS02; SE02; WBP+03]. Für komplexe Systeme, die nicht auf einen Blick überschaut werden können, ist eine vollständig-manuelle Erfassung und Annotierung der Features nicht nur sehr zeitaufwändig, sondern auch fehlerbehaftet [WR07]. Mit dem Ziel der weiteren Senkung der Transitionsbarriere, wird in dieser Arbeit daher vorgeschlagen die Bearbeitung jener Aufgabe durch geeignete Werkzeugunterstützung zu automatisieren.

Im Folgenden wird umfassend diskutiert, welche (Teil-)Aufgaben und Herausforderungen hierbei konkret zu lösen sind und auf welche Annahmen man sich grundsätzlich berufen kann.

**Primäre Aufgabenstellung: Feature Mining**

Durch vorangegangene Aktivitäten wird der Quellcode des Altsystems sowie ein Feature-Modell bzw. Feature-Diagramm bereitgestellt. Die primäre Aufgabe besteht dann darin, zu definierten Features zugehörige Quellcodefragmente *vollständig* und *feingranular* zu *lokalisieren*, zu *expandieren* und so zu *dokumentieren*, dass aus der entstehenden annotationsbasierten Plattform gültige, fehlerfreie Produktvarianten generiert werden können. Hierbei müssen unterstützende Techniken ein *inkrementelles Vorgehen für einzelne Features ermöglichen,* damit die Plattform bedarfsorientiert aufgebaut werden kann. Darüber hinaus müssen sie eine *möglichst hohe Automatisierung anstreben*, sodass der Aufwand für den Anwender im Vergleich zu einem manuellen Vorgehen maßgeblich reduziert wird.

Die grundlegende Aufgabenstellung Quellcodefragmente zu einem abstrakterem Konstrukt wie z.B. einem Konzept, Belang oder Feature zu zuordnen ist nicht neu [Rob05]. Viele unterschiedliche Forschungsgebiete befassen sich mit solch einer Fragestellung. In *Kapitel 3.2* wird ein Überblick erarbeitet, welche das konkret sind. Dennoch wird in dieser Arbeit für die beschriebene Aufgabenstellung bewusst der neue Name *Feature Mining from Source Code* oder abgekürzt *Feature Mining (FM)*[1] eingeführt, da aktuell verwendete Begriffe, zum Teil abweichende Annahmen treffen, ein anderes Ziel verfolgen oder uneinheitlich verwendet werden.

[1] Der Begriff *Feature Mining* versteht sich als Spezialisierung des *Feature Asset Mining* - Vorschlages von Eisenbarth und Koschke [ES01, SE02] (vgl. *Kapitel 3.2*). Er lehnt sich zudem an verwandte Forschungsgebiete wie *Aspect, Asset, Component* oder *Data Mining* an. Jene benutzen das Wort *Mining* als Synonym für die Identifikation und Extraktion von gewissen Informationen oder Konstrukten aus einer großen Menge von Rohdaten (vgl. *Kapitel 3.2*). In diesem Fall sind die Rohdaten Quellcode-Artefakte, die zu herauslösenden Konstrukte *Features*. In der Langform könnte der Forschungsbereich daher auch *Feature Mining from Source Code* heißen.

*Abbildung 3.1* veranschaulicht für ein besseres Verständnis die Feature Mining Aufgabe noch einmal grafisch und hebt zugleich einige zentrale Begriffe hervor, die folgend näher erläutert werden.

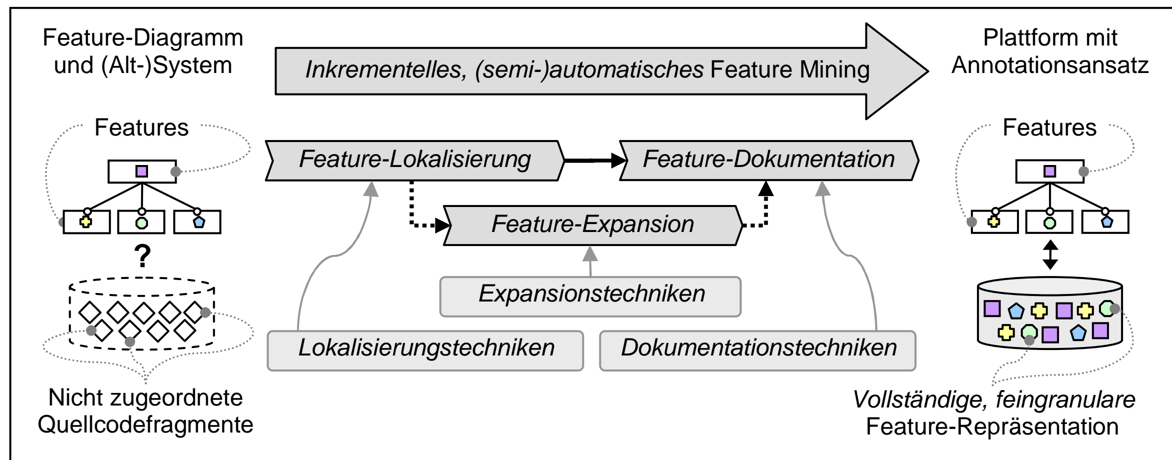



Abbildung 3.1: Werkzeuggestützte Realisierung einer annotationsbasierten Plattform

## Feature-Lokalisierung / Lokalisierungstechniken

Feature Mining wird, wie auch viele verwandte Forschungsgebiete (vgl. *Kapitel 3.2*), von der Aufgabe dominiert eine *Zuordnung zwischen Konstrukten zweier Ebenen mit einem unterschiedlichen Abstraktionsgrad* herstellen zu müssen. Auf der einen Seite befinden sich die Features (vgl. *Kapitel 2.1*). Sie dienen der Kommunikation zwischen Menschen, sind stark kontextabhängig und erfordern Domänenwissen. Meist werden sie nur durch ein Wort oder eine kleine Wortgruppe zusammengefasst. Demgegenüber stehen Feature-Repräsentationen im Quellcode auf einer maschinellen Ausführungsebene mit stark eingegrenztem Vokabular und einer simplen Grammatik (vgl. *Kapitel 2.1*). Diverse Konstanten, Variablen, Funktionen, Klassen, Schlüsselwörter und Beziehungen zwischen ihnen setzen die Repräsentation um. Jene Quellcodefragmente referenzieren aber nicht zwangsläufig den selben „menschennahen Kommunikationskontext".

Das heißt, wenn beispielsweise von *Rechtschreibprüfung* in einer Textverarbeitungsanwendung gesprochen wird, können unterschiedliche Akteure sich (relativ eindeutig) etwas unter diesem Feature vorstellen. Dessen Repräsentation im Quellcode kann allerdings auf unterschiedlichste Weise, unter Verwendung der Programmiersprachenkonstrukte, implementiert sein. Zudem müssen die deklarierten Bezeichner auch nicht zwangsläufig Wörter enthalten, die in der Fach- oder Umgangssprache auf eine *Rechtschreibprüfung* hindeuten, wie z.B. „Orthographie", „Buchstaben", „Fehler" usw.

Die Herausforderung der Zuordnungsaufgabe, welche im Folgenden als *Feature-Lokalisierung* bezeichnet wird, begründet sich demzufolge nicht nur durch die unterschiedliche

Granularität der Informationen, sondern vor allem durch den beschriebenen Paradigmenwechsel [BMW93, BMW94]. Es werden spezielle *Lokalisierungstechniken* benötigt, welche ungeachtet der dargestellten Probleme zu Features zugehörige Quellcodefragmente lokalisieren können.

### Feature-Expansion / Expansionstechniken

Nicht zuletzt durch die Dekompositionsbeschränktheit von traditionellen Programmierparadigmen ist im Allgemeinen *mit querschneidenden Features zu rechnen* [LAL+10; MR05; MRB+05, WBP+03] (vgl. *Kapitel 2.3.2*). Sollte es also durch die Feature-Lokalisierung gelingen beispielsweise selbst eine gesamte Klasse oder sogar ein Paket einem Feature zuzuordnen, muss trotzdem immer zusätzlich geprüft werden, ob der lokalisierte Bereich eventuell nur eine Teilmenge eines querschneidenden Gesamtkonstrukts ist. Prinzipiell bedeutet dieses, dass *anhand der bereits lokalisierten Fragmente Schlussfolgerungen zu möglichen weiteren syntaktisch und semantisch zugehörigen Elementen* getroffen werden müssen.

Im Kontext des Feature Mining wird diese Aufgabe als *Feature-Expansion* bezeichnet. Verschiedene Untersuchungen zeigen, dass in den meisten Fällen querschneidende Features durch unterschiedliche Erweiterungen an verschiedenen Stellen umgesetzt werden [Ape07; LAL+10]. Aufgrund dessen ist eine „einfache" Suche nach korrespondierenden Quellcodeduplikaten in der Regel unzureichend. Gerade in großen und komplexen Systemen, die nicht auf einen Blick erfasst werden können, stellt diese Aufgabenstellung eine große Herausforderung dar [SE02; WS95]. Ein vollständig-manuelles Vorgehen ist dann nicht nur sehr zeitaufwändig, sondern meist auch fehlerbehaftet [WR07]. Aufgrund dessen werden strukturierte, werkzeuggestützte *Expansionstechniken* benötigt, um den Suchraum zu reduzieren bzw. zu präzisieren, den der Anwender manuell untersuchen muss.

### Feature-Dokumentation / Dokumentationstechniken

Verwandte Problembereiche zielen darauf ab ein unbekanntes System zu erschließen oder konkrete Feature-Repräsentationen für Wartungsaufträge zu lokalisieren. Ihnen geht es vor allem darum Einstiegspunkte zu finden oder den Einflussbereich eines Features abzustecken. Ergebnis dieses Vorganges ist daher lediglich eine lose Sammlung von Quellcodeabschnitten (vgl. *Kapitel 3.2*). Im Feature Mining müssen allerdings die Features nicht nur lokalisiert, sondern auch so dokumentiert werden, dass gültige, fehlerfreie Produktvarianten erzeugt werden können.

In dieser Arbeit wird diese *konsistente Beschreibung von Feature-Repräsentationen* auch als *Feature-Dokumentation* bezeichnet. Eine unbedachte Dokumentation kann zu Produktvarianten führen, die beispielsweise falsche Klammersetzungen aufweisen, fehlende Deklarationen referenzieren oder ein Fehlverhalten des Programms verursachen. Kästner und Apel [KA08; Käs10] ordnen solche Fehler in drei Gruppen ein – *Syntax-Fehler* (in Bezug zu

der Sprachengrammatik), *Typ-Fehler* (in Bezug zu Sprachensemantiken) sowie *Verhaltensfehler* (in Bezug zu der Programmspezifikation). Unter Verwendung von geeigneten *Dokumentationstechniken* müssen die Quellcodefragmente so erfasst und verwaltet werden, dass die beschriebenen Fehler minimiert oder im optimalen Fall ganz ausgeschlossen werden.

### Vollständigkeit

Im Feature Mining wird eine vollständige Erfassung aller Feature-Quellcodefragmente angestrebt. Schließlich kann erst hierdurch gewährleistet werden, dass optionale Features nicht in Produktvarianten enthalten sind, für die sie nicht selektiert wurden. Allerdings stellen Robillard et al. [RSH+07] in einer empirischen Untersuchung fest, dass unterschiedliche Personen Zugehörigkeit von einem Quellcodefragment zu einem abstraktem Konstrukt unterschiedlich bewerten. Allein aus diesem Umstand heraus ist es schwierig einzuschätzen, ob ein Feature vollständig erfasst ist oder nicht.

In dieser Arbeit wir daher von *Vollständigkeit* gesprochen, wenn sich eine Feature-Repräsentation so entfernen lässt, dass die Ausführung des Programms keine Rückschlüsse auf die Existenz des Features zulässt, kein ungenutzter Quellcode hinterlassen wird und keine Fehler (im Sinne der drei eingeführten Gruppen) in allen gültigen Varianten verursacht werden.

Gerade für den Bereich der eingebetteten Systeme, ist die Einschränkung im letzten Halbsatz wichtig, da Speicher- und Rechenressourcen stark eingeschränkt sind. Hier ist man besonders bestrebt jede Form von Verschwendung zu vermeiden. Für gewöhnlich genügt es nicht oberflächlich Features zu deaktivieren, wie z.B. nur Menüeinträge zu entfernen, welche die Funktionalität auslösen. Auch in Hinblick auf die Wartbarkeit der Systeme lassen sich Nutzenvorteile durch den SPL-Ansatz meist nur dann erfolgreich umsetzen, wenn auch bekannt ist welche Quellcodebereiche tatsächlich zu den Features gehören.

Es sei an dieser Stelle klar betont, dass durch Lokalisierungstechniken ermittelte Quellcodefragmente das Vollständigkeitskriterium erfüllen *können*. Expansionstechniken (in Kombination mit Dokumentationstechniken) *müssen* hingegen Feature-Repräsentationen *vollständig* identifizieren. Im Gegensatz zu Lokalisierungstechniken können die zuletzt Genannten aber auf bekannte Quellcodefragmente, die eine Teilmenge der Feature-Repräsentation bilden, zurückgreifen.

### (Fein-)Granularität

Liebig et al. [LAL+10] untersuchen in einer Studie 40 Opensource-Anwendungen aus unterschiedlichen Domänen, die Features mit Präprozessor-Anweisungen umsetzen. Sie stellten dabei fest, dass im Durchschnitt rund 23% des gesamten Quellcodes variabel sind. Wobei davon zu gleichen Teilen ganze Methoden und Klassen sowie einzelne Blöcke und

Ausdrücke innerhalb von Methoden annotiert sind. Für das Feature Mining sind diese Ergebnisse insofern wichtig, da hieraus erkennbar wird, dass Lokalisierungs-, Expansions-, und Dokumentationstechniken bis hin zu *feingranularen Erweiterungen innerhalb von Methoden notwendig* sind. Eine „gröbere“ Unterstützung würde dazu führen, dass relevante Elemente möglicherweise unerkannt bleiben. Eine weitere Folge wäre, dass unter Umständen der Quellcode massiv verändert werden muss, damit er annotiert werden kann.

### Inkrementelles Vorgehen

Aus dem technischen Blickwinkel der Plattformumsetzung lässt sich nur schwer eine klare Grenze zwischen Einführung, Betrieb, Erweiterung und einer optionalen Restrukturierung ziehen [CN02, S. 56]. Es sind durchaus Fälle vorstellbar in denen jene Phasen für einzelne Features versetzt ablaufen. So könnte sich beispielsweise ein Unternehmen dazu entscheiden zunächst nur die zehn umsatzstärksten Features zu überführen und die hieraus resultierenden Varianten zu vertreiben. Neue Kundenwünsche könnten dann Weiterentwicklungen der bereits umgesetzten Features, Neuentwicklungen oder vorhandene Features aus Altsystemen erfordern.

Aufgrund dessen müssen Techniken welche den Feature Mining Prozess unterstützen in der Lage sein, einzelne Features bedarfsorientiert zu überführen ohne dabei die Konsistenz der Plattform zu gefährden. Nach Abschluss aller Transitionsaktivitäten für jeweils ein Feature müssen sich also gültige, fehlerfreie Produktvarianten generieren lassen.

### (Semi-)Automatisches Vorgehen

Erfahrungen aus verwandten Forschungsbereichen, wie z.B. Concept Assignment oder Feature Location zeigen (vgl. *Kapitel 3.2*), dass bestimmte Teilprobleme weder von Computern noch Menschen allein erfolgreich gelöst werden können [BMW93; BMW94; CR00]. Eine vollautomatische Bearbeitung der gesamten Feature Mining Aufgabe scheint daher ebenfalls illusorisch.

In dieser Arbeit wird daher vereinbart, dass im Rahmen des Feature Mining spezifische Aufgaben je nach Stärken auf Mensch oder Maschine verteilt werden dürfen. Allerdings muss sichergestellt werden, dass der Aufwand für den Anwender im Vergleich zu einer manuellen Transition maßgeblich reduziert wird.

### Grundsätzliche Annahmen

Im Rahmen des Feature Mining wird angenommen, so wie die SPLE es auch fordert (vgl. *Kapitel 2.1; 2.2*), dass die Transition in Übereinstimmung mit den Unternehmenszielen erfolgt. Daher ist davon auszugehen, dass *hinreichend technische und personelle Ressourcen* für den gesamten Prozess zur Verfügung stehen. Weiterhin wird wie auch in anderen Arbeiten [BKPS04, S. 10; Joh06; SE02; SEI10] zugrunde gelegt, dass der Aufbau und die

Funktionsweise des zu *überführenden Systems weitgehend bekannt* sind. Sodass technisches Personal, welches das System entwickelt und gewartet hat, in der Lage ist für ein gegebenes Quellcodefragment unter Beachtung der definierten Vollständigkeitskriterien zu entscheiden, ob es zu einem gewissen Feature gehört oder nicht.

## 3.2 Verwandte Forschungsgebiete

Der Feature Mining Problembereich steht nicht im luftleeren Forschungsraum. Es gibt eine Vielzahl von verwandten Forschungsgebieten, die sich mit vergleichbaren Fragestellungen beschäftigen. Die meisten von ihnen lassen sich in den größeren Software-Engineering-Kontext der *Wartung*, *Wiederverwendung* und des *Reengineerings* einordnen. Die detailliertere Untersuchung dieser und weiterer spezialisierter Themengruppen ist aus drei verschiedenen Hinsichten interessant. Einerseits zeigt sie aus welchen Bereichen bereits erzielte Ergebnisse und verfügbares Wissen genutzt werden können. Andererseits wird aber auch deutlich, welche Gebiete im Umkehrschluss von Forschungsanstrengungen im Bereich des Feature Mining profitieren könnten. Zudem veranschaulicht die Diskussion auch, dass obwohl Feature Mining viele Schnittmengen mit unterschiedlichen Bereichen aufweist, es dennoch ein eigenständiges Forschungsgebiet ist.

Tabelle 3.1 fasst verwandte Gebiete und ihre Abweichungen zu dem Feature Mining zusammen. Jene Tabelle dient als Gliederung für die folgende nähere Betrachtung.

| Verwandte Forschungsgebiet/e | Unterschied zu Feature Mining | |
|---|---|---|
| Wartung; Wiederverwendung; Reengineering | abweichende Zielsetzung, Annahmen, Anforderungen – z.B. ohne Dokumentation, Vollständigkeitskriterium | übergeordnete, unspezifische Aufgaben |
| Program Understanding / Comprehension | | Feature-Modell nicht per se gegeben |
| Reverse Engineering; Redocumentation; Design-, Specification-, Architecture Recovery / Reconstruction; Component Mining; Aspect Mining; Asset Mining; Mining Existing Assets | | umfassende Quellcodeanalyse zur Identifizierung von „interessanten" Konstrukte; Interessante Konstrukte werden in andere Repräsentationsform überführt – korrespondieren nicht zwangsläufig zu (annotierten) Features |
| Concept Assignment Problem; Concept-, Concern-, Fault Location; Concern Mapping; Requirements Tracing | | keine Lokalisierung von Features (Feature Definition nach Kapitel 2.1) |
| Feature-to-Code Trace Detection and Analysis; Feature Identification; Feature Analysis; Feature Location | | uneinheitliche Verwendung der Begriffe; adressierte Aufgabenstellung behandelt nur Teilbereich des Feature Mining |
| Feature Asset Mining | | nicht festgelegt auf Annotationsansatz |
| Impact Analysis; Change Propagation | | adressierte Aufgabenstellung behandelt nur Teilbereich des Feature Mining |
| Data Mining; Knowledge Discovery from Data | | Liefert mögliches Rüstzeug für Teilaufgaben des Feature Mining |

Tabelle 3.1: Unterschiede verwandter Forschungsgebiete zum Feature Mining

### Software-Wartung, -Wiederverwendung, -Reengineering

*Software-Wartung* bezeichnet den Prozess der Veränderung eines Softwaresystems nach dessen Auslieferung, um Fehler zu beheben (präventive/korrektive Wartung), Performanz, Wartbarkeit oder andere Attribute zu verbessern (perfektionierende Wartung) oder Anpassungen an die veränderte Umgebung vorzunehmen (adaptive Wartung) [IEEE90]. Im Rahmen der adaptiven oder perfektionierenden Wartung, können die Änderungen des Systems aber so weit reichend sein, dass im Grunde ein neues System entsteht, welches lediglich vorhandene Elemente wiederverwendet. Je nach Umfang und Intension der Anpassung, wird in der Literatur meist dann von *Software-Wiederverwendung* oder *Reengineering,* und nicht mehr von Wartung, gesprochen. Im Unterschied zur (normalen) perfektionierenden Wartung, erfolgt ein umfangreiches Reengineering immer dann, wenn kaum Kenntnisse zu dem System verfügbar sind. Es werden hierfür die Phasen Reverse Engineering, Restrukturierung und Forward Engineering durchlaufen, um die Wartbarkeit oder Wiederverwendbarkeit des System zu erhöhen bzw. sie erst zu ermöglichen [GT03, S. 134; LL07, S. 547ff.; MTW93]. Software-Wiederverwendung zielt hingegen darauf ab, neue Systeme so zu entwickeln, dass durch Verwendung von bestehenden Systemressourcen der Aufwand minimiert wird [Bas90; FK05; GT03, S. 155, S.181; Kru92].

Wartungs-, Reengineerings- und Wiederverwendungsaktivitäten ähneln sich vor allem dahingehend, weil sie auf bereits bestehenden Systemen erfolgen. Jenes bedeutet, dass sie allesamt durch eine Analyse oder durch das Verständnis des Systems eingeleitet werden. Erst nach diesem initialen und basisschaffenden Schritt können spezifische Aufgaben zielgerichtet ausgeführt werden [CZD+08; EKS03; GT03, S. 73ff., S. 92; KQ05; RW02; WR07]. Im Folgenden werden einige spezialisierte Forschungsgebiete aus diesem Analyse- und Verständnisaufgabenfeld näher betrachtet, da sie Schnittmengen mit dem Feature Mining aufweisen.

### Program Understanding / Comprehension

Häufig existiert das Systemwissen lediglich in den Köpfen der Software-Ingenieure, da schriftliche Dokumentationen vernachlässigt werden. Über die Zeit hinweg verblasst allerdings die genaue Erinnerung, nicht selten verlassen auch Schlüsselmitarbeiter das Unternehmen. Es entsteht dann eine Verständnislücke zu benötigten Informationen, um Wartungs-, Reengineerings-, oder Wiederverwendungsvorhaben durchführen zu können. Der Aufwand relevanten Code zu finden und zu verstehen ist in solchen Fällen zum Teil größer, als die eigentliche Modifikation. Der Forschungsbereich *Program Understanding* (auch: *Program Comprehension*) sucht daher nach Konzepten, Techniken und Werkzeugen, mit denen jene Verständnislücke effizient geschlossen werden kann [CZD+08; GT03, S. 98, S. 303; HPS07; MJS+00; MTW93; RW02].

Eine wesentliche Aufgabe des Verständnisprozesses besteht darin, durch die Untersuchung von feingranularen Informationen, wie z.B. Quelltext, eine Zuordnung zu einem abstrakten Semantik-Modell aufzubauen, es zu verwalten und auch langfristig aktuell zu halten [MJS+00; MTW93]. Müller et al. [MJS+00] stellen fest, dass auch in der Zukunft gerade in diesem Bereich Forschungsanstrengungen ansetzen müssen.

Im Unterschied zu Feature Mining muss das Semantik-Modell, welches beispielsweise auch der Entwickler des Systems vor sich hatte, nicht zwangsläufig ein Feature-Diagramm sein. Dieses mentale Modell ist auch nicht per se gegeben, häufig muss es erst schrittweise rekonstruiert werden [GT03, S. 109ff., S. 125; MJS+00]. Weiterhin beschränkt sich Program Understanding nicht konkret auf eine Aufgabe mit klaren Annahmen und Anforderungen (vgl. *Kapitel 3.2*). Was genau, in welcher Richtung und in welchem Umfang zu verstehen ist, hängt von zahlreichen Faktoren ab, einer davon ist z.B. die Perspektive und Zielsetzung der beteiligten Akteure [GT03, S. 92, S.110ff.].

### Reverse Engineering; Redocumentation; Design-, Specification-, Architecture-Recovery/Reconstruction; Component-, Aspect-, Asset Mining; Mining Existing Assets

Ein Weg um für große Systeme die Verständnisaktivitäten zu verbessern, ist die Anwendung von Techniken aus dem *Reverse Engineering*, wie z.B. die strukturelle und verhaltensorientierte Quellcode-Analyse [GT03, S. 135; LL07, S.551; MJS+00; MTW93]. Reverse Engineering bezeichnet allgemein den Analyseprozess eines Systems, mit dem Komponenten und deren Beziehungsabhängigkeiten identifiziert werden, um daraus eine andere, zum Teil abstraktere, Repräsentationsform des Systems zu erstellen [CC90; GT03, S. 134, S. 169, MTW93].

Werden semantisch äquivalente Repräsentationen, innerhalb der selben Abstraktionsebene, in einer verbesserten, übersichtlicheren und leichtverständlicheren Form erstellt, spricht man von *Redocumentation* [CC90; GT03, S. 139; LL07, S. 553, MTW93]. Werden andererseits intermediäre, abstraktere Repräsentationsformen in Gegenrichtung zu dem traditionellen Entwicklungszyklus erzeugt, so bezeichnet man das Vorgehen als *Design*, *Architecture* oder *Specification Recovery* (auch: *Reconstruction*) [CC90; BOS00; DP09; Fow99, S.xvi, GT03, S. 141f.; LL07, S.553].

*Component Mining* diskutiert spezialisierte Recovery-Techniken, mit denen wiederverwendbare Komponenten erkannt werden können [BGW+99; Spi99]. Im Kontext der SPLE wird auch von *Asset Mining* [BKPS04, S. 146] bzw. *Mining Existing Assets* [BOS00; BOS01; SEI10] gesprochen, wenn die Komponenten in eine Plattform überführt werden sollen. In jüngster Vergangenheit hat auch der Recovery-Forschungsbereich *Aspect Mining* einen besonderen Auftrieb erfahren. Dort werden Ansätze gesucht, mit denen in objektorien-

tiertem Quellcode querschneidende Belangrepräsentationen identifiziert werden können, die sich in Aspekten modularisieren lassen [Dre09; KMT07].

Zusammenfassend lässt sich festhalten, dass alle hier betrachteten Reverse Engineering Arten darauf abzielen Konstrukte zu identifizieren bzw. zu lokalisieren, die in eine andere Repräsentationsform überführt werden sollen. Im Gegensatz dazu sind im Feature Mining schon Konstrukte aus unterschiedlichen Repräsentationsebenen gegeben. Gesucht werden dann die ebenenübergreifenden Zuordnungen zwischen den „Paaren“ (vgl. *Kapitel 3.2*). Dennoch könnte Feature Mining insofern von Techniken aus diesem Bereich profitieren, da sie dabei helfen Konstrukte zu identifizieren, die eventuell zu Features korrespondieren, sowie Beziehungen und Abhängigkeiten von Quellcodefragmenten für ein einfacheres Verständnis abstrahieren.

### Concept Assignment Problem; Concept-, Concern-, Fault Location; Concern Mapping; Requirements Tracing

Ein vollständiges Verständnis eines Programms, wie in traditionellen Program Understanding und Reverse Engineering Ansätzen, ist nicht immer möglich und häufig auch nicht notwendig. Schließlich umfassen Änderungs- und Modifikationsanfragen häufig nur eine Teilmenge der gesamten Anwendung. Jene Anfragen werden in der Regel unter Verwendung von abstrakten Domänenkonstrukten formuliert, wie z.B. „Ändere oder füge Kreditkartenzahlung zu den Zahlungsarten hinzu“. Meist gilt, dass der Programmierer die Domäne kennt, aber nicht genau weiß, wie diese im Quellcode (oder auch in anderen Artefakten) umgesetzt ist. Daher besteht die wesentliche Aufgabe dann darin zu verstehen, wo und wie die relevanten, abstrakten Konstrukte des Änderungsauftrages im Quellcode implementiert sind. Die Komplexität und Bedeutung dieser zentralen Aufgabenstellung im Software-Engineering wächst proportional mit der Größe des zu untersuchenden Systems. Aufgrund dessen werden für umfangreiche Systeme, werkzeuggestützte Ansätze benötigt, die den Suchraum und -aufwand reduzieren können [EAAG08; EKS03; EV05; LMPR07; MRB+05; MSRM04; RSH+07; RW02; SEW+06; WBPR01; WBP+03].

Biggerstaff et al. [BMW93; BMW94] waren die ersten, welche explizit diese Lokalisierungs- bzw. Zuordnungsaufgabe als *Concept Assignment Problem* formuliert haben. Sie stellten zudem die Vermutung auf, dass dieses Problem wahrscheinlich niemals vollautomatisch zu lösen sein wird. In späteren Arbeiten wurden für eng verwandte Problemstellungen Begriffe wie *Concept Location* [MRB+05; MSRM04; RW02], *Concern Mapping* [RSH+07], *Concern Location* [EAAG08], *Fault Location* [AHLW95; CC87; JHS01] sowie *Requirements Tracing* [ACC+02; HDS06] verwendet.

Im Unterschied zu all diesen Gebieten ist das Ziel im Feature Mining die Lokalisierung von (spezifischen) Features, die sich auf Anforderungen von Interessengruppen zurückführen lassen und eine Konfigurationsoption anbieten (vgl. *Kapitel 2.1*), und nicht von

(allgemeinen) Belangen bzw. Konzepten, Anforderungen oder Fehlern (vgl. *Kapitel 3.2*).[2] Darüber hinaus werden in jenen Gebieten nicht explizit die selben Annahmen getroffen, Anforderungen formuliert oder vergleichbare Vollständigkeitskriterien definiert.

### Feature-to-Code Trace Detection and Analysis; Feature Identification; Feature Analysis; Feature Location

Wilde und Scully [WS95] gelten als Pioniere der Lokalisierung von Features, die eine vom Nutzer ausführbare bzw. beobachtbare Teilmenge der Belange darstellen [AG05; EKS03; EV05; LMPR07; MR05; RP09]. In der Zwischenzeit wurden in vielen Arbeiten hierfür unter anderem die eigenständigen Begriff *Feature-to-Code Trace Detection and Analysis* [EBG07], *Feature Identification* [AG05; AG06], *Feature Analysis* [SE02; SMAP06] und *Feature Location* [AG06; CR00; EKS03; EV05; KQ05; LMPR07; SE02; SEW+06; WBP+03; WGHT99; WS95; ZZL+06] verwendet. Im Allgemeinen ist es nicht möglich eindeutige Abgrenzungen für die jeweiligen Bereiche zu formulieren, da die Begriffe uneinheitlich verwendet werden. Zum einen lassen sich aus allen vier Bereichen Arbeiten benennen, die eine fast identische Aufgabenstellung verfolgen, wie z.B. [AG05], [EBG07], [EKS05] sowie [SMAP06]. Zum anderen gibt es aber auch innerhalb der Gruppen unterschiedliche Anschauungen zum Umfang der Aufgabe. Beispielsweise verfolgen Revelle und Poshyvanyk [RP09] mit ihrem Feature Location Ansatz eine nahezu vollständige Erfassung als Ziel, wobei Marcus et al. [MRB+05], oder Liu et al. [LMPR07] nur eine Identifizierung von Startpunkten anstreben. Salah et al. [SMAP06] erörtern unter dem Begriff *Feature Analysis* einige Publikationen, welche im dem *Feature Location* Themenbereich veröffentlicht wurden. Simon und Eisenbarth [SE02] ordnen hingegen *Feature Analysis* als eine übergeordnete Aufgabe des *Feature Location* ein.

Nicht zuletzt aufgrund der uneinheitlichen und zum Teil undurchsichtigen Verwendung jener diskutierten Begriffe, ist eine klare Positionierung notwendig. In keiner der untersuchten Arbeiten aus diesem Problembereich, werden explizit die Anforderungen oder Vollständigkeitskriterien definiert, welche das Feature Mining fordert (vgl. *Kapitel 3.2*). Allerdings wird aufgrund der vergleichbaren Zuordnungsaufgabe, Feature Location (und dessen Synonyme) als eine wesentliche Teilaufgabe des Feature Mining angesehen. Diese Teilaufgabe wird im FM-Kontext, in Anlehnung an dessen englischsprachiges Pendant als Feature-Lokalisierung bezeichnet. Wobei hierfür die Feature-Definition aus *Kapitel 2.1* gilt.

---

[2] *Konzept* (engl. concept) und *Belang* (engl. concern) beschreiben im Wesentlichen den selben Gegenstand. In der gegenwärtigen Literatur setzt sich aber zunehmend der zuletzt Genannte durch [MR05; RSH+07] (vgl. Kapitel 2.3.2). Im Kontext der Problemdomäne, wird häufig auch der Begriff *Anforderung* (engl. requirement) verwendet. In dieser Arbeit werden, in Anlehnung an Eaddy et al. [EAAG08] oder auch Apel [Ape07], Anforderungen als Teilmenge von Belangen angesehen. *Fehler* (engl. *fault*) können auch als ungewollte Belange bezeichnet werden [RW02].

### Feature Asset Mining

Simon und Eisenbarth [SE02] strukturieren wesentliche Aktivitäten und Prozesse, welche für die Überführung von Altsystemressourcen notwendig sind, in ihrem methodischen Rahmenwerk Namens *Feature Asset Mining.* Sie verfolgen hierbei die Idee eines leichtgewichtigen, partiellen Reengineerings für relevante Features. In diesem werden zunächst wichtige Features definiert, im Quellcode lokalisiert und analysiert. Im Anschluss daran, erfolgt eine teilweise Umstrukturierung bzw. Anpassung des produktiven Systems. Jene Aktivitäten strukturieren die beiden Autoren in einem detaillierten Aktivitätsdiagramm und ordnen spezifische Aufgaben festgelegten Rollen zu.

Dieses Vorgehensmodell berücksichtigt nicht explizit die Unterscheidung zwischen Annotations- und Kompositionsansatz. Mit dem generischen Begriff „Reengineering" wird die konkrete Repräsentationsform der Features in Quellcode-Artefakten offen gelassen. *Feature Mining* legt sich hingegen konkret auf den Annotationsansatz fest und adressiert detailliert Aufgaben und Anforderungen, die hiermit verbunden sind (Feature-Dokumentation, Vollständigkeit, Feingranularität, etc.). Prinzipiell lässt sich *Feature Mining* daher auch als Spezialisierung und Präzisierung des *Feature Asset Mining* verstehen.

### Impact Analysis; Change Propagation

Mit Feature-Lokalisierungstechniken allein wird es im Allgemeinen, auch aufgrund der unterschiedlichen Zielsetzung, nicht gelingen Features *vollständig* zu lokalisieren. Genau deswegen werden im Feature Mining zusätzliche Expansionstechniken benötigt, welche basierend auf lokalisierten Fragmenten, Rückschlüsse auf die Existenz weiterer korrespondierender Bereiche ziehen (vgl. *Kapitel 3.2*). *Impact Analysis* und *Change Propagation* sind zwei Forschungsgebiete, die einen vergleichbaren Aufgabenbereich adressieren. Impact Analysis ist ein spezialisierter Programmverständnisprozess, welcher dem Programmierer dabei hilft, die Art, den Umfang sowie alle betroffenen Komponenten einer bevorstehenden Software-Modifikation festzustellen. Typischerweise beginnt der Prozess mit initial bekannten Startpunkten. Es werden dann Komponentenabhängigkeiten einer definierten Granularität genutzt, um weitere Elemente zu erschließen, welche der Programmierer zu untersuchen hat [BA96; PMFG09; PR09; QVWM94]. Der Vorgang wird solange rekursiv wiederholt bis der Programmierer alle Elemente identifiziert hat. Impact Analysis ist ein Mittel in der Software Wartung mit dem der Aufwand und das Risiko einer Änderung abgeschätzt werden kann, um daraus eine Modifikationsstrategie abzuleiten [GT03, S. 271; RG04].

Eine Fallstudie hat gezeigt, dass Programmierer häufig die Menge der betroffenen Komponenten nicht vollständig bestimmen. Daher können während der tatsächlichen Änderung unerwartet Effekte auftreten. Change Propagation hilft dabei die Auswirkungen

dieser Effekte zu eruieren. Im Wesentlichen werden hierfür vergleichbare Schritte wie in der Impact Analysis durchlaufen [RG04].

Obwohl die beiden vorgestellten Forschungsgebiete, eine andere Intension verfolgen und abweichende Annahmen zur Vollständigkeit treffen, können sie für die Expansionsaufgabe des Feature Mining wertvolle Ergebnisse einbringen.

#### Data Mining; Knowledge Discovery from Data

Abschließend sei noch auf einen Forschungsbereich hingewiesen, auf dessen Sammlung von Techniken sich Program Understanding und Reverse Engineering Ansätze nicht selten beziehen – gemeint ist das *Data Mining* [KMT07]. Einfach ausgedrückt umfasst Data Minig einen Prozess sowie eine Methodik, mit dem Wissen aus einer großen Menge von Daten „extrahiert“ werden kann. Data Mining ordnet sich wiederum in den größeren Kontext des Forschungsgebietes *Knowledge Discovery from Data* ein, welches alle Schritte von der Datenaufbereitung über das Data Mining bis hin zur Wissenspräsentation umfasst [FSSU96; HK06, S. 5].

Feature- und Data Mining verfolgen beide das Ziel relevante Informationen aus einer großen Datenmenge zu gewinnen, dennoch können sie nicht ohne weiteres gleichgesetzt werden. Feature Mining beschreibt nämlich die konkrete Aufgabe Features im Quellcode vollständig zu lokalisieren und zu annotieren. Data Mining ist hingegen weiter gefasst und liefert mögliches Rüstzeug für ganz verschiedene Aufgaben.

## 3.3 Zusammenfassung

Einleitend wurde in diesem Kapitel zum Zweck der Senkung der Transitionsbarriere vorgeschlagen ein so genanntes Feature Mining durchzuführen.

**Feature Mining** beschreibt ein (semi-)automatisches, werkzeuggestütztes Vorgehen mit dem ein (Alt-)System inkrementell in eine annotationsbasierte Plattform überführt werden kann. Hierbei ist aus vorangegangenen SPLE-Aktivitäten ein Feature-Modell bzw. Feature-Diagramm verfügbar. Die primäre Aufgabe besteht dann darin zu definierten Features zugehörige Quellcodefragmente vollständig und feingranular zu lokalisieren, zu expandieren und so zu dokumentieren, dass aus der Plattform gültige, fehlerfreie Produktvarianten generiert werden können. Die besondere Schwierigkeit dieser Aufgabe begründet sich darin, dass die bestehenden Systeme nicht zwangsläufig für die Wiederverwendung im Rahmen der SPLE geplant waren und vorhandene Features daher nur selten schon kommentiert oder modular repräsentiert sind.

In diesem Kapitel wurden über jene Definition hinaus wesentliche Teilaufgaben, Herausforderungen und Annahmen des Feature Mining ausführlich diskutiert, sodass auch zukünftige Arbeiten diesen Problembereich eindeutig adressieren können.

Der Feature Mining Problembereich steht nicht im luftleeren Forschungsraum. Es gibt eine Vielzahl von verwandten Forschungsgebieten, die sich mit vergleichbaren Fragestellungen beschäftigen. Im Wesentlichen lassen sie sich alle in den größeren Software-Engineering-Kontext der Wartung, Wiederverwendung und des Reengineerings einordnen. Die detaillierte Untersuchung dieser und weiterer spezialisierter Themengruppen hat gezeigt, dass Feature Mining ein neues und eigenständiges Forschungsgebiet ist. Allerdings wurden hierbei aber auch diverse Schnittmengen mit den verwandten Forschungsgebieten identifiziert. Hierdurch wird zum einen deutlich aus welchen Bereichen bereits erzielte Ergebnisse und verfügbares Wissen genutzt werden können. Zum anderen zeigt sich aber auch, welche Gebiete im Umkehrschluss von Forschungsanstrengungen im Bereich des Feature Mining profitieren könnten.

# 4 Aktueller Stand der Technik

In dem *3. Kapitel* wurde gezeigt, dass Feature Mining ein neues Themenfeld in der Forschung aufspannt. Aus diesem Blickwinkel lässt sich argumentieren, dass es noch keinen aktuellen Stand der Technik gibt. Allerdings hat die Diskussion im vorherigen Kapitel auch gezeigt, dass Feature Mining in diversen Aspekten Schnittmengen mit verwandten Forschungsgebieten aufweist, von dessen Ergebnissen man durchaus bei der Entwicklung bzw. Implementierung von konkreten Feature Mining Konzepten profitieren kann. In dieser Arbeit werden jene Ergebnisse als Stand der Technik angesehen und nachfolgend dargestellt.

Da im Rahmen dieser Diplomarbeit eine vollständige Betrachtung aller Arbeiten aus den verwandten Forschungsgebieten weder möglich noch erstrebenswert ist, werden Kategorien gebildet und durch einige häufig zitierte Ansätze veranschaulicht. Die jeweiligen Konzeptkategorien werden wiederum je nach dem, welche Phase bzw. welches Aufgabengebiet sie bereichern könnten, in den Kontext der *Lokalisierungs-, Expansions-,* oder *Dokumentationstechniken* eingeordnet (vgl. *Kapitel 3.1*). Abschließend erfolgt jeweils kurz eine Einschätzung hinsichtlich der aktuellen Eignung für das Feature Mining.

Jene Darstellung des aktuellen Stands der Technik richtet sich *primär an Forscher, die interessiert sind eigene Lösungen für die Feature Mining Aufgabenstellung zu entwickeln*. Für sie bietet dieses Kapitel einen idealen Einstiegspunkt, um grundlegende Techniken und Werkzeuge aus diesem Bereich kennen zu lernen.

In dem folgenden *5. Kapitel* wird ein eigenes Lösungskonzept vorgeschlagen, welches unter anderem auf Techniken aus dieser Darstellung aufbaut. Um jenes Konzept einordnen zu können, genügt es an dieser Stelle sich zunächst einen oberflächlichen Überblick zu verschaffen. So können durchaus alle Beschreibungen der *konkreten Ansätze* „überflogen" oder gar „übersprungen" werden. Alle in *Kapitel 5* verwendeten Techniken werden explizit referenziert, sodass bei Interesse noch einmal genauer nachgeschlagen werden kann.

## 4.1 Lokalisierungstechniken

Aufgrund der Größe und Komplexität heutiger Systeme, ist eine vollständig-manuelle Transition zu einer annotationsbasierten Plattform höchstwahrscheinlich nicht nur zeitaufwändig, sondern vor allem auch fehlerbehaftet. Daher fordert das Feature Mining explizit eine (semi-)automatische Werkzeugunterstützung für den gesamten Prozess. *Lokalisierung* ist hierbei der erste entscheidende Schritt für den effiziente Konzepte, Methoden und Techniken benötigt werden. Die besondere Schwierigkeit der Lokalisierung von Features im Quellcode ergibt sich aus dem Paradigmenwechsel sowie der Informationsgranularität der Ebenen, in denen die Features bzw. die Feature-Repräsentationen enthalten sind. *Lokalisierungstechniken* müssen nicht zwangsläufig *vollständige* Feature-Repräsentationen erfassen. Ihr Fokus liegt vielmehr darin, die *Ebenendifferenz zu überwinden und Teilmengen bzw. Einstiegspunkte des Features im Quellcode zu identifizieren* (vgl. *Kapitel 3.1*).

Werden für die Feature-Lokalisierung Informationen verwendet, die durch die Ausführung der Anwendung gewonnen wurden, wird der Ansatz als *dynamisch* bezeichnet. *Statische* Techniken basieren hingegen nur auf dem „reinen" Quellcode [MRB+05; LMPR07]. In dieser Arbeit wird zudem von *Hybriden Techniken* gesprochen, wenn sowohl statische als auch dynamische Aspekte eine Rolle spielen. Im Folgenden werden Ansätze aus allen drei Gruppen betrachtet.

### 4.1.1 Statische Techniken

Statische Techniken nutzen Abhängigkeiten und textuelle Informationen in dem Quelltext oder in zugehörigen Dokumentationen [LMPR07]. Aufgrund dessen vereinen sie alle gemeinsame Anforderungen und vergleichbare Vorverarbeitungsschritte [MRB+05]. In dieser Arbeit werden daher nur die populärsten Ansätze, wie z.B. *Textbasierte-Mustersuche*, *Textbasierte Suche mit Techniken aus dem Information Retrieval*, *Quellcodestrukturbasierte Mustersuche* sowie *Explorative Suche entlang von Abhängigkeiten und Beziehungen* betrachtet.

#### 4.1.1.1 Textbasierte Mustersuche

Wenn durch Intuition oder persönliche Erfahrung mit dem System kein Startpunkt für ein Feature im Quellcode benannt werden kann, nutzen Programmierer in der Praxis vor allem eine *textbasierte Mustersuche*. Hierbei wird die mögliche Beziehung zwischen Belangnamen sowie Programmbezeichnern ausgenutzt. Soll beispielsweise das Feature „Zahlung" lokalisiert werden, könnte nach Stellen gesucht werden, welche die Wörter „Zahlung", „zahle", „Rechnung", etc. enthalten. Werden keine oder zu viele Treffer gefunden, ist eine Anpassung bzw. Veränderung des Suchmusters erforderlich. Bei einer überschau-

baren Ergebnisgröße muss der Entwickler im nächsten Schritt jeden Treffer untersuchen und entscheiden, ob jener zu dem Feature gehört oder nicht [MRB+05; MSRM04; WBP+03; WR07].

**Konkrete Ansätze**

Diese Technik wird durch eine ganze Familie von Unix-Werkzeugen, wie z.B. *grep*, *egrep*, *fgrep*, *ed*, *sed*, *awk* und *lex* unterstützt, wobei grep sicherlich zu den populärsten in dieser Reihe gehört. Mit grep werden Zeilen ausgewiesen, welche Übereinstimmungen zu einem definierten regulären Ausdruck aufweisen [ESW04; MN96; MRB+05; MSRM04; RW02; WBP+03].

Für grep wurden zahlreiche fortgeschrittenerer Erweiterungen vorgeschlagen [MRB+05]. Eine davon ist der *Aspect Browser* [GKY99]. Dort werden gefundene lexikalische Muster, d.h. Zeichenketten oder reguläre Ausdrücke, in verschiedenen Sichten des Quellcodes farblich hervorgehoben. Ursprünglich wurde das Tool für Fortran und C entwickelt. Mittlerweile wird die Software auch als Eclipse Plug-in[1] für Java Quelltext angeboten. [2]

**Eignung für das Feature Mining**

Die textbasierte Mustersuche hat den großen Vorteil, dass sie feingranular, sehr schnell und einfach angewendet werden kann [ESW04; WBP+03]. Allerdings ist sie ebenso bestimmt von einem nicht zu vernachlässigenden Problem: In *Kapitel 3.1* wurde festgestellt, dass Quellcodefragmente nicht zwangsläufig den selben Kommunikationskontext der Feature-Ebene referenzieren müssen. Diese Technik-Gruppe vernachlässigt aber diese Einschränkung. Wird daher beispielsweise ein unterschiedliches Vokabular verwendet, so scheitert das gesamte Verfahren, da keine Übereinstimmungen erzielt werden können. Auftretende Synonyme im Quelltext werden zudem nicht (automatisch) erfasst. Polyseme[3] können darüber hinaus zu verfälschten oder unpräzisen Suchergebnissen führen [ESW04; MSRM04; SP03; RW02; WBP+03; WR07].

## 4.1.1.2 Textbasierte Suche mit Information Retrieval Techniken

*Information Retrieval (IR)* ist ein eigenständiger Themenbereich innerhalb des Data Mining bzw. Knowledge Discovery from Data. IR Systeme werden verwendet um (meist unstruk-

---

[1] http://www.eclipse.org

[2] http://cseweb.ucsd.edu/~wgg/Software/AB

[3] Ein Wort wird als Polysem bezeichnet, wenn es mehrere semantische Bedeutungen hat, wie z.B. „Zug“ – Fortbewegungsmittel, Bewegung bei Brettspielen, etc.

turierte) Daten, wie z.B. Text-Dokumente, zu speichern, zu verwalten und bei entsprechenden Anfragen anzubieten. Sie werden häufig in Bibliotheken oder Web-Suchmaschinen verwendet. Ein weiteres Einsatzfeld ist aber auch die Lokalisierung von abstrakten Konstrukten in Quellcode-Artefakten [BN99; HK06, S. 615ff.; MRB+05; SM83].

IR-basierte Lokalisierungstechniken folgen im Wesentlichen fast alle einem einheitlichen Muster. Im ersten Schritt werden Bezeichner und Kommentare aus dem Quellcode extrahiert. Anschließend werden diese weiter nachbearbeitet. Beispielsweise werden hierbei Stoppwörter entfernt, zusammengesetzte Namen in Einzelwörter zerlegt, Abkürzungen in ihre Langform überführt oder aus gebeugten Wörtern die Stammform gebildet. Im dritten Schritt werden die auf diese Weise entstandenen *Terme* ihren *Dokumenten* zu geordnet. Je nach Granularität, kann ein Dokument einer Klasse, einer Methode, einem Block oder auch einer Zeile entsprechen. Mit Hilfe von unterschiedlichen IR-Methoden (*Vector Space Model* [SM83], *Latent Semantic Indexing* [LFL98], etc.), werden die Dokumente indexiert und durch einen Vektor repräsentiert.

Eine vom Anwender gestellte Anfrage in natürlicher Sprache, welche den Belang oder das Feature beschreibt, wird nach der gleichen Methodik in ein Dokument bzw. Vektor überführt. Durch eine Gleichheitsfunktion, wie z.B. der Kosinuswert des eingeschlossenen Winkels zweier Vektoren, kann die Nähe zwischen den Vektoren bzw. Dokumenten bestimmt werden. Als Rückgabewert erhält der Anwender, ähnlich wie auch bei Suchmaschinen, eine Liste von Dokumenten geordnet nach der Übereinstimmungsgenauigkeit zu der gestellten Anfrage. Abschließend muss nacheinander (manuell) für jeden Suchtreffer entschieden werden, ob dieser zu dem gesuchten Konstrukt gehört oder nicht [MRB+05; MSRM04]. Die einzelnen Arbeiten in diesem Bereich unterscheiden sich im Wesentlichen hinsichtlich der ausgeführten Normalisierungsschritte sowie nach der Art der verwendeten IR-Methode. Im Folgenden wird auf einige von ihnen näher eingegangen.

### Konkrete Ansätze

Einer der ersten Versuche, Zuordnungen nach dem beschriebenen Muster herzustellen, geht auf Antoniol et al. [ACC+02; ACCD00] zurück. Sie verwenden als IR-Methode ein *Vector Space Model*. Vereinfacht dargestellt, spannen die Autoren eine *Matrix aus allen auftretenden Termen und Dokumenten* auf. In den einzelnen Zellen wird die gewichtete Häufigkeit des Auftretens der jeweiligen Terme in den Dokumenten festgehalten. Alle genau zu einem Dokument korrespondierenden Zellen determinieren dessen Vektor. Einen ähnlichen Ansatz verfolgen auch Zhao et al. [ZZL+06] und Eaddy et al. [EAAG08]. Ein allgemeiner Kritikpunkt am klassischen Vector Space Model ist, das es keine Beziehungen zwischen den Termen berücksichtigt. Das heißt, obwohl in einem Dokument das Wort „Automobil“ und in einem anderen „Fahrzeug“ in ähnlichem Kontext auftreten, werden sie als ungleich eingestuft. Daher liefert eine Suchanfrage nach „Fahrzeug“ auch keine Dokumente die „Automobil“ enthalten – und umgekehrt.

*Latent Semantic Indexing* basiert ebenfalls auf dem Vector Space Model, ist aber in der Lage einige Aspekte der (semantischen) *Bedeutung von Wörtern* oder Passagen *aus ihrer Benutzung in den Dokumenten* abzuleiten. Hierdurch können Synonymie- und Polysemieprobleme im Quelltext weitestgehend gelöst werden. Es ist daher denkbar, dass diese Technik Dokumente als sehr ähnlich ausweist, obwohl sie keinen einzigen gemeinsamen Term haben. Zudem können Dokumente bei denen „Fahrzeug" nur erwähnt wird von solchen unterschieden werden, bei denen tatsächlich ein „Fahrzeug" umfassend beschrieben wird [MMS05; PGM+07]. Für technische Details der IR-Methode sei auf Landauer et al. [LFL98] verwiesen. Im Kontext der Lokalisierung von abstrakten Konstrukten wurde Latent Semantic Indexing unter anderem in den Arbeiten von Marcus et al. [MSRM04; MMS05; MRB+05], Poshyvanyk et al. [PGM+07; RP09], Liu et al. [LMPR07] sowie von McMillan et al. [MPR09] angewendet. Zudem wurde gezeigt, dass *Latent Semantic Indexing* gegenüber anderen IR-Methoden, wie z.B. dem Vector Space- oder Probabilistischen IR Model [ACDM99], in jedem Fall vorgezogen werden sollte [MMS05].

*Lucene*[4], ein von der Apache Software Foundation[5] gefördertes Opensource-Projekt, ist eine in Java implementierte Bibliothek für eine Hochleistungstextsuchmaschine. Jene Bibliothek ist grundsätzlich für fast alle Anwendungen geeignet, die eine Volltextsuche realisieren müssen. Im Kern basiert Lucene auf dem Vector Space Model mit der TF-IDF-Termgewichtungsformel [BN99]. Aufgrund ihrer Performanz und Stabilität wird die Lucene Bibliothek mittlerweile in über 300 unterschiedlichen Systemen verwendet.[6]

*IriSS* (Information Retrieval based Software Search) von Poshyvanyk et al. [PMDS05] implementiert den Latent Semantic Indexing Ansatz in einem Add-on für das Microsoft Visual Studio. Mit Hilfe dieses Werkzeuges können Belangbeschreibungen in natürlicher Sprache als Suchanfragen über C++ Quellcode und zugehörige Dokumentationsunterlagen angewendet werden. Je nach dem Grad der Übereinstimmung wird eine geordnete Liste von Klassen und Methoden zurückgegeben.

*JIriSS* ist hingegen das Eclipse-Pendant für Java Quellcode. Es bietet im Wesentlichen die selbe Kernfunktionalität wie IriSS. Darüber hinaus sind aber noch einige weitere Features implementiert, wie z.B. eine Rechtschreibprüfung und Autovervollständigung basierend auf dem genutzten Vokabular. Hilfreich ist auch die Erweiterung mit der gesamte Funktionen markiert und als Suchanfrage ausgelöst werden können, um „verwandte" Quellcodefragmente zu finden [PMD06].

[4] http://lucene.apache.org

[5] http://www.apache.org

[6] http://wiki.apache.org/lucene-java/PoweredBy

Poshyvanyk und Marcus [PM07] erweitern das „Standardvorgehen" indem sie die sortierte Ergebnisliste mit der *formalen Begriffsanalyse* [Sne98] zusätzlich nachbearbeiten. Dabei werden aus den ersten *n* Dokumenten (Methoden bzw. Klassen) *k* Terme (Wörter) isoliert, welche die Suchtreffer am besten beschreiben. Es wird dann ein Konzept-Gitter aufgebaut, das die Ergebnisse zu relevanten Themen gruppiert und zueinander in Beziehung setzt. Es wurde festgestellt, dass durch diese Methode der Benutzer die Ergebnisse deutlich besser strukturieren und erfassen kann.

#### Eignung für das Feature Mining

In Studien wurde gezeigt, dass mit IR-Ansätzen signifikante Verbesserungen gegenüber rein textbasierten Mustersuchen, wie z.B. mit Grep, erzielt werden können [ACC+02]. Bei gleicher Flexibilität und einfacher Anwendbarkeit können Probleme der Synonymie und Polysemie innerhalb des Quellcodes weitestgehend überwunden werden. Allerdings bleibt wie auch bei den textbasierten Mustersuchen die Schwierigkeit bestehen, Anfragen zu formulieren, welche dem Quellcode-Vokabular entsprechen [EAAG08; MRB+05; MSRM04].

### 4.1.1.3 Quellcodestrukturbasierte Mustersuche

Reine textbasierte Suchanfragen können bis auf die Ebene einzelner Zeichen angewendet werden. Allerdings erfassen sie nicht strukturelle Quellcode-Informationen, welche aber ebenso zu einer Feature-Repräsentation gehören [MRB+05]. In den letzten Jahren wurden zahlreiche Techniken und Werkzeuge vorgeschlagen, die genau dieses Problem adressieren. Die meisten von ihnen verwenden eigene Muster-Spezifizierungssprachen, basierend auf regulären oder logischen Ausdrücken. Hierdurch können abstrakt formulierte Strukturen unter Verwendung von Faktendatenbanken, (Abstrakten-)Syntaxbäume, oder auch des reinen Quellcodes, lokalisiert werden [MN96; MRB+05].

#### Konkrete Ansätze

In den 90er Jahren wurden in diesem Kontext die Werkzeuge *A* tool*, *Code Miner*, *GENOA, LSME, Ponder, Refinery, Scrimshaw, Scruple, tawk* sowie *TXL* vorgestellt, um nur einige zu nennen. [MN96; MRB+05]. Um dem *aktuellen* Stand der Technik gerecht zu werden, wird im Folgenden detaillierter auf einige Techniken eingegangen, die in dem letzten Jahrzehnt diskutiert wurden.

Hannemann und Kiczales [HK01] erweitern in ihrem *Aspect Mining Tool* den Aspect Browser um eine *klassentypbezogene Suche* (vgl. *Kapitel 4.1.1.1*), welche sich mit der vorhandenen Lexikalischen komplementär ergänzt. Somit könnten Anfragen verarbeitet

werden, die wie folgt aussehen: „Zeige alle Klassen, die den Typ java.lang.Exception nutzen oder den Ausdruck ^.*catch.*$ enthalten".

Zhang und Jacobsen [ZJ05] entwickeln, basierend auf dem Aspect Mining Tool, das Eclipse Plug-in *Prism Aspect Miner*. Durch die Einführung der deklarativen Anfragesprache *Prism Query Language* können in der Folge noch komplexere Suchmuster bearbeitet werden, wie z.B.: „Finde Methoden, die Aufrufe zu Klassen haben, welche ‚Figure' als Feld deklarieren". Prism Query Language ermöglicht die Java-Quellcode-Suche auf der Ebene von Typen, Methoden und Feldern. Im Hintergrund wird ein angepasster AspectJ Compiler[7] verwendet, um eine effiziente Ausführung der Anfragen zu realisieren.

*JQuery*, ein Eclipse Plug-in von Janzen und De Volder [JV03], ermöglicht es durch die Anwendung von regulären Ausdrücken und logischen Anfragen Java Quellcode explorativ zu erschließen. Basierend auf Prädikaten der logischen Programmiersprache TyRuBa[8] können eigene spezifische Anfragen formuliert werden [Vol98]. Allerdings stehen dem Anwender auch vordefinierte Strukturanfragen zur Verfügung, die über ein Kontextmenü auf Pakete, Typen, Methoden und Felder angewendet werden können. Auf Elementen der Ergebnismenge können wiederum dieselben Analysen erfolgen. JQuery speichert in einer baumartigen Darstellung alle Pfade, die während der gesamten Expansion besucht wurden. Hierdurch erleichtert man dem Benutzer den Quellcode, gleichzeitig in verschiedene Richtungen entlang der definierten Beziehungen, zu erschließen dabei aber nicht den Gesamtüberblick zu verlieren. Dieses Konzept ist somit flexibel wie eine Anfragesprache, aber auch übersichtlich wie ein hierarchischer Browser [JV03]. JQuery lässt sich für kleine Anwendungen sehr einfach installieren und benutzen, allerdings skaliert es nicht für große Systeme [KHR07].

Ein vergleichbarer Ansatz zu JQuery wurde auch in dem Eclipse Plug-in *Sextant* von Eichberg et al. [EHM06] umgesetzt. Mit diesem Werkzeug werden strukturierte Artefakte in eine XML-Repräsentation überführt und in einer Datenbank abgelegt. Mit der Anfragesprache XQuery[9] können im Anschluss Anfragen (artefaktunabhängig) auf den XML-Strukturen erfolgen. Für Standardbeziehungen sind hierbei entsprechende Muster schon hinterlegt. Sextant bietet darüber hinaus die Möglichkeit Anfrageergebnisse visuell in einem Graph darzustellen und diesen explorativ zu erschließen. Für Java Quellcode können auf diese Weise sämtliche Beziehungen zwischen Typen, Methoden und Feldern untersucht werden [SEM05].

---

[7] http://www.eclipse.org/aspectj

[8] http://tyruba.sourceforge.net

[9] www.w3.org/TR/xquery

Motiviert durch den Erfolg, aber auch durch die schwäche hinsichtlich der Skalierbarkeit, von JQuery wurde das Eclipse Plug-in *CodeQuest* entwickelt. Hajiyev et al. [HVMV05] verwalten Typen, Methoden und Felder, sowie ihre Beziehungen untereinander, als Prolog-Fakten mit Hilfe eines skalierbaren und effizienten Relationalen Datenbank Management Systems. Als Anfragesprache verwenden sie den logikbasierten Prolog-Dialekt DataLog. CodeQuest wurde kürzlich in das kommerzielle Portfolio von *Semmle*[10] überführt [KHR07]. In diesem Zusammenhang wurde auch die neue objektorientierte, SQL-ähnliche, Anfragesprache *.QL* eingeführt, welche das Ziel verfolgt besonders nutzerfreundlich und intuitiv zu sein [MVH+07].

Der *JTransformer*[11], ebenso ein Eclipse Plug-in, lässt sich ähnlich gut auf große Systeme anwenden, ermöglicht aber zudem einen weitaus detaillierten Blick auf Java Quellcode. In einem initialen Schritt wird ein Java Projekt und zugehörige Bibliotheken in eine Prolog-Fakten-Datenbank überführt. Diese bildet den vollständigen Abstrakten-Syntaxbaum ab, d.h. auch Intra-Methodenelemente und sämtliche Abhängigkeiten zwischen den Konstrukten. Durchgeführte Änderungen, sowohl in der Datenbank als auch im Quellcode, werden in der jeweils anderen Repräsentationsform synchronisiert. Mit Hilfe von logikbasierten Anfragen können selbst feingranulare Beziehungen untersucht werden. Die Erstellung der Datenbank führt zwar zu längeren Startzeiten, ermöglicht aber eine schnellere Bearbeitung der einzelnen Anfragen, welche durch SWI Prolog[12] ausgewertet werden [KHR07]. JTransformer wurde darüber hinaus auch schrittweise zu dem sprachenunabhängigen *StarTransformer*[13] erweitert.

*JQueryScapes* von Markle und de Volder [MV08] integriert in Eclipse eine Reihe von Standardsichten, wie z.B. Package Explorer, Type-Hirarchy, Call-Hirarchy, u.v.m., welche alle durch JQuery-Anfragen generiert werden. Der Vorteil dieser Sichten besteht darin, dass sie sich explorativ und flexibel erweitern bzw. anpassen lassen. JQueryScapes gibt es auch als Variante mit dem JTransformer als Hintergrundplattform [KHR07].

### Eignung für das Feature Mining

Die in dieser Kategorie diskutierten Anfragekonzepte ermöglichen es sehr komplexe und zum Teil auch feingranulare Quellcodestruktur-Muster zu suchen. Entwickler können hierbei Anfragen formulieren, welche diverse Element- und Beziehungstypen enthalten. Anfrageergebnisse, z.B. dargestellt als Baum oder Graph, können auf die gleiche Weise

[10] http://semmle.com

[11] http://sewiki.iai.uni-bonn.de/research/jtransformer/start

[12] http://www.swi-prolog.org

[13] http://sewiki.iai.uni-bonn.de/research/jtransformer/startransformer

(explorativ) verfeinert werden, sodass sich schrittweise vollständigere Strukturen ergeben.

Solche Ansätze weisen im Hinblick auf die Feature-Lokalisierung ein erhebliches Defizit auf: Während bei einer rein textbasierten Suche, ein Anwender recht unkompliziert verschiedene Feature-Namen (und dessen Synonyme) angeben kann, erfordert das Aufstellen solcher Strukturanfragen ein viel genaueres Verständnis von den Zusammenhängen innerhalb einer Feature-Repräsentation. Erfahrungen zeigen aber, dass dem Anwender meist gar nicht eindeutig klar ist was er sucht, bevor er es auch tatsächlich gefunden hat. Selbst wenn es gelingt jene Zusammenhänge zu modellieren, so ist das Aufstellen von korrekten Anfragen, welche auch genau das liefern was man sucht, entweder unmöglich oder unpraktikabel [JV03; MRB+05; RM02b].

In *Kapitel 3.1* wurde festgestellt, dass „querschneidende Features durch unterschiedliche Erweiterungen an verschiedenen Stellen umgesetzt werden". Aufgrund dessen können in der Regel auch keine „pauschalen" Anfragen formuliert werden, die sich während der Feature-Lokalisierung effektiv wiederverwenden lassen.

### 4.1.1.4 Explorative Suche entlang von Strukturbeziehungen

Eine weitere Möglichkeit Feature-Repräsentation statisch zu lokalisieren, ist die explorative Suche entlang von Quellcodefragment-Abhängigkeiten und -Beziehungen. Jene lässt sich, als Variante der Tiefensuche, besonders strukturiert umsetzten: Schritt 1 – als Startpunkt wird die Klasse festgelegt, welche die *init()* oder *main()* – Methode enthält. Schritt 2 – der Anwender prüft die Klasse und legt fest ob jene das gesuchte Feature implementiert oder nicht. Wenn die Klasse das gesuchte Feature enthält wird die Suche beendet (Fall A), andernfalls müssen alle unterstützenden bzw. referenzierten Klassen ermittelt werden. Hiervon wird eine ausgewählt und nach dem Schema von Schritt 2 überprüft (Fall B). Ist man bei einer Klasse angekommen, die nicht das Feature implementiert, keine unterstützenden Klassen hat (oder schon alle von ihnen überprüft wurden), erfolgt der Rückschritt zu der zuvor betrachteten Klasse. Nun wird der Vorgang mit der nächsten unterstützenden Klasse nach dem Schema von Schritt 2 wiederholt (Fall C) [MRB+05]. Die Art der Beziehungen, welche verfolgt werden, sowie die Granularität der untersuchten Quellcodefragmente können bei diesem Verfahren beliebig variiert werden.

#### Konkrete Ansätze

Chen und Rajlich [CR00] stellen fest, dass die Aufgabe der Feature-Lokalisierung weder von Computern noch von Menschen allein erfolgreich gelöst werden kann. Sie schlagen daher ein strukturiertes, semiautomatisches Vorgehen vor, bei dem sie die beschriebenen Schritte der explorativen Suche, je nach Stärken auf Mensch oder Maschine verteilen.

Nach diesem Schema muss zunächst durch ein computergestütztes Werkzeug aus dem Quellcode ein *abstrakter System-Abhängigkeitsgraph* erzeugt und visualisiert werden. In diesem werden Typen, Methoden und Felder als Knoten abstrahiert. Kontrollfluss- und Datenflussbeziehungen zwischen den Knoten werden als Kanten abgetragen. Weiterhin muss durch das Werkzeug ein separater *Suchgraph* verwaltet werden. Jener nimmt während der explorativen Suche schrittweise die lokalisierten Feature-Knoten und -Kanten auf. Der Anwender muss hingegen einen Starpunkt festlegen, entscheiden welcher Knoten als nächstes besucht werden soll und ob dieser für das Feature relevant ist. Im Unterschied zu dem beschriebenen Standardvorgehen, zielen Chen und Rajlich in ihrer Methode auf eine vollständige Erfassung der Feature-Repräsentation ab. Daher muss der Anwender nach jedem lokalisierten Knoten festlegen, ob die Suche weiter fortgesetzt werden soll oder nicht.

Das Eclipse Plug-in *JRipples* setzt die Methode von Chen und Rajlich [CR00] für Java Quellcode um. Dort wird Feature-Lokalisierung als erster Schritt eines System-Änderungsauftrags angesehen, mit dem Startpunkte im Quellcode identifiziert werden können. Der Systemabhängigkeitsgraph wird auf der Ebene von Typen, Methoden und Feldern erzeugt und verwaltet. Knoten-Abhängigkeiten können optional durch Markierungen einzelner Quellcodezeilen innerhalb der Methoden eingeschränkt werden. Lokale Variablen bzw. Parameter werden nicht unterstützt. Der Such- und der Systemabhängigkeitsgraph werden in der gleichen Sicht dargestellt. Sie unterscheiden sich visuell, je nach Art der ermittelten Knoten-Markierungen, wie z.B. „Leer“, „Nächster“, „Unverändert“, „Propagiert“, „Lokalisiert“, etc. [BBPR05; PR09]. Neben der strukturellen Exploration wird durch die Integration von *Grep* sowie der *Lucene-Bibliothek* auch eine textbasierte Feature-Lokalisierung unterstützt (vgl. *Kapitel 4.1.1.1; 4.1.1.2*). JRipples wird in *Kapitel 4.2* im Kontext von Expansionstechniken noch einmal näher untersucht.

Es sei zudem angemerkt, dass sich die Suche nach Feature-Repräsentationen entlang von Abhängigkeiten und Beziehungen durch eine Reihe weiterer Werkzeuge umsetzen ließe. Grundsätzlich sind hierfür diverse hierarchische Browser geeignet, welche die Navigation entlang von Quellcodestrukturen unterstützen. Fast jede moderne *integrierte Entwicklungsumgebung*, wie z.B. *Eclipse*[14] oder *Microsoft Visual Studio*[15], stellt Sichten und Perspektiven bereit, mit denen unter anderem Zugehörigkeits-, Aufruf-, Zugriffs- sowie Vererbungsbeziehungen untersucht werden können.

Neben diesen gibt es auch frei verfügbare Werkzeuge, die auf eine integrierte Quellcode-Navigation spezialisiert sind, wie z.B. *The Source-Navigator IDE*[16] oder auch die bereits

---

[14] http://www.eclipse.org

[15] http://www.microsoft.com/germany/visualstudio/products/default.aspx

[16] http://sourcenav.sourceforge.net

diskutierten Plug-ins *JQuery*, *Sextant*, *CodeQuest*, *JTransformer* und *JQueryScapes* (vgl. *Kapitel 4.1.1.3*). Hilfreich in diesem Zusammenhang könnten auch Ansätze sein, die insbesondere auf eine graphische Exploration setzen, wie u.a. diskutiert in Storey et al. [SWM97; SBM01], Ducasse und Lanza [DL05] sowie Bohnet und Döllner [BD06a].

**Eignung für das Feature Mining**

Wenn die explorative Suche entlang von Abhängigkeiten und Beziehungen systematisch erfolgt, kann sie insofern vorteilhaft sein, da Feature-Repräsentationen (nahezu) vollständig lokalisiert werden können. Grundvoraussetzung ist aber hierbei zum einen, dass der Anwender mit dem System sehr gut vertraut ist und zum anderen, dass das System als solches übersichtlich strukturiert ist. Die Methode hat allerdings den Nachteil, dass sie für feingranulare Elemente und dessen Abhängigkeiten einen sehr großen Suchraum aufspannt – dessen Untersuchung kann durchaus sehr zeitintensiv sein. Für komplexe, objektorientierte, multi-thread Systeme können zudem statische Abhängigkeiten nicht immer exakt zugeordnet werden [EKS03; RW02; MRB+05; SMAP06].

### 4.1.2 Dynamische Techniken

Die generelle Idee der dynamischen Feature-Lokalisierung basiert auf der Annahme, dass Features durch bestimmte Test-Fälle, Szenarien oder Eingabedaten „ausgelöst" werden können. Daher werden während der Ausführung der Systeme so genannte *Ausführungsspuren* (engl. *execution traces*) protokolliert, d.h. ausgelöste Ereignisse, „besuchte" Blöcke, Methoden und Klassen oder gelesene bzw. geschriebene Variablen. Im Anschluss wird das Ziel verfolgt genau die Spuren zu isolieren, welche zu gewissen Features korrespondieren. Jenen können dann im Anschluss eindeutige Quellcodebereiche der Feature-Repräsentation zugeordnet werden [MRB+05; LMPR07; RW02; RP09; SE02; WS95]. Im Folgenden werden drei wesentliche Filter- bzw. Isolationskonzepte vorgestellt und mit einigen Arbeiten veranschaulicht.

#### 4.1.2.1 Mengenoperationen auf Ausführungsspuren

Wilde et al. [WGGS92; WS95, WC96] gelten als Pioniere des vollständig dynamischen Ansatzes, den sie als *Software Reconnaissance* bezeichnen. In diesem wird zunächst das zu untersuchende System so instrumentalisiert, dass Ausführungsinformationen protokolliert werden. Im Anschluss werden zwei Gruppen von Testfällen ausgeführt. Zum einen solche, die *ein* Feature „auslösen" ($\{a_1, a_2, \ldots, a_n\} \in A$) und zum anderen, welche die es nicht tun ($\{b_1, b_2, \ldots, b_n\} \in B$). Durch Mengenoperationen auf den erzeugten Ausführungsspuren, wie z.B. „$a_1 \cap a_2 \cap \ldots \cap a_n$" oder „$A \setminus B$", können Elemente isoliert werden, die zu dem Fea-

ture (höchst wahrscheinlich) korrespondieren. Die auf diese Weise identifizierten Elemente müssen dann manuell inspiziert und bewertet werden.

### Konkrete Ansätze

Zur Unterstützung der Reconnaissance-Technik wurde das Werkzeug *RECON*[17] implementiert. Aktuell ist dieses bereits in der vierten Version verfügbar, welche die Programmiersprachen C, C++ und Fortran77 unterstützt.

Reps et al. [RBDL97] wenden ein vergleichbares Verfahren im Kontext des Jahr-2000-Problems an. Sie verwenden anstatt von Testfällen Eingabedaten, die kleiner bzw. größer sind als „2000". Die Ausführungsspuren, welche durch die instrumentalisierte Anwendung erzeugt werden, interpretieren sie als *Pfadspektren*. Zur Ermittlung der Fehler verursachenden Quellcodebereiche führen sie so genannte *Spektralvergleiche* durch.

Wong et al. [WGHT99] zeigen wie sich der Ansatz von Wilde et al. auch auf feingranularer Ebene, bis hin zu lokalen Variablen, umsetzen lässt. Sie instrumentalisieren ein System so, dass nicht nur Kontroll- sondern auch Datenflüsse erfasst werden. Solche Ausführungsspuren bezeichnen sie als *Dynamic Slices*. Mit dieser Methode werden deutlich präzisere und vollständigere Feature-Repräsentationen lokalisiert, die nur in geringem Maße nachträglich manuell erweitert werden müssen.

Deprez und Lakhotia [DL00] stellen fest, dass die Identifizierung und Zuordnung von Szenarien, welche das Feature auslösen bzw. es nicht auslösen, den zeitaufwändigsten Schritt darstellt. Zur Verbesserung von Software Reconnaissance diskutieren sie eine *Formalisierung*, mit dem jener Schritt weitestgehend automatisiert werden kann. Ihr Vorschlag sieht eine Feature-Annotierung einer *Eingabedaten-Grammatik* vor, mit der dann feature-spezifische Werte erzeugt werden können.

### Eignung für das Feature Mining

Ansätze welche auf Mengenoperationen basieren zeichnen sich durch ihr simples Prinzip sowie ihre einfache Anwendbarkeit aus. Erfahrungen haben gezeigt, dass auf diese Weise hilfreiche Startpunkte einzelner Features identifiziert werden können. Jene Methoden haben allerdings den Nachteil, dass sie nicht explizit zufällige Störgeräusche, Beobachtungseinflüsse oder auch Abhängigkeiten zu anderen Features beachten. Es wird z.B. auch keine Unterscheidung zwischen „einmalig" und „mehrmalig" auftretenden Elementen getroffen. Darüber hinaus umfassen solche Ansätze keine konkreten Maßnahmen mit

[17] http://www.cs.uwf.edu/~recon

denen irrelevante Ereignisse, wie z.B. Start- oder Abschaltungsvorgänge, herausgefiltert werden können [AG05; AMGS05; EKS03; ESW04; LWS00].

### 4.1.2.2 Markierung von Ausführungsspuren

In der Vergangenheit wurden Methoden vorgeschlagen, die auf das separate Ausführen von Szenarien, welche ein Feature *nicht* auslösen, verzichten. Stattdessen wird das System lediglich einmal gestartet, wobei *feature-auslösende* Szenarien nacheinander abgearbeitet werden. Parallel dazu werden in den Ausführungsspuren die Start- und Endzeitpunkte der Szenarien festgehalten. Ein Bereich zwischen solch einem Start- und Endpunkt bezeichnet man als *markierte Ausführungsspur* [RP09]. Elemente werden als feature-relevant eingestuft, wenn sie beispielsweise zum ersten Mal in einem markierten Bereich auftreten.

**Konkrete Ansätze**

*TraceGraph* von Lukoit et al. [LWS00] implementiert genau diese Methode. Mit Hilfe eines Monitors kann die Ausführungsspur eines instrumentalisierten Programms während dessen Ausführung beobachtet werden. Für jede Komponente, deren Granularität beliebig eingestellt werden kann, wird die Verwendung entlang einer Zeitskala protokolliert – ähnlich wie bei einem Oszilloskop. Ein Anwender kann Veränderung, die durch das Auslösen eines Features hervorgerufen werden, daher gleich direkt beobachten. TraceGraph ist mittlerweile auch fester Bestandteil, in dem bereits angesprochenen Werkzeug, RECON (vgl. *Kapitel 4.1.2.1*).

Der *Multi-Threaded Tracer - MuTT*[18] ist ein Opensource-Werkzeug, basierend auf der *Java Platform Debugger Architecture*[19], mit dessen Hilfe markierte Ausführungsspuren für Java Anwendungen erzeugt werden können. Für (über MuTT gestartete) Programme wird eine „Start/Stop - Aufzeichnungsfunktion“ zur Verfügung gestellt, sodass während der Ausführung aufgerufene Methoden je nach Bedarf protokolliert werden können.

Salah und Mancoridis [SM04] lokalisieren mit Hilfe der Markierungstechnik mehrere Features nacheinander. Aus den markierten Ausführungsspuren wird abgeleitet, welche Objekte und Klassen aufgerufen wurden und wie sie zueinander in Beziehung stehen. Jene Beziehungen werden in einer *Objekt-Interaktions-* sowie *Klassen-Interaktionssicht* dargestellt. Darauf aufbauend werden zudem eine *Feature-Interaktions-* und *Feature-Implementierungssicht* abgeleitet. Die zuletzt genannte Sicht zeigt (recht grobgranular) auf Klassenebene die Zuordnung von unterschiedlichen Features zu ihren Repräsentationen.

[18] http://sourceforge.net/projects/muttracer

[19] http://java.sun.com/javase/technologies/core/toolsapis/jpda

In Folgearbeiten präzisieren Salah, Mancoridis et al. [SMAP05; SMAP06] ihren Ansatz bis hin zu der Methodenebene und führen hierfür auch entsprechende Sichten ein. Darüber hinaus schlagen sie Metriken vor, welche basierend auf der Methodenzuordnung die *Gleichheit zweier Features quantifizieren* können.

**Eignung für das Feature Mining**

Im Unterschied zu Ansätzen, die Mengenoperationen nutzen, haben Markierungstechniken den Vorteil, dass sie irrelevante Ereignisse, wie z.B. Start- oder Abschaltungsvorgänge, herausfiltern können. Hierdurch wird die zu untersuchende Datenmenge deutlich reduziert. Zudem brauchen nur noch feature-auslösende Szenarien formuliert werden, welches eine Zeit- und Aufwandsersparnis bedeuten kann. Wie in dem Beispiel von TraceGraph, ist es sogar möglich die Auswertung stark zu vereinfachen, indem zum ersten Mal auftretende Elemente in einem markierten Bereich direkt in einem Überwachungsmonitor hervorgehoben werden. Markierungstechniken allein, d.h. nicht unterstützt durch weitere Konzepte, können aber durchaus anfällig für Störeinflüsse sein, da sie keine Möglichkeiten bieten unterschiedliche Ausführungsspuren desselben Features oder von anderen Features direkt zu kombinieren [AG06; LMPR07; LWS00; RP09; SMAP06].

## 4.1.2.3 Rangordnung nach Zugehörigkeit

Aufgrund von möglichen Störereignissen, die beispielsweise durch eine verteilte Ausführung von Prozessen oder durch abhängige Features verursacht werden, können Komponenten bzw. Elemente nur selten eindeutig einem Feature zugeordnet werden. Daher werden Ansätze benötigt, die unter Berücksichtigung solcher Nebeneffekte eine (abgestufte) Empfehlung zur Relevanz bzw. Spezifität eines Elements gegenüber einem Feature ausdrücken können. Jüngst wurden einige Metriken und Konzepte vorgeschlagen, die genau das umsetzen können [EAAG08; PGM+07].

**Konkrete Ansätze**

Edwards et al. [ESW04] unterbreiten einen Vorschlag, wie eine dynamische Feature-Lokalisierung in verteilten Systemen aussehen könnte. Sie erweitern die Markierungstechnik auf parallele Ausführungseinheiten und ermitteln für Elemente (vgl. *Kapitel 4.1.2.2*), die innerhalb von markierten Bereichen liegen einen Relevanzindex. Hierbei wird die *(absolute) Aufrufhäufigkeit einer Komponente innerhalb von markierten Bereichen zur gesamten (absoluten) Aufrufhäufigkeit dieser Komponente, d.h. auch in nicht markierten Bereichen, ins Verhältnis* gesetzt.

Antoniol und Guéhéneuc [AG05; AG06] bauen auf den Ergebnissen von Edwards et al. auf. Sie verwenden aber nicht die Markierungstechnik, sondern zwei Gruppen von Szena-

rien, um zum einen *feature-relevante* ($F'$) und *feature-irrelevante* ($F$) Ausführungsspuren zu generieren. Sie erweitern den Ansatz um das so genannte *Knowledge-based Filtering*, bei dem zunächst offensichtlich irrelevante Ereignisse aus $F'$ manuell entfernt werden. Um Störeinflüsse weiter einzudämmen, passen sie auch die Relevanzmetrik an. Antoniol und Guéhéneuc setzen die *absolute Aufrufhäufigkeit einer Komponente bzw. eines Ereignisses, zum einen in $F'$ und zum anderen in $F$, zur gesamten Aufrufhäufigkeit aller Ereignisse in $F'$ bzw. in $F$ ins Verhältnis*. Erst jene gewichteten Häufigkeiten werden dann in die Formel von Edwards et al. [ESW04] eingesetzt. Sie nennen die angepasste Metrik *Probabilistic Ranking*.

Eisenberg und de Volder [EV05] berücksichtigen in ihrem Rangordnungsansatz für *Methoden*, nicht nur *auslösende* und *nicht auslösende* Testfälle eines Features, sondern *einer ganzen Reihe von Features*. Aus einer verfügbaren *JUnit Test Suite* [MH03] für eine Java Anwendung werden manuell Mengen von Testfällen gebildet, die jeweils ein bestimmtes Feature auslösen. Alle übrig gebliebenen Testfälle werden ohne Feature-Zuordnung in semantischen Gruppen zusammengefasst. Für jedes Feature gibt es demzufolge genau *eine auslösende* Menge und *viele nicht-auslösende* Testfallmengen. Mit Hilfe des eigens entwickelten *Dynamic Feature Trace* Werkzeuges werden alle Testfälle automatisch nacheinander durchgeführt und anschließend ausgewertet. Für jede Testfallmenge werden alle aufgerufenen Methoden aus der Ausführungsspur (in [EV05] als Dynamic Feature Trace bezeichnet) ermittelt und anhand einer Rangfolge sortiert.

Jene Rangwerte ergeben sich aus der Kombination der Metriken *Multiplizität*, *Spezialisierung* und *Tiefe*. Die Multiplizität sorgt dafür, dass eine Methode als besonders feature-relevant eingestuft wird, wenn sie in der aktuell betrachteten Testfallmenge von „vielen“ Testfällen aufgerufen wird. Die Spezialisierung bezeichnet eine Methode hingegen als feature-relevant, wenn sie nicht im Kontext von „vielen“ unterschiedlichen Features verwendet wird. Mit Hilfe der Metrik Tiefe wird ausgedrückt, dass eine Methode, welche sehr „nah“ an der Wurzel eines Aufrufgraphs liegt, d.h. direkt aufgerufen wird, für das aktuell betrachtete Feature relevanter ist, als eine Methode, die nah an einem Blatt liegt, d.h. indirekt durch andere Methoden aufgerufen wird. Unpräzise Angaben, wie „viele“ bzw. „nah“, verstehen sich in dem Kontext als relative Werte über alle Testfallmengen. Die sortierte Methoden-Ergebnismenge wird als *JQuery*-Sicht dargestellt (vgl. *Kapitel 4.1.1.3*).

Eaddy et al. [EAAG08] *kombinieren* in ihrer Arbeit *Probabilistic Ranking* [AG05] mit der *Spezialisierungsmetrik aus dem Dynamic Feature Trace Ansatz* [EV05]. Die daraus resultierende Metrik *Element Frequency-Inverse Concern Frequency* ist zudem angelehnt an die im Information Retrival häufig genutzte *TF-IDF-Termgewichtungsformel* [BN99]. Im Wesentlichen wird das Probabilistic Ranking dahingehend angepasst, dass zusätzlich für das untersuchte Element die Wahrscheinlichkeit der Ausführung durch ein anderes Feature berücksichtigt wird. Jene neue Metrik wird sowohl für Methoden als auch Felder angewendet [EAAG08].

Eisenbarth et al. [EKS01a; EKS01b] stellen fest, dass sich Szenarios, welche genau nur ein Feature auslösen meist sehr schwer umsetzen lassen. Gelingt jenes nicht, können allerdings Feature-Abhängigkeiten in einfachen Analysen, wie z.B. in Software Reconnaissance (vgl. *Kapitel 4.1.2.1*), zu unpräzisen oder fehlerhaften Ergebnissen führen. Eisenbarth et al. ordnen daher Szenarios alle Features zu, welche ausgelöst werden. Aus den markierten Ausführungsspuren wird eine Relationstabelle erzeugt, welche für jedes Feature alle zugehörige Methoden bzw. für jede Methode alle zugehörigen Features enthält. Basierend auf jener Tabelle werden mit Hilfe der *formalen Begriffsanalyse* [Sne98] Konzepte gebildet, die aus einer Menge von Features und Methoden bestehen sowie definierte Bedingungen erfüllen. Jene Konzepte werden in einem Konzept-Gitter angeordnet. Eisenbarth et al. formulieren unterschiedliche Interpretationshinweise für solche Gitter, mit denen Feature-zu-Feature, Methode-zu-Methode sowie *Methode-zu-Feature* Beziehungen abgeleitet werden können. Hierdurch lässt sich im Anschluss unter anderem eine Aussage darüber treffen, *wie spezifisch bzw. relevant Methoden für einzelne Features (bzw. für eine Menge von Features)* sind.

Tonella und Ceccato [TC04] wenden im Kontext des Aspect Mining ein vergleichbares Vorgehen wie Eisenbarth et al. [EKS01a; EKS01b] an (vgl. *Kapitel 3.2*), um ein Konzept-Gitter zu erzeugen. Sie sind allerdings in der Auswertung an Konzepten interessiert, die nur ein Szenario bzw. Feature enthalten und dessen Methoden *Tangling- und Scattering-Effekte* (vgl. *Kapitel 2.3.2*) aufweisen.

Koschke und Quante [KQ05] untersuchen die Anwendbarkeit des Ansatzes von Eisenbarth et al. [EKS01a; EKS01b] auf der feingranularen Ebene einzelner Anweisungsblöcke. Sie stellen fest, dass gerade bei sehr vielen Features und komplexen Systemen das erzeugte Konzeptgitter (für Menschen) in der Regel unhandhabbar wird, sodass dann eher andere Ansätze zu bevorzugen sind. Für *kleine und übersichtliche Systeme* mit nur wenigen Features kann ein solches Vorgehen allerdings *wertvolle Erkenntnisse* bringen.

### Eignung für das Feature Mining

Die beschriebenen Ansätze, welche eine Zugehörigkeitsrangordnung erzeugen können, verwenden für ihre Analysen fast alle markierte Ausführungsspuren. Infolgedessen ist die Diskussion hinsichtlich der Vorteile von Markierungstechniken auch für diese Kategorie zutreffend. Deren Nachteile gelten hier allerdings nicht, da Rangordnungskonzepte direkte Hilfestellungen geben, um den Effekt von „Störeinflüssen“, verursacht durch Feature-Abhängigkeiten oder verteilte Prozesse, zu quantifizieren. Durch die Einführung von Rangfolgen, in denen alle ausgeführten Komponenten auftreten, müssen aber Nutzerinteraktionen oder Grenzwertannahmen erfolgen, um ein finales Urteil hinsichtlich der Zugehörigkeit zu treffen [AMGS05; EV05]. Manuelle Nutzerinteraktionen verursachen zusätzlichen Aufwand. Ungeeignete Grenzwerte können wiederum zu einem sehr großen Suchraum führen oder unter Umständen relevante Elemente ausblenden.

### 4.1.3 Hybride Techniken

Werden für die Feature-Lokalisierung sowohl statische als auch dynamische Informationen verwendet, so wird von hybriden Techniken gesprochen. Im Folgenden werden einige Arbeiten und Ansätze aus dieser Gruppe vorgestellt, welche hybride Informationen verwenden, um *präzisere* Startpunkte bzw. Teilmengen von Feature-Repräsentationen zu *lokalisieren*. Hybride Techniken, welche hingegen das Ziel verfolgen *vollständigere* Feature-Repräsentationen zu erfassen, und dabei auf *Expansionstechniken* zurückgreifen, werden in *Kapitel 4.2* besprochen.

**Konkrete Ansätze**

Simmons et al. [SEW+06] stellen fest, dass für eine praxistaugliche Kombination von dynamischen und statischen Techniken auch zuverlässige und erprobte Werkzeuge gebraucht werden. Als geeignete Vertreter haben sie hierfür *CodeTest* von Metrowerks Corp (bzw. Freescale)[20] und *inSight* von Klocwork Inc[21] identifiziert. CodeTest instrumentalisiert Software so, dass während der Ausführung Informationen zur Systemleistung, Testabdeckung und Speicherverbrauch gesammelt werden können. Hierdurch können aber auch Quellcode-Fragemente lokalisiert werden, die auf eine Feature-Repräsentation hinweisen. InSight ermöglicht hingegen darauf aufbauen diverse statische Kontext- und Beziehungsanalysen der jeweiligen Kandidaten, um sie genauer zu untersuchen. Erste Fallstudien der Autoren weisen darauf hin, dass sich beide Werkzeuge sehr gut gegenseitig ergänzen.

Poshyvanyk et al. [PGM+07] kombinieren die *Probabilistic Ranking Metrik* (vgl. *Kapitel 4.1.2.3*) mit einer textbasierten Suche, welche durch die *Latent Semantic Indexing* Methode aus dem *Information Retrieval* unterstützt wird (vgl. *Kapitel 4.1.1.2*). Abgekürzt verwenden sie hierfür den Namen *PROMESIR – Probabilistic Ranking Of Methods based on Execution Scenarios and Information Retrieval*. Den statischen sowie den dynamischen Teil sehen sie in ihrem Ansatz als eigenständige Experten, welche unabhängig von einander Quellcodefragmente (in dem Fall Methoden und Klassen) hinsichtlich ihrer Relevanz zu einem Feature bewerten. Durch die *affine Transformation* [Jac95] wird, gemäß einer Vertrauensgewichtung, das Urteil beider „Experten“ vereint. Darüber hinaus zeigen Poshyvanyk et al. [PGM+07] anhand von unterschiedlichen Fallstudien, dass die Kombination beider Experten bessere Ergebnisse liefert als die jedes Einzelnen.

Der *SITIR (Single Trace and Information Retrieval)* Ansatz von Liu et al. [LMPR07] verwendet ebenso eine textbasierte Suche, welche durch die *Latent Semantic Indexing* - Methode

[20] http://www.metrowerks.com/MW/Develop/AMC/CodeTEST

[21] http://www.klocwork.com/products/insight/

aus dem *Information Retrieval* unterstützt wird (vgl. *Kapitel 3.4.1.1*). Der wesentliche Unterschied zu PROMESIR ist allerdings hierbei, dass für die Erzeugung von feature-relevanten Ausführungsspuren Markierungstechniken oder *nur feature-auslösende Szenarien verwendet* werden. Da infolgedessen die Probabilistic Ranking Metrik nicht ermittelbar ist (vgl. *Kapitel 4.1.2.3*), wird die Relevanz der in den Ausführungsspuren enthaltenen Methoden und Klassen mit ihrem Übereinstimmungsgrad zu einer natürlich sprachlichen Feature-Beschreibung gleichgesetzt.

Das freiverfügbare Eclipse Plug-in *Feature Location and Textual Tracing Tool – FLAT³* von Savage et al. [SRP10] setzt für Java Anwendungen die SITIR Methode um.[22] FLAT³ nutzt intern für den dynamischen Teil das Werkzeug *MuTT* (vgl. *Kapitel 4.1.2.2*). *Lucene* realisiert hingegen die IR-basierte Suche (vgl. *Kapitel 4.1.1.2*). Mit Hilfe eines weiteren verwendeten Werkzeuges, dem ConcernTagger, lassen sich alle definierten Features und die entsprechend lokalisierten Typen, Methoden und Felder verwalten. Auf den *ConcernTagger* wird in *Kapitel 4.3* noch einmal näher eingegangen. FLAT³ bieten darüber hinaus die Möglichkeit Feature-Repräsentationen oder Suchergebnisse visuell darzustellen. Hierbei werden, ähnlich wie in dem *AspectBrowser* (vgl. *Kapitel 4.1.1.1*), Klassen als Rechtecke dargestellt, in denen die identifizierten Elemente als Linien hervorgehoben werden. Je nach Relevanz- bzw. Zugehörigkeitsgrad, wird den Linien eine unterschiedliche Farbsättigung zugewiesen.

### Eignung für das Feature Mining

Aus der Diskussion in *Kapitel 4.1.1* und *4.1.2* lässt sich ableiten, dass *statische Ansätze* nur dann anwendbar sind, wenn die Struktur oder das verwendete Vokabular des Systemquellcodes bekannt sind. Mit Hilfe von textbasierten Methoden und zielsicheren Anfragen lassen sich „kleine“ Teilmengen bzw. Startpunkte einer Feature-Repräsentation einfach und schnell lokalisieren. Struktur- bzw. abhängigkeitsbasierte Ansätze ermöglichen wiederum eine feingranulare, gründliche und nahezu vollständige Lokalisierung von Features jeglicher Art – erfordern aber in der Regel tiefgehenderes Systemverständnis und höheren manuellen Aufwand. Statische Techniken haben darüber hinaus den Vorteil, dass sie sich auch auf fehlerhaften oder unvollständigen Quellcode anwenden lassen [EKS01b; PGM+07; RM03; SEW+06; WBP+03; WBPR01].

*Dynamische Ansätze* bestechen hingegen in der Regel durch eine einfache und schnelle Anwendbarkeit, wenn nach außen sichtbare bzw. auslösbare Features grobgranular lokalisiert werden müssen. Im Vergleich zu statischen Techniken sind sie besonders dann hilfreich, wenn ein Feature durch verstreute, nicht zusammenhängende Quellcodebereiche repräsentiert wird. Dynamische Ansätze lassen sich allerdings nur für Systeme anwen-

[22] http://www.cs.wm.edu/semeru/flat3

den, die fehlerfrei kompilier- und ausführbar sind. Zudem verlangen sie von dem Anwender (anstatt „interne" Quellcode-) „externe" Systemfunktionskenntnisse, damit Eingabedaten oder Szenarien formuliert werden können, die ein Feature zur Laufzeit auslösen. Um eine Feature-Repräsentation durch dynamische Techniken vollständig und präzise erfassen zu können, sind Szenarien notwendig, welche das korrespondierende Feature in all seinen Facetten auslösen, aber gleichzeitig ein Minimum an „Fremdeinflüssen" durch Basisfunktionen oder korrelierte Features zulassen. In der Praxis ist diese Bedingung vermutlich nur bedingt realisierbar, sodass die Qualität der Ergebnismenge stark variieren kann. Lässt sich ein Feature nicht durch eine Systeminteraktion oder spezifische Eingabedaten „erreichen", so kann es auch nicht durch dynamische Techniken lokalisiert werden [EKS01a; EKS01b; EKS03; ESW04; CZD+08; LMPR07; PGM+07; RM03; Rob05; RW02; SEW+06; WBP+03; WBPR01; WGH00].

Zusammenfassend lässt sich festhalten, dass sich die beschriebenen Eigenschaften von statischen und dynamischen Konzepten komplementär ergänzen. Es werden also durch das eine Konzept stets gewisse Aspekte berücksichtigt, die dem jeweils anderen in der Regel verborgen bleiben. Die Kombination statischer und dynamischer Techniken führt daher zu positiven Synergieeffekten, sodass (im Vergleich zu der Ausführung beider Konzepte für sich allein) Startpunkte bzw. Teilmengen von Feature-Repräsentationen präziser lokalisiert werden können [AMGS05; EAAG08; LMPR07; MRB+05; MSRM04; PGM+07].

## 4.2 Expansionstechniken

In dieser Arbeit werden Expansionstechniken interpretiert als jegliche Konzepte, Methoden und Werkzeuge, welche *anhand bereits lokalisierter Quellcodefragmente Schlussfolgerungen zu möglichen weiteren syntaktisch und semantisch zugehörigen Elementen* treffen können und den Anwender dabei unterstützen *vollständigere* Repräsentationen zu *lokalisieren* (vgl. *Kapitel 3.1).*

Im Folgenden werden verschiedene solcher Techniken dargestellt. In einigen dieser Ansätze werden dynamische Informationen genutzt, um Startpunkte oder Teilmengen der Feature-Repräsentation zu lokalisieren (vgl. *Kapitel 4.1)*. Die *eigentlichen Expansionstechniken* beruhen allerdings alle auf statischem Quellcode. Daher wird im Gegensatz zu *Kapitel 4.1* keine zusätzliche Einteilung in dynamische, statische und hybride Techniken vorgenommen.

### 4.2.1 Iterative Suchanfragen und Szenarien

Grundsätzlich lässt sich festhalten, dass die Feature-Lokalisierung und -Expansion eng mit einander verwandt sind und ein ähnliches Ziel verfolgen. Der wesentliche Unterschied begründet sich vielmehr in dem Fokus des zu lösenden Problems und der verfügbaren Informationen (vgl. *Kapitel 3.1; 4.1*). Aufgrund jener Nähe können viele der in *Kapitel 4.1* diskutierten Lokalisierungstechniken auch für die Feature-Expansion adaptiert werden. Hierbei werden allerdings Suchanfragen oder Szenarien nicht nur einmalig ausgeführt sondern mehrfach hintereinander eingesetzt. Hierbei wird in jeder Iteration das neu gewonnene Wissen zu dem System verwendet, um in der Folge „bessere" Anfragen oder Szenarien zu formulieren.

#### Konkrete Ansätze

Soll beispielsweise das Feature „Zahlung" lokalisiert werden, könnte initial mit Hilfe der *textbasierten Mustersuche* nach Stellen gesucht werden (vgl. *Kapitel 4.1.1.1*), welche die Wörter „Zahlung", „zahle", etc. enthalten. Die manuelle Inspektion der Suchtreffer im Quellcode könnte unter anderem ergeben, dass auch Wörter wie „Rechnung" und „Geld" höchstwahrscheinlich zu dem Feature korrespondieren. In der nächsten Iteration wird daher eine Anfrage formuliert, die zusätzlich auch diese beiden neuen Wörter umfasst, usw.

An dieser Stelle sei darauf verzichtet alle Lokalisierungstechniken noch einmal hier unter diesem Blickwinkel zu diskutieren. Die grundsätzliche Idee, wie sie für die textbasierte Mustersuche exemplarisch dargestellt wurde, lässt sich auch für andere Lokalisierungstechniken, welche in den *Kapiteln 4.1.1.2, 4.1.1.3, 4.1.2, 4.1.3* beschrieben sind, adaptieren.

#### Eignung für das Feature Mining

Iterative Suchanfragen und Szenarien können durchaus hilfreich sein, um Teilmengen von Feature-Repräsentationen zu expandieren. Jene Techniken sind allerdings in der Regel darauf ausgerichtet, dass sie nur einmalig ausgeführt werden. Zwischenergebnisse werden daher nicht gespeichert bzw. nicht geeignet verwaltet, sodass der Anwender ab einer hohen Zahl von Iterationen möglicherweise den Gesamtüberblick verlieren kann.

### 4.2.2 Expansion basierend auf Textinformationen

Im Rahmen der Feature-Expansion sind schon einige Quellcodefragmente einer Feature-Repräsentation gegeben. Aus den Namen der entsprechenden Bezeichner oder auch der zugehörigen Kommentare können Informationen hinsichtlich des im Quellcode verwendeten Feature-Vokabulars abgeleitet werden. Jene Informationen können genutzt werden,

um nicht identifizierte Typen, Methoden, Felder und Variablen hinsichtlich ihrer Feature-Zugehörigkeit zu bewerten.

### Konkrete Ansätze

Shepherd und Pollock [SP05] haben in ihrem *Aspect Miner and Viewer* eine einfache Abstandsfunktion implementiert, die auf der Länge gemeinsamer Teilzeichenketten von Methodennamen basiert. Mit Hilfe des *agglomerativen hierarchischen Clusterings* [TSK06, S.515ff.] gruppieren sie schrittweise alle Methoden, die hinsichtlich dieser Abstandsfunktion besonders nah zu einander sind. Jene ermittelten Gruppen bzw. Cluster von Methoden können dafür genutzt werden, um zu einer gegebenen Methode mögliche weitere ähnliche Kandidaten der Feature-Repräsentation zu ermitteln.

Mens und Tourwe' [MT05] wenden die *formale Begriffsanalyse* [Sne98] *auf Teilzeichenketten von Klassen- und Methodennamen* an. Gesucht werden Konzepte bestehend aus möglichst großen Gruppen von Methoden und Klassen, welche gemeinsame Teilzeichenketten haben. Alle Mitglieder von Konzepten, die lokalisierte Fragmente enthalten, gelten als mögliche weitere Repräsentationskandidaten.

Die Annahme, dass querschneidende Belange oft auf einem einheitlichen Vokabular basieren, nutzen Shepherd et al. [STP05] auch in ihrem Ansatz *Language Clues*. Hierbei werden sämtliche Kommentare, Typen-, Methoden- und Feldnamen in Teilzeichenketten zerlegt und unter Zuhilfenahme einer *Wortdatenbank* zu *semantisch zusammengehörigen Ketten* verbunden. Ähnlich wie in dem Clustering-Ansatz können die korrespondierenden Quellcodefragmente der Wortketten, welche lokalisierte Elemente enthalten, auf potentielle Kandidaten hinweisen.

Revelle und Poshyvanyk [RP09] vergleichen in einer empirischen Studie unterschiedliche Lokalisierungstechniken. In diesem Zusammenhang schlagen sie auch vor, den *Quellcode von bereits lokalisierten Methoden oder Typen als Anfragen für eine IR-basierte Textsuche* zu verwenden. Gerade in Kombination mit dem Latent Semantic Indexing können auf diese Weise „verwandte" Fragmente identifiziert werden, welche nicht exakt die selben Terme sondern mögliche Synonyme verwenden (vgl. *Kapitel 4.1.1.2*).

### Eignung für das Feature Mining

Textinformationen können zweifelsohne einen wertvollen Beitrag für die Feature-Expansion leisten. Allerdings sind sie an die grundlegende Voraussetzung gebunden, dass die Teilmenge des bekannten Vokabulars im Quellcode einheitlich und unter Beachtung gewisser Namenskonventionen benutzt wurde. Von dieser Annahme kann höchstwahrscheinlich fast immer ausgegangen werden, dennoch könnte es aber auch Fälle geben für die sie nicht zutrifft.

### 4.2.3 Expansion basierend auf Klonen/vergleichbaren Eigenschaften

Studien haben gezeigt, dass sich in einigen Systemen zum Teil bis zu 59% des Gesamtquellcodes auf Klone zurückführen lassen. Sprich, auf Segmente die bezüglich einer Gleichheitsdefinition identisch zu einander sind [Kos07]. Diese Beobachtung lässt sich unter Umständen auch für die Feature-Expansion verwenden. Unter der Annahme, dass identische Umsetzungen das selbe Feature repräsentieren, könnten zu bereits lokalisierten Quellcodebereichen alle Klone ermittelt werden und als weitere Repräsentationskandidaten vorgeschlagen werden.

#### Konkrete Ansätze

Die Erkennung von Codeklonen oder -Duplikaten ist ein eigenständiges etabliertes Forschungsfeld. Hierzu wurden bereits unzählige Publikationen veröffentlich, die an dieser Stelle nicht alle dargestellt werden können. Für eine umfassende Übersicht sei z.B. die Arbeit von Koschke [Kos07] empfohlen. Um zumindest einen groben Überblick zu geben, werden im Folgenden exemplarisch Ansätze aus den drei typischen Teilgebieten der Duplikaterkennung herausgegriffen.

Bruntink et al. [BDET04] untersuchen Klonerkennung auf *Zeichen- und abstrakter Syntaxbaumebene*. *SimScan*[23] ist ein (für nicht kommerzielle Zwecke freies) Eclipse Plug-in, welches Klone auf jener Ebene zuverlässig erkennen kann.

Krinke [Kri01] und Shepherd et al. [SGP04] entwickeln hingegen Ansätze für *semantische Beziehungen auf Systemabhängigkeitsgraphen*. Das Eclipse Plug-in *Ophir* [SP03] unterstützt Klonerkennung auf jener semantischen Ebene.

Ein weiteres Konzept zur Identifizierung von Quellcodebereichen mit *vergleichbaren Eigenschaften* setzt das Eclipse Plug-in *Timna* um [SPPC05]. Anhand von bereits identifizierten Trainingsbeispielen werden Werte für zahlreiche Eigenschaften von querschneidenden Belangen ermittelt. Diese dienen als Eingabe für einen Algorithmus mit dem ein Klassifikator abgeleitet wird, welcher mit einer gewissen Sicherheit eine Entscheidung darüber treffen kann, ob ein Quellcodefragment zu dem selben Belang gehört oder nicht [TSK06, S. 145ff.]. Dieser Klassifikator wird dann auf alle unbekannten Quellcodefragmente angewendet, um weitere Repräsentationskandidaten zu identifizieren [SPPC05].

[23] http://blue-edge.bg/download.html

### Eignung für das Feature Mining

Auch wenn Klonerkennungstechniken sehr zuverlässig Duplikate lokalisieren können, impliziert es nicht zwangsläufig, dass alle identifizierten Bereiche auch zu der selben Feature-Repräsentation gehören. Teilweise entstehen Klone unabsichtlich oder beziehen sich auf einen anderen Kontext [Kos07]. Infolgedessen müssen alle auf diese Weise ermittelten Kandidaten sorgfältig geprüft werden. In *Kapitel 3.1* wurde zudem festgestellt, dass eine „einfache“ Suche nach Quellcodeduplikaten in der Regel unzureichend ist, da in den meisten Fällen querschneidende Features durch unterschiedliche Erweiterungen an verschiedenen Stellen umgesetzt werden [Ape07; LAL+10].

## 4.2.4 Explorative Expansion entlang von Strukturbeziehungen

Die explorative Suche entlang von Quellcodefragment-Abhängigkeiten und -Beziehungen lässt sich nicht nur als Lokalisierungs- sondern auch als Expansionstechnik anwenden (vgl. *Kapitel 4.1.1.4*). Als Einstiegspunkte werden die bereits lokalisierten Quellcodefragmente genutzt. Alle hierzu abhängigen bzw. in Beziehungen stehenden Elemente stellen potentielle Repräsentationskandidaten dar. Infolgedessen müssen sie alle systematisch nacheinander überprüft werden. Für alle weiteren auf diese Weise lokalisierten Fragmente wird der Vorgang rekursiv wiederholt.

Aufgrund der Nähe dieser Vorgehensweise zu dessen Pendant in der Feature-Lokalisierung, gilt die Diskussion in *Kapitel 4.1.1.4* bis auf den Unterschied der Startpunktwahl ohne Abstriche auch in diesem Kontext. Im Folgenden werden weitere Arbeiten, Techniken und Werkzeuge aus diesem Themenbereich vorgestellt, welche allerdings das Verfolgen von Strukturabhängigkeiten und -Beziehungen eher im Zusammenhang mit der Feature-Expansion diskutieren.

### Konkrete Ansätze

Das *Feature Exploration and Analysis Tool – FEAT* lässt sich als klassischer Vertreter für die Umsetzung des semiautomatischen Vorgehens von Chen und Rajlich [CR00] im Kontext der Feature-Expansion bezeichnen (vgl. *Kapitel 4.1.1.4*). Dieses Eclipse Plug-in von Robillard und Murphy [RM02a; RM07] abstrahiert Java Quellcode in einem programmiersprachen-unabhängigem *Systemabhängigkeitsgraph*. Felder, Methoden und Typen repräsentieren hierbei Knoten. Diverse Abhängigkeiten zwischen den Elementen werden durch verbindende, gerichtete Kanten umgesetzt. FEAT stellt eine hierarchische Sicht zur Verfügung mit welcher der Graph entlang aller Abhängigkeiten erschlossen werden kann. Als Startpunkte könnten hierfür z.B. bereits lokalisierte Fragmente verwendet werden. Im Unterschied zu vergleichbaren Werkzeugen wie JRipples, JQuery oder dem JTranformer können in einer weiteren Sicht unterschiedliche Features angelegt werden (vgl. *Kapitel*

*4.1.1.3*). Zu jedem dieser Features wird ein *Concern Graph* erzeugt, der alle bisher lokalisierten Knoten und Kanten aufnehmen und persistent speichern kann. Concern Graphs werden in *Kapitel 4.3* noch einmal aufgegriffen.

Antoniol und Guéhéneuc [AG05; AG06] nutzen das in *Kapitel 4.1.2.3* vorgestellte *Knowledge-based Filtering* und *Probabilistic Ranking*, um in objektorientierten, multi-threading Systemen feature-relevante Ereignisse, wie z.B. Variablenzugriffe, Objektinstanziierungen und Methodenaufrufe zu identifizieren. Sie umfassen in ihrem Ansatz ebenso die Möglichkeit jene Ereignisse, basierend auf einem statischen Programm- bzw. Abhängigkeitsmodell, zu vollständigeren Strukturen erweitern zu können. Die Autoren gehen in ihren Publikationen allerdings nicht im Detail darauf ein, wie eine solche Expansion konkret aussieht und unter welchen Umständen sie beendet wird.

#### Eignung für das Feature Mining

Wenn die explorative Expansion entlang von Abhängigkeiten und Beziehungen systematisch erfolgt, kann sie insofern vorteilhaft sein, da Feature-Repräsentationen (nahezu) vollständig lokalisiert werden können. Grundvoraussetzung ist aber hierbei zum einen, dass der Anwender mit dem System sehr gut vertraut ist und zum anderen, dass das System als solches übersichtlich strukturiert ist. Die Methode hat den Nachteil, dass sie für feingranulare Elemente und dessen Abhängigkeiten einen sehr großen Suchraum aufspannt – dessen vollständige manuelle Untersuchung kann durchaus sehr zeitintensiv sein [EKS03; RW02; MRB+05; SMAP06].

### 4.2.5 Regelbasierte Expansion entlang von Strukturbeziehungen

Die im vorherigen *Kapitel 4.2.4* dargestellten Techniken extrahieren zu einem Quellcode-Element zugehörige strukturelle Nachbarn. Der Anwender muss hierbei jedes dieser Elemente hinsichtlich der Feature-Zugehörigkeit prüfen. Für komplexe Systeme entsteht somit ein erheblicher manueller Aufwand. In diesem Kapitel werden Techniken dargestellt, die ebenso auf strukturellen Beziehungen im Quellcode aufbauen. Allerdings verwenden diese bestimmte Heuristiken oder Regeln, um die Exploration abzukürzen oder gar zu automatisieren.

#### Konkrete Ansätze

Eisenbarth et al. [EKS01c; EKS03] ermitteln mit ihrem dynamischen Ansatz, der auf der formalen Begriffsanalyse basiert, zunächst eine Teilmenge einer Feature-Repräsentation (vgl. *Kapitel 4.1.2.3*). Jene Elemente verwenden sie als Startpunkte, um die Repräsentation zu verfeinern bzw. zu erweitern. Sie adaptieren hierfür ebenfalls den semiautomatischen Ansatz von Chen und Rajlich [CR00] (vgl. *Kapitel 4.1.1.4*). Darüber hinaus werden zwei

Heuristiken auf dem Systemabhängigkeitsgraph angewendet mit denen die Suche abgekürzt werden kann. Zum einen ist das eine Zyklen- und zum anderen eine Dominanzbeziehungserkennung. Bis dato wurde von den Autoren keine integrierte werkzeuggestützte Lösung für einen praktischen Einsatz veröffentlich. In ihren Fallstudien verwenden sie einzelne Werkzeuge (auf Methoden-Ausführungsebene), die sie je nach Bedarf mit einander kombinieren.

Zhao et al. [ZZL+06] streben in ihrem *SNIAFL (Static Noninteractive Approach to Feature Location)* Ansatz eine vollständig automatisiertes Vorgehen an. Hierfür lokalisieren sie zunächst mit Hilfe einer *IR-Textsuche* Methoden, die zu *unterschiedlichen Feature-Repräsentationen* gehören (vgl. *Kapitel 4.1.1.2*). Methoden die einen definierten Grenzwert überschreiten bezeichnen sie als „Feature-spezifisch". Darüber hinaus erzeugen sie für jede Methode in dem Quellcode einen so genannten *Branch-Reserving Call Graph*. Im Wesentlichen entspricht dieser einem (hierarchischen) Baum, dessen innere Knoten zu Anweisungsblöcken und Blätter zu Anweisungsblöcken oder Methodenaufrufen korrespondieren. Die Wurzel entspricht dem gesamten Methodenblock. In jedem Baum werden nach definierten Regeln Pfade entlang der Knoten markiert, welche mindestens durchlaufen werden müssen, um eine spezifische Methode aufzurufen. Alle so markierten Blöcke des Pfades gelten dann als *relevante* Kandidaten für eine Feature-Repräsentation. Um zu überprüfen, ob die Kandidaten nur zu einem Feature gehören oder ob sie Teil der Basisfunktionalität sind, werden aus den Pfaden für jedes Feature Pseudo-Ausführungsspuren erzeugt. Ähnlich wie in dynamischen Verfahren werden darauf aufbauend Mengenoperationen verwendet, um eine finale Entscheidung treffen zu können (vgl. *Kapitel 4.1.2.1*). Dieses Konzept zeigt für einfachstrukturierte C Programme gute erste Ergebnisse. Für komplexere (objektorientierte) Anwendungen gibt es bislang keine angepasste Variante dieser Technik.

Eine andere Technik, die automatisch relevante Struktur- bzw. Daten- oder Kontrollflussbeziehungen verfolgt, ist das *Program Slicing* [HPS07]. Slicing wurde einst als statische Analysetechnik definiert [Wei84], wobei auch mittlerweile dynamische Varianten existieren [AH90]. Mit dieser Technik kann für eine spezifische Bedingung eine (ausführbare) Teilmenge eines Programms, auch *Slice* genannt, identifiziert werden. Im Allgemeinen ist man an einem Slice bestehend aus allen Programmanweisungen interessiert, der bestimmte Variablen(-werte) beeinflusst bzw. durch bestimmte Variablen(-werte) beeinflusst wird [ESW04]. Program Slicing lässt sich unter anderem mit den kommerziellen Werkzeugen *CodeSurfer*[24] oder auch *Imagix4D*[25] durchführen. Eine kostenfreie Lösung bie-

[24] http://www.grammatech.com/products/codesurfer/overview.html

[25] http://www.imagix.com/products/source_code_analysis.html

ten hingegen die Opensource-Projekte *WALA*[26] und *Indus*[27] an. Obwohl Slicing aus konzeptueller Sicht gesehen eine sehr interessante Technik ist, wird sie in der Praxis von zahlreichen Problemen dominiert. Zum einen ist die Berechnung von feingranularen Extrakten sehr zeit- und ressourcenaufwändig. Zum anderen werden aufgrund von transitiven Beziehungen die Slices sehr groß. Unter diesen Umständen ist es für einen Anwender sehr schwierig zuzuordnen, welche Teilbereiche tatsächlich zu einem Feature und welche zu dem Basiscode gehören [ESW04; HPS05; RM03; Rob05; WBPR01].

Eaddy et al. [EAAG08] lokalisieren in ihrem *Cerberus* Konzept mit Hilfe von PROMESIR Teilmengen eines Belangs (vgl. *Kapitel 4.1.3*). Um jene zu möglichst vollständigen Repräsentationen zu expandieren, verwenden sie die von ihnen entwickelte *Prune Dependency Analysis*. Für diese gilt folgende zentrale Belang-Zugehörigkeitsregel: „Ein Quellcodefragment gehört zu einem Belang genau dann, wenn es geändert oder entfernt werden muss, sobald der gesamte Belang entfernt wird". Zu Beginn des Expansionsprozesses wird angenommen, dass die bereits lokalisierten Fragmente aus dem Quellcode entfernt werden. Anschließend wird anhand von drei Abhängigkeitsregeln (*Reference, Element Containment, Inheritance*) geprüft, ob aufgrund dieses Schrittes weitere Fragmente geändert oder entfernt werden müssen. Wenn ja, gehören sie nach der definierten Zugehörigkeitsregel ebenso zu dem Belang – die Analyse wird unter Berücksichtigung der neu aufgenommenen Fragmente wiederholt. Der Prozess wird beendet, sobald keine neuen Fragmente mehr aufgenommen werden müssen. Die *Reference-Regel* besagt, dass wenn eine Deklaration eines Typs, Feldes oder einer Methode entfernt wird, alle Methoden, welche auf dieses Element zugreifen, geändert werden müssen. Laut der Element *Containment-Regel* ist ein Typ zu entfernen, wenn alle darin enthaltenen Methoden und Felder bereits entfernt wurden. Im Umkehrschluss gilt auch, dass alle Methoden und Felder zu entfernen sind, wenn der Typ in dem sie deklariert sind bereits entfernt wurde. Die *Inheritance-Regel* fordert, dass eine Basisklasse entfernt werden muss, wenn alle Klassen, welche von ihr erben, schon entfernt wurden. Erste Fallstudien haben gezeigt, dass mit der Prune Dependency Analysis deutlich vollständigere Ergebnisse erzielt werden können, als mit PROMESIR allein [EAAG08]. Allerdings ist Cerberus aufgrund der recht grobgranularen Ausrichtung aktuell für das Feature Mining unzureichend.

Das bereits in *Kapitel 2.3.2* erwähnte Werkzeug *CIDE* implementiert einige zentrale Regeln eines *SPL-bewussten Typsystems* für Java Quellcode [KA08; KAT+08]. Jene Funktionalität hilft dem Anwender dabei unter Berücksichtigung eines Feature-Modells semantisch korrekte Annotierungen von Features vorzunehmen. Das heißt beispielsweise sicherzustellen, dass keine gültigen Varianten erstellt werden können, bei denen auf Variablen, Methoden, Felder oder Typen zugegriffen wird, für die es keine Deklaration gibt. Obwohl

[26] http://wala.sourceforge.net/

[27] http://indus.projects.cis.ksu.edu

das Typsystem unter einem anderen Gesichtspunkt entwickelt wurde, lässt es sich durchaus auch auf für eine feingranulare Feature-Expansion verwenden. Mit CIDE können zunächst alle bekannten Fragmente eines Features annotiert werden. Parallel dazu werden die verschiedenen Regeln des Typsystems angewendet und Bereiche hervorgehoben, die ebenfalls zu dem Feature zugeordnet werden müssen, damit es in keiner gültigen Variante zu Kompilierungsfehlern kommen kann. Mit der Annotierung neuer Fragmente beginnt der Vorgang von vorn und endet erst, wenn keine Fehler mehr angezeigt werden. Dieses Expansionskonzept ist vergleichbar mit der Prune Dependency Analysis von Eaddy et al. [EAAG08]. CIDE hat aber den entscheidenden Vorteil, dass Fragmente präziser, d.h. auch Anweisungen und Variablen innerhalb von Methoden, lokalisiert werden können. Im Hinblick auf das Feature-Mining muss zudem auch keine „Nachdokumentation" mehr erfolgen, um gültige Systemvarianten zu generieren. Allerdings werden durch das Typsystem keine logischen Schlüsse zu weiteren Fragmenten gezogen, wie z.B. in der Containment oder Inheritance-Regel [EAAG08], wenn sie keine Fehler verursachen.

#### Eignung für das Feature Mining

Konzeptionell gesehen bieten die hier dargestellten Ansätze ein großes Potential, um die Expansion zu beschleunigen, indem bestimmte Elemente automatisch zu einem Feature zugeordnet werden. Die praktische Umsetzung gestaltet sich allerdings bisweilen schwierig. Zum Teil sind die dargestellten Ansätze zu rechenaufwändig, zu unpräzise oder nicht für komplexe Programmierparadigmen erprobt. Das SPL-bewusste Typsystem in CIDE bildet hierbei eine Ausnahme. Allerdings ist jener Ansatz nur für einige (wenige) „Spezialfälle" anwendbar. Bei den definierten Regeln handelt es sich zudem um Heuristiken. Man muss sich daher bewusst sein, dass eine solche Automatisierung des Prozesses ein gewisses Risiko für Fehlentscheidungen birgt.

### 4.2.6 Relevanzwertbasierte Empfehlungen struktureller Nachbarn

Die Komplexität (und Bedeutung) der Feature-Expansion wächst proportional mit der Größe des zu untersuchenden Systems. Während in kleinen Systemen Anwender die verschiedenen Beziehungen sowie die Explorationstiefe noch selbstständig überblicken können, gelingt das in umfangreichen Systemen nur schwer. Empirische Untersuchungen zeigen allerdings, dass für eine erfolgreiche Software-Erschließung ein strukturiertes, methodisches Vorgehen notwendig ist, bei dem konsequent ein Plan verfolgt wird. Daher ist eine integrierte Werkzeugunterstützung unabdingbar, welche *nicht nur Beziehungen extrahiert und ihre Exploration anbietet, sondern auch dem Anwender hilft eng an seinem Plan zu bleiben bzw. schlichtweg die Orientierung zu behalten* [AMR07; JV03; RCM04; RM02a; RM02b; RM03; SEM05]. Eine mögliche Strategie, die dabei helfen könnte einen Anwender während einer Feature-Expansion gezielt zu lenken, wäre es ihm *Vorschläge zu weiteren, rele-*

*vanten Elementen zu unterbreiten, für die eine manuelle Inspektion „empfehlenswert“ wäre* [WR07]. Nachfolgend werden Ansätze vorgestellt, die jenes realisieren können.

**Konkrete Ansätze**

Der klassische Vertreter für diese Kategorie wurde von Robbillard [Rob05; Rob08] vorgeschlagen und in dem Eclipse Plug-in *Suade* implementiert. In diesem Ansatz wird einleitend Java Quellcode auf Methoden- und Feld-Ebene in einem *Systemabhängigkeitsgraph* abstrahiert. Mit Hilfe dieses Graphen werden zu einer gegebenen Menge von bereits lokalisierten Fragmenten, dem so genannten *Kontext*, alle struktur-abhängigen Quellcodeelemente identifiziert. Diese werden basierend auf den Metriken *Specifity* und *Reinforcement* sowie einem Vereinigungsalgorithmus hinsichtlich ihres potentiellen Interessegrades bewertet und in sortierter Reihenfolge ausgegeben. Werden weitere Quellcodefragmente in den Kontext mit aufgenommen bzw. zu einem Feature hinzugefügt, erfolgt eine erneute Ermittlung relevanter Kandidaten nach dem gleichen Schema. Der Kontext eines Features bzw. dessen Repräsentation wird mit Hilfe des *ConcernMappers* verwaltet. Dieses Werkzeug wird noch einmal näher in *Kapitel 4.3* besprochen.

Innerhalb von *Suade* beschreibt die *Specifity*, wie spezifisch bzw. wie einzigartig eine Verbindung zwischen einem Kontext- und einem Abhängigkeitselement ist. Werden beispielsweise aus der Kontextmenge das Feld *x* in zehn Methoden $xm_1 \ldots xm_{10}$ sowie ein Feld y in zwei Methoden $ym_1$ und $ym_2$ gelesen, ordnet die Metrik $ym_1$ und $ym_2$ einen höheren Spezifitäts- und damit höheren Interessegrad zu. Die *Reinforcement*-Metrik „bestärkt“ hingegen ein Element mit einem höheren Interessegrad, wenn viele seiner verwandten Elemente bereits in der Kontextmenge sind. Angenommen $xm_1 \ldots xm_9$ sind schon Teil der Kontextmenge – $xm_{10}$ sowie $ym_1$ und $ym_2$ sind es nicht. Dann wird in dem Fall $xm_{10}$, trotz der niedrigen Spezifität, ein hohes potentielles Interesse beigemessen. *Specifity* und *Reinforcement* werden für jeden Abhängigkeitstyp separat berechnet. Wird ein und das selbe Element über verschiedene Beziehungstypen „vorgeschlagen“, so wird der Interessegrad entsprechend heraufgesetzt. Empirische Untersuchungen haben gezeigt, dass auf diese Weise wertvolle Vorschläge unterbreitet werden können. Gerade wenn in komplexen Systemen sehr viele Abhängigkeiten existieren, kann ein solches Vorgehen dabei helfen den Suchraum auf wesentliche Elemente zu fokussieren. Die aktuelle Version von Suade kann allerdings nur Methoden oder Felder vorschlagen – andere Fragmente und Beziehungen zwischen ihnen werden nicht berücksichtigt [Rob05; Rob08; WR07]. Nicht zuletzt deswegen, ist dieses Plug-in mit den Anforderungen des Feature Mining nur eingeschränkt vereinbar.

Hill et al. [HPS07] schlagen eine weitere Möglichkeit vor mit der strukturabhängige Nachbarn bezüglich ihrer Relevanz bewertet werden können. In ihrem Eclipse Plug-in *Dora* ermitteln sie alle Aufrufbeziehungen zwischen Methoden in einem Java Programm. Zusätzlich muss für jedes Feature eine natürlich sprachliche Beschreibung angegeben

werden. Im Anschluss daran wird anhand des Aufrufgraphs jede bereits lokalisierte Methode um ihre entsprechenden Nachbarn erweitert. Jeder dieser neuen Methoden erhält einen Relevanzwert, welcher den Übereinstimmungsgrad mit der Feature-Beschreibung widerspiegelt. Für diesen textuellen Abgleich wird zum einen auf die *TF-IDF-Termgewichtungsformel* [BN99] aus dem Information Retrieval zurückgegriffen und zum anderen eine zusätzlich Gewichtung hinsichtlich des Auftrittorts (im Methodennamen, in einer Anweisung, etc.) sowie der Länge eines Worts vorgenommen. Überschreitet dieser *Method Relevance Score* einen gewissen Grenzwert, wird die Methode automatisch zu dem Feature aufgenommen und nach der gleichen Vorgehensweise expandiert. In der Evaluation von Dora konnte gezeigt werden, dass dieser Ansatz bessere Ergebnisse liefert als die *Specifity* Komponente von Suade. In Hinblick auf das Feature Mining kann auch hier festgehalten werden, dass die Ausrichtung auf eine Methodenebene zu grobgranular ist.

Revelle und Poshyvanyk [RP09] schlagen für die Feature-Expansion ebenfalls ein Konzept vor, welches Struktur- und Textinformationen vereint. Das grundsätzliche Vorgehen ist äquivalent zu Dora, mit dem wesentlichen Unterschied, dass die Methodenrelevanzmetrik das *Latent Semantic Indexing* aus dem Information Retrieval verwendet (vgl. *Kapitel 4.1.1.2*).

Das bereits in *Kapitel 4.1.1.4* vorgestellte Eclipse Plug-in *JRipples* enthält ebenso verschiedene Ansätze mit denen eine Feature-Expansion strukturiert und methodisch fundiert unterstützt werden kann. Der Systemabhängigkeitsgraph dient hierbei als Basis, um zu den im ersten Schritt lokalisierten Quellcodefragmenten strukturelle Nachbarn zu ermitteln. Als Parameter kann die Granularität der Beziehung festgelegt werden. Alle dadurch ermittelten Elemente werden mit dem Attribut „Next“ markiert. Die Aufnahme eines neuen Elements in die Feature-Menge erzeugt dabei wiederum weitere Next-Elemente. Da unter Umständen aufgrund von zahlreichen Abhängigkeiten sehr viele Elemente auf diese Weise hinzukommen können, gibt es zusätzlich die Möglichkeit jene hinsichtlich unterschiedlicher Relevanzmetriken zu sortieren.

Die *Impact Set Conncections* - Metrik gibt hierbei den absoluten Wert an, wie viele Beziehungen ein Next-Element zu der bereits lokalisierten Menge hat. Mit der *PIM-Heuristik* wird ermittelt wie oft Methoden aus einer Klasse eine andere Klasse aufrufen – und umgekehrt. Die Annahme ist, je stärker ein Element verknüpft ist, desto relevanter ist es. Die *Grep-Metrik* gibt Aufschluss darüber, wie viele Suchtreffer ein Next-Element zu einer (als regulärer Ausdruck formulierten) Feature-Beschreibung enthält. Unter Verwendung der Lucene-IR-Bibliothek können darüber hinaus Feature-Beschreibungen in verschiedenen anderen Anfrageformen formuliert werden (vgl. *Kapitel 4.1.1.2*). Ähnlich wie in dem Ansatz von Revelle und Poshyvanyk [RP09] bzw. in Dora, werden dann alle Next-Elemente hinsichtlich ihres Übereinstimmungsgrades sortiert. In JRipples heißen die dazu entsprechenden Metriken *Change Request to Class Similarity Evaluated by Information Retrieval Heu-*

*ristic – RCIR* sowie *Class To Class Similarity Evaluated By Information Retrieval Heuristic – CCIR* [BBPR05; PR09].[28]

JRipples ist darauf ausgerichtet System-Änderungsaufträge in ihrem gesamten Auftragszyklus zu unterstützen. Hierbei müssen Schritte erfolgen und Techniken angewendet werden, die auch für eine Feature-Lokalisierung und -Expansion relevant sind. Daher können die verwendeten statischen Konzepte auch einen hilfreichen Beitrag für das Feature Mining leisten. JRipples als solches lässt sich aber für das Feature Mining nicht eins zu eins übernehmen, auch wenn der erste Eindruck es vermuten lässt. Gerade aufgrund des unterschiedlichen Fokus, werden die Defizite in der Handhabung deutlich. Es ist beispielsweise nicht möglich mehrere Features gleichzeitig anzulegen und zu verwalten. Zudem kann sobald die Expansion von Fragmenten begonnen wurde, nicht mehr in den Lokalisierungsmodus zurückgeschaltet werden, um neue Startpunkte aufzunehmen. Dieses ist aber manchmal erforderlich, da relevante Elemente auch „zufällig“ entdeckt werden können, die in keiner direkten Strukturabhängigkeit zu der aktuellen Feature-Menge stehen. Weiterhin können die Metriken nicht miteinander gekoppelt werden, sodass nur ein einheitlicher Wert errechnet wird. So bleibt es dem Anwender überlassen für sich selbst herauszufinden, welche der Metriken am zuverlässigsten ist. Wie in vielen anderen Werkzeugen in diesem Bereich, werden auch hier lokale Variablen und Parameter, weder in dem Abhängigkeitsgraph noch in den Relevanzmetriken berücksichtigt.

### Eignung für das Feature Mining

In komplexen Systemen kann der zu explorierende Beziehungsgraph sehr umfangreich werden. Eine Untersuchung *aller* strukturellen Nachbarn von lokalisierten Feature-Elementen führt zwar mit Sicherheit immer zu der besten Lösung ist aber dafür sehr zeitaufwändig. Die in diesem Kapitel dargestellten Ansätze helfen dem Anwender dabei all jene Elemente hinsichtlich ihrer Relevanz für das Feature zu priorisieren. Hierdurch wird eine Empfehlung gegeben, welche Elemente es sich eher lohnt näher zu untersuchen und welche möglicherweise nicht. Durch diese zielgerichtete Führung kann der Aufwand für den Anwender erheblich reduziert werden. Es ist allerdings wichtig zu betonen, dass es sich bei den dargestellten Ansätzen um Heuristiken handelt, die sich unter Umständen auch „irren können“. Relevante Elemente könnten einen derart niedrigen Relevanzwert beigemessen bekommen, dass sie von dem Anwender nicht berücksichtigt werden.

Eine wesentliche Beobachtung ist, dass keiner von den betrachteten Ansätzen den Ansprüchen hinsichtlich der *Feingranularität* für das Feature Mining erfüllt (vgl. *Kapitel 3.1*).

[28] http://jripples.sourceforge.net/jripples/manual/reference/analysis/analysismain.html

## 4.3 Dokumentationstechniken

Feature-Mining verfolgt das Ziel eine Plattform „zu erstellen“, welche auf einem Annotationsansatz basiert, d.h. bei dem Feature-Repräsentationen innerhalb einer Codebasis mit Hilfe von textuellen oder graphischen Markierungen ausgezeichnet werden. Im Rahmen der gesamten Aufgabenstellung müssen daher Features nicht nur als „lose Sammlungen“ lokalisiert, sondern auch so dokumentiert werden, dass gültige, fehlerfreie Produktvarianten erzeugt werden können. Diese konsistente Beschreibung von Feature-Repräsentationen wird als Feature-Dokumentation bezeichnet. Für diese werden geeignete Dokumentationstechniken benötigt, welche Quellcodefragmente so erfassen und verwalten können, dass in erzeugten Varianten Syntax-, Typ- und Verhaltensfehler minimiert oder im optimalen Fall ganz ausgeschlossen werden (vgl. *Kapitel 2.3.2; 3.1*).

Zur Dokumentation von Feature-Repräsentationen sind grundsätzlich alle Annotationstechniken geeignet, welche in *Kapitel 2.3.2* erwähnt wurden. Kästner [Käs10] diskutiert diese und noch viele weitere in seiner Dissertation. Daher sei der interessierte Leser für eine nähere Betrachtung der unterschiedlichen Ansätze auf seine Arbeit verwiesen. Im Folgenden werden lediglich einige einzelne Konzepte herausgegriffen, welche über die traditionellen textuellen Konzepte hinausgehen und/oder explizit die Vermeidung der erwähnten Fehlertypen anstreben.

#### Konkrete Ansätze

Es können unterschiedliche text- oder strukturbasierte Anfragen verwendet werden, um Features zu lokalisieren (vgl. *Kapitel 4.1.1.1; 4.1.1.2; 4.1.1.3*). Anfragen, welche korrekte Suchergebnisse liefern, lassen sich aber auch als Möglichkeit interpretieren, um Feature-Repräsentationen „indirekt“ zu beschreiben. Dieser Ansatz wird unter anderem von Mens et al. [MMW02; MPG03] näher verfolgt. Mit Hilfe von logischen Anfragen generieren sie so genannte *Intentional source-code Views* für einen bestimmten Belang. Mit einem zusätzlichen Modell definieren sie zudem Abhängigkeiten zwischen den unterschiedlichen Sichten. Obwohl mit diesem Vorgehen unterschiedlichste Strukturen dokumentiert werden können, sind sie eigentlich nur für grobe Zusammenhänge (wie z.B. zwischen Klassen) geeignet. Anfragen, welche feingranulare, heterogene Fragmente erfassen sollen, lassen sich entweder gar nicht oder nur mit großer Mühe formulieren. Aufgrund dessen ist jener Ansatz nur bedingt für das Feature Mining geeignet.

*JQuery* sowie *JQueryScapes* gehören ebenfalls zu der Gruppe von Werkzeugen, mit denen es möglich ist durch Anfragen Quellcodefragmente einer Feature-Repräsentation zu lokalisieren bzw. indirekt zu beschreiben (vgl. *Kapitel 4.1.1.3*). Im Unterschied zu anderen ver-

gleichbaren Techniken lassen sich aber hiermit *Java Annotations*[29] suchen. Jene können im Quellcode genutzt werden, um beispielsweise Deklarationen von Typen, Feldern, Methoden und Variablen als feature-relevant zu markieren. Mit Hilfe von einfachen, separaten Anfragen lassen sich diese Markierungen dann suchen. Der Vorteil hierbei ist, dass sich strukturierte und explorierbare Feature-Sichten ohne einen erheblichen Mehraufwand erzeugen lassen [JV03; MV08]. Mit Java Annotations können allerdings nicht einzelne Anweisungen, wie z.B. Aufrufe, Schreib- und Lesezugriffe markiert werden. Nicht zuletzt aufgrund dieser Einschränkung hinsichtlich der Granularität, ist ein solches Vorgehen im Rahmen des Feature Mining unzureichend.

Robillard und Murphy [RM02a, RM07] schlagen zur Dokumentation von Belangen so genannte *Concern Graphs* vor, welche Quellcodefragmente als Knoten und Beziehungen zwischen ihnen als Kanten abstrahieren. Nach diesem Ansatz entspricht jede Belang-Dokumentation einer Teilmenge des Systemabhängigkeitsgraphs, die sich für ein besseres Verständnis explorativ erschließen lässt. In dem bereits in *Kapitel 4.2.4* erwähnten Werkzeug FEAT wurde das Konzept der Concern Graphs auf Typ-, Feld- und Methodenebene implementiert. FEAT ermöglicht nicht nur die Verwaltung und Exploration der Concern Graphs, sondern auch eine Analyse von Feature-Interaktionen und gemeinsam verwendeten Elementen. Ein Konsistenzmanagement stellt darüber hinaus sicher, dass Änderungen im Quelltext auch in den Concern Graphs vollzogen werden. Mit diesem Konzept wird das Ziel verfolgt Belange in ihrem Kontext zu erfassen und sie zu verstehen. Hierfür wird bewusst auf ein abstraktes, grobgranulares Modell abstrahiert. So werden z.B. auch zwei unterschiedliche Aufrufe einer Methode B innerhalb des Methodenblocks A als ein und die selbe Abhängigkeitsbeziehung deklariert. Aufgrund des unterschiedlichen Fokus und der damit verbundenen Unschärfe sind Concern Graphs in der beschriebenen Form für das Feature Mining nicht geeignet.

Robillard und Weigand-Warr [RW05; WR07] haben im Vergleich zu Concern Graphs in ihrem Eclipse Plug-in *ConcernMapper* einen *leichtgewichtigeren Ansatz* umgesetzt. Dieser bietet die Möglichkeit Typen, Felder und Methoden eines Java Programms definierten Belangen zu zuordnen. Die Besonderheit hierbei ist, dass über einen Parameter noch jeweils ein Relevanzgrad zwischen 0 (nicht relevant) und 1 (sehr relevant) mit angegeben werden kann. Das resultierende Modell lässt sich im XML Format speichern und laden. Aufgrund der einfachen Anwendbarkeit wurde der ConcernMapper auch in dem Eclipse Plug-in Suade verwendet (vgl. *Kapitel 4.2.6*). In diesem Konzept können feingranulare Aufruf-, Lese-, oder Schreibanweisungen nicht verwaltet werden. Für das Feature Mining wird allerdings jenes benötigt.

[29] http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

ConcernMapper wird ebenso als Basis in dem Eclipse Plug-in *ConcernTagger* [30] verwendet. In dieser Erweiterung werden zu den jeweiligen dokumentierten Belangen *zusätzlich unterschiedliche Metriken* ermittelt, wie z.B. Anzahl der Quellcodezeilen, Verstreuungsgrad über Methoden bzw. Klassen, etc. Zudem können die Belange in dem Attribute-Relation File Format – ARFF exportiert werden. Dieses Format wird häufig verwendet um Datensätze von Fallstudien zu veröffentlichen. Alle erzeugten Daten können darüber hinaus auch in einer Datenbank abgelegt werden. Der ConcernTagger wird unter anderem in dem Feature-Lokalisierungswerkzeug FLAT³ verwendet (vgl. *Kapitel 4.1.3*). Trotz der zahlreichen Erweiterungen bleibt das wesentliche Zuordnungskonzept unverändert. Somit ist der ConcernTagger ebenfalls aufgrund der grobgranularen Ausrichtung in der aktuellen Variante für das Feature Mining nur bedingt geeignet.

*Spotlight* von Coppit et al. [CPR07], ebenfalls ein Eclipse Plugin, ermöglicht es im Gegensatz zu den bisher vorgestellten Ansätzen Quellcode sogar auf Zeichenebene zu verschiedenen Sichten, den so genannten *Software Plans*, zu zuordnen. Jene Sichten, die Belange repräsentieren, können im Anschluss variabel kombiniert angezeigt werden. Spotlight ist ein Prototyp eines Editors, der darauf ausgelegt ist neue Anwendungen zu erstellen oder neue Belange in bestehenden Code „einzuweben". In beiden Fällen müssen Codefragmente nach einem definierten Schema angelegt werden. Aktuell werden keine unterstützenden Mechanismen angeboten, um bestehenden Code in das „Software Plan Schema" zu überführen – jene sind lediglich als zukünftige Erweiterungen angedacht. In wie fern Spotlight für das Feature Mining geeignet ist, kann erst beurteilt werden, sobald die Erweiterungen auch umgesetzt werden.

Das programmiersprachen-unabhängige Eclipse Plug-in *CIDE,* welches bereits in *Kapitel 2.3.2* und *4.2.5* besprochen wurde, unterstützt ebenso wie Spotlight eine feingranulare Markierung von Quellcodefragmenten. Als Annotationstechnik werden hierbei aber bewusst *Farben eingesetzt*, um die Struktur des Quellcodes nicht zu verändern und die Lesbarkeit zu erhöhen. Auf der Basis des „eingefärbten" Quellcodes lassen sich unterschiedliche Sichten und Auswertungen zu den Features (bzw. Feature-Kombinationen) erzeugen. CIDE grenzt sich von den in diesem Kapitel vorgestellten Werkzeugen dahingehend ab, dass es explizit das Ziel verfolgt Variabilität zu modellieren und zu implementieren, um daraus verschiedene Produktvarianten erzeugen zu können. In diesem Zusammenhang werden zwei Konzepte angewendet, welche sowohl Syntax- als auch Typ-Fehler minimieren bzw. vermeiden sollen:

- Die Korrektheit auf *Syntaxebene* wird erreicht indem nur *disziplinierte Annotierungen erlaubt* werden. Hierzu erfolgt eine interne Verwaltung von Farbmarkierungen auf der Ebene eines abstrakten Syntaxbaumes. Jener enthält keine Klammern oder

[30] http://www.cs.columbia.edu/~eaddy/concerntagger

jegliche Trennzeichen, sodass solche Elemente erst gar nicht einzeln markiert werden können. Darüber hinaus erfolgt bei jedem Markierungsversuch eine Berücksichtigung der jeweiligen Programmiersprachen-Grammatik. Mit Hilfe dieser wird sichergestellt, dass nur optionale Elemente annotiert werden können. Beispielsweise darf nach dieser Regel innerhalb des Deklarationsblocks einer Klasse nicht der Name allein markiert werden. Zudem wird eine weitere Regel angewendet in der Feature-Markierungen eines übergeordneten Elementes stets an seine untergeordneten propagiert werden. Sprich, wird z.B. eine Methode einem Feature zugeordnet, so werden alle darin enthaltenen Anweisungen ebenfalls entsprechend markiert [KAT+08].

Unter Verwendung eines *SPL-bewussten Typsystems* wird hingegen die Minimierung von *Typ-Fehlern* angestrebt. Hierbei werden Feature-Markierungen hinsichtlich „einiger zentraler" Regeln überprüft. Kommt es gegebenenfalls zu einem Widerspruch, werden Bereiche hervorgehoben, die ebenfalls zu dem Feature zugeordnet werden müssen, damit es in keiner gültigen Variante zu Kompilierungsfehlern kommen kann. Eine der Regeln untersucht beispielsweise, ob gültige Varianten erzeugt werden können, in denen auf Variablen, Methoden, Felder oder Typen zugegriffen wird, für die es keine Deklaration gibt. Ein solches Typsystem ist innerhalb von CIDE für die Programmiersprache Java nur in Ansätzen verfügbar. Aufgrund der Komplexität dieser Sprache wurde bislang hierfür kein vollständiges Typsystem implementiert. Deswegen können semantische Fehler nicht gänzlich ausgeschlossen werden [KA08].

Obwohl CIDE bislang nicht alle Fehlertypen vermeiden kann, bietet es dennoch perspektivisch gesehen, auch wegen seiner SPL-Ausrichtung, eine sehr gute Grundlage auf der zukünftige Forschungsanstrengungen im Feature Mining aufsetzen können.

### Eignung für das Feature Mining

Eine wesentliche Beobachtung ist, dass die meisten betrachteten Ansätze aufgrund ihrer grobgranularen Ausrichtung *in der aktuellen Umsetzung* nicht den Ansprüchen des Feature Mining genügen. Eine Ausnahme stellen die unterschiedlichen textuellen Annotationsansätze und CIDE dar. Kästner [Käs10] zeigt, dass CIDE gegenüber traditionellen (textuellen) Ansätzen deutliche Verbesserungen erreichen kann, wie z.B. konsistentere Feature-Annotationen oder verbesserte Lesbarkeit des Quellcodes.

## 4.4 Fazit und Zusammenfassung

Der Feature Mining Problembereich steht nicht im luftleeren Forschungsraum. Es gibt eine Vielzahl von verwandten Forschungsgebieten, die sich mit vergleichbaren Fragestellungen beschäftigen. Die detaillierte Untersuchung dieser und weiterer spezialisierter Themengruppen hat gezeigt, dass Feature Mining ein neues und eigenständiges Forschungsgebiet ist. Zudem veranschaulichte die Diskussion aber auch, dass es viele gemeinsame Bereiche mit einzelnen Feature Mining Teilaufgaben gibt.

Aus diesen angrenzenden Forschungsgebieten wurden in diesem Kapitel diverse Ergebnisse und Ansätze diskutiert, welche für die Entwicklung bzw. Implementierung von konkreten Feature Mining Konzepten hilfreich sein könnten. Hierbei wurden unterschiedliche Ansätze zu Kategorien zusammengefasst und hinsichtlich ihrer aktuellen Eignung für das Feature Mining bzw. für dessen Teilaufgaben bewertet. Durch diese umfassende Aufarbeitung der unterschiedlichen Konzepte liefert dieses Kapitel einen idealen Einstiegspunkt für zukünftige Forschungsaktivitäten in dem FM-Bereich.

Aus dieser Darstellung des aktuellen Stands der Technik lässt sich festhalten, dass auf den ersten Blick viele verschiedene Techniken und Werkzeuge, wie z.B. Cerberus, CIDE, Dora, FEAT, FLAT[3], JIriSS, JQuery, JRipples, JTransformer, Klonerkennung, Slicing oder auch Suade, prädestiniert für das Feature Mining zu sein scheinen. Allerdings zeigen sich die Abweichungen hinsichtlich der Anforderungen meist erst im Detail. Viele der besprochenen Ansätze sind zu grobgranular, streben keine vollständige Annotierung an, berücksichtigen nur Teilaufgaben oder verfolgen eine grundsätzlich verschiedene Zielsetzung. Eine nahtlose, kombinierte Nutzung vorhandener Werkzeuge ist aufgrund unterschiedlich verwendeter Modelle nicht ohne weiteres möglich.

Empirische Untersuchungen zeigen, dass für eine erfolgreiche Software-Erschließung ein strukturiertes, methodisches Vorgehen notwendig ist, bei dem konsequent ein Plan verfolgt wird. Daher ist eine integrierte Werkzeugunterstützung unabdingbar, welche dem Anwendender hilft eng an seinem Plan zu bleiben und ihm die Möglichkeit gibt erzielte Ergebnisse in irgendeiner Form festzuhalten. Feature Mining ist zu großen Teilen eine solche Software-Erschließungsaufgabe, sodass jene Erkenntnis mit Sicherheit auch hiefür gilt. Basierend auf der Untersuchung der verwandten Ansätze lässt sich daher schlussfolgern, dass Forschungsanstrengungen in der Zukunft vor allem ihren Fokus auf die Entwicklung von Werkzeugunterstützungen setzten müssen, welche den gesamten Prozess des Feature Mining integriert abdecken. Jene Konzepte müssen primär sicherstellen, dass eine annotationsbasierte Plattform mit einem Minimum an manuellem Aufwand und Risiko aus bestehenden Ressourcen aufgebaut werden kann.

# 5 Lösungskonzept: LEADT

Empirische Untersuchungen zeigen, dass für eine erfolgreiche Software-Erschließung ein strukturiertes und methodisches Vorgehen notwendig ist, bei dem konsequent ein Plan verfolgt wird. Eine umfassende Werkzeugunterstützung ist hierbei unabdingbar, welche dem Anwender hilft während *allen* erforderlichen Aktivitäten eng an seinem Plan zu bleiben (vgl. *Kapitel 4.2.6*). Die Analyse des aktuellen Stands der Technik in dem *4. Kapitel* hat gezeigt, dass für das Feature Mining bislang keine Konzepte und Werkzeuge existieren, die jenes für den *gesamten Prozess* umsetzen.

Vor diesem Hintergrund wird in dem ersten Teil dieses Kapitels das *Location, Expansion and Documentation Tool (LEADT)* vorgeschlagen, mit dem erstmalig die Möglichkeit geboten wird *alle Aufgaben des Feature Mining strukturiert* und *integriert* zu bearbeiten.

In LEADT wird das Aufgabenspektrum der Lokalisierung und Dokumentation durch die Integration von bereits verfügbaren Werkzeugen abgedeckt. Für die Feature-Expansion konnte allerdings nicht vollständig auf fertige Werkzeuge von anderen zurückgegriffen werden, welche die Anforderungen des Feature Mining erfüllen. Infolgedessen wurde *ein eigenes vollständigkeitsfokussiertes, automatisiertes Expansionskonzept entwickelt*. Der zweite Teil dieses Kapitels beschreibt ausführlich die konkrete Umsetzung jener Neu-Entwicklung.

## 5.1 Strukturiertes, integriertes Konzept für das Feature Mining

In diesem Kapitel wird das *Location, Expansion and Documentation Tool (LEADT)* vorgeschlagen, mit dem erstmalig die Möglichkeit geboten wird *alle Aufgaben des Feature Mining strukturiert* und *integriert* zu bearbeiten. LEADT bietet eine einheitliche Umgebung in der Quellcode und Feature-Diagramme verwaltet, zugehörige Features semiautomatisch in dem (Alt-)System annotiert und gültige Produktvarianten generiert werden können.

*Abbildung 5.1* gibt einen groben Überblick zu den grundlegenderen Bestandteilen des gesamten LEADT Konzepts. Jene Darstellung dient zugleich als Orientierungshilfe für die anschließende detaillierte Besprechung der Funktionsweise sowie des implementierten Prototypen.

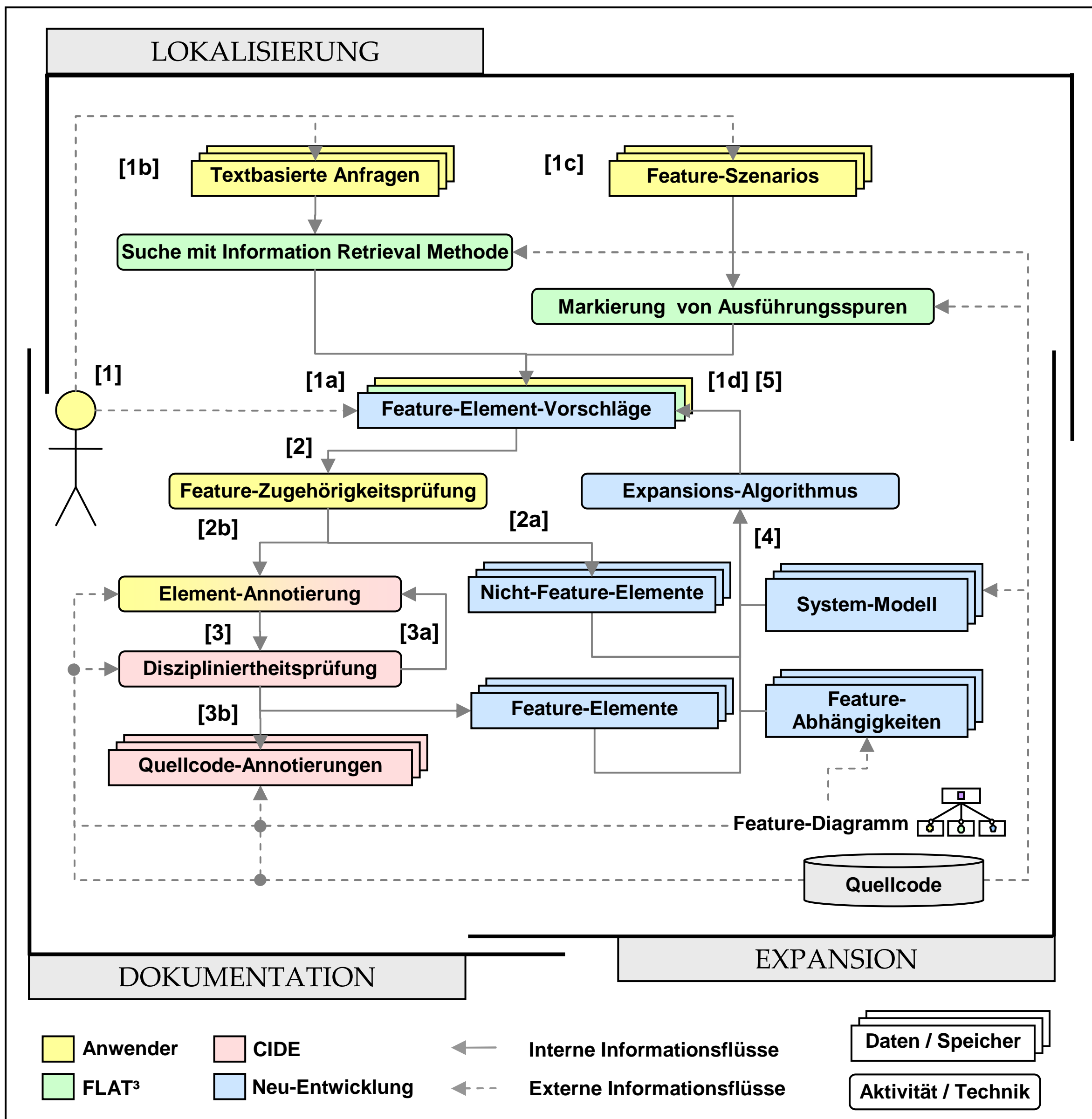



Abbildung 5.1: LEADT Konzept im Überblick

### 5.1.1 Funktionsweise

Erfahrungen aus verwandten Forschungsbereichen wie z.B. Feature Location zeigen, dass bestimmte Probleme weder von Computern noch Menschen allein erfolgreich gelöst werden können (vgl. *Kapitel 3.1; 4.1.1.4*). In LEADT wird daher auf eine Kombination der komplementären Stärken von beiden gesetzt. Hierbei werden die unterschiedlichen Aktivitäten der Feature-Lokalisierung, -Expansion, -Dokumentation folgendermaßen auf Mensch oder Maschine verteilt:

### Feature-Lokalisierung

Nachdem durch den Anwender festgelegt wurde, welches Feature in die SPL-Plattform überführt werden soll, wird der eigentliche Feature Mining Prozess angestoßen. Im *1.Schritt* müssen die verfügbaren abstrakten Informationen zu dem Feature derart verarbeitet werden, dass sich hieraus korrespondierende Start-Elemente im Quellcode lokalisieren lassen.

Zu aller erst hat der Anwender immer die Möglichkeit durch seine Erfahrung oder Intuition konkrete Feature-Elemente direkt anzugeben (*1a*). Ist er hierzu nicht in der Lage, hat er in LEADT drei weitere Optionen. Einerseits kann er statische, *textbasierte Suchanfragen* (in natürlicher Sprache) formulieren, welche durch *Methoden aus dem Information Retrieval* zu Suchtreffer im Quellcode umgesetzt werden (*1b*; vgl. *Kapitel 4.1.1.2*). Andererseits besteht die Möglichkeit dynamische *Feature-Szenarios* auszuarbeiten und sie während des laufenden Betriebs durchzuführen, um *markierte Ausführungsspuren* zu erzeugen (*1c*; vgl. *Kapitel 4.1.2.2*). Die letzte Alternative basiert auf einer indirekten Lokalisierung. Statt direkter Informationen des Anwenders, werden hierbei Feature-Abhängigkeiten aus dem Feature-Diagramm verwendet (*1d*). Der Anwender annotiert in diesem Fall, wenn vorhanden, zuerst ein anderes, „verwandtes" Feature, zu dem er zumindest einige abstrakte Informationen besitzt. Unter bestimmten Voraussetzungen kann der *Expansions-Algorithmus* dann aus dem bereits annotierten Feature, Element-Vorschläge für das tatsächlich gesuchte Feature generieren.

LEADT verwendet für die Umsetzung der Alternativen *1b* und *1c* das Eclipse Plug-in *FLAT*[3] von Savage et al. [SRP10] (vgl. *Abbildung 5.1*). Für nähere Information zu dieser *hybriden Lokaliserungstechnik*, der zugrunde liegenden Techniken sowie deren Vor- und Nachteile sei auf die Analyse in *Kapitel 4.1.3* verwiesen. Die zuletzt genannte Option *1d* wird in *Kapitel 5.2.2* noch näher erläutert.

### Feature-Dokumentation

Alle ermittelten *Feature-Element-Vorschläge*, sowohl aus der Lokalisierung als auch der Expansion, werden in einer nach dem Relevanzwert sortierten Liste verwaltet. Der Anwender wählt beispielsweise das Element mit dem höchsten Wert aus und prüft es manuell hinsichtlich seiner Feature-Zugehörigkeit (2.*Schritt*). In *Kapitel 3.1* wurde zugrunde gelegt, dass Experten des Systems in der Lage sein sollten hierbei eine korrekte Entscheidung zu treffen.

Lehnt der Anwender den Vorschlag ab, wird das entsprechende Element in den *Nicht-Feature-Datenspeicher* aufgenommen (*2a*). Stellt er hingegen fest, dass der Vorschlag zu der gesuchten Feature-Repräsentation gehört, folgt die *Annotierung im Quellcode* (*2b*). Hierbei markiert der Anwender in einem Texteditor das vorgeschlagene Element (und evtl. des-

sen Kontext) und versucht diesem die zugehörige *Feature-Farbe* zuzuweisen (vgl. *Kapitel 4.3*). Eine interne *Diszipliniertheitsprüfung* entscheidet, ob eine solche Markierung *hinsichtlich* der Syntax zulässig ist (*3. Schritt*; vgl. *Kapitel 4.3*). Schlägt die Prüfung fehl (*3a*), muss der Kontext des Elements solange erweitert werden bis die Prüfung erfolgreich ist. In *Abbildung 5.2* werden zur Veranschaulichung von undisziplinierten und erweiterten, disziplinierten Markierungen drei Beispiele gezeigt.

| | **Undisziplinierte Element-Vorschläge** | | **Disziplinierte erweiterte Markierungen** |
|---|---|---|---|
| 1) | this.**feld** = variable; | 1) | **this.feld = variable;** |
| 2) | If (**feld**) {<br>methode();<br>} | 2) | **If (feld) {<br>methode();<br>}** |
| 3) | class **Klasse** {<br>int feld;<br>void methode(){…}<br>} | 3) | **class Klasse {<br>int feld;<br>void methode() {…}<br>}** |

Abbildung 5.2: Undisziplinierte und erweiterte disziplinierte Markierungen

Erfolgreich zugewiesene *Quellcode-Annotierungen* werden abschließend in separaten Dateien abgelegt. Alle darin enthaltenen Schlüssel-Elemente werden zudem in den *Feature-Elemente-Datenspeicher* aufgenommen (*3b*). In dem Beispiel 1 aus *Abbildung 5.2* sind *feld* und *variable* Schlüssel-Elemente, in Beispiel 2 *feld* und *methode*, in Beispiel 3 *Klasse*, *feld* und *methode*. In *Kapitel 5.2.1* wird noch ausführlich diskutiert werden, was unter einem Schlüssel-Element zu verstehen ist.

Die Feature-Dokumentation in LEADT basiert zu großen Teilen auf dem Eclipse Plug-in *CIDE* (vgl. *Abbildung 5.1*). Für ausführliche Darstellungen zu diesem Werkzeug sei auf die *Kapitel 2.3.2* und *4.3* sowie die Arbeiten von Kästner et al. [KAK08; Käs10; KTA08] verwiesen.

## Feature-Expansion

Jede Veränderung in dem *Nicht-Feature-Elemente-* oder *Feature-Elemente-Datenspeicher* führt dazu, dass sich der Informationsbestand erhöht (*2a*; *3b*). Hierdurch können zum Teil Rückschlüsse auf weitere relevante Feature-Elemente automatisiert gezogen werden. Bei jeder solchen Änderung wird ein *Expansions-Algorithmus* aktiviert, der neue bzw. aktualisierte *Element-Vorschläge für Features generiert* (*4. Schritt*). Neben den bereits erwähnten Datenspeichern verwendet der Algorithmus noch ein *System-Modell*, welches verschiedene Beziehungen zwischen Schlüssel-Elementen des Quellcodes abbildet, sowie extrahierte *Abhängigkeitsbeziehungen aus dem Feature-Diagramm.*

Die in LEADT umgesetzte Methode für die Feature-Expansion ist ein zentrales Hilfsmittel, mit dem der manuelle (Such-)Aufwand für die *vollständige* Lokalisierung reduziert

werden kann. Der Anwender wird nicht nur seiner Erfahrung und Intuition überlassen, sondern zielgerichtet und effektiv geführt. Gerade für komplexe Systeme, bei denen ein Anwender aufgrund von *Verstreuungs- und Vermischungseffekten* nur mit großer Mühe alle Beziehungen überblicken kann, könnte dieser Ansatz eine wertvolle Bereicherung sein.

Der *Vorschlags-Algorithmus* sowie zugehörige Vorverarbeitungsschritte und Datenspeicher sind wesentliche Neu-Entwicklungen innerhalb von LEADT. In den folgenden Kapiteln wird sich daher noch umfassend der genauen Umsetzung gewidmet.

Jede Aktualisierung der *Feature-Element-Vorschläge,* wie angedeutet in *Schritt 5,* bedeutet zugleich eine mögliche neue Iteration beginnend bei *Schritt 2*. Spätestens sobald alle Feature-Element-Vorschläge abgearbeitet sind, und keine neuen mehr generiert werden können, endet der Vorgang für das gesuchte Feature. Über die von CIDE bereitgestellte Funktionalität, lassen sich aus der annotierten Plattform nun System-Varianten mit und ohne dem annotierten Feature erzeugen. Mit *Schritt 1* könnte dann das Mining für ein weiteres Feature beginnen bzw. für ein bereits begonnenes fortgesetzt werden.

## 5.1.2 Prototyp

Das in *Abbildung 5.1* bzw. *Kapitel 5.1.1* schematisch dargestellte Konzept wurde in einem ersten Prototyp als Eclipse Plug-in implementiert. Für die Umsetzungsart haben drei wesentliche Gründe gesprochen:

- Eclipse ist eine freie, häufig genutzte Plattform für diverse Software-Engineering Aufgaben, die vielen Anwendern vertraut ist.
- Durch die Framework- bzw. Plug-in-Technologie in Eclipse können viele Standardkomponenten (Editoren, Sichten, etc.) einfach und schnell wiederverwendet werden. Der Hauptentwicklungsaufwand kann sich dadurch auf die Umsetzung von neuen Ideen und Konzepten konzentrieren.
- FLAT³ und CIDE konnten fast nahtlos eingebunden werden.

FLAT³ wurde für die Programmiersprache Java entwickelt. CIDE ist sprachenunabhängig, wurde aber bislang besonders häufig auf Java Quellcode erprobt. Die Neuentwicklungen in LEADT sind daher ebenso auf Java ausgerichtet. Allerdings ist die Idee des LEADT Konzepts als solches ohne weiteres auch auf andere Programmiersprachen übertragbar.

In *Abbildung 5.3* ist ein Screenshot des Prototypen abgebildet. Folgend werden die darin sichtbaren Bereiche kurz erklärt, um das Konzept aus der praktischen Umsetzung noch einmal zu veranschaulichen.

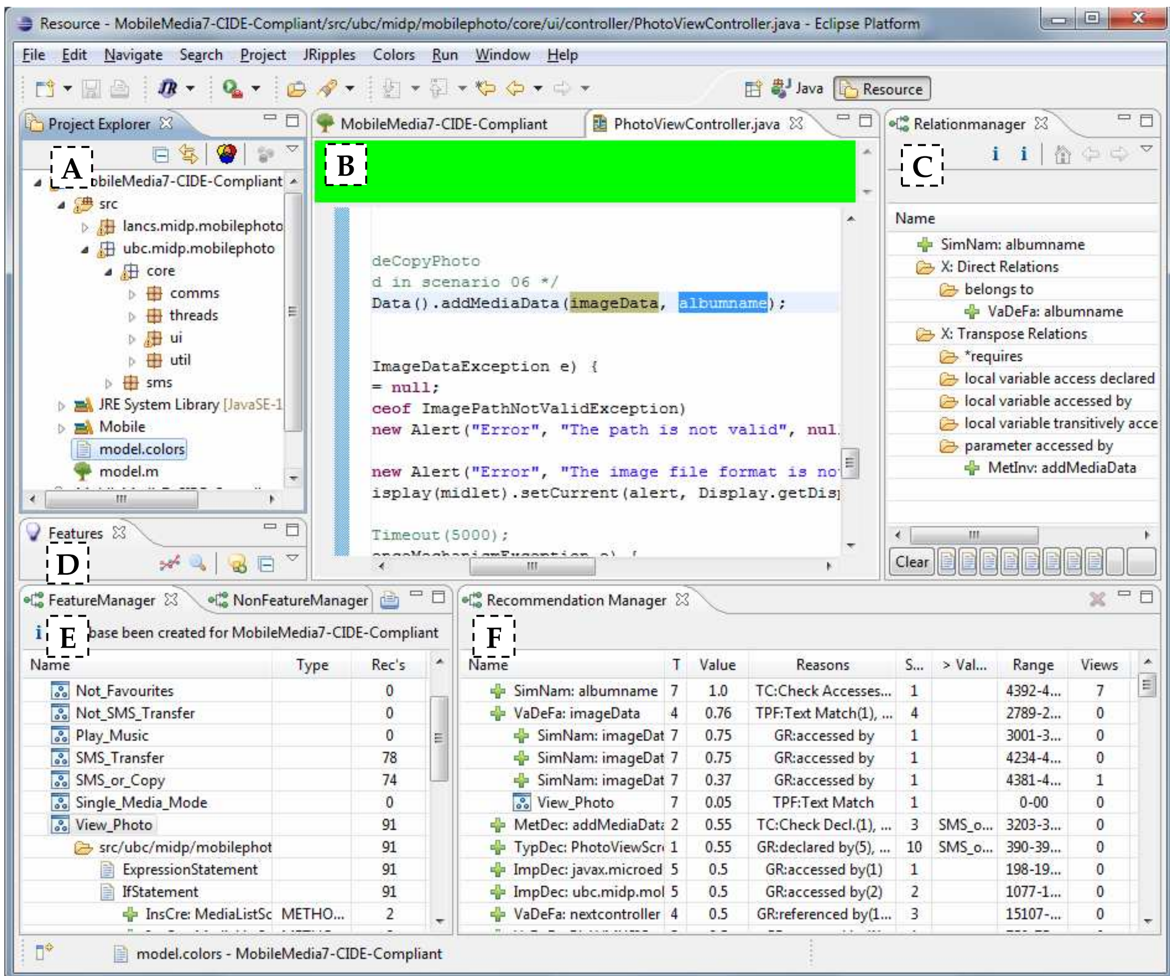



Abbildung 5.3: Screenshot des LEADT Prototypen

A) Diese Standardsicht von Eclipse gibt ein Überblick über alle in einem Projekt enthaltenen Dateien (*Feature-Diagramm, Quellcode-Dateien, Quellcode-Annotierungen*, etc.). Über das Kontextmenü des Wurzel-Elements lassen sich mit Hilfe von CIDE verschiedene Projekt- bzw. Systemvarianten aus der annotierten Plattform generieren.

B) In diesem Editor-Bereich können unterschiedliche Dateien betrachtet und modifiziert werden. Im Wesentlichen werden zwei konkrete Editortypen verwendet, die beide CIDE bereitstellt. Zum einen ein Quelltext-Editor mit dessen Hilfe *Quellcode-Annotierungen* in Java-Dateien angelegt und visualisiert werden können (vgl. *Abbildung 5.1*). Zum anderen wird ein Feature-Modell-Editor genutzt, mit dem *Feature-Diagramme* verwaltet werden können.

C) Der RelationManager ist Teil der neu entwickelten Komponenten und basiert im Wesentlichen auf dem *System-Modell*, d.h. also auf einer abstrakten Repräsentation

des Quellcodes (vgl. *Abbildung 5.1*; *Kapitel 5.2.1*). Jener Manager gibt dem Anwender, z.B. während der *Feature-Zugehörigkeitsprüfung* (vgl. *Abbildung 5.1*), die Möglichkeit Elemente in ihrem Kontext besser zu verstehen. Der Texteditor und der RelationManager sind miteinander verknüpft. Wird im Quellcode ein „Schlüssel-Element markiert, so stellt der RelationManager dieses mit seinen Beziehungen zu anderen Schlüssel-Elementen dar. Aus dem RelationManager heraus können wiederum abhängige Elemente im Quellcode angezeigt werden. Auf diese Weise kann das gesamte System-Modell explorativ erschlossen werden.

D) Diese Sicht wird durch *FLAT*[3] bereitgestellt. Über die Schaltknöpfe lässt sich unter anderem das zu überführende System starten, um *markierte Ausführungsspuren* zu erzeugen. Zudem können hierüber auch die *textuellen Anfragen formuliert und ausgeführt werden* (vgl. *Abbildung 5.1*).

E) Die neu entwickelten Feature- bzw. NonFeatureManager-Sichten korrespondieren zu dem *Feature-Elemente-* bzw. *Nicht-Feature-Elemente-Datenspeicher* (vgl. *Abbildung 5.1*). In diesen beiden Sichten werden alle zu einem Feature zugeordneten bzw. abgelehnten Schlüssel-Elemente dargestellt. Die Ansicht der Elemente lässt sich nach unterschiedlichen Attributen sortieren, wie z.B. Name, Element-Typ, Bereich im Quellcode, Einfügereihenfolge, Anzahl der Betrachtungen im Editor sowie Anzahl der hieraus aktuell generierten Element-Vorschläge. Wie auch in dem RelationManager lassen sich die einzelnen Elemente über einen Doppelklick im Editor öffnen und hervorheben.

F) Der RecommendationManager ist ebenso Teil der Neuentwicklung. Er visualisiert alle aktuellen Element-Vorschläge zu dem markierten Feature in *E* (vgl. *Abbildung 5.3*). Standardmäßig ist die Ansicht nach dem Relevanzwert sortiert - beginnend mit dem höchsten. Bei gleichen Werten erfolgt, jeweils in aufsteigender Reihenfolge, zunächst eine Sortierung nach der Anzahl der zugehörigen Textzeichen im Quellcode und dann nach dem Elementnamen. Für jeden Feature-Element-Vorschlag lässt sich in der Sicht ablesen durch welche bereits identifizierten Elemente es vorgeschlagen wurde – und aus welchem Grund. Besitzt ein Element-Vorschlag einen höheren Relevanzwert für ein anderes Feature als das aktuell dargestellte, wird jene Information explizit hervorgehoben. Ähnlich wie in *E* und *C* lässt sich weiterhin ablesen wie oft ein Element schon in dem Editor betrachtet wurde und in welchem Quellcode-Bereich es liegt. Markierte Elemente können per Doppelklick im Editor geöffnet werden.

## 5.2 Vollständigkeitsfokussierte, automatisierte Feature-Expansion

Durch die Integration von FLAT[3] und CIDE, welche von anderen Forschern entwickelt wurden, konnte das Aufgabenspektrum der Lokalisierung und Dokumentation sehr gut abgedeckt werden (vgl. *Kapitel 5.1*). Für den Teil der *Feature-Expansion* war es allerdings nicht möglich auf fertige Werkzeuge zurückzugreifen, welche die Anforderungen des Feature Mining erfüllen (vgl. *Kapitel 4.2; 4.4*). Fehlende (automatisierte) Unterstützung in diesem Bereich bedeutet, dass ein Anwender komplett seiner Erfahrung und Intuition überlassen wird, um aus bekannten Start-Elementen *vollständige* Feature-Repräsentationen zu identifizieren. In komplexen Systemen, in denen die verschiedenen syntaktischen und semantischen Beziehungen nur mit großer Mühe eigenständig überblickt werden können, ist ein rein manuelles Vorgehen nicht nur sehr zeitaufwändig sondern vermutlich auch fehlerbehaftet (vgl. *Kapitel 3.1*). Um den Suchraum und somit den Suchaufwand zu reduzieren, wurde im Rahmen dieser Arbeit *ein eigenes Konzept entwickelt* in dem Vorschläge zu weiteren möglichen Feature-Elementen *automatisch generiert* werden. In diesem Kapitel wird die konkrete Umsetzung jener Neu-Entwicklungen beschrieben.

Im Folgenden werden einleitend zwei Vorverarbeitungsschritte diskutiert, welche notwendige Eingabedaten für den Algorithmus liefern – und zwar zum einen das *System-Modell* und zum anderen die *Feature-Abhängigkeiten* (vgl. *Abbildung 5.1*). Darauf anknüpfend wird dann der eigentliche *Expansions-Algorithmus* besprochen.

### 5.2.1 System-Modell

Bevor Feature-Element-Vorschläge generiert werden können, muss der Quellcode hinsichtlich der *auftretenden Schlüssel-Elemente und -Beziehungen interpretiert* werden. Das Ergebnis entspricht dann einer *abstrakteren, leichter handhabbaren Repräsentationsform* des Quellcodes, welche im Folgenden als *System-Modell* bezeichnet wird. Die Granularität eines solchen Modells kann je nach der verfolgten Zielsetzung sehr unterschiedlich sein. In der Untersuchung des aktuellen Stands der Technik wurde bei anderen Werkzeugen eine Spanne beobachtet, die von exakten *Systemabhängigkeitsgraphen,* wie bei Indus und WALA (vgl. *Kapitel 4.2.5*), bis hin zu *leichtgewichtigen Modellen*, wie z.B. bei Cerberus, FEAT, JRipples und Suade, reicht (vgl. *Kapitel 4.2.5; 4.2.6*). Für den Expansions-Algorithmus in LEADT wird eine Lösung benötigt, die in der Mitte der beiden Extreme liegt. Auf der einen Seite müssen selbst feingranulare Elemente innerhalb von Methoden berücksichtigt werden, um möglichst präzise Vorschläge generieren zu können. Auf der anderen Seite muss das Modell aber derart abstrakt sein, dass grundsätzliche Abhängigkeiten und Beziehungen ablesbar sind. Aus der Literaturrecherche konnte für die Programmiersprache Java kein frei verfügbares Konzept identifiziert werden, welches ohne Abstriche für LEADT geeignet ist.

Aufgrund dessen wurden ein eigenes System-Modell und ein zugehöriger Extraktor entwickelt. Obwohl sich das System-Modell an der *Java Sprachen Spezifikation*[1] orientiert, erhebt es keinen Anspruch auf eine komplette Abdeckung aller theoretisch möglichen Facetten. Der in dieser Arbeit vorgeschlagene Expansions-Ansatz versteht sich als ein erster Prototyp. Daher wird sich auch in diesem frühen Stadium zunächst nur auf wesentliche Elemente und Beziehungen konzentriert. Allerdings wird für das vorgeschlagene Modell angenommen, dass es trotzdem die meisten Java-Anwendungen hinreichend genau approximiert, um damit eine hohe Vollständigkeit der Feature-Repräsentationen erreichen zu können.

## Schlüssel-Elemente

In dieser Arbeit wird ein Element als *Schlüssel-Element* bezeichnet, wenn es innerhalb des Quellcodes (für die Feature-Expansion) wesentliche Informationen kodiert. Insgesamt wurden elf wesentliche Typen von Schlüssel-Elementen identifiziert. *Tabelle 5.1* fasst sie überblicksartig zusammen. In der zweiten Spalte werden alle *Java-Elemente* benannt, die zu dem jeweiligen *Schlüssel-Element* korrespondieren. Grundsätzlich lassen sich die Schlüssel-Elemente in drei Haupttypen einteilen: (a) *Strukturierungseinheiten*, (b) *Deklarationen* und (c) *Referenzen*. Jene Dreiteilung ist auch in der *Tabelle 5.1* abgebildet.

| Typ | Schlüssel-Element | Java-Element/e |
|---|---|---|
| a | **Kompilierungseinheit** | Java-Quellcode-Datei |
| b | **Import** | Import-Deklaration |
| | **Typ** | Klassen- Deklaration; abstrakte Klassen-Deklaration; Schnittstellen-Deklaration; Enumerations-Deklaration |
| | **Feld** | Feld-Deklaration; Enumerations-Element-Deklaration |
| | **Methode** | Methoden-Deklaration; abstrakte Methoden-Deklaration; Konstruktor-Deklaration |
| | **Lokale Variable** | Lokale -Variablen-Deklaration; Parameter-Deklaration |
| c | **Typ Referenz** | Zeichenfolge eines „Typ"-Namens ≙ Referenz auf „Typ"-Element und ggf. „Import"-Element z.B. in Instanzerstellungen, Casts, Instance-of-Prüfungen |
| | **Feld Referenz** | Zeichenfolge eines „Feld"-Namens ≙ Referenz auf „Feld"-Element z.B. in Schreib-/Lese-Operationen |
| | **Methoden Referenz** | Zeichenfolge eines „Methoden"-Namens ≙ Referenz auf „Methoden"-Element z.B. in Aufrufen |
| | **Lokale Variablen Referenz** | Zeichenfolge eines „Lokalen Variablen"-Namens ≙ Referenz auf „Lokales Variablen"-Element z.B. als Schreib-/Lese-Operation |
| | **Parameter Referenz** | Bereich innerhalb eines „Methoden Zugriff"-Elements ≙ Übergabe eines Parameterwerts |

Tabelle 5.1: Schlüssel-Elemente des System-Modells

[1] http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html

Erfahrenen Java- bzw. Eclipse-Entwicklern fällt vermutlich auf, dass Elemente wie *Paket*, *Projekt* oder *Bibliothek* nicht enthalten sind. In der aktuellen Version von LEADT liegt der primäre Fokus bislang auf feingranularen Elementen *innerhalb von Quellcode-Dateien*. Daher wurde zunächst aus Vereinfachungsgründen auf *übergeordnete Strukturierungseinheiten* verzichtet.

Darüber hinaus fehlen *Block-*, *If-Else-*, *Schleifen-* und *Try-Catch-Anweisungen* sowie *Zuweisungs-*, *Instanzerstellungs-* und *Cast-Ausdrücke* – um einige der Wichtigsten zu nennen. Solche Struktureinheiten werden aus zwei folgenden Gründen dem Anwender nicht *separat* empfohlen und somit auch nicht in dem System-Modell erfasst:

1. Anweisungen und Ausdrücke umfassen *zusammenhängende Quellcode-Bereiche*, die *meist* nur *aus wenigen Zeilen* bestehen. Es wird angenommen, dass wenn ein Anwender ein Element innerhalb eines solchen Bereiches empfohlen bekommt, er in der *Feature-Zugehörigkeitsprüfung* auch den unmittelbaren, lokalen Kontext überprüfen wird, um eine Entscheidung treffen zu können. Daher werden *umschließende Anweisungen und Ausdrücke zwangsläufig mit erfasst*.

2. In vielen Fällen können vorgeschlagene Referenz-Elemente nicht direkt, diszipliniert markiert werden. Die *Erweiterung zu* dem *nächst größeren, disziplinierten Bereich* führt häufig *automatisch* dazu, dass umschließende Anweisungen oder Ausdrücke ebenfalls zu dem Feature zugeordnet werden (vgl. Beispiele in *Abbildung 5.2, S. 80*).

Es sei ausdrücklich darauf hingewiesen, dass der Extraktor nur Schlüssel-Elemente von Quellcode-Dateien innerhalb eines Java-Projektes erfasst. Verknüpfte Bibliotheken werden in dem aktuellen Prototyp nicht in das System-Modell überführt.

## Schlüssel-Beziehungen

Schlüssel-Elemente stehen im Quelltext zu einander in Beziehung. Wenn eine solche Beziehung (für die Feature-Expansion) wesentliche Informationen kodiert, wird sie in dieser Arbeit als *Schlüssel-Beziehung* bezeichnet. In dem aktuellen System-Modell werden drei Typen von Schlüssel-Beziehungen erfasst:

- *enthält* (vgl. *Abbildung 5.4; 5.5*)
- *referenziert* (vgl. *Abbildung 5.6; 5.7*)
- *verwendet* (vgl. *Abbildung 5.8; 5.9*)

Die *Enthält-Beziehung* bildet die typische hierarchische Struktur der Schlüssel-Elemente innerhalb einer Java-Quellcode-Datei ab. Demnach *enthält* eine Kompilierungseinheit

Import-Ausdrücke sowie Typen. Ein Typ kann weitere Typen *enthalten*, sowie Felder und Methoden. Eine Methode *enthält* wiederum lokale Variablen sowie die diversen Referenzen. Die meist einzeiligen Deklarations-Anweisungen von Felder und lokalen Variablen können selbstverständlich auch Referenzen auf andere Elemente enthalten. Jene Beziehung wird hier nicht erfasst, weil sie bei einer manuellen Zugehörigkeitsprüfung direkt erkennbar ist (vgl. Begründung in Punkt 1 und 2 im vorherigen Unterkapitel). Die Enthält-Beziehung zwischen Methoden- und Parameter Referenz ist ebenfalls manuell ablesbar. Jene wird innerhalb des Expansionsalgorithmus allerdings explizit benötigt und infolgedessen auch in dem Modell erfasst (vgl. *Abbildung 5.4*). *Abbildung 5.5* veranschaulicht die *Enthält-Beziehung* anhand einiger Beispiele.

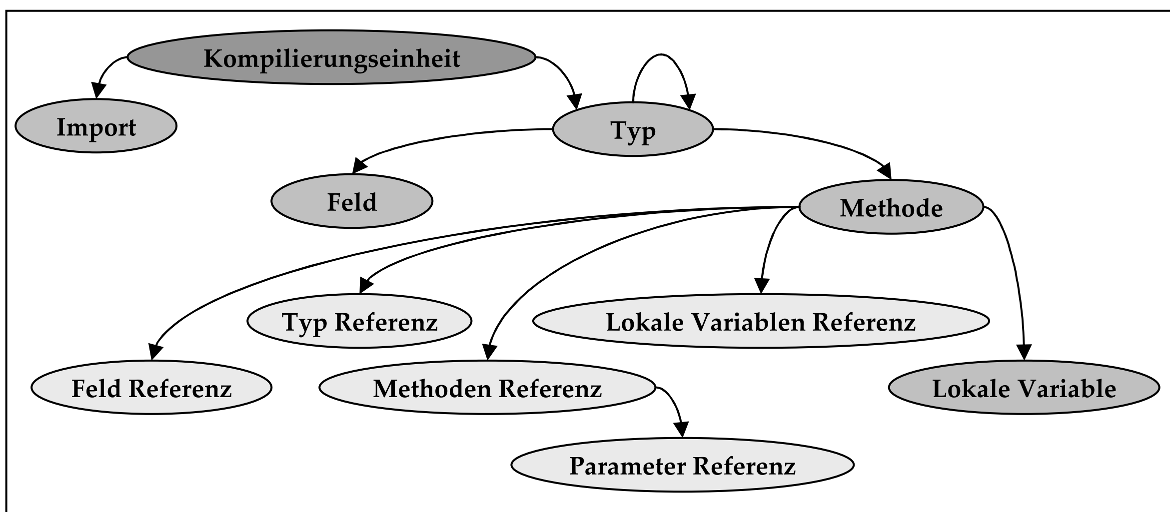



Abbildung 5.4: Enthält-Beziehung des System-Modells

```
1  public TypA {
2        private int feldA = 0;
3        public void methodeA(TypB lokVarA) {
4              methodeB(feldA);
5              …
6        }
7        …
8  }
```

**2: Typ - TypA** *enthält* **Feld - feldA**
**3: Typ - TypA** *enthält* **Methode - methodeA**
**4: Methode - methodeA** *enthält* **Met. Referenz - methodeB**
**4: Met. Referenz - methodeB** *enthält* **Param. Referenz - feldA**

Abbildung 5.5: Beispiele für Enthält-Beziehung

Die *Referenziert-Beziehung* verknüpft alle Deklarationen mit ihren Referenzen im Quellcode. Eine Besonderheit gilt hierbei für die Typ-Referenz. Jene kann zwei Schlüssel-Elemente *referenzieren*. Zum einen ihre zugehörige Typ-Deklaration und zum anderen die Import-Anweisung des Typen innerhalb einer Kompilierungseinheit. Eine weitere Ausnahmestellung nehmen lokale Variablen ein, wenn sie als Parameter für Methoden definiert werden. Jene werden dann einerseits von allen Stellen *innerhalb der Methode referen-*

*ziert*, in denen auf die Variable zugegriffen wird. Andererseits wird von allen zugehörigen Stellen *innerhalb von Methoden-Aufrufen referenziert*, in denen der Parameterwert festgelegt wird (vgl. *Abbildung 5.6*). Zu der *Referenziert-Beziehung* wird hier auch das Überschreiben bzw. das Implementieren einer Methode A von einer geerbten Methode B gezählt. In diesem Fall werden zwei Deklarationen mit einander verbunden. In dem Expansionsalgorithmus wird jene spezielle Abhängigkeit gesondert behandelt und daher in *Abbildung 5.6* auch in einer abweichenden Form dargestellt. Liegt eine Deklaration nicht innerhalb der analysierten Quellcode-Dateien, wird für die zugehörige Referenz keine Beziehung in dem System-Modell hinterlegt. *Abbildung 5.7* veranschaulicht die *Referenziert-Beziehung* anhand einiger Beispiele.

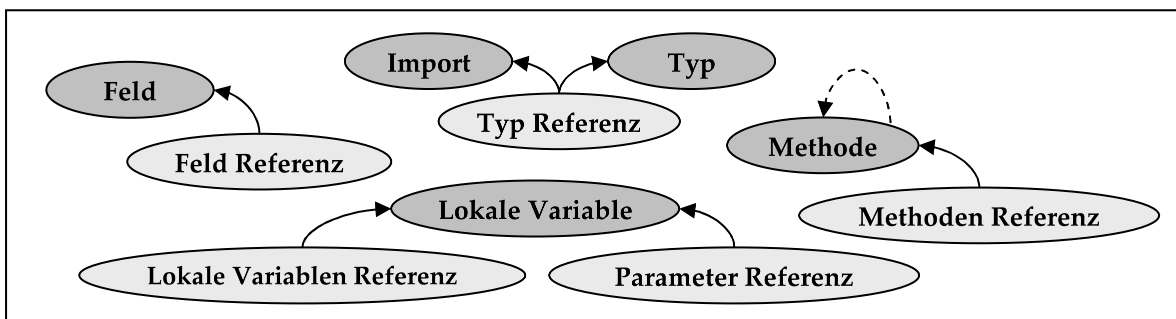



Abbildung 5.6: Referenziert-Beziehung des System-Modells

```
1  import paket.TypB;
2  public TypA {
3        private int feldA = 0;
4        public void methodeA(TypB lokVarA) {
5              methodeB(feldA);
6              …
7        }
8  …
9  }
```

**4: Typ Refrenz - TypB** *refrenziert* **Typ - TypB**
**4: Typ Refrenz - TypB** *refrenziert* **Import - TypB**
**5: Met. Referenz methodeB** *referenziert* **Methode - methodeB**
**5: Feld Referenz - feldA** *referenziert* **Feld feldA**
**5: Param. Referenz - feldA** *referenziert* **Lok. Variable - lokVarB**

Abbildung 5.7: Beispiele für Referenziert-Beziehung

*Verwendet* ein Deklarations-Schlüssel-Element ein anderes Element *in einem gewissen Kontext*, so wird jener Zusammenhang als *Verwendet-Beziehung* in dem Modell ausgedrückt. Für Felder und lokale Variablen kann der Kontext ein *Deklarations-, Zuweisungs-, Instance of-* oder *Cast-Ausdruc*k sein. Für Methoden besteht der Kontext aus dem gesamten zugehörigen *Deklarations-Block* sowie aus möglichen *Aufruf- und Instanzerstellungs-Ausdrücken*. Typen können nur von anderen Elementen verwendet werden und haben daher keinen eigenen Kontext. Die Beziehungen werden immer nur dann in das System-Modell aufgenommen, wenn beide Deklarationen Teil des analysierten Quellcodes sind. In *Abbildung 5.8* werden alle unterschiedlich möglichen Abhängigkeiten dargestellt. *Abbildung 5.9* veranschaulicht zudem die *Verwendet-Beziehung* anhand einiger Beispiele.

Abbildung 5.8: Verwendet-Beziehung des System-Modells

```
1   …
2   public void methodeA(TypA lokVarA)
3                                   throws TypB {
4      methodeB(feldA);
5      If (lokVarA instance of TypC)
6         this.feldB = lokVarA;
7   }
8   …
```

**2: Lok. Variable - lokVarA** *verwendet* **Typ - TypA**
**3: Methode - methodeA** *verwendet* **Typ - TypB**
**4: Methode - methodeA** *verwendet* **Methode - methodeB**
**4: Methode - methodeB** *verwendet* **Feld - feldA**
**5: Lok. Variable - lokVarA** *verwendet* **Typ - TypC**
**6: Feld - feldB** *verwendet* **Lok. Variable - lokVarA**

Abbildung 5.9: Beispiele für Verwendet-Beziehung

## Aktuelle Implementierung

Die Erzeugung dieses System-Modells erfolgt einmalig in einer initialen Analysephase. Hierbei werden die Quellcode Dateien des Java Projektes bzw. dessen Repräsentationen im abstrakten Syntaxbaum zweimal durchlaufen. In der ersten Iteration werden alle Schlüssel-Elemente extrahiert und die Enthält-Beziehungen abgebildet. Die zweite Iteration sorgt dafür, dass Querverweise im Quellcode, d.h. die Referenziert- sowie Verwendet-Beziehung, angeleget werden. Für die Verwaltung der benötigten Graph-Datenstruktur wurden Teile der *JayFX* Bibliothek[2] verwendet. In der aktuellen Version von LEADT muss nach jeder Änderung im Quellcode die Analyse wiederholt werden. Im Feature Mining wird allerdings angenommen, dass *bereits bestehende Systeme* überführt werden sollen. Aufgrund dessen ist davon auszugehen, dass Änderungen im Quellcode nur in Ausnahmefällen erfolgen werden. Das vollständige System-Modell wird bislang nur in dem Hauptspeicher verwaltet. Die Auslagerung des Modells für größere Systeme in eine Datenbank ist bereits in Planung.

---

[2] http://www.cs.mcgill.ca/~swevo/jayfx/

Es sei zudem darauf hingewiesen, dass in dem System-Modell zu allen Beziehungstypen darüber hinaus auch die transitiven Pendants erfasst werden. In *Abbildung 5.9* wäre es beispielsweise *„Methode - methodeA verwendet transitiv Feld - feldA“*. Jene Informationen werden nur für den RelationManager benötigt (vgl. *Abbildung 5.3/C, S. 82*), um dem Anwender die Navigation während der Feature-Zugehörigkeitsprüfung zu erleichtern. Der Expansions-Algorithmus verwendet hingegen nur „direkte“ Abhängigkeiten.

## 5.2.2 Feature-Abhängigkeiten

In einem weiteren Vorverarbeitungsschritt werden *Abhängigkeitsbeziehungen aus dem Feature-Diagramm extrahiert* und gespeichert. Diese Domänen-Informationen werden verwendet, um zusätzlich den Suchraum bzw. -aufwand zu reduzieren. Nach dem Kenntnisstand des Autors, der auf der umfangreichen Analyse des aktuellen Stands der Technik basiert (vgl. *Kapitel 4*), ist nicht bekannt, dass andere Expansionstechniken existieren, die vergleichbare Informationen in ihrem Vorgehen berücksichtigen.

Im Folgenden wird diskutiert, welche *Feature-Abhängigkeitsbeziehungen* verwendet werden und *in wie fern sie nützlich* sein können. Zudem wird schematisch dargestellt, *wie* sie aus einem Feature-Modell *ausgelesen* werden können. Hierbei wird direkt auf Grundlagen zu Feature-Modellen und ihren Darstellungsweisen des *Kapitels 2.3.1* aufgebaut.

### Art der Feature-Abhängigkeiten und ihr Nutzen

Zur Veranschaulichung des möglichen Nutzens von Feature-Abhängigkeitsbeziehungen für die Expansion sei exemplarisch der Ausschnitt eines Feature-Modells in *Abbildung 5.10* näher betrachtet. Aus diesem lässt sich zunächst erkennen, dass das zugehörige System ein EXPORT Feature enthalten kann. Wird dieses Feature in einer Variante ausgewählt, muss das konkrete Format näher spezifiziert werden. Hierbei muss entweder PDF oder HTML genutzt werden.

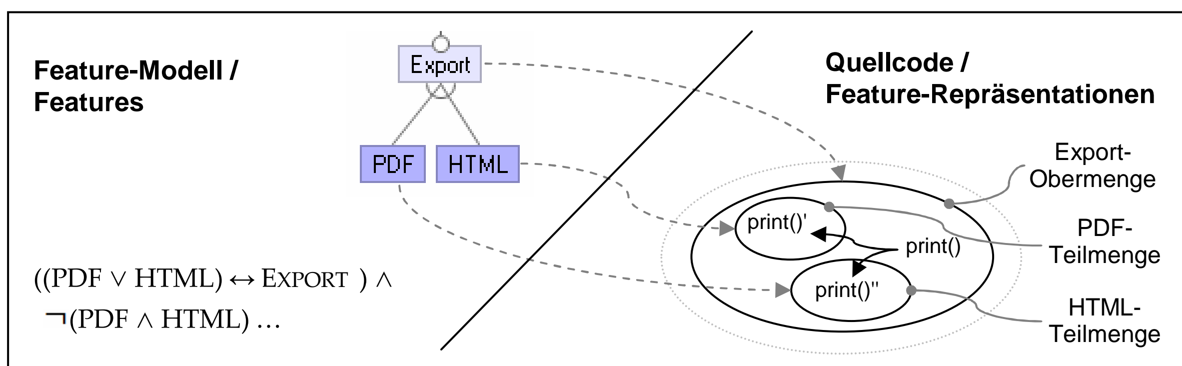



Abbildung 5.10: Von Feature- zu Feature-Repräsentations-Abhängigkeiten

Aus dem in *Abbildung 5.10* dargestellten Feature-Diagramm bzw. dem zugehörigen aussagenlogischen Ausdruck können für die zugehörigen Feature-Repräsentationen zwei grundsätzliche Abhängigkeitsannahmen getroffen werden:

- *Ausschluss-Abhängigkeit* – Die Features PDF und HTML können nie in der selben Variante auftreten (es gilt: PDF → ¬HTML sowie HTML → ¬PDF). Quellcode-Elemente, die zu PDF *und* HTML gehören wären in keiner Variante sichtbar. Es ist also davon auszugehen, dass die Feature-Repräsentationen von PDF *und* HTML *disjunkte Quellcode-Element-Mengen* bilden (vgl. *Abbildung 5.10*). Während der Expansion für das Feature PDF sollten somit keine Elemente vorgeschlagen werden von denen bekannt ist, dass sie zu HTML gehören – und umgekehrt.

- *Notwendigkeits-Abhängigkeit* – In allen Varianten in denen PDF oder HTML enthalten ist, *muss auch notwendigerweise* EXPORT enthalten sein (es gilt: PDF → EXPORT sowie HTML → EXPORT). Es kann daher angenommen werden, dass die PDF- und HTML-Feature-Repräsentationen *Teilmengen* von EXPORT sind und Quellcode-Elemente aus dieser *Obermenge* verwenden. Jener Sachverhalt ist in *Abbildung 5.10* mit der Deklaration *print()* in EXPORT und den beiden zugehörigen Referenzen *print()'* in HTML und *print()''* in PDF angedeutet. Bekannte Schlüssel-Elemente von PDF und HTML können somit unter gewissen Umständen Rückschlüsse auf die (übrige) Feature-Repräsentation von EXPORT erlauben und sollten daher während dessen Expansion mit berücksichtigt werden. Entsprechend sollten für EXPORT auch keine Elemente mehr vorgeschlagen werden von denen bekannt ist, dass sie schon zu einer der beiden Teilmengen gehören.

Es lässt sich zusammenfassen, dass in dieser Idee *Ausschluss-* und *Notwendigkeits-Abhängigkeiten* aus dem Feature-Modell auf Feature-Repräsentationen übertragen werden. Wenn Repräsentationen von entsprechend abhängigen Features bekannt sind, kann hierdurch *die Anzahl fehlerhafter Element-Vorschläge für ein gesuchtes Feature reduziert werden*. Speziell mit Hilfe von *Notwendigkeits-Abhängigkeiten* können darüber hinaus *Vorschläge für das (benötigte) Obermengen-Feature, anhand von bekannten Elementen eines bereits identifizierten Teilmengen-Features*, erzeugt werden. Auf diese Weise ist es auch indirekt möglich Start-Elemente einer Feature-Repräsentation zu lokalisieren, wenn keine weiteren externen Informationen, wie z.B. textbasierte Anfragen oder Ausführungsspuren, vorhanden sind.

### Ermittelung der Feature-Abhängigkeiten

In dem kleinen Beispiel aus *Abbildung 5.10* lassen sich Notwendigkeits-Abhängigkeiten der Form „A → B“ bzw. Ausschluss-Abhängigkeiten der Form „A → ¬B“ noch recht einfach in dem Feature-Modell erkennen. In realen Anwendungen können Feature-Modelle allerdings durchaus mehrere hunderte oder tausende Features enthalten (vgl. *Kapitel*

*2.3.1*), sodass ein manuelles Vorgehen unpraktikabel ist. Wie bereits in *Kapitel 2.3.1* angekündigt wird daher die Funktionsweise des in CIDE enthaltenen Produktkonfigurators von Thüm et al. [TBK09] adaptiert, um in LEADT aus einem beliebigen Feature-Diagramm automatisiert Notwendigkeits- oder Ausschluss-Abhängigkeiten zu ermitteln.

Hierbei wird initial das zugehörige Feature-Diagramm in einen aussagenlogischen Ausdruck *FM* überführt (siehe Beispiele *Abbildung 2.4, S. 20; Abbildung 5.10*). *FM* umfasst auch in dieser Repräsentationsform alle für das zugehörige System möglichen bzw. gültigen Feature-Selektionen, lässt sich aber besser „maschinell" verarbeiten (vgl. *Kapitel 2.3.1*). Mit Hilfe der Bibliothek *SAT4J*[3] lassen sich dann die Features *R(S)* bestimmen, die in *allen* (gültigen) Selektionen beschrieben durch den Ausdruck *S* enthalten sind. *R(FM)* enthält demnach Features, welche in jeder Variante des Systems ausgewählt sein müssen. Beispiel hierfür könnte gewisse Kern-Funktionalität sein, die immer benötigt wird. Analog hierzu können die Features *I(S)* ermittelt werden, die in *keiner* (gültigen) Selektion, beschrieben durch den Ausdruck *S*, enthalten sind. *I(FM)* zeigt Features, die nie ausgewählt werden können. In der Regel sollte dieser Fall nie auftreten, da solche Features für das System wertlos sind.

Zur Ermittelung von *Notwendigkeits-Abhängigkeiten* wird ein Feature $A_i$ bzw. dessen Repräsentation als Boolsche-Variable über einen UND-Operator mit *FM* verknüpft. Hiervon wird die Menge $R(FM \wedge A_i)$ ermittelt. Subtrahiert man dann von *R* das Feature $A_i$ sowie alle Features *R(FM)*, die ohnehin in jeder Variante enthalten sind, so können Features $B_{ij}$ verbleiben, die genau dann ausgewählt werden *müssen*, wenn $A_i$ ausgewählt wird. Für alle $B_{ij}$ gilt somit die Abhängigkeit $A_i \rightarrow B_{ij}$. In diesem Zusammenhang wird im Folgenden $B_{ij}$ auch als *Obermengen-Feature* und $A_i$ als *Teilmengen-Feature* bezeichnet. Dieses Vorgehen wird für alle Features $A_i$ aus dem Feature-Modell durchgeführt. Abschließend werden Abhängigkeiten gelöscht, *welche zu einer Äquivalenz führen*. Gilt beispielsweise $X \rightarrow Y$ und $Y \rightarrow X$, kann letztlich die Zusammenhangsrichtung nicht eindeutig angegeben werden. Abgeleitete Teil- und Obermengen der Feature-Repräsentationen wären in diesem Fall widersprüchlich. In *Abbildung 5.11* werden zur Verdeutlichung einige Beispiele gezeigt, in denen solche Fälle auftreten können.

Zur Ermittelung von *Ausschluss-Abhängigkeiten* wird ebenfalls ein Feature $A_i$ bzw. dessen Repräsentation als Boolsche-Variable über einen UND-Operator mit *FM* verknüpft. Hiervon wird allerdings die Menge $I(FM \wedge A_i)$ ermittelt. Subtrahiert man dann von *I* das Feature $A_i$ sowie alle Features *I(FM)*, die *nie* in einer Variante enthalten sein dürfen, so können Features $B_{ij}$ verbleiben, die genau dann *nicht* ausgewählt werden *dürfen*, wenn $A_i$ ausgewählt wird. Für alle $B_{ij}$ gilt somit die Abhängigkeit $A_i \rightarrow \neg B_{ij}$. Dieses Vorgehen wird für

[3] SAT4J ist eine Open-Source Bibliothek mit der das Erfüllbarkeitsproblem in der Aussagenlogik effizient bearbeitet werden kann. SAT4J ist verfügbar unter: http://www.sat4j.org/

alle Features $A_i$ aus dem Feature-Modell durchgeführt, aber nicht für die Negationen der Features. Daher können in diesem Fall auch keine Äquivalenz-Abhängigkeiten auftreten.

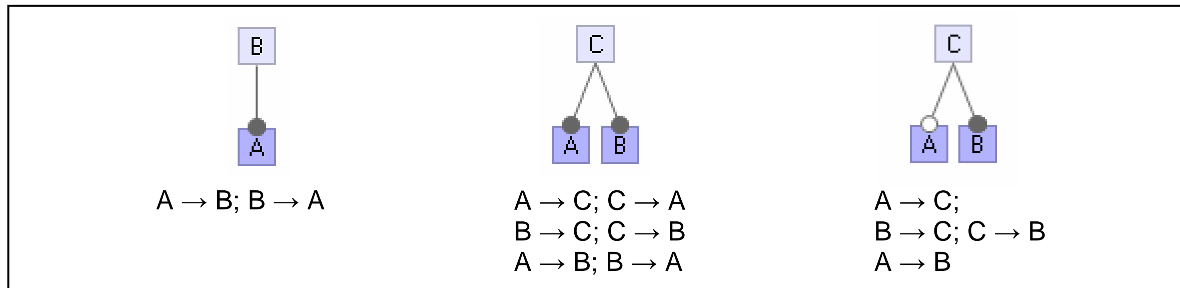



Abbildung 5.11: Beispiele für mögliche Äquivalenz-Abhängigkeiten

Es sei angemerkt, dass sich Ausschluss-Abhängigkeiten zwischen *Feature A* und *B* auch als Notwendigkeits-Abhängigkeiten zwischen einem *Feature A* und einem *Nicht-Feature B* interpretieren lassen. Schließlich können Elementen von A auch Rückschlüsse zu Quellcode-Elementen erlauben, die *nicht* zu B gehören. Durch die Expansion von *Nicht-Feature B* ließe sich *Feature B* in einer Art Ausschluss-Methode indirekt identifizieren. LEADT setzt bislang den Fokus allerdings nur auf eine *direkte Expansion von Features.* Aufgrund dessen werden Notwendigkeitsbeziehungen zwischen Features und Nicht-Features bislang nicht weiter berücksichtigt.

Der aktuelle Prototyp bezieht sich zudem nur auf direkte Abhängigkeiten zwischen Features. Obwohl auch geschachtelte Ausdrücke wie z.B. $(A \wedge B) \rightarrow C$ ebenso wertvolle Informationen mit einbringen könnten. Die Idee Feature-Abhängigkeiten für die Expansion zu nutzen ist neu. Bis dato existieren hierzu keine praktischen Erfahrungen. Das Vorgehen beschränkt sich daher bewusst zunächst auf einfache, elementare Zusammenhänge, um die *grundsätzliche* Nützlichkeit besser untersuchen zu können.

### 5.2.3 Expansions-Algorithmus

Der Expansions- bzw. Vorschlags-Algorithmus in LEADT ist das zentrale Hilfsmittel, um den Anwender bei der Erschließung von Feature-Repräsentationen zu unterstützen. Zur Reduzierung des manuellen (Such-)Aufwandes werden durch den Algorithmus Schlüssel-Elemente im Quellcode vorgeschlagen, von denen angenommen wird, dass sie für das gesuchte Feature relevant sein könnten.

Um eine möglichst hohe *Vollständigkeit* der Feature-Repräsentationen zu erzielen (vgl. *Kapitel 3.1*), kombiniert der Expansions-Algorithmus drei Vorschlagsansätze:

- Eine adaptierte *Spezifitäts-* und *Bestärkungsheuristik* basierend auf Schlüssel-Element-Beziehungen
- Eine *Textmuster-Erkennung*
- Ein rudimentäres *SPL-bewusstes Typsystem*, welches Typfehler in gültigen Systemvarianten ermittelt

Teile dieser drei Ansätze wurden bislang in unterschiedlichen Werkzeugen separat verwendet (vgl. *Kapitel 4.2.6; 4.2.2; 4.2.5*). In LEADT werden sie allerdings aufgrund ihrer komplementären Eigenschaften gebündelt und in bestimmten Details für das Feature Mining angepasst bzw. verbessert.

Im Folgenden werden das Grundgerüst des Algorithmus sowie das Vereinigungsschema der unterschiedlichen Empfehlungen besprochen. Im Anschluss daran werden die Umsetzung der drei implementierten Empfehlungsansätze sowie die Vorteile, welche sich durch eine Kombination ergeben, diskutiert.

### 5.2.3.1 Grundgerüst und Vereinigungsschema

*Abbildung 5.12* zeigt das *Grundgerüst* für den Algorithmus im Pseudocode-Format. Hieraus lässt sich erkennen, dass als Eingabeparameter ein Feature übergeben wird, für welches Element-Vorschläge generiert und zurückgegeben werden (*Zeile 1*). In *Zeile 3* wird das zu dem System zugehörige Feature-Modell *FM* geladen. Jenes enthält auch die in *Kapitel 5.2.2* beschriebenen Feature-Abhängigkeiten. Mit Hilfe dieses Modells werden folgend bereits identifizierte Elemente aus dem Feature- sowie Nicht-Feature-Elemente Datenspeicher geladen und in den Variablen *I* bzw. *Nicht_I* abgelegt (vgl. *Kapitel 5.1.1*; *Abbildung 5.1*; S. 78). Hierbei werden, wie bereits in *Kapitel 5.2.2* beschrieben, nicht nur Elemente des Features selbst, sondern auch Elemente von dessen abhängigen Features berücksichtigt (*Zeile 4-5*). In einem weiteren Initialisierungsschritt wird das in *Kapitel 5.2.1* erläuterte System-Modell *SM* geladen sowie eine Variable *EV* angelegt, welche die Element-Vorschläge aufnehmen soll (*Zeile 7-8*).

Die eigentliche Generierung der Vorschläge wird an die einzelnen hinterlegten Ansätze delegiert, welche in den folgenden Kapiteln vorgestellt werden (*Zeile 10-13*). Jeder der Ansätze erhält als Eingabe das Feature *F*, alle Elemente aus *I* (die schon zu *F* oder zu entsprechenden Teilmengen-Features gehören), alle Elemente aus *Nicht_I* von denen man weiß, dass sie nicht zu *F* gehören, das System-Modell *SM* sowie das Feature-Modell *FM*. Zurückgegeben wird eine Menge von Vorschlägen $EV_x$, bestehend aus Paaren von Schlüssel-Elementen *e* sowie zugehörigen Relevanzwerten *r* aus dem Intervall von 0 bis 1 (*Zeile 11*).

```
1   Menge<Element-Vorschlag> gibElementVorschlaege(Feature F) {
2
3       Feature-Modell FM = gibFeatureModell()
4       Menge<Element> I = gibIdentifizierteElemente( F ∪ FM.gibAbhaengigeTeilmengenFeatures(F))
5       Menge<Element> Nicht_I = gibIdentifizierteElemente (FM.gibNichtFeature(F) ∪
6                                                   FM.gibAbhaengigeAusschlussFeatures(F))
7       System-Modell SM = gibSystemModell()
8       Menge<Element-Vorschlag>  EV = new Menge<Element-Vorschlag>()
9
10      for (VorschlagsAnsatz x : gibVorschlagsAnsaetze()) {
11          Menge<Element-Vorschlag>  EVx = x.generiereElementVorschlaege(F, I, Nicht_I, SM, FM))
12          EV.vereinigeVorschlaege(EVx)
13      }
14
15      for (Element-Vorschlag ev : EV) {
16          if (ev.gibVorgeschlagenesElement() ∈ (I  ∪ Nicht_I))
17              EV.entferne(ev)
18      }
19
20      return EV
21
22  }
```

Abbildung 5.12: Grundgerüst des Expansions-Algorithmus

In *Zeile 12* werden die Vorschläge aus $EV_x$ mit der Gesamtmenge *EV* vereinigt. Hierbei werden die Relevanzwerte von Elementen, welche *mehrfach von unabhängigen Vorschlags-ansätzen empfohlen* werden, nach einem bestimmten Schema kombiniert. In *Kapitel 5.2.3.5* wird noch explizit geklärt in wie fern jenes vorteilhaft sein kann.

In dem aktuellen Prototyp von LEADT erfolgt die Vereinigung nach einem von Robilliard [Rob08] adaptierten Ansatz, welcher wie folgt aussieht:

- gilt *ev.gibVorgeschlagenesElement()* $\notin$ *gibVorgeschlageneElemente(EV)*, so wird *ev*(*e* ,*r*) in *EV* aufgenommen
- sonst, so wird *ev(e, r)* überschrieben mit $ev'(e, r_{alt} + r_{neu} - r_{alt} * r_{neu})$

Der resultierende Relevanzwert ist nach diesem Konzept also stets größer oder gleich dem Maximum aus altem sowie neuem Wert, liegt aber immer noch in dem Intervall von 0 bis 1. Ist also beispielsweise in der ersten Iteration der Relevanzwert für ein Element *0,4* so wird dieser „einfach“ übernommen. Ermittelt in der zweiten Iteration ein anderer Ansatz für das gleiche Element einen Wert von *0,2* ergibt sich ein resultierender Wert von *0,4 + 0,2 - 0,4 * 0,2 = 0,52*. Der dritte Ansatz kommt schließlich auf einen Wert von *0,7*. Abschließend ergibt sich somit der Relevanzwert von *0,52 + 0,7 - 0,52 * 0,7 = 0,86* – usw. Die Vereinigungsreihenfolge spielt in diesem Schema keine Rolle, da sowohl die Eigenschaft der Kommutativität als auch Assoziativität erfüllt wird.

Bei dieser Methode besteht die Gefahr, dass durch korrelierte Vorschlagsansätze der Relevanzwert „künstlich“ größer wird [Rob08]. Es sei exemplarisch der schlechteste Fall angenommen, dass immer wieder der selbe Vorschlagsansatz verwendet wird, der jedes mal auf einen Wert von *0,5* kommt. So beträgt der resultierende Wert nach drei Iterationen bereits *0,88* nach acht Iteration ist er letztlich sogar rund *1,0*. In dieser Arbeit werden allerdings drei komplementäre Ansätze verwendet, sodass dieser Effekt prinzipiell ausgeschlossen werden kann.

Das hier verwendete Vereinigungsschema stellt eine mögliche Umsetzung von vielen dar, die zwar aufgrund ihrer Eigenschaften gut geeignet zu sein scheint, aber nicht zwangsläufig die beste Lösung sein muss. In zukünftigen Arbeiten sollte daher eine fundierte Evaluation von unterschiedlichen Schemata erfolgen, um hierfür eine optimale Antwort zu finden.

In den *Zeilen 15-18* des Expansions-Algorithmus werden abschließend aus der Menge aller erzeugten Vorschläge *EV* diejenigen entfernt, die schon in *I* oder *Nicht_I* enthalten sind. Hierbei wird noch einmal deutlich in wie fern abhängige Feature-Repräsentationen, die in *I* und *Nicht_I* enthalten sind, dabei helfen können die Anzahl fehlerhafter Element-Vorschläge zu reduzieren (vgl. *Kapitel 5.2.2*).

Die verbleibenden Element-Vorschläge werden in *Zeile 20* zurückgegeben und dem Anwender in geeigneter Weise präsentiert (vgl. *Kapitel 5.1.2*).

#### 5.2.3.2 Vorschlagsansatz: Spezifitäts- und Bestärkungsheuristik

Robillard et al. [Rob05; Rob08; WR07] wenden in ihrem Expansionswerkzeug Suade die sogenannte *Specifity* und *Reinforcement-Metrik* an, um Vorschläge zu relevanten Elementen zu generieren. Hierbei werden zu bereits identifizierten Schlüssel-Elementen anhand eines System-Modells alle strukturellen Nachbarn ermittelt. Anschließend werden sie hinsichtlich ihrer Relevanz für das gesuchte Feature bewertet. Die *Specifity-Komponente* berücksichtigt dabei wie spezifisch bzw. wie einzigartig eine Verbindung zwischen einem identifizierten Element und einem potentiellen Kandidaten ist. Die *Reinforcement-Komponente* „bestärkt“ hingegen ein Kandidat mit einem höheren Interessegrad, wenn viele seiner verwandten Elemente bereits Teil der Feature-Repräsentation sind. Die aktuelle Version von Suade verwendet lediglich Methoden und Felder als Schlüssel-Elemente, welche über Aufruf- bzw. Zugriffsbeziehungen mit einander verknüpft sind. Feingranularere Elemente und Beziehungen zwischen ihnen werden nicht erfasst (vgl. *Kapitel 4.2.6*).

Aufgrund der viel versprechenden ersten Ergebnisse wird die grundlegende Idee der Spezifitäts- und Bestärkungsheuristik folgend *für das in Kapitel 5.2.1 eingeführte System-Modell* adaptiert [Rob08] (vgl. *Kapitel 4.2.6*).

Durch die im System-Modell verwendeten Beziehungstypen wird ein Vorgehen umgesetzt, welches an die *Prune Dependency Analysis* in dem Werkzeug *Cerberus* erinnert. Wobei allerdings nicht nur offensichtlich zugehörige Elemente automatisch zu einem Feature zugeordnet werden, sondern *Wahrscheinlichkeiten* für *alle* benachbarten Elemente *geschätzt* werden. Trotz gewissen Ähnlichkeiten von einigen Beziehungstypen bestehen deutliche Abweichungen hinsichtlich der Granularität des System-Modells, da LEADT sich nicht nur auf Typen, Methoden und Felder beschränkt (vgl. *Kapitel 4.2.5; 5.2.1*).

LEADT unterscheidet sich von Suade und Cerberus zudem darin, dass nicht nur *Elemente des Features* selbst berücksichtigt werden, sondern auch jene von *abhängigen Teilmengen-Features*, *Nicht-Features* sowie *Ausschluss-Features* (vgl. *Kapitel 5.2.2; 5.2.3.1*).

### Berechnung der Relevanzwerte

In *Abbildung 5.13* ist die Implementierung der Methode zur Generierung von Element-Vorschlägen nach der Spezifitäts- und Bestärkungsheuristik schematisch dargestellt. *Abbildung 5.14* zeigt zudem ein Beispiel-Szenario an dem das Vorgehen kurz erklärt wird.

```
1   Menge<Element-Vorschlag> generiereElementVorschlaege(Feature F, Menge<Element> I,
2                                 Menge<Element> Nicht_I, System-Modell SM, Feature-Modell FM)) {
3
4       Menge<Element-Vorschlag>  EVx = new Menge<Element-Vorschlag>()
5
6       for (Element e : I) {
7           for (Beziehung b : SM.gibVorwaertsUndRueckwaertsBeziehungen()) {
8               Menge<Element> S = SM.gibNachbarElemente(e, b)
9               for (Element s : S) {
10                  if (s ∉ (I ∪ Nicht_I)) {
11                      Menge<Element> SUmkehr = SM.gibNachbarElemente(s,
12                                                   SM.gibUmkehrBeziehung(b))
13                      double rI = (1+ | S ∩ I |) / | S | * | SUmkehr ∩ I | / | SUmkehr |
14                      double rNicht_I = (1+ | S ∩ Nicht_I |) / | S | * | SUmkehr ∩ Nicht_I | / | SUmkehr |
15                      double  r = rI - rNicht_I
16                      if (r >= 0 )
17                          EVx.fuegeHinzu(s, r)
18                  }
19              }
20          }
21      }
22
23      return EVx
24
25  }
```

Abbildung 5.13: Vorschlagsansatz – Spezifitäts- und Bestärkungsheuristik

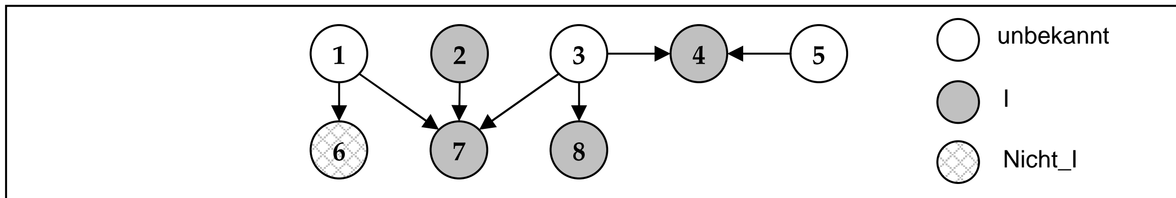



Abbildung 5.14: Verwendet- bzw. „Wird verwendet von“-Beziehung für ein Beispiel

Initial wird eine Variable $E_x$ angelegt, welche die Element-Vorschläge aufnehmen soll, um sie dann an den Hauptalgorithmus zurück zugeben (*Zeile 4, 23*).

Im Anschluss werden für jedes Element *e* aus *I* (*Zeile 6*) sowie jede Beziehung *b* aus dem System-Modell (*Zeile 7*)[4], die benachbarten Schlüssel-Elemente *S* ermittelt (*Zeile 8*). In dem Beispiel von *Abbildung 5.14* ist für *e = 7* sowie *b = „wird verwendet von“* $S_{[7,\ verwendet\ von]}$ = *{1, 2, 3}*. Gehört eines dieser Nachbar-Elemente *s* weder zu *I* noch zu *Nicht-I*, so kann es dem Anwender als potentielles Feature-Element vorgeschlagen werden (*Zeile 10*). In $S_{[7,\ verwendet\ von]}$ wären *1* und *3* verbleibende gültige Kandidaten.

Für jeden dieser Kandidaten muss ein Relevanzwert ermittelt werden. Hierbei wird auf die Kombination der Spezifitäts- und Bestärkungsmetrik zurückgegriffen. Jene erfasst, wie spezifisch bzw. einzigartig eine Verbindung zwischen zwei Elementen ist. Zusätzlich wird berücksichtigt wie viele der benachbarten Elemente bereits in *I* sowie *Nicht_I* enthalten sind. Für die Metrik werden nicht nur die Nachbar-Elemente des vorschlagenden Elementes *e* sondern auch die des Kandidaten *s* benötigt – und zwar hinsichtlich der Umkehrbeziehung von *b* (*Zeile 11-12*) [Rob08]. Für den Element-Kandidaten *s = 1* des betrachteten Beispiels ist $S_{Umkehr\ [1,\ verwendet]}$ = *{6, 7}*. Gemäß der Formel aus *Zeile 13* ergibt sich für $r_I$ der Wert ⅔ * ½ = ⅓. Es wird also angenommen, dass Element *1* mit einer Wahrscheinlichkeit von rund *0,33* zu *F* gehört. Für $r_{Nicht_I}$ ergibt sich der Wert ⅓ * ½ = ⅙ (*Zeile 14*), welcher wiederum die Wahrscheinlichkeit dafür schätzt, dass Element *1* nicht zu F gehört. Der finale Relevanzwert *r* entspricht dann der Differenz aus $r_I$ und $r_{Nicht_I}$ und beträgt in dem Beispiel rund *0,17* (*Zeile 15*). Da *r* größer als 0 ist, wird in diesem Fall Element *1* mit dem Wert von 0,17 in $EV_x$ aufgenommen (*Zeile 16-17*).

Es ist prinzipiell möglich, dass ein Element mehrfach vorgeschlagen wird. In dem Beispiel aus *Abbildung 5.14* wird Element *3* allein mit dem Beziehungstyp *„wird verwendet von“* durch Element *4, 7* und *8* vorgeschlagen. Als Vereinigungslogik ist festgelegt, dass in $EV_x$ immer der maximal ermittelte Wert gespeichert wird. Nach Abschluss des Algorithmus für das Beispiel in *Abbildung 5.14* enthält $EV_x$ daher die Paare *(1; 0,17)*, *(3; 1,0)* und *(5; 0,5)*.

[4] Für das eingeführte System-Modell in Kapitel 5.2.1 sind es die Beziehungen *enthält*, *referenziert* und *verwendet*. Hinzu kommen die Beziehungen der Umkehrrichtungen, d.h. *„ist enthalten in“*, *„wird referenziert von“* sowie *„wird verwendet von“*.

### 5.2.3.3 Vorschlagsansatz: Textmuster-Erkennung

Die im vorherigen Kapitel dargestellte Spezifitäts- und Bestärkungsheuristik betrachtet immer nur Beziehungen zwischen direkt benachbarten Schlüssel-Elementen (vgl. *Kapitel 5.2.3.2*). Empfehlungen, welche sich erst durch eine „zusammenhängende Analyse“ von größeren Element-Mengen erschließen, können auf diese Weise nicht ermittelt werden. Darüber hinaus kann der Vorschlagsansatz *gewichtete Beziehungen* zwischen Elementen *nicht interpretieren*. *Textuelle Beziehungen*, welche meist ohne Abstufungen des Abhängigkeitsgrades nicht auskommen, können *daher nicht sinnvoll verarbeitet werden*. Aufgrund dessen bleibt eine für die Expansion wichtige Informationsquelle ungenutzt (vgl. *Kapitel 4.2.2*). Im Folgenden wird ein *Textmuster-Erkennungsansatz* vorgeschlagen, der genau an diesen Kritikpunkten ansetzt.

Die grundsätzliche Idee ist, dass alle auftretenden Teilzeichenketten der Bezeichner von bereits zugeordneten Typen, Methoden, Feldern und lokalen Variablen mit ihren Auftrittshäufigkeiten in Registern verwaltet werden. Es wird angenommen, dass ein Register das verwendete Vokabular der Feature-Repräsentationen abbildet, wobei die Auftrittshäufigkeit ein Maß für die Wichtigkeit der einzelnen Wörter für das Feature darstellt. Alle Bezeichner der nicht zugeordneten Schlüssel-Elemente werden hinsichtlich ihrer Übereinstimmung mit dem ermittelten Vokabular, des gesuchten sowie dessen verwandten Features, überprüft. Wird hierbei ein gewisser Grenzwert erreicht, so werden die Elemente mit ihrem korrespondierenden Übereinstimmungsgrad empfohlen.

Die Bereinigung, Verarbeitung und Interpretation von Textinformationen in großen Datenmengen spannt ein sehr weites Feld in der Forschung auf, welches nicht primär im Fokus dieser Arbeit steht. Im Rahmen der Untersuchung von textbasierten Expansionstechniken wurden lediglich exemplarisch einige Konzepte angedeutet (*Kapitel 4.2.2*). Viele von ihnen sind mit der beschriebenen Idee in gewissen Teilen verwandt, setzen sie aber nicht in der notwendigen Form um. Die folgende Lösung basiert daher im Wesentlichen auf eigenen, einfachen Überlegungen. Es ist davon auszugehen, dass mit tieferen Fachkenntnissen im Bereich der Computerlinguistik höchst wahrscheinlich deutliche Verbesserungen erzielt werden könnten. Die aktuelle Implementierung in dem Prototyp versteht sich daher eher als ein erster Schritt zur Prüfung des grundsätzlichen Konzepts.

**Berechnung der Relevanzwerte**

In *Abbildung 5.15,* welche sich über die Teile *a* und *b* erstreckt, ist die Implementierung der Methode zur Generierung von Element-Vorschlägen mit Hilfe der Textmuster-Erkennung schematisch dargestellt. *Abbildung 5.16* zeigt zudem ein Beispiel-Szenario an dem das Vorgehen erklärt wird.

```
1   Menge<Element-Vorschlag> generiereElementVorschlaege(Feature F, Menge<Element>  I,
2                                     Menge<Element> Nicht_I, System-Modell SM, Feature-Modell FM)) {
3
4       Menge<Element-Vorschlag>  EVx = new Menge<Element-Vorschlag>()
5       RegisterI = new Abbildung<String, Double>()
6       RegisterNicht_I = new Abbildung<String, Double>()
7
8       for (Element e : gibRegisterKompatibleElemente(I)) {
9           RegisterI.fuegeHinzuUndInkrementiere(gibTeilZeichenketten(e))
10      }
11      for (Element e : gibRegisterKompatibleElemente(Nicht_I)) {
12          RegisterNicht_I.fuegeHinzuUndInkrementiere(gibTeilZeichenketten(e))
13      }
14
15      int SummeI  = gibSummeAusWerten(RegisterI)
16      int SummeNicht_I = gibSummeAusWerten(Nicht_I_Register)
17      for (String k : RegisterNicht_I.gibSchluessel()) {
18          if (k ∈ RegisterI) {
19              double relWichtigkeitI  = RegisterI.gib(k) /  SummeI
20              double relWichtigkeitNicht_I = RegisterNicht_I.gib(k) /  SummeNicht_I
21              if (relWichtigkeitI  < relWichtigkeitNicht_I) {
22                  RegisterI.entferne(k)
23                  RegisterNicht_I.gib(k) -= relWichtigkeitI  * SummeNicht_I
24              }
25              else  if (relWichtigkeitI  > relWichtigkeitNicht_I) {
26                  RegisterNicht_I_I.entferne(k)
27                  RegisterI.gib(k) -= relWichtigkeitNicht_I * SummeI
28              }
29              else {
30                  RegisterNicht_I.entferne(k)
31                  RegisterI.entferne(k) = 0
32              }
33          }
34      }
…
```

Abbildung 5.15.a: Vorschlagsansatz – Textmuster-Erkennung

Initial wird eine Variable $E_x$ angelegt, welche die Element-Vorschläge aufnehmen soll, um sie dann an den Hauptalgorithmus zurück zugeben (*Zeile 4, 50*). Zudem werden die Variablen $Register_I$ und $Register_{Nicht_I}$ deklariert, welche Teilzeichenketten sowie zugehörige Auftrittshäufigkeiten verwalten können.

Es wird grundsätzlich angenommen, dass die Teilzeichenketten in $Register_I$ das verwendete Vokabular von Feature *F* abbilden. Wobei $Register_{Nicht_I}$ Wörter enthält, von denen vermutet wird, dass sie nicht mit *F* vereinbar sind. Die Auftrittshäufigkeit gibt Aufschluss darüber, welchen Stellenwert das jeweilige Wort in dem Register hat.

In *Zeile 8* werden aus *I* alle Elemente *e* ermittelt, welche in das $Register_I$ aufgenommen werden sollen. In der aktuellen Implementierung verbleiben dann lediglich Schlüssel-

Elemente vom Typ *Lokale Variable*, *Feld*, *Methode* sowie *Typ*. Da jedes solche Deklarations-Element nur einmal im Quellcode auftreten darf, kann vereinfacht angenommen werden, dass jeder korrespondierende Bezeichner gleich bedeutsam für das Feature-Vokabular ist. Mit der zusätzlichen Berücksichtigung von zugehörigen Referenz-Elementen, welche den selben Bezeichner haben und mehrfach auftreten können, würde jenes Gleichgewicht verletzt. Aufgrund dessen werden alle Referenzen herausgefiltert. Für jedes Element *e* wird anschließend der Bezeichner in Teilzeichenketten zerlegt. In der Programmiersprache Java wird häufig das so genannte *CamelCaseFormat* verwendet, bei dem Teilzeichenketten durch einen Wechsel von Groß- und Kleinschreibung getrennt werden. Jene Eigenschaft wird hierbei ausgenutzt. Zudem erfolgt eine Zerlegung von Wörtern, die durch einen *Unterstrich* getrennt sind. Bei der erstmaligen Aufnahme einer Teilzeichenkette zu dem $Register_I$ wird als zugehörige absolute Auftrittshäufigkeit der Wert *1* hinterlegt. Bei jedem weiteren Auftreten wird dieser Wert *um 1* inkrementiert (*Zeile 9*). In den *Zeilen 11-13* wird analog das $Register_{Nicht_I}$ aus der Element-Menge *Nicht_I* erzeugt. *Abbildung 5.16 (a, b)* zeigt exemplarisch wie jene Register nach einer solchen Extraktion aussehen könnten.

Sind Teilzeichenketten in beiden Registern enthalten (*Zeile 17-18*), kann davon ausgegangen werden, dass es sich um unspezifische Füllwörter handelt. Jene verfälschen „die tatsächliche“ Wichtigkeit der einzelnen Teilzeichenketten. In einem Normalisierungsschritt werden daher in solchen Fällen beide Register „gegeneinander verrechnet“ (*Zeile 19-33*). Es wird hierbei zugrunde gelegt, dass $Register_I$ und $Register_{Nicht_I}$ gleichberechtigt sind, auch wenn sie unterschiedlich viele Elemente verwalten. Um die jeweiligen Wichtigkeiten einer Teilzeichenkette vergleichen zu können, werden sie als relative Werte $relWichtigkeit_I$ bzw. $relWichtigkeit_{Nicht_I}$ ausgedrückt (*Zeile 19-20*).

In dem Beispiel aus *Abbildung 5.16 (a,b)* lässt sich ablesen, dass die Teilzeichenkette DRUCK in beiden Registern gleich oft vertreten ist. Relativ gesehen ist sie aber für *I* mit einem Wert von $relWichtigkeit_I = 0{,}1$ weniger wichtig als für *Nicht_I* ($relWichtigkeit_{Nicht_I} = 0{,}2$). Bei der folgenden Normalisierung wird daher DRUCK aus dem $Register_I$ entfernt (*Zeile 22*). Im Gegenzug wird die Wichtigkeit von DRUCK um *0,1* in $Register_{Nicht_I}$ gemindert. Jenes entspricht einer absoluten Aufrufhäufigkeit von $0{,}1 * 5 = 0{,}5$. Abschließend verbleit daher ein finaler Wert von $1 - 0{,}5 = 0{,}5$ (*Zeile 23*). In dem Fall von FORMAT, $relWichtigkeit_I = 0{,}3$ und $relWichtigkeit_{Nicht_I} = 0{,}2$ erfolgt ein Ausgleich in umgekehrter Richtung (*Zeile 25*). Die Teilzeichenkette wird zunächst aus $Register_{Nicht_I}$ entfernt (*Zeile 26*). Im Anschluss wird in $Register_I$ der ursprüngliche Wert um $0{,}2 * 10 = 2$ gemindert, sodass ein finaler Wert von $3 - 2 = 1$ verbleibt (*Zeile 27*). Die Teilzeichenkette GIB ist sowohl in $Register_I$ als auch $Register_{Nicht_I}$ mit einer relativen Wichtigkeit von *0,2* vertreten und wird daher in der Normalisierung aus beiden Registern entfernt (*Zeile 30-31*). *Abbildung 5.16(c,d)* zeigt den Zustand der Register nach Abschluss der Normalisierung.

Für die Erzeugung von Feature-Element-Vorschlägen werden alle Lokalen Variablen, Felder, Methoden und Typen *s*, welche weder zu *I* noch zu *Nicht_I* gehören, hinsichtlich ihrer Übereinstimmung mit dem Feature-Vokabular untersucht (*Zeile 35-36*).

```
…
35      for (Element s : gibRegisterKompatibleElemente(SM.gibElemente()) {
36          if (s ∉ (I ∪ Nicht_I)) {
37              Menge<String> TZK = gibTeilZeichenketten(s)
38              double r = 0
39              for (String tzk : TZK) {
40                      double relWichtigkeit_I = Register_I.gib(k) /  gibSummeAusWerten(Register_I)
41                      double relWichtigkeit Nicht_I =  Register_Nicht_I.gib(k) /
42                                                      gibSummeAusWerten(Nicht_I_Register)
43                       r += (relWichtigkeit_I  - relWichtigkeit Nicht_I) * gibDaempfungsWert(tzk.laenge())
44              }
45              if ( r > 0)
46                  EV_x.fuegeHinzu(s, r)
47          }
48      }
49
50      return EV_x
51
52  }
```

Abbildung 5.15.b: Vorschlagsansatz – Textmuster-Erkennung

**Nach Extraktion**

a) **$Register_I$**

| Schlüssel | Wert | $relWichtigkeit_I$ |
|---|---|---|
| DRUCK | 1 | 0,1 |
| FORMAT | 3 | 0,3 |
| GIB | 2 | 0,2 |
| HTML | 4 | (0,4) |
| $Summe_I$ | 10 | (1,0) |

b) **$Register_{Nicht_I}$**

| Schlüssel | Wert | $relWichtigkeit_{Nicht_I}$ |
|---|---|---|
| DRUCK | 1 | 0,2 |
| FORMAT | 1 | 0,2 |
| GIB | 1 | 0,2 |
| PDF | 2 | (0,4) |
| $Summe_{Nicht_I}$ | 5 | (1,0) |

**Nach Normalisierung**

c)

| Schlüssel | Wert | $relWichtigkeit_I$ |
|---|---|---|
| | | |
| FORMAT | 1 | 0,2 |
| | | |
| HTML | 4 | 0,8 |
| $Summe_I$ | 5 | 1,0 |

d)

| Schlüssel | Wert | $relWichtigkeit_{Nicht_I}$ |
|---|---|---|
| DRUCK | 0,5 | 0,2 |
| | | |
| | | |
| PDF | 2 | 0,8 |
| $Summe_{Nicht_I}$ | 2,5 | 1,0 |

Abbildung 5.16: Beispiel-Register

Hierzu wird jeweils der Bezeichner von *s*, nach dem gleichen Schema wie in der Extraktion, in Teilzeichenketten *TZK* zerlegt (*Zeile 37)*. Zudem wird die Hilfsvariablen *r* für den Relevanzwert mit dem Wert *0* initialisiert (*Zeile 38*). Im Anschluss daran wird für jede Teilzeichenkette *tzk* aus *TZK die relative Wichtigkeit* für das Feature *F* ermittelt. Hierbei wird in dem jeweiligen, normalisierten Register die Aufrufhäufigkeit von *tzk* zu allen

Aufrufen ins Verhältnis gesetzt. Ist *tzk* in einem der Register nicht enthalten, so gilt $relWichtigkeit_I = 0$ bzw. $relWichtigkeit_{Nicht_I} = 0$ (*Zeile 40-42*). Es wird angenommen, dass sehr kurze Teilzeichenketten einen geringen Informationsgehalt (für das Feature) kodieren. Daher wird die relative Wichtigkeit, je nach Länge von *tzk,* gegebenenfalls gemindert. In der aktuellen Implementierung ist der Dämpfungsfaktor für Teilzeichenketten der Länge *1* gleich *0*, der Länge *2* gleich *0,33*, der Länge *3* gleich *0,67* und ab einer Länge von *4* gleich *1*. Ist *tzk* in $Register_I$ enthalten, so wird die resultierende relative Wichtigkeit zu *r* addiert. Ist sie in $Register_{Nicht_I}$ enthalten wird sie von *r* subtrahiert. Andernfalls bleibt *r* unverändert (*Zeile 43*).

In dem Beispiel aus *Abbildung 5.16* würde sich nach dem beschriebenen Algorithmus für einen Bezeichner, welcher aus den Teilzeichenketten „GIB, DRUCK, IM, PDF, FORMAT" besteht, ein Wert von $w = (0 * 0) + (-0,2 * 1) + (0 * 0) + (0,8 * 0,67) + (0,2 * 1) = 0,54$ ergeben.

Gibt es für das Element *s* eine Übereinstimmung mit dem Vokabular aus *I* bzw. gilt $r > 0$ (*Zeile 45*), so wird jenes Element mit dem zugehörigen *r* zu der Menge der Element-Vorschläge $EV_x$ hinzugefügt (*Zeile 46*).

### 5.2.3.4 Vorschlagsansatz: SPL-bewusstes Typsystem

In der Vollständigkeits-Definition von Feature-Repräsentationen wurde unter anderem festgelegt, dass die Entfernung eines Features keine Fehler in gültigen Varianten verursachen darf. Zu der Expansion gehört es daher auch bereits identifizierte Bereiche auf mögliche *Verhaltens-, Syntax- und Typ-Fehler* zu überprüfen und gegebenenfalls Lösungsstrategien vorzuschlagen (vgl. *Kapitel 3.1*).

*Verhaltensfehler* heißt: Eine Variante verhält sich abweichend zu einer definierten Spezifikation. Solche Fehlertypen lassen sich am schwierigsten identifizieren, da üblicherweise eine statische Analyse des Quellcodes hierfür nicht ausreichend ist [KA08; Käs10] (vgl. *Kapitel 3.1).* In LEADT werden Verhaltensfehler bislang nicht weiter adressiert.

*Syntax-Fehler* heißt: Eine Variante hat eine ungültige Form bezüglich zu der Programmiersprachen-*Syntax.* Bei der Behandlung von Syntax-Fehlern kann für jeden Fall eine eindeutige, automatisierte Lösungsstrategie verfolgt werden, wie beispielsweise: Füge den nächst größeren disziplinierten Bereich zu dem Feature hinzu. Hierbei genügt es den lokalen Kontext einer Klasse zu prüfen, die den Fehler verursacht [KA08; Käs10] (vgl. *Kapitel 3.1)*. Durch die Verwendung von CIDE als Dokumentations-Komponente wird sichergestellt, dass Syntax-Fehler in Feature-Repräsentationen erst gar nicht auftreten können. Aufgrund dessen ist es nicht erforderlich zusätzliche (Expansions-)Maßnahmen zu treffen, um diesen Fehlertyp zu behandeln (vgl. *Kapitel 5.1.1; 4.3).*

*Typ-Fehler* heißt: Eine Variante hat eine ungültige Form bezüglich zu dem Typsystem einer Programmiersprache. Das heißt beispielsweise, dass gültige Varianten existieren bei denen auf Variablen, Methoden, Felder oder Typen zugegriffen wird für die es keine Deklaration gibt [KA08; Käs10] (vgl. *Kapitel 3.1)*. In dem für die Feature-Dokumentation verwendeten CIDE ist für die Programmiersprache Java ein *SPL-bewusstes Typsystem* umgesetzt, welches dabei hilft auch solche Fehlertypen zu erkennen. Hierbei wird der Quellcode hinsichtlich „einiger zentraler" Regeln überprüft. Kommt es gegebenenfalls in einer gültigen Variante zu einem Widerspruch werden Quellcode-Fragmente hervorgehoben, welche eine erfolgreiche Kompilierung verhindern würden. Zudem werden einige Lösungsmöglichkeiten angeboten, um den Fehler zu beheben. In der Regel bedeutet jenes, dass Elemente vorgeschlagen werden, die ebenfalls zu dem Feature hinzugefügt werden müssen, damit es in keiner Variante mehr zu einem Widerspruch kommen kann (vgl. *Kapitel 4.2.5)*.

In LEADT wird die Funktionsweise des vorhandenen Typsystems in CIDE aus folgenden zwei Gründen in den *Expansions-Algorithmus* ein- bzw. umgegliedert:

- Im Unterschied zu Syntax-Fehlern lässt sich bei der Behandlung von Typfehlern nicht immer eine eindeutige, automatisierte Lösungsstrategie verfolgen. Es können Fälle auftreten, bei denen dem Anwender *mehrere gleichrangige Alternativen* vorgeschlagen werden „müssen", welche den Fehler beheben könnten. Jene Element-Vorschläge müssen sich nicht zwangsläufig auf den lokalen Kontext der Fehlerursache beschränken und können durchaus verstreut in dem gesamten Quelltext liegen. Das Vorschlagen von potentiellen Feature-Elementen, die von dem Anwender geprüft werden müssen, deckt sich konzeptionell gesehen in LEADT eher mit dem Vorgehen der Expansion als mit dem der Dokumentation. Die Ein- bzw. Umgliederung des Typsystems ermöglicht dem Anwender daher ein einheitliches Vorgehen und Bedienkonzept für die Feature-Expansion.

- Einzelne Feature-Element-Vorschläge des Typsystems können durch andere Vorschlagsansätze zusätzlich gestützt werden. Hierdurch kann unter Umständen die Auswahl zwischen (ursprünglich) gleichrangigen Alternativen erleichtert werden. Jener Aspekt wird in *Kapitel 5.2.3.5* noch umfangreicher diskutiert.

Es sei erinnert, dass auch die *Prune Dependency Analysis* in dem Werkzeug *Cerberus* eine ähnliche Idee zur Expansion verfolgt. Sie fügt ein Quellcodefragment einem Belang genau dann hinzu, wenn es geändert oder entfernt werden muss, sobald der gesamte Belang entfernt wird. Im Unterschied zu dem im Folgenden adaptierten Typsystem von CIDE, berücksichtigt Cerberus nicht das Feature-Modell für dessen „Entfernungsregeln". Die Prune Dependency Analysis fügt beispielsweise *immer* eine Referenz zu einem Belang hinzu, wenn die Deklaration Teil des Belangs ist (vgl. *Kapitel 4.2.5)*. Aufgrund von Feature-Abhängigkeiten ist jenes aber nicht immer erforderlich bzw. auch nicht richtig. Es

kann durchaus Fälle geben, in denen Deklaration und Referenz unterschiedlichen Features zu geordnet sind, aber dennoch in keiner gültigen Variante Typfehler entstehen. Wie bereits in *Kapitel 4.2.5* oder *Kapitel 5.2.3.2* erwähnt, benötigt Cerberus zudem ein System-Modell, welches für das Feature Mining zu grobgranular ist.

**Berechnung der Relevanzwerte**

In *Abbildung 5.17* ist die Implementierung der Methode zur Generierung von Element-Vorschlägen mit Hilfe des SPL-bewussten Typsystem Ansatzes schematisch dargestellt. In dieser werden Feature-Element-Kandidaten empfohlen, die ebenfalls zu dem Feature hinzugefügt werden müssen, damit es in keiner Variante zu einem Typ-Fehler kommen kann. Der zugehörige Relevanzwert entspricht jeweils dem Kehrwert der Anzahl aller gleichwahrscheinlichen Kandidaten.

```
1   Menge<Element-Vorschlag> generiereElementVorschlaege(Feature F, Menge<Element>  I,
2                                           Menge<Element> Nicht_I, System-Modell SM, Feature-Modell FM)) {
3
4       Menge<Element-Vorschlag>  EVx = new Menge<Element-Vorschlag>()
5
6       for (Element e : I) {
7           for (Pruefregel p : gibTypSystemPruefregeln(e, SM, FM)) {
8               if (p.hatWiderspruch(F)) {
9                   Menge<Element> S = p.gibLoesungsKandidaten(I, Nicht_I)
10                  If (|S| > 0) {
11                      double r = 1 / |S|
12                      for (Element s : S))  {
13                          EVx.fuegeHinzu(s, r)
14                      }
15                  }
16                  else  {
17                      informiereAnwender(p.gibFehlerMeldung())
18                  }
19              }
20          }
21      }
22
23      return EVx
24
25  }
```

Abbildung 5.17: Vorschlagsansatz – Typsystem

Initial wird eine Variable $E_x$ angelegt, welche die Element-Vorschläge aufnehmen soll, um sie dann an den Hauptalgorithmus zurück zugeben (*Zeile 4, 23*).

Für jedes Element *e* aus *I* werden definierte, elementtyp-spezifische Typsystem-Prüfregeln erzeugt, welche als Eingaben das Element *e*, das Feature F, das System-Modell *SM* sowie

das Feature-Modell *FM* benötigen (*Zeile 6-7*). Anschließend wird anhand jeder Regel *p* geprüft, ob die Zugehörigkeit von *e* zu F (und anderen Features) Kompilierungsfehler in irgendeiner gültigen Variante des Systems verursacht (*Zeile 8*).

Treten Typ-Fehler auf, werden aus der selben Prüfregel in *Zeile 9* mögliche, gleichrangige Lösungsalternativen *S* abgefragt. Das heißt solche Elemente *s*, die zu dem Feature *F* aufgenommen werden können bzw. müssen damit die Prüfung der Regel widerspruchsfrei erfolgen kann.[5] Ein Lösungskandidat darf weder in *I* noch in *Nicht_I* enthalten sein. Es ist vereinbart, dass jede Prüfregel typischerweise mindestens ein Element vorschlagen muss (*Zeile 10*). Sollte jenes für bestimmte Regeltypen nicht möglich sein, muss der Anwender auf geeignete Art und Weise informiert werden, dass andere Maßnahmen erforderlich sind, wie z.B. das Umschreiben des Quellcodes (*Zeile 17*).

*S* kann mehrere Elemente enthalten, die für die Behebung des Typ-Fehlers alle gleichrelevant bzw. gleichwahrscheinlich sind. Der Relevanzwert *r* definiert sich daher für jedes *s* als Kehrwert der Anzahl aller möglichen Lösungskandidaten (*Zeile 11*). Abschließend wird jeder Element-Vorschlag *s* mit seinem zugehörigen Relevanzwert *r* in $EV_x$ aufgenommen (*Zeile 13*).

**Umgesetzte Typsystem-Prüfregeln**

Für den aktuellen LEADT Prototypen wurden aufgrund des hohen Entwicklungsaufwandes nicht alle verfügbaren Regeln aus CIDE überführt. Nach einer Priorisierung wurden in erster Annäherung folgende zwei Prüfregeln umgesetzt:

- Für *Deklarationen von Feldern, Imports, Lokale Variablen, Methoden und Typen*, die in dem System-Modell erfasst sind (vgl. *Kapitel 5.2.1*), werden über die Umkehrung der „referenziert"-Beziehung alle zugehörigen Referenzen ermittelt. Für jedes Paar von Deklaration und Referenz wird die so genannte *Referenz-Prüfregel* erzeugt. Jene analysiert, ob die Deklaration in allen Varianten enthalten ist in denen auch die Referenz enthalten ist. Kommt es hierbei zu einem Widerspruch, weil die Referenz nicht zu Feature *F* gehört - die Deklaration aber schon, so wird die Aufnahme der Referenz zu *F* empfohlen.

- Für *Parameter Referenzen*, sprich die Wertübergabebereiche in Methoden Referenzen (vgl. *Kapitel 5.2.1*), wird die *Aufruf-Prüfregel* erzeugt. In dieser wird analysiert, ob die Signatur des Aufrufs in allen Varianten mit der zugehörigen Deklaration

[5] Es kann Fälle geben in denen s nicht nur zu F sondern noch zu anderen Features hinzugefügt werden muss, damit der Typ-Fehler nicht mehr auftritt. Dennoch wird hier von Lösungsalternativen gesprochen. Die Zuordnungen zu den verbleibenden Features werden über dessen Expansionen sichergestellt.

übereinstimmt.[6] Kommt es hierbei zu einem Widerspruch, weil die Parameter Referenz zu Feature *F* gehört, so werden (wenn möglich) zwei gleichrangige Lösungsalternativen empfohlen. Eine davon ist immer die Aufnahme der übergeordneten Methoden Referenz zu *F*. Liegt die zugehörige Variablen- bzw. Parameter Deklaration nicht „außerhalb" des analysierten Projekt-Quellcodes, wird auch deren Aufnahme zu Feature *F* empfohlen.

Um die Zusammenhänge in den unterschiedlichen Varianten prüfen zu können, wird basierend auf dem Feature-Modell bzw. dessen Repräsentation als aussagenlogischer Ausdruck ein Erfüllbarkeitsproblem formuliert. Dieses wird schrittweise mit der bereits in *Kapitel 5.2.2* erwähnten Bibliothek SAT4J gelöst. Das grundsätzliche Vorgehen hierbei wurde aus dem SPL-bewussten Typsystem von CIDE übernommen, welches ausführlich in der Dissertation von Kästner [Käs10] erläutert wird.

### 5.2.3.5 Vorteile der Vorschlagsansatz-Kombination

Die drei beschriebenen Vorschlagsansätze bzw. Teile von ihnen wurden bislang in Werkzeugen separat verwendet (vgl. *Kapitel 5.2.3.2; 5.2.3.3; 5.2.3.4*). Durch ihre Kombination in LEADT werden allerdings zum einen *Feature-Repräsentationen besser abstrahiert* und zum anderen *zuverlässigere und aussagekräftigere Vorschläge* generiert. Beide Eigenschaften führen zu einer Erhöhung der Vollständigkeit bei der Feature-Expansion. Im Folgenden werden jene Verbesserungen näher besprochen und veranschaulicht.

#### Bessere Abstraktion von Feature-Repräsentationen

Jeder der drei Ansätze hat einen anderen Blickwinkel auf den Quellcode bzw. die identifizierten Repräsentationen. Es lassen sich typische Szenarien angeben, in denen ein Ansatz immer deutlich bessere Leistungen erzielen kann als die jeweils beiden anderen. Eine Kombination dieser komplementären Stärken hat zur Folge, dass *Feature-Repräsentationen besser abstrahiert und erfasst werden können*. In *Abbildung 5.18* ist ein Ausschnitt aus dem System-Modell dargestellt, der jene Eigenschaft veranschaulicht.

---

[6] Werden Parameter Referenzen in Methodenaufrufen annotiert, verändert sich in einigen Varianten die Signatur. Hierdurch kann es passieren, dass jene Aufrufe auf einmal mit anderen, unerwarteten Methodensignaturen übereinstimmen. In dem Fall würden demnach auch keine Typ-Fehler auftreten. In der aktuellen Implementierung in LEADT wird dieser Fall nicht beachtet. Es erfolgt immer nur eine Prüfung der Beziehung von Aufruf und Deklaration, wie sie in der nicht annotierten Form des Systems besteht.

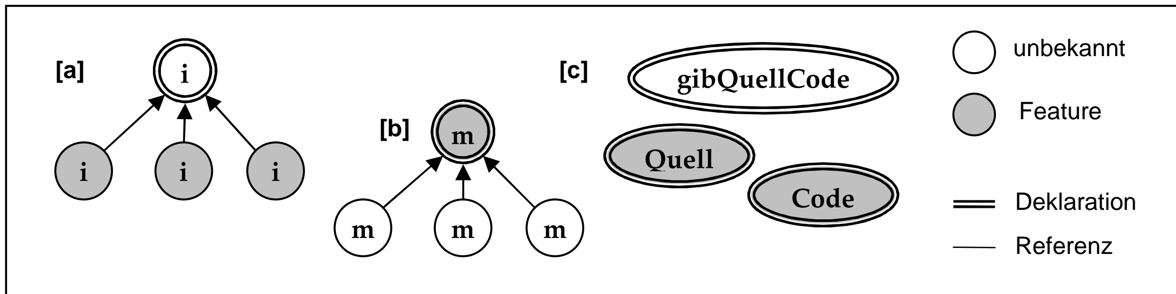



Abbildung 5.18: Beispiel – Bessere Abstraktion von Feature-Repräsentationen

In der *Abbildung 5.18* lässt sich erkennen, dass während der Feature-Expansion bislang die drei Referenzen zu *i*, die Deklaration *m* sowie die Deklarationen *Quell* und *Code* annotiert wurden. Die ebenfalls zu dem Feature zugehörige Deklaration *i*, die Referenzen zu *m* sowie die Deklaration *gibQuellCode* sind noch nicht identifiziert wurden.

## Teilausschnitt a

Die *Textmuster-Erkennung* enthält in den Registern nur Deklarationen. Referenz-Bezeichner werden nicht verwaltet. Somit wird hierbei die ebenfalls zu dem Feature zugehörige Deklaration *i* nicht vorgeschlagen. Selbst wenn Referenzen erfasst würden, wäre in diesem Fall der Informationsgehalt mit nur einem Zeichen sehr gering, sodass die Relevanzwerte sehr niedrig wären (vgl. *Kapitel 5.2.3.3*).

Unter der Annahme, dass die Deklaration *i* zu keinen anderen Features gehört, treten in diesem Fall keine Typfehler auf. Das *SPL-bewusste Typsystem* würde demzufolge ebenfalls nicht die zugehörige Deklaration *i* erkennen (vgl. *Kapitel 5.2.3.4*).

Im Gegensatz hierzu ist die *Spezitäts- und Bestärkungsheuristik* in der Lage in jedem Geflecht von Beziehungen eine Wahrscheinlichkeit für die Feature-Zugehörigkeit von direkten Schlüssel-Element-Nachbarn zu schätzen. Mit jenem Ansatz und den aktuell berücksichtigten Beziehungstypen wird sichergestellt, dass ungenutzter bzw. mit großer Sicherheit zu einem Feature gehörender Quellcode in vielen Fällen erkannt wird. Sind beispielsweise, so wie in diesem Fall, alle Referenzen *i* Teil des Features, so wird auch die zugehörige Deklaration *i* mit einem maximalen Relevanzwert von *1,0* empfohlen. Im Fall der „enthält"-Beziehung wird immer das übergeordnete Schlüssel-Element vorgeschlagen, wenn alle dessen untergeordneten Elemente Teil des Features sind. Verwendet eine Deklaration nur Feature-Elemente, so wird auch sie selbst für das entsprechende Feature empfohlen (vgl. *Kapitel 5.2.3.2*).

## Teilausschnitt b

Für die *Textmuster-Erkennung* trifft hier der gleiche Sachverhalt wie in dem *Teilausschnitt a* zu.

Die *Spezifitäts- und Bestärkungsheuristik* behandelt alle Beziehungstypen gleich. Der errechnete Relevanzwert bezieht sich nur auf die Lage bzw. den Grad der Vernetzung innerhalb des System-Modells. Die Bedeutung der Beziehungen wird nicht berücksichtigt. In diesem Fall wird jede der drei Referenzen von *m* mit einem Relevanzwert von *0,33* vorgeschlagen. Bei beispielsweise *100* zugehörigen, „unbekannten" Referenzen würde jedes Element einen Wert von nur noch *0,01* erhalten. Ein Anwender könnte Vorschläge mit einem derart geringen Wert leicht übersehen (vgl. *Kapitel 5.2.3.2*).

Werden die Referenzen *m* nicht zu dem Feature hinzugefügt, so können in Varianten, in denen die Deklaration *m* nicht enthalten ist, allerdings Typfehler auftreten. Die *Referenz-Prüfregel* aus dem *SPL-bewussten Typsystem* führt in diesem Fall daher zu einem Widerspruch. Jenes hat zur Folge, dass alle Referenzen *m* mit dem maximalen Relevanzwert von 1,0 vorgeschlagen werden (vgl. *Kapitel 5.2.3.4*).

### Teilausschnitt c

Die hier implementierte *Spezifitäts- und Bestärkungsheuristik* basiert auf der „enthält"-, „referenziert"- sowie „verwendet"-Beziehung zwischen direkten Nachbarn. Textuelle Zusammenhänge können nicht (sinnvoll) verarbeitet werden (vgl. *Kapitel 5.2.1; 5.2.3.3*). Aufgrund dessen wird mit diesem Vorschlagsansatz die Deklaration *gibQuellCode* nicht erkannt.

In diesem Teilausschnitt treten für die definierten Typsystem-Prüfregeln keine Widersprüche auf. Das *SPL-bewusste Typsystem* erkennt demzufolge ebenfalls nicht die Deklaration *gibQuellCode* (vgl. *Kapitel 5.2.3.4*).

Die *Textmuster-Erkennung* zielt darauf ab, anhand von bereits annotierten Repräsentationen feature-relevante Teilzeichenketten zu identifizieren (vgl. *Kapitel 5.2.3.3*). In diesem Fall sind es CODE, M und QUELL. Da jene gleichhäufig vertreten sind, wird angenommen, dass sie mit einem Wert von *0,33* auch gleich wichtig in dem Feature-Vokabular sind. Da in der Deklaration *gibQuellCode* zwei solche feature-relevante Teilzeichenketten enthalten sind, wird jenes Element mit einem Relevanzwert von *0,66* vorgeschlagen.

### Aussagekräftigere und zuverlässigere Vorschlagsgenerierung

Verlässt man sich auf jeden Vorschlagsansatz allein, kann es zu Fällen kommen in denen Feature-Element-Vorschläge von dem Anwender nicht berücksichtigt werden, da die errechneten Relevanzwerte zu gering sind. Durch die Kombination der unabhängigen Ansätze erhöht sich allerdings die Informationsmenge, sodass *ein Element aussagekräftiger und zuverlässiger bewertet werden kann*. In *Abbildung 5.19* ist ein Ausschnitt aus dem System-Modell dargestellt, der jene Eigenschaft veranschaulicht.

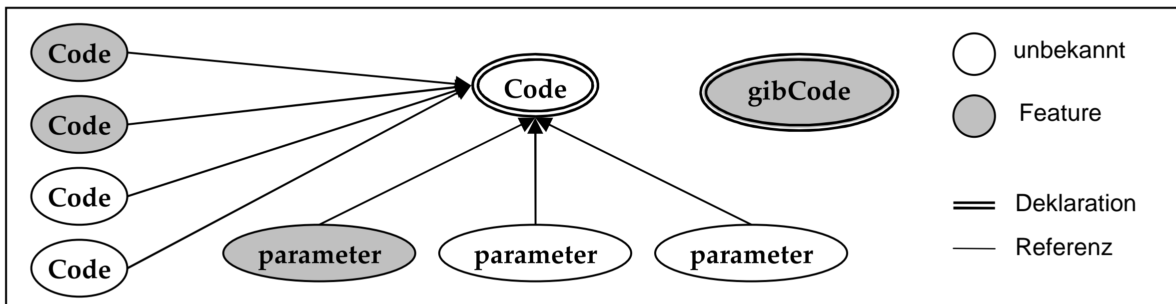



Abbildung 5.19: Beispiel – Aussagekräftigere, zuverlässigere Vorschlagsgenerierung

In der *Abbildung 5.19* lässt sich erkennen, dass von dem Methoden-Parameter *Code* zwei von vier Referenzen Teil des Features sind. Zudem ist in einem von drei Methodenaufrufen die zugehörige Parameter-Referenz annotiert. Die Deklaration selbst, welche auch zu dem Feature gehört, wurde bislang noch nicht identifiziert. Die annotierte Repräsentation umfasst darüber hinaus die Deklaration *gibCode*.

Jeder der drei Vorschlagsansätze empfiehlt die Deklaration *Code* jeweils mit einem Relevanzwert von 0,5:

- Die *Spezitäts- und Bestärkungsheuristik* generiert anhand der „referenziert"-Beziehung zu den Parameter-Referenzen ein Relevanzwert von *0,3*. Dieser wird aber durch die Beziehung zu den Code-Referenzen innerhalb der Methode dominiert (vgl. *Kapitel 5.2.3.2*).

- Die *Textmuster-Erkennung* errechnet den Relevanzwert von *0,5* aufgrund der Übereinstimmung der Teilzeichenkette CODE mit dem bislang bekannten Feature-Vokabular, welches aus CODE und GIB besteht (vgl. *Kapitel 5.2.3.2*).

- Die annotierte Parameter-Referenz verändert die Signatur des Methoden-Aufrufs. Hierdurch kommt es zu einem Widerspruch der Aufruf-Prüfregel in dem *SPL-bewussten Typsystem.* Eine der beiden Lösungsalternativen ist die Aufnahme der *Code* Deklaration zu dem Feature (vgl. *Kapitel 5.2.3.4*).

Aus der Perspektive jedes einzelnen Vorschlages gehört *Code* mit einer Wahrscheinlichkeit von *50%* zu dem Feature. Über das in *Kapitel 5.2.3.1* beschriebene Vereinigungsschema der Ansätze ergibt sich allerdings ein genauerer Relevanzwert von *0,88* bzw. eine Zugehörigkeitswahrscheinlichkeit von *88%*. Jene Erhöhungen des Wertes können entscheidend sein, um (korrekte) Empfehlung näher in den Fokus des Anwenders zu rücken.

## 5.3 Zusammenfassung

Die Analyse des Stands der Technik hat gezeigt, dass bislang keine Konzepte bzw. Werkzeuge existieren, die den gesamten Prozess des Feature Mining strukturiert und integriert unterstützen können. Vor diesem Hintergrund wurde in diesem Kapitel mit dem *Location, Expansion and Documentation Tool* eine Lösung vorgeschlagen, die jene Lücke schließt.

In dem ersten Hauptteil des Kapitels wurden die Funktionsweise und die Bestandteile des gesamten LEADT Konzepts vorgestellt. Hierbei wurde vor allem deutlich gemacht, welche Aktivitäten des Feature Mininig durch den Anwender zu erfolgen haben und welche durch Werkzeuge automatisiert werden. Für Aufgabenbereiche im Kontext der Feature-Lokalisierung und -Dokumentation wurde hierbei auf die Werkzeuge FLAT³ und CIDE zurückgegriffen, welche von anderen Forschern entwickelt wurden. Für die Feature-Expansion wurde ein eigens entwickeltes Konzept eingebracht. In dem aktuellen LEADT Prototyp für die Entwicklungsumgebung Eclipse sind die Grenzen der einzelnen Werkzeuge nicht mehr eindeutig sichtbar. Die Konzepte präsentieren sich dem Anwender integriert in einer einheitlichen Nutzerschnittstelle. Im Rahmen dieses Kapitels wurde zudem jene Schnittstelle als Screenshot abgebildet und überblicksartig beschrieben.

Das vollständigkeitsfokussierte, automatisierte Expansionskonzept ist eine wesentliche Neu-Entwicklung innerhalb von LEADT. Im zweiten Hauptteil dieses Kapitels wurde daher umfassend dessen konkrete Umsetzung dargestellt.

Das Kernziel der Expansionsunterstützung in LEADT besteht darin den manuellen Suchaufwand für die vollständige Lokalisierung zu reduzieren. Jenes soll erreicht werden indem basierend auf bereits identifizierten Feature-Elementen über einen Algorithmus weitere (mögliche) relevante Element-Vorschläge generiert werden. Der Anwender soll demnach nicht nur seiner Erfahrung und Intuition überlassen, sondern zielgerichtet und effektiv geführt werden.

Der Algorithmus benötigt neben den Datenspeichern, welche die Feature-Elemente bzw. die Nicht-Feature-Elemente enthalten, noch zwei weitere wesentliche Eingaben – zum einen das System-Modell und zum anderen spezifische Feature-Abhängigkeiten. Das System-Modell abstrahiert Schlüssel-Elemente und -Beziehungen im Quellcode in einer leichter handhabbaren Repräsentationsform. Aus der Literaturrecherche konnte kein frei verfügbares Konzept identifiziert werden, welches ohne Abstriche für LEADT geeignet ist. Aufgrund dessen wurden ein eigenes System-Modell und ein zugehöriger Extraktor entwickelt. Die Feature-Abhängigkeiten umfassen Notwendigkeits- und Ausschlussbeziehungen in dem Feature-Modell, welche Rückschlüsse auf Repräsentationen im Quellcode geben können. Nach dem Kenntnisstand des Autors ist nicht bekannt, dass andere Expansionstechniken existieren, die vergleichbare Domänen-Informationen in ihrem Vorgehen

berücksichtigen. In diesem Kapitel wurde umfassend erläutert, wie jene beiden Eingaben konkret aufgebaut sind und auf welche Art und Weise sie erzeugt bzw. verwaltet werden.

Um eine möglichst hohe Vollständigkeit der Feature-Repräsentationen zu erzielen kombiniert der Expansions-Algorithmus drei Vorschlagsansätze – eine adaptierte Spezifitäts- und Bestärkungsheuristik, eine Textmuster-Erkennung sowie ein SPL-bewusstes Typsystem. Teile dieser drei Ansätze wurden bislang in unterschiedlichen Werkzeugen separat verwendet. In LEADT werden sie allerdings aufgrund ihrer komplementären Eigenschaften gebündelt und in bestimmten Details für das Feature Mining angepasst bzw. verbessert. In diesem Kapitel wurde abschließend gezeigt, wie jene Vorschlagsansätze in dem aktuellen Prototyp umgesetzt sind, wie sie mit einander vereint werden und welche Vorteile sich durch ihre Kombination ergeben.

# 6 Evaluation

Im vorherigen *5. Kapitel* wurde der Prototyp *LEADT* vorgeschlagen, welcher alle Schritte des Feature Mining integriert unterstützt. Dieses Kapitel zielt nun darauf ab exemplarisch die Leistungsfähigkeit dieses Werkzeuges zu prüfen.

Basierend auf einer kurzen Diskussion von grundsätzlich in Frage kommenden Evaluationsmethoden wird sich auf einen konkreten Ansatz festgelegt. Darauf anknüpfend wird der Untersuchungsaufbau detailliert erläutert. In dem Hauptteil des Kapitels erfolgt die Auswertung der gemessenen Ergebnisse. Hierzu werden insgesamt vier unterschiedliche Bewertungsaspekte betrachtet. Ein kurzes Fazit fasst die wesentlichen Erkenntnisse zusammen. Jene werden darüber hinaus hinsichtlich ihrer Wiederholbarkeit und Generalisierbarkeit kritisch gewürdigt. Dieses Kapitel schließt mit einer überblicksartigen Zusammenfassung.

## 6.1 Evaluationsmethode

Für die Evaluation von Feature Mining Konzepten und Werkzeugen gibt es bis dato keine festgelegten Standards und Methoden. Im Rahmen dieser Arbeit steht die Definition jener auch nicht im Fokus. Damit dennoch eine erste Aussage zur Tauglichkeit von LEADT für das Feature Mining gemacht werden kann, wird im Folgenden auf Erfahrungen aus den verwandten Gebieten des Reverse Engineering und Program Understanding zurückgegriffen. In der Regel werden dort drei unterschiedliche Hauptkonzepte für eine Werkzeug-Evaluation verwendet – *quantitative* und *qualitative* Methoden sowie *Fallstudien* [MJS+00; Sto05; WP03]. Im Folgenden wird kurz auf die unterschiedlichen Arten eingegangen.

Der erklärende, *qualitative* Ansatz zeichnet sich durch große Offenheit und Flexibilität aus. Standardisierte Vorgaben spielen eine untergeordnete Rolle. Zu dieser Gruppe können unter anderem *Ausführungs- oder Bewertungsprotokolle* von *Anwendern* und *Experten* eingeordnet werden. Bei der Nutzung des Werkzeuges sowie der Erstellung der Protokolle erfolgt häufig eine direkte Interaktion bzw. Diskussion zwischen Probanden und dem Experimentator. Hierdurch ist es möglich umfassende Informationen zu erhalten und daraus neue Sachverhalte, Muster oder Theorien zu erschließen. Diese Art der Informati-

onsgewinnung ist sehr effektiv. Sie hängt allerdings stark von unkontrollierten Variablen, wie beispielsweise der Qualifikation der Probanden, ab und erlaubt aufgrund der subjektiven, kontextabhängigen Sicht keinen allgemeinen Schluss [MJS+00; Sto05; WP03].

Der *quantitative* Ansatz zielt auf die exakte, objektive Messung und Quantifizierung von Sachverhalten ab. Basierend auf dem bestehenden theoretischen Wissen über den Untersuchungsgegenstand werden Hypothesen formuliert, wie z.B.: „Das Werkzeug A hilft dabei den zeitlichen Aufwand signifikant zu reduzieren". Im Rahmen der Untersuchungen werden diese dann (statistisch) getestet bzw. überprüft. Die hierfür nötige Datenbasis wird durch eine *standardisierte Befragung* oder *Beobachtung* einer möglichst großen und repräsentativen Stichprobe in *formalen Experimenten* gewonnen. Für die Einhaltung der Objektivität, Reliabilität sowie Validität der Ergebnisse ist es dabei zwingend erforderlich, alle abhängigen und unabhängigen Variablen zu kontrollieren. Der unflexible Rahmen von quantitativen Ansätzen verhindert ein individuelles Eingehen auf die Testpersonen. Allerdings lässt sich mit Hilfe der Ergebnisse ein Schluss auf die Grundgesamtheit ziehen, sodass gewisse Beobachtungen und Effekte auch unter Feldbedingungen vorhersehbar werden [Fei09; FKAL09; MJS+00; Sto05; WP03].

Im Bereich des Reverse Engineering und Program Understanding erfreuen sich zudem *Fallstudien* großer Beliebtheit. In diesen wird das zu untersuchende Werkzeug in *einem* spezifischen Fall oder auf spezifischen Systemen angewendet, wobei der Experimentator sein Vorgehen detailliert, introspektiv dokumentiert. Fallstudien weisen Merkmale sowohl von quantitativen als auch qualitativen Methoden auf. Auf der einen Seite bemühen sie sich um eine objektive und kontextfreie Sicht, sodass die Studien reliabel und wiederholbar sind. Bei der Wahl eines sinnvollen und repräsentativen Falls können sogar durchaus auch allgemeine Schlüsse gezogen werden, obwohl nur eine einzige spezifische Situation betrachtet wird. Auf der anderen Seite lassen sie aber Raum, um Erkenntnisse in einem iterativen Lernprozess zu gewinnen und daraus beispielsweise Hypothesen abzuleiten. Fallstudien sind insbesondere dann geeignet, wenn der Experimentator nur minimalen Einfluss auf die zu untersuchenden Faktoren hat [Fly06; MJS+00; Sto05].

LEADT ist momentan ein Forschungsprototyp, der wesentliche Kernideen implementiert. Aspekte der Nutzerfreundlichkeit und Bedienbarkeit standen während der Entwicklung bislang nicht in dem Vordergrund, sodass konkrete Messungen der Komplexitäts- und Aufwandsreduktion mit Experten und Anwendern in Feldversuchen oder unter formalen Laborbedingungen noch stark verfälscht wären. Qualitative und quantitative Evaluationsmethoden stellen daher eher eine Option für mögliche spätere Stadien dar. In dieser Arbeit liegt der Fokus darin exemplarisch zu demonstrieren, dass mit diesem Ansatz ein Feature Mining überhaupt realisiert werden kann. Hierfür bietet sich vor allem eine Fallstudie an. Aufgrund dessen wird jene Evaluationsmethode auch im Folgenden angewendet.

## 6.2 Untersuchungsaufbau

Feature Mining Werkzeuge müssen das Ziel verfolgen, im Vergleich zu dem manuellen Vorgehen, die Qualität bzw. Vollständigkeit der Plattform zu erhöhen und den Transitionsaufwand zu reduzieren. LEADT unterstützt ein *semiautomatisches* Vorgehen (vgl. *Kapitel 3.1; 5.1.1*). Inwieweit es in der Lage ist die geforderte Zielsetzung zu erreichen lässt sich daher final nur mit Experimenten beantworten, in denen Probanden zum einen mit und zum anderen ohne die Werkzeugunterstützung eine Transitionsaufgabe bearbeiten. Auf solche Ergebnisse kann aber aktuell nicht zurückgegriffen werden (vgl. *Kapitel 6.1*).

In dieser Arbeit wird die Tauglichkeit von LEADT für das Feature Mining anhand einer Fallstudie *indirekt* demonstriert, indem das aktuelle Potential *des automatisierten Teils* herausgearbeitet wird. Hierbei wird grundsätzlich angenommen, dass der manuelle Anteil der Arbeit abnehmen wird, wenn Teile der Aufgabe mit einer hohen Qualität automatisiert bearbeitet werden können. Dieses Kapitel beschreibt den Untersuchungsaufbau, mit dem eine entsprechende Evaluation umgesetzt werden soll. Hierbei wird insbesondere festgelegt wie ein geeignetes Feature Mining Vorgehen zu erfolgen hat und auf welchem System es angewendet wird.

### 6.2.1 Grundsätzliches Vorgehen

In dieser Fallstudie wird im Wesentlichen die Idee verfolgt mit Hilfe von LEADT, korrespondierend zu einem Feature-Modell, ein (Alt-)System in eine annotationsbasierte Software-Produktlinienplattform zu überführen und die hierbei erzielten Ergebnisse zu bewerten. Damit sichergestellt werden kann, dass tatsächlich Rückschlüsse auf die Fähigkeit des *automatisierten Teils des Prototypen* möglich sind, muss das Feature Mining Vorgehen reglementiert werden. Konkret bedeutet dieses, dass der Anteil an Informationen, die ein Anwender durch Wissen, Erfahrung oder Intuition mit einbringen kann, möglichst gering gehalten werden muss. Aufgrund dessen wird folgendes Vorgehen festgelegt:

1. *Schritt – Festlegen der Feature-Kategorien*

   Aus dem zugehörigen Feature-Modell werden zunächst nach dem in *Kapitel 5.2.2* beschriebenen Schema alle Obermengen- und Teilmengen-Features ermittelt

2. *Schritt – Feature-Lokalisierung*

   *a:* Für jedes *Teilmengen-Feature* wird ein Startfragment lokalisiert. Hierbei wird der Feature-Name als Anfragetext für die statische IR-Suche genutzt. Das Element mit dem höchsten ermittelten Relevanzwert in der Trefferliste, welches zu dem Feature gehört, bildet den Ausgangspunkt für die anschließende Expansion (siehe

weiter 3. Schritt). Wurden alle Teilmengen-Features betracht, folgen die Obermengen-Features (siehe weiter b).

*b:* An dieser Stelle wird der Effekt genutzt, dass bereits lokalisierte Teilmengen-Features mögliche Vorschläge zu der Existenz von *Obermengen-Feature-Elementen* geben können. Für die weitere Expansion wird daher das Element mit dem höchsten Relevanzwert aus der zugehörigen Vorschlagsliste gewählt. Eine IR-Suche ist in diesem Fall nicht notwendig (siehe weiter 3. Schritt). Wurden auch alle Obermengen-Features berücksichtigt, gilt das Feature Mining als beendet.

3. *Schritt – Feature-Expansion; Feature-Dokumentation*

   Während der Expansion ermittelt der Vorschlagsalgorithmus weitere Elemente, die ebenfalls zu dem Feature gehören könnten und sortiert sie absteigend nach der errechneten Relevanz. Iterativ wird dann das jeweils oberste Element näher betrachtet. Gehört das Element (bzw. dessen nächst größere disziplinierte Einheit) zu dem Feature, wird es mit Hilfe der Dokumentationskomponente annotiert. Führt der Vorschlag nicht zu der Feature-Repräsentation, so muss das Element zu der Nicht-Feature-Repräsentation zugeordnet werden. Als Entscheidungsgrundlage ist hierfür eine bereits annotierte Version des Systems zu nutzen. Basierend auf den neuen Informationen wird die Vorschlagsliste aktualisiert; der 3. Schritt beginnt von vorn. Sind *zehn* nacheinander folgende Element-Vorschläge nicht korrekt, so wird die Expansion für dieses Feature beendet.[1] Es wird dann mit dem 2. Schritt fortgefahren.

Es sei betont, dass in diesem Vorgehen zur Ermittlung von potentiellen zugehörigen Feature-Elementen tatsächlich nur generierte Vorschläge von LEADT berücksichtigt werden. Aufgrund dessen wird beispielsweise auf dynamische Lokalisierungstechniken verzichtet, da ein Anwender durch seine formulierten Ausführungsszenarien indirekt Element-Kandidaten festlegt. Die IR-Suchanfragen basieren hingegen auf einem automatisierbaren Schema. Um die von „außen" zugeführte Informationsmenge zusätzlich gering zu halten, wird aus der Feature-Lokalisierung nur *ein* Startelement für *Teilmengen-Features* bzw. *gar keins* für *Obermengen-Features* verwendet. Während der Expansion darf nur das korrekt vorgeschlagene Element annotiert werden bzw. der nächst größere disziplinierte Bereich, den die Dokumentationskomponente erlaubt – selbst wenn ein Anwender intuitiv auch in dem gleichen Schritt noch benachbarte zugehörige Elemente erkennen würde.

Die Bewertung, ob ein Element-Vorschlag korrekt ist oder nicht, muss aber letzten Endes von einem Anwender bzw. in diesem Fall von dem Experimentator erfolgen. Im Rahmen

[1] Es wurde festgestellt, dass Nutzer selten mehr als zehn Empfehlungen überprüfen [RP09; ZZWD05]. Daher wurde an dieser Stelle jene Zahl als Grenze festgelegt.

dieser Fallstudie wird als Entscheidungsgrundlage hierfür ein bereits (von anderen) annotiertes Original-System verwendet, sodass auch in diesem Fall die Objektivität der Ergebnisse gewahrt bleibt. In dem folgenden Kapitel wird die Wahl des (Test-)Systems näher besprochen.

### 6.2.2 Wahl des Systems

Eine Bewertung der Korrektheit der einzelnen Vorschläge sowie des erzielten Vollständigkeitsgrades ist in der Regel nur dann sinnvoll, wenn auch die tatsächlichen Feature-Repräsentationen des Systems bekannt sind. Natürlich wäre es möglich ein System eigenhändig zu annotieren und hinterher mit LEADT die Gegenprobe zu machen. Allerdings bleibt in diesem Fall die Frage offen, ob die Vergleichmarkierungen tatsächlich korrekt und vollständig sind. Es besteht eindeutig die Gefahr der Verfälschung der Ergebnisse durch den Experimentator. Es könnten schließlich nur Vergleichannotierungen festgelegt werden von denen man weiß, dass sie durch das zu untersuchende Werkzeug auch gefunden werden können. Um aus diesem Blickwinkel die Zuverlässigkeit und Glaubwürdigkeit der Ergebnisse abzusichern, wurde ein von der Lancaster Universität entwickelte und bereits annotierte Software-Produktlinie verwendet – die *MobileMedia* Applikation.

MobileMedia ist eine *ausführlich dokumentierte, frei verfügbare, mittelgroße Java (ME) - Anwendung* mit der Multimedia-Dateien auf mobilen Geräten bearbeitet und verwaltet werden können. Die Feature-Repräsentationen sind durch textuelle Antenna-Präprozessor Anweisungen im Quellcode annotiert. Insgesamt existieren von dem System acht unterschiedliche Releases, in denen inkrementell Features hinzugefügt wurden. Die Entwickler der Anwendung nutzten MobileMedia als Fallstudie in unterschiedlichen Untersuchungen im Kontext der aspektorientierten Softwareentwicklung, wie z.B. in Figueiredo et al. [FCS+08] oder Conejero et al. [CFG+09]. Aufgrund der günstigen Eigenschaften dieser Software wurden einige Releases zudem in Fallstudien von Kästner [Käs10] und in einem formalen Experiment von Feigenspan et al. [FKAL09] verwendet.

In dieser Arbeit wurde das zweitumfangreichste 7. Release der *MobileMedia* Applikation genutzt, bestehend aus 46 Dateien und rund 4600 Quellcodezeilen. *Tabelle 6.1* zeigt den genauen Umfang des Systems in der Genauigkeit der abstrahierten Schlüssel-Elemente des Programmquellcodes (vgl. *Kapitel 5.2.1*). Man kann unter anderem hieraus ablesen, dass insgesamt 524 Lokale Variablen, 239 Methoden und 132 Felder in 47 unterschiedlichen Typen auftreten. Eine solche Anwendung kann daher als mittelgroß bezeichnet werden.

In dem 7. Release werden die optionalen Haupt-Features PLAY MUSIC, VIEW PHOTO, SMS TRANSFER, COPY MEDIA, FAVOURITES sowie COUNT/SORT umgesetzt. In CIDE, der Dokumentationskomponente von LEADT, können für markierte Quellcodebereiche keine aus-

sagenlogischen Ausdrücke hinterlegt werden, damit jene nur in definierten Varianten erscheinen. Aktuell werden lediglich „einfache“ Markierungen für ein Feature sowie UND-Beziehungen für mehrere Features durch Farbüberlagerungen unterstützt. Allerdings können zusätzliche „Hilfs-Features“ definiert werden, für welche spezifische, logische Bedingungen gelten. Für das 7. Release werden für eine vollständige Annotierung mit CIDE aufgrund dessen noch die Features SMS TRANSFER OR COPY MEDIA, NOT SMS TRANSFER, NOT FAVOURITES sowie MUSIC XOR PHOTO benötigt. *Abbildung 6.1* zeigt das vollständige, zugehörige Feature-Diagramm mit all seinen Abhängigkeiten. Es sei noch einmal betont, dass jene Hilfs-Features, die unter dem Knoten ADDITIONAL zusammengefasst sind, nicht Teil des eigentlichen Feature-Modells sind, sondern nur aufgrund der Repräsentationsbeschränktheit von CIDE benötigt werden.[2]

Die Hilfs-Features NOT SMS TRANSFER (6 Quellcodezeilen) NOT FAVOURITES (1 Quellcodezeile) sowie MUSIC_XOR_PHOTO (12 Quellcodezeilen) werden aufgrund ihrer geringen Größe und Verstreuung im Quellcode in der Fallstudie nicht weiter berücksichtigt. Für diese drei Features ist eine automatisierte Transition überflüssig, da jene auch mit geringem Aufwand manuell erfolgen kann.

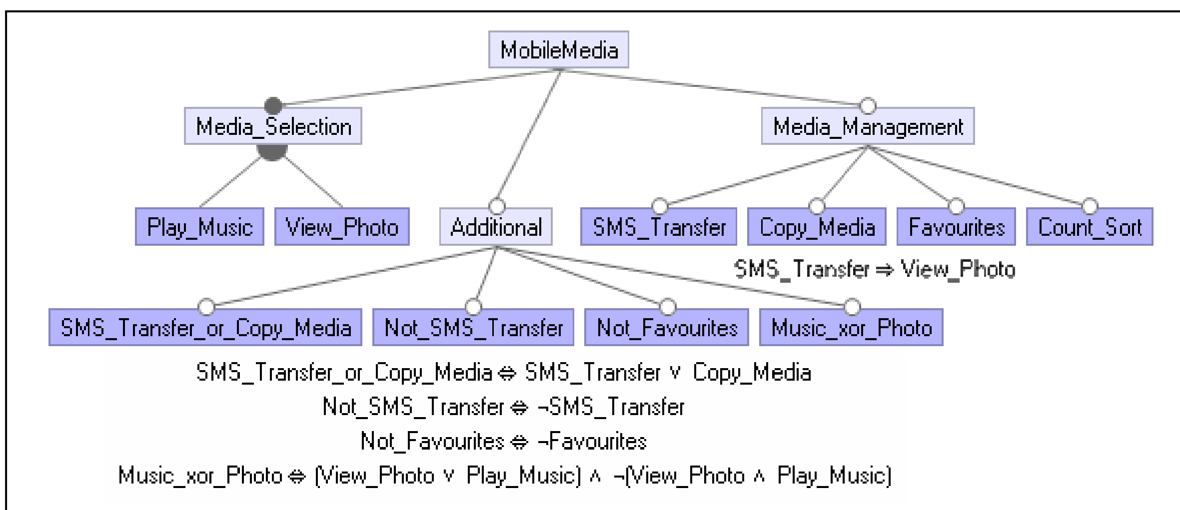



Abbildung 6.1: Feature-Diagramm von MobileMedia (7. Release)

In der verwendeten MobileMedia Applikation werden in einigen Ausnahmefällen Präprozessor Anweisungen eingesetzt, um die Haupt-Blockstruktur von *if-else-Anweisungen* zu

[2] Aus dem Blickwinkel der Skalierbarkeit von LEADT wäre prinzipiell das 8. Release aufgrund des etwas umfangreicheren Quelltextes (ca. 5700 Quellcodezeilen) vorzuziehen. Jenes benötigt aber sehr viele „Hilfs-Features“ für die Abbildung in CIDE. Hierdurch wird das originale Feature-Diagramm stark „aufgebläht“ und deutlich komplizierter. Zum Zweck der einfachen Nachvollziehbarkeit für den Leser wurde sich hier daher für das etwas „kleinere“ 7. Release entschieden.

verändern oder sogar aufzulösen. In CIDE gelten solche Annotierungen allerdings als undiszipliniert und werden daher nicht erlaubt (vgl. *Kapitel 4.3; 5.1.1*). Damit eine disziplinierte Markierung möglich ist, musste der Quellcode an entsprechenden Stellen modifiziert werden. Hierbei wurden in allen Fällen, ähnlich wie in der Fallstudie von Kästner [Käs10], alle betroffenen if-else-Blöcke in seperate Strukturen überführt. Hierdurch sind zum Teil Quellcode-Replikationen entstanden. Die Semantik der Anwendung wurde dadurch allerdings nicht verändert. In *Tabelle 6.1* sieht man den Stand des Systems vor und nach der Modifikation. Hieraus wird deutlich, dass die durchgeführten Modifikationen marginal sind. Zur Wahrung der Wiederholbarkeit der Fallstudie sind in *Tabelle 6.2* zudem alle Anpassungen kurz dokumentiert.

| | **Original-Version** | **Angepasst für CIDE** |
|---|---|---|
| **KOMPILIERUNGSEINHEIT** | **46** | **46** |
| **FELD** | **132** | **132** |
| FELD REFERENZ | 528 | 538 |
| IMPORT | 264 | 264 |
| **LOKALE VARIABLE** | **524** | **524** |
| LOKALE VARIABLEN REFERENZ | 665 | 674 |
| **METHODE** | **239** | **239** |
| METHODEN REFERENZ | 1065 | 1086 |
| PARAMETER REFERENZ | 1260 | 1285 |
| **TYP** | **47** | **47** |
| TYP REFERENZ | 1541 | 1556 |
| | 6311 | 6395 |

Tabelle 6.1: Schlüssel-Elemente des 7. Releases vor und nach Anpassung für CIDE

| **Datei** | **Zeile/n** | **Grund** | **Änderung** |
|---|---|---|---|
| MediaController.java | 74-270 | If else if else…-Struktur wird verändert | Einzelne if else - Blöcke |
| MediaListController.java | 87 | If else - Struktur wird verändert | Einzelne if else - Blöcke |
| | 123-131 | If else - Struktur wird verändert | Einzelne if else - Blöcke |
| MediaListScreen.java | 80 | If else - Struktur wird verändert | Einzelne if else - Blöcke |
| MusicPlayController.java | 57-104 | If else if else…-Struktur wird verändert | Einzelne if else - Blöcke |
| PhotoViewController.java | 83-86 | If else - Struktur wird verändert | Einzelne if else - Blöcke |
| | 105-107 | If else - Struktur wird verändert | Einzelne if else - Blöcke |

Tabelle 6.2: Modifikationen des originalen MobileMedia Quellcodes

Abschließend sei angemerkt, dass das 7. Release der MobileMedia Applikation prinzipiell von zwei unterschiedlichen Stellen herunter geladen werden kann:

1. *http://www.lancs.ac.uk/postgrad/figueire/spl/icse08/* (zu der Publikation von Figueiredo et al. [FCS+08] zugehörige Webseite; unveränderter Stand des Quellcodes seit der ersten Veröffentlichung)

2. *http://mobilemedia.cvs.sf.net/viewvc/mobilemedia/* (Open-Source Software Plattform SourceForge; kontinuierlich verbesserter Quellcode) [3]

In dieser Arbeit wird sich auf die „Erste“ Version von *Stelle 1* bezogen, welche auch von den Entwicklern der Software direkt referenziert wird [FCS+08]. Vorteilhaft ist hierbei, dass ein eindeutiger Stand sehr einfach benannt und „bequem“ als ZIP-Archiv herunter geladen werden kann. Wohingegen die unterschiedlichen Revisionen in SourceForge mit unter verwirrend sein könnten. Für das herunter laden der einzelnen Dateien wird zudem ein SVN Client benötigt.

Das zugehörige Feature-Diagramm (ohne die CIDE spezifischen Features) wurde von den Autoren per E-Mail erfragt.

## 6.3 Auswertung der Fallstudie

In diesem Kapitel wird die Leistungsfähigkeit von LEADT exemplarisch am Beispiel des 7. Release der MobileMedia Applikation demonstriert. Hierzu werden insgesamt vier unterschiedliche Bewertungsaspekte betrachtet – und zwar *erzielte Vollständigkeit der Features*, *Effektivität der Element-Vorschläge*, *Automatisierungspotential* sowie *benötigter Zeitaufwand*. Für jeden dieser Aspekte wird kurz motiviert weshalb dieser zu untersuchen ist. Im Anschluss daran werden zudem jeweils konkrete Metriken definiert, erzielte Ergebnisse quantifiziert und interpretiert.

In der folgenden Auswertung der Fallstudie wird auf Messwerte aus dem durchgeführten Feature Mining zurückgegriffen (vgl. *Kapitel 6.2*). Die Datenerhebung als solches steht hier allerdings nicht in dem Fokus. Der interessierte Leser sei hierzu für eine umfassende Darstellung auf den *Anhang A* verwiesen, in welcher alle einzelnen Schritte der Lokalisierung und Expansion für jedes Feature genau nachvollzogen werden können.

[3] Hinweis: Die Bezeichnung der einzelnen Releases zwischen Stelle 1 und 2 ist nicht identisch. Das 7. Release von Stelle 1 entspricht dem 6. Release von Stelle 2.

### 6.3.1 Erzielte Vollständigkeit der Features

Ein zentrales Ziel von Feature Mining ist die *vollständige* Erfassung und Dokumentation der Features (vgl. *Kapitel 3.1)*. Daher wird auch zuerst geprüft in wie weit LEADT hierbei unterstützend helfen kann.

#### Metriken

In MobileMedia sind alle Feature-Repräsentationen bereits annotiert. Es wird davon ausgegangen, dass alle Varianten fehlerfrei sind. Die Leistungsfähigkeit hinsichtlich der erzielten Vollständigkeit lässt sich daher sehr gut in einen Soll-Ist-Vergleich für die jeweiligen Feature Mining Schritte darstellen. Hierbei wird jeweils die Anzahl der lokalisierten bzw. expandierten Elemente mit Hilfe von LEADT (Ist) zu der Anzahl der originalen Feature-Elemente (Soll) ins Verhältnis gesetzt. Im optimalen Fall, wenn durch LEADT alle Elemente identifiziert werden, müsste jener Quotient genau 1 (bzw. 100%) sein – im schlechtesten Fall, wenn keines der Elemente identifiziert wird, wäre er 0 (%).

Es sei noch einmal erinnert, dass in dem Vollständigkeitskriterium festgelegt wurde (vgl. *Kapitel 3.1*), dass ein Feature solange expandiert werden muss bis kein ungenutzter Quellcode mehr übrig bleibt. Während des durchgeführten Feature Mining wurden in einigen Fällen Elemente identifiziert, welche laut des geforderten Vollständigkeitskriteriums zu einem Feature zugeordnet werden müssten, aber in dem Original bzw. in der Entscheidungsgrundlage nicht als solches markiert sind. In eindeutig begründbaren Fällen wurden entsprechende Elemente in die „Soll-Feature-Repräsentationen" mit aufgenommen, obwohl jenes streng genommen nach dem definierten Vorgehen in *Kapitel 6.2.1* nicht zulässig ist. Allerdings kann LEADT nicht zu lasten gelegt werden, wenn es *zusätzliche*, korrekte Bereiche findet, die durch die Entwickler der Software wahrscheinlich übersehen wurden. Zur Wahrung der Nachvollziehbarkeit und Reproduzierbarkeit der erzielten Ergebnisse werden in *Anhang A* alle solche *zusätzlichen Feature-Elemente* sowie die zugehörigen Begründungen explizit benannt.

Für die Erfassung der erzielten Vollständigkeit werden also drei *Feature-Vollständigkeitsmaße* wie folgt definiert:

$$Vollständigkeitsgrad_{Lokalisierung} = \frac{\#Lokalisierte\ Feature\text{-}Elemente}{\#Originale\ Feature\text{-}Elemente + \#Zusätzliche\ Feature\text{-}Elemente} * 100$$

$$Vollständigkeitgrad_{Expansion} = \frac{\#Expandierte\ Feature\text{-}Elemente}{\#Originale\ Feature\text{-}Elemente + \#Zusätzliche\ Feature\text{-}Elemente} * 100$$

$$Vollständigkeitgrad_{Gesamt} = Vollständigkeitsgrad_{Lokalisierung} + Vollständigkeitgrad_{Expansion}$$

### Ergebnisse

*Tabelle 6.3* zeigt, unter Berücksichtigung der zusätzlichen Elemente, die ermittelten Messwerte, die erzielten Vollständigkeiten für die einzelnen Features sowie den Mittelwert und die Standardabweichung über alle Features hinweg.

| | PLAY MUSIC | SMS TRANSFER | COPY MEDIA | FAVOURI-TES | COUNT / SORT | VIEW PHOTO | SMS. OR COPY. | **Mittelwert** | **Standard-abweich-ung** |
|---|---|---|---|---|---|---|---|---|---|
| #Lokalisierte Feature-Elemente | 15 | 2 | 2 | 1 | 9 | - | - | 5,8 | 5,4 |
| #Expandierte Feature-Elemente | 1180 | 1074 | 213 | 89 | 166 | 812 | 313 | 549,6 | 425,9 |
| #Identifizierte Elemente | 1195 | 1076 | 215 | 90 | 175 | 812 | 313 | 553,7 | 427,9 |
| #Originale Feature-Elemente | 1375 | 660 | 240 | 210 | 197 | 808 | 310 | 542,9 | 405,8 |
| #Zusätzliche Feature-Elemente | 11 | 431 | 8 | 6 | 6 | 284 | 14 | 108,6 | 162,3 |
| #Feature-Elemente | 1386 | 1091 | 248 | 216 | 203 | 1092 | 324 | 651,4 | 476,2 |
| Vollständigkeitsgrad$_{Lokalisierung}$ | 1,08% | 0,18% | 0,81% | 0,46% | 4,43% | - | - | 0,99% | 1,57% |
| Vollständigkeitsgrad$_{Expansion}$ | 85,14% | 98,44% | 85,89% | 41,20% | 81,77% | 74,36% | 96,60% | 80,49% | 17,79% |
| Vollständigkeitsgrad$_{Gesamt}$ | **86,22%** | **98,63%** | **86,69%** | **41,67%** | **86,21%** | **74,36%** | **96,60%** | **81,48%** | **17,85%** |

Tabelle 6.3: Erzielte Feature-Vollständigkeit

Im Durchschnitt wurden rund *82%*, bei einer Standardabweichung von ca. *18%*, der Feature-Repräsentationen identifiziert. Die aus der Feature-Lokalisierung stammenden Startelemente tragen mit nur ca. *1%* einen geringen Anteil zu der gesamten Feature-Repräsentation bei. Der entscheidende Beitrag wurde mit rund *81%* von der Expansionskomponente geleistet. Das beste Gesamtergebnis wurde für das Feature SMS TRANSFER erzielt (99%). Das schlechteste hingegen für FAVOURITES (*42%*).

### Interpretation

Die erzielten Ergebnisse sind durchaus viel versprechend. Vor allem wenn man berücksichtigt, dass menschliche Intuition und Erfahrung bei der Suche nach weiteren Elementen hierbei vollständig unberücksichtigt geblieben sind. Zudem wurden unter Verwendung des Feature-Namen nur sehr wenige initiale Informationen eingebracht, um den Feature Mining Prozess anzustoßen. Für VIEW PHOTO sowie SMS TRANSFER OR COPY MEDIA wurden sogar nicht einmal die Feature-Namen berücksichtigt, sondern nur die bereits identifizierten Teilmengen-Feature-Repräsentationen (vgl. *Kapitel 6.2.1*).

Um diese Behauptung zu stützen, wurde für jedes Feature analysiert welche Bereiche unerkannt geblieben sind bzw. durch welche zusätzlichen Maßnahmen ein maximaler Vollständigkeitsgrad von 100% erreicht hätte werden können. *Tabelle 6.4* fasst die Ergebnisse hierzu zusammen. Es ist hierbei deutlich erkennbar, dass sich die fehlenden Bereiche auf das stark reglementierte, automatisierte Vorgehen zurückführen lassen. In sechs von sieben Fällen hätten deutlich mehr Elemente gefunden werden können, wenn weitere Start-

elemente aus der textbasierten Suche, mit dem Feature-Namen als Anfrage, berücksichtigt wurden wären. Erfahrung und Intuition des Anwenders, bei der Betrachtung des lokalen Kontexts von vorgeschlagenen Elementen, hätte ebenso die Vollständigkeit von sechs Features bedeutend erhöht.

| | PLAY MUSIC | SMS TRANSFER | COPY MEDIA | FAVOURITES | COUNT / SORT | VIEW PHOTO | SMS. OR COPY. |
|---|---|---|---|---|---|---|---|
| Textbasierte Suche nach dem Feature-Namen (oder Gewichtung wichtiger Wörter des Vokabulars) durch den Anwender würde weitere relevante (Start-)Elemente liefern | x | x | x | x | | x | x |
| Eingeschlossene Elemente fehlen, obwohl umrandender Kontext Teil des Features; Würde erkannt durch menschliche Erfahrung / Intuition, Anpassung von LEADT möglich | x | x | x | x | x | x | |
| Übergeordneter Block fehlt, obwohl einige untergeordnete Elemente Teil des Features; Würde erkannt durch menschliche Erfahrung / Intuition, Anpassung von LEADT möglich | x | | | | x | | |
| Zu markierten Anweisungen im „Try-Block“ fehlen zugehörige „Catch-Blöcke“; Würde erkannt durch menschliche Erfahrung / Intuition, Anpassung von LEADT möglich | x | | | x | | | |

Tabelle 6.4: Maßnahmen zur Erreichung eines maximalen Vollständigkeitsgrads

Es sei erneut betont, dass LEADT ein semiautomatisches, iteratives Vorgehen unterstützt. Es erhebt nicht den Anspruch Features mit minimalen Startinformationen zu *100%* vollautomatisch identifizieren zu können. Unter diesem Blickwinkel sind die im Durchschnitt erreichten *82%* als sehr gut zu bewerten. Die Analyse der fehlenden Elemente hat auch gezeigt, dass die Ergebnisse durch einige Anpassungen in LEADT sogar noch durchaus verbessert werden könnten. Eine minimale Erweiterung des Beziehungsgraphen würde zu weiteren relevanten Vorschlägen führen. So würden beispielsweise zusätzliche Kontrollflussbeziehungen der Fehlerbehandlung in dem „schlechtesten“ FAVOURITES Fall allein rund *100* weitere Elemente (ca. *46%* des Features) aufzeigen. Von der Repräsentation der Struktureinheit „Block“, welche innerhalb von Methoden auftreten kann, würde man in den Fällen von PLAY MUSIC und COUNT / SORT erheblich profitieren.

## 6.3.2 Effektivität der Element-Vorschläge

Grundsätzlich kann die maximale Vollständigkeit immer dann erreicht werden, wenn dem Anwender einfach alle Elemente des Systems hintereinander vorgeschlagen werden. Allerdings hätte der Anwender in diesem Fall den gleichen Aufwand wie bei einem manuellen Vorgehen. Daher ist es wichtig bei der Bewertung von unterstützenden Werkzeugen auch auf die *Effektivität der Vorschläge* einzugehen.

### Metriken

In Anlehnung an Liu et al. [LMPR07] und Poshyvanyk et al. [PGM+07] wird die *Vorschlagseffektivität* definiert als der prozentuale Anteil *der korrekten Element-Vorschläge* an *allen Element-Vorschlägen*. Eine Vorschlagseffektivität von 100% bedeutet, dass alle Ele-

ment-Vorschläge eines Werkzeuges (korrekterweise) zu dem gesuchten Feature gehören. Wobei hingegen 0% ausdrücken, dass keines der Element-Vorschläge Teil der gesuchten Feature-Repräsentation ist.

Die Vorschlagseffektivität lässt sich formal wie folgt ausdrücken:

$$\textit{Vorschlagseffektivität} = \frac{\#\textit{Korrekte Element-Vorschläge}}{\#\textit{Alle Element-Vorschläge}} * 100$$

Als *korrekte* Element-Vorschläge werden in dieser Arbeit nicht nur diejenigen gezählt, die zu der *originalen* Feature-Repräsentation, sondern auch zu der *zusätzlich lokalisierten* Feature-Repräsentation gehören. Wie bereits in *Kapitel 6.3.1* diskutiert, stellen gemäß der Festlegung in *Kapitel 6.2* eigentlich nur allein die originalen Annotierungen der MobileMedia-Entwickler die Entscheidungsgrundlage dar. Allerdings würde es aber die Leistung von LEADT fälschlicherweise abwerten, wenn laut des Vollständigkeitskriteriums zweifelsfrei korrekte Element-Vorschläge als nicht-korrekt eingestuft werden. Um die Glaubwürdigkeit der erzielten Ergebnisse zu wahren, sei in diesem Zusammenhang daher erneut auf den *Anhang A* verwiesen. Dort werden alle zusätzlichen korrekten Elemente-Vorschläge sowie die zugehörigen Begründungen dargestellt.

Es sei explizit darauf hingewiesen, dass die Anzahl der korrekten Element-Vorschläge nicht per se mit der Anzahl der identifizierten Feature-Elemente gleichgesetzt werden kann. Ein Element-Vorschlag kann z.B. eine ganze Methode, Klasse oder sogar eine Kompilierungseinheit sein (vgl. *Kapitel 5.2.1*). Jene umschließen in der Regel weitere Elemente wie lokale Variablen, Zugriffe auf Methoden, Typen oder Felder, etc. Es gilt daher der Zusammenhang:

$$\#\textit{Identifizierte Feature-Elemente} >= \#\textit{Korrekte Element-Vorschläge}$$

In *Kapitel 6.2.1* wurde festgelegt, dass die Expansion für jedes Feature nach zehn falschen Element-Vorschlägen terminiert. Auch im optimalen Fall, d.h. wenn das Feature vollständig identifiziert wird, werden noch zehn weitere Elemente betrachtet. Es wird daher darüber hinaus noch die *bereinigte Vorschlagseffektivität* eingeführt, welche das Stoppkriterium explizit herausrechnet.

$$\textit{Bereinigte Vorschlagseffektivität } (VE_{Ber}) = \frac{\#\textit{Korrekte Element-Vorschläge}}{\#\textit{Alle Element-Vorschläge - 10}} * 100$$

In dieser Fallstudie wurde aus der Feature-Lokalisierung nur ein Vorschlag pro unabhängigem Feature berücksichtigt, der immer korrekt war – für abhängige Features wurden erst gar keine benötigt. Die folgende Untersuchung zu der (bereinigten) Vorschlagseffek-

tivität vernachlässigt jenen minimalen Einfluss und bezieht sich nur auf die Feature-Expansion, welche für den Hauptteil der identifizierten Elemente verantwortlich war (vgl. *Kapitel 6.3.1*). Für Evaluationen der Effektivität von der (in der Feature-Lokalisierung verwendeten) IR-basierten Textsuche sei auf die vorgestellten Arbeiten in *Kapitel 4.1.1.2* verwiesen.

## Ergebnisse

In *Tabelle 6.5* sind die ermittelten Kennzahlen für die einzelnen Features sowie die Mittelwerte und die Standardabweichungen über alle Features hinweg dargestellt. Zur Einordnung jener Werte wird zudem auch der jeweilige Element-Anteil der Features am Gesamt-System angegeben. Dieser Anteil lässt sich ebenso als *Vorschlagseffektivität eines Zufallsgenerators* verstehen. Sprich, würde man beispielsweise einem Anwender, welcher das Feature PLAY MUSIC expandiert, rein zufällige Element-Vorschläge unterbreiten, so wäre im Schnitt ca. jeder Fünfte von ihnen korrekt – bei SMS TRANSFER OR COPY MEDIA nur jeder Zwanzigste, usw.

| | PLAY MUSIC | SMS TRANSFER | COPY MEDIA | FAVOURI-TES | COUNT / SORT | VIEW PHOTO | SMS. OR COPY. | Mittelwert | Standard-abweich-ung |
|---|---|---|---|---|---|---|---|---|---|
| Anteil am System / Vorschlagseffektivität eines Zufallsgenerators | 21,67% | 17,06% | 3,88% | 3,38% | 3,17% | 17,08% | 5,07% | 10,19% | 7,45% |
| #Korrekte Element-Vorschläge | 120 | 155 | 41 | 33 | 40 | 85 | 26 | 71,4 | 46,3 |
| #Alle Element-Vorschläge | 152 | 176 | 72 | 51 | 56 | 100 | 50 | 93,9 | 47,6 |
| Vorschlagseffektivität | 78,95% | 88,07% | 56,94% | 64,71% | 71,43% | 85,00% | 52,00% | 71,01% | 12,81% |
| Bereinigte Vorschlagseffektivität | **84,51%** | **93,37%** | **66,13%** | **80,49%** | **86,96%** | **94,44%** | **65,00%** | **81,56%** | **11,06%** |

Tabelle 6.5: Effektivität der Vorschläge im Vergleich zu einem Zufallsgenerator

Im Durchschnitt waren rund *82%* der (bereinigten) Vorschläge korrekt, bei einer Standardabweichung von ca. *11%*. Die höchste bereinigte Vorschlagseffektivität wurde für das Feature VIEW PHOTO erzielt (rund *94%)*, die niedrigste für SMS TRANSFER OR COPY MEDIA (*65%*).

## Interpretation

Betrachtet man die Werte aller sieben Features, so fällt auf, dass die Vorschläge für COPY MEDIA und SMS TRANSFER OR COPY MEDIA deutlich unpräziser waren als für die Übrigen. Aktuell kann mit dieser Fallstudie nicht beantwortet werden was genau die Ursache hierfür ist. Eine mögliche Erklärung, die es in Zukunft zu prüfen gilt, ist, dass im Vergleich zu den ähnlich kleinen Features FAVOURITES und COUNT/SORT keine charakteristischen Wörter auftreten, welche entscheidende Informationen für die Expansion enthalten.

Dessen ungeachtet lässt sich gemäß des $\chi^2$-Tests für Alternativdaten mit einer Irrtumswahrscheinlichkeit von weniger als *1%* sagen, dass die bereinigte Vorschlagseffektivität *für jedes Feature* signifikant höher ist als die eines Zufallsgenerators. [4] Gerade für recht umfangreiche Features wie PLAY MUSIC, SMS TRANSFER oder auch VIEW PHOTO lässt sich daher ableiten, dass die hohe Effektivität der Vorschläge nicht nur auf der Größe der Features basiert sondern vor allem auf dem Beitrag von LEADT. Über alle Features hinweg gesehen, waren rund *acht von zehn Vorschläge korrekt*. Ein Zufallsgenerator würde im Mittel nur *einen* Korrekten *von insgesamt zehn* generieren.

Es ist daher anzunehmen, dass die in der Feature-Expansion verwendete Relevanzwertfunktion Rückschlüsse auf die Wahrscheinlichkeit für die Korrektheit eines Element-Vorschlages erlaubt. Denn erst durch diese Eigenschaft wird es möglich (nur) vermeintlich korrekte Elemente vorzuschlagen und somit die Effektivität zu erhöhen. Jene Annahme wird in dem folgenden Exkurs bekräftigt.

### Beziehung zwischen Relevanzwert und Wahrscheinlichkeit für Korrektheit

Um zu prüfen, ob höhere Relevanzwerte im Durchschnitt häufiger zu korrekten Vorschlägen korrespondieren als niedrige Relevanzwerte wird das Vorgehen von Robbillard [Rob08] adaptiert. Hierbei werden alle generierten Vorschläge hinsichtlich ihres Relevanzwertes in ein bestimmtes Intervall eingeordnet. Im Folgenden werden zehn Intervalle zwischen *1,0* und *0* mit einer Breite von *0,1* verwendet. Für jedes dieser Intervalle wird die relative Häufigkeit der tatsächlichen Klassen, d.h. der korrekten und nicht-korrekten Vorschläge, ermittelt. Bei einer theoretisch perfekten Funktion sollte der errechnete Relevanzwert für einen Element-Vorschlag der Wahrscheinlichkeit für dessen Korrektheit entsprechen, sodass sich nach der Idee der Analyse das Diagramm in *Abbildung 6.2* ergibt.

Trägt man *alle* betrachteten Vorschläge der MobileMedia-Fallstudie nach dem gleichen Schema ab ergibt sich die *Abbildung 6.3* (vgl. *Anhang A*).

---

[4] Mit dem $\chi^2$-Test für Alternativdaten kann geprüft werden wie gut eine beobachtete Verteilung von Alternativdaten mit, einer in der Nullhypothese, erwarteten Verteilung übereinstimmt [BL08, S. 68]. In diesem Fall entspricht die erwartete Verteilung jeweils der Vorschlagseffektivität eines Zufallsgenerators. Die Alternativhypothese besagt hingegen, dass die bereinigte Vorschlagseffektivität höher ist als die des Zufallsgenerators. Der $\chi^2$-Test ermittelt die Wahrscheinlichkeit, mit der das gefundene empirische Ergebnis sowie Ergebnisse, die noch extremer sind als das gefundene Ergebnis, auftreten können, wenn die Populationsverhältnisse der Nullhypothese entsprechen. Diese Wahrscheinlichkeit bezeichnet man als Irrtumswahrscheinlichkeit p, mit der man die Nullhypothese fälschlicherweise zugunsten der Alternativhypothese verwirft. Ist p kleiner als ein definiertes Signifikanzniveau $\alpha$, bezeichnet man das Stichprobenergebnis als statistisch signifikant. Stichprobenergebnisse mit $p \leq 0,05$ ($\alpha = 0,05$) sind signifikant, mit $p \leq 0,01$ ($\alpha = 0,01$) sehr signifikant [BD06b, S .494].

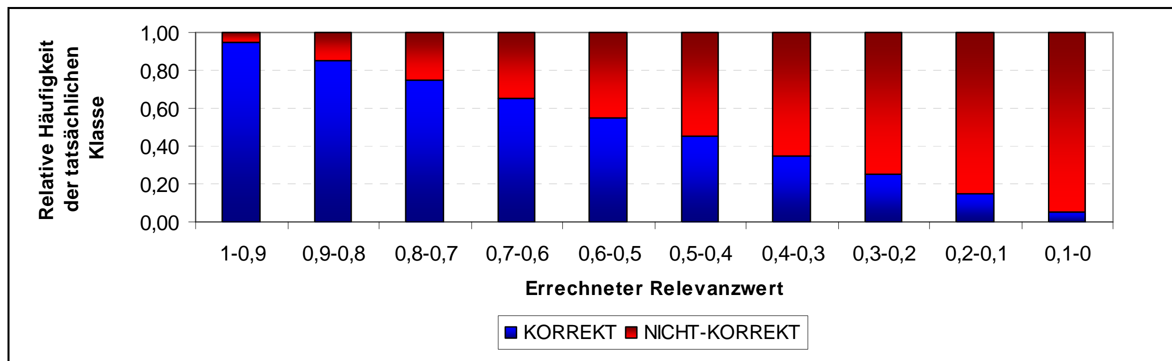



Abbildung 6.2: Perfekte Element-Vorschläge in der Theorie

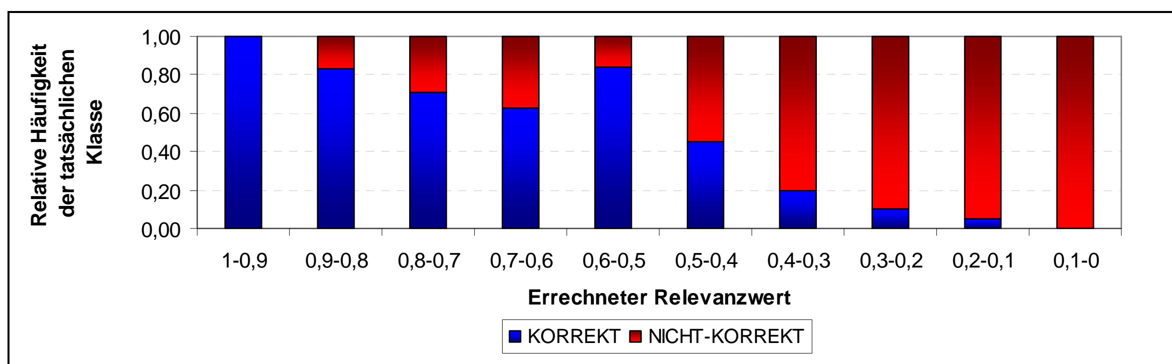



Abbildung 6.3: Element-Vorschläge durch LEADT in der Fallstudie

Durch eine rein visuelle Betrachtung beider Diagramme lässt sich, bei den in der Fallstudie beobachteten Element-Vorschlägen, keine bedeutsame Abweichung von der theoretisch erwarteten perfekten Verteilung ausmachen. Zur Überprüfung dieser Annahme wurde der $\chi^2$-Anpassungstest durchgeführt. Der Tests kommt auf einem Signifikanzniveau von *$\alpha$ = 0,2* zu dem selben Ergebnis.[5]

Aufgrund dessen kann davon ausgegangen werden, dass die in LEADT verwendete Vorschlagsfunktion dabei helfen kann Elemente hinsichtlich ihrer Zugehörigkeitswahrscheinlich zu bewerten. Sollte ein Anwender keinen besseren Anhaltspunkt haben, so ist es daher stets besser den Vorschlag mit dem höchsten Relevanzwert bzw. der höchsten Zugehörigkeitswahrscheinlichkeit näher zu untersuchen, als zufällig irgendein Element aus-

[5] Im Unterschied zu dem durchgeführten $\chi^2$-Test für Alternativdaten ist man hier daran interessiert die Nullhypothese als Wunschhypothese beizubehalten ($H_0$: Die erhobenen Daten sind gemäß der erwarteten perfekten Vorschlagsfunktion verteilt). Hierfür muss eine möglichst niedrige $\beta$-Fehlerwahrscheinlichkeit, also die Wahrscheinlichkeit $H_0$ irrtümlicherweise beizubehalten, angestrebt werden. Eine niedrige $\beta$ - Fehlerwahrscheinlichkeit kann indirekt sichergestellt werden indem ein möglichst hohes Signifikanzniveau gewählt wird. Empfehlungswert ist hierbei $\alpha$ = 0,2. Durch diese Maßnahme erleichtert man Entscheidungen zugunsten der Alternativhypothese bzw. gegen die Nullhypothese [BL08, S. 76; S. 82; S.251].

zuwählen. Hierdurch kann gewährleistet werden, dass die Feature-Expansion mit LEADT im Vergleich zu einem unstrukturierten Vorgehen zielgerichteter und deutlich effektiver erfolgen kann.

### 6.3.3 Automatisierungspotential

In dem vorherigen *Kapitel 6.3.2* wurde gezeigt, dass die verwendete Vorschlagsfunktion der Feature-Expansion in der Lage ist die Relevanz von System-Elementen für ein Feature zu bewerten. Die errechneten Werte lassen sich hierbei im Mittel als Wahrscheinlichkeiten für die Feature-Zugehörigkeit eines Elements interpretieren. So bedeutet beispielsweise der Relevanzwert *1*, dass (*basierend auf den aktuellen Informationen)* das vorgeschlagene Element immer zu der Feature-Repräsentation gehört. Ein Wert von *0* verweist hingegen unter gleichen Voraussetzungen nie auf ein korrektes Element. Die beobachteten Werte in der Fallstudie stützen diese Hypothese (vgl. *Abbildung 6.3*).

Für die Einschätzung des tatsächlich Feature Mining Aufwandes für den Anwender, ist es daher wesentlich die Verteilung der Relevanzwerte der zu *untersuchenden Top-Element-Vorschläge*[6] zu kennen. Es ist leicht ersichtlich, dass Vorschläge mit einem Relevanzwert von *1* oder *0* besonders hilfreich sind, weil sie bezüglich der verfügbaren, lokalen Informationen eine *eindeutige* und *sichere Entscheidung* ermöglichen.[7] Jene Entscheidungen lassen sich zum Zweck der Aufwandsreduktion automatisieren, sodass der Anwender im Grunde nur noch verbleibende „unsichere" Element-Vorschläge hinsichtlich ihrer Feature-Zugehörigkeit manuell prüfen muss. In diesem Kapitel wird untersucht welches Potential zur Automatisierung der Korrektheitsüberprüfung der Element-Vorschläge für die einzelnen Features besteht.

#### Metriken

Der prozentuale Anteil lokal sicherer (Top-)Element-Vorschläge (mit dem Wert 1 oder 0) an allen Element-Vorschlägen wird im Folgenden als *Automatisierungspotential* definiert. Ein Wert von 100% stellt den optimalen Fall dar, bei dem ein Anwender keinen der Vor-

---

[6] Unter einem Top-Element-Vorschlag wird der Vorschlag mit dem höchsten Relevanzwert innerhalb eines Expansionsschrittes verstanden.

[7] In LEADT wird im Rahmen eines Expansionsschrittes nicht jedes Schlüssel-Element des Systems hinsichtlich seiner Relevanz bewertet (vgl. Kapitel 5.2.3). Unberücksichtigte Elemente haben keinen Relevanzwert. Sie werden dem Anwender nicht präsentiert, weil keine Aussage hinsichtlich ihrer Feature-Zugehörigkeit getroffen werden kann. Solche Elemente dürfen nicht mit jenen gleichgesetzt werden, die einen Relevanzwert von 0 aufweisen. Ein Relevanzwert von 0 drückt nämlich aus, dass ein Element höchstwahrscheinlich nicht zu dem Feature gehört. Im Allgemeinen ist davon auszugehen, dass „0-Elemente" nur selten Top-Vorschläge sein werden. Aufgrund dessen spielen sie hier eine untergeordnete Rolle.

schläge manuell prüfen muss. Beträgt das Automatisierungspotential hingegen 0%, so muss in diesem schlechtesten Fall der Anwender alle Vorschläge manuell prüfen.

$$Automatisierungspotential = \frac{\#Sichere\ Element\text{-}Vorschläge}{\#Alle\ Element\text{-}Vorschläge} * 100$$

Wie auch bei der Vorschlagseffektivität wird zudem eine bereinigte Variante dieser Kennzahl eingeführt, um das Stoppkriterium herauszurechnen (vgl. *Kapitel 6.3.2*).

$$Bereinigtes\ Automatisierungspotential = \frac{\#Sichere\ Element\text{-}Vorschläge}{\#Alle\ Element\text{-}Vorschläge - 10} * 100$$

Es sei darauf hingewiesen, dass sich das Automatisierungspotential im Folgenden nur auf die Feature-Expansion bezieht, weil bislang nur für die in dieser Phase verwendete Funktion der Zusammenhang zwischen Relevanzwert und Zugehörigkeitswahrscheinlichkeit gezeigt wurde. Der Beitrag der Feature-Lokalisierung mit je einem Element-Vorschlag für ein unabhängiges Feature ist ohnehin vernachlässigbar gering, sodass eine Automatisierung jenes Schrittes auch keine wesentliche Verbesserung des gesamten Prozesses erzielen würde.

### Ergebnisse

Gemäß der Expansionsprotokolle der Fallstudie ergeben sich die in *Tabelle 6.6* zusammengefassten Kennzahlen (vgl. *Anhang A*).

| | PLAY MUSIC | SMS TRANSFER | COPY MEDIA | FAVOURITES | COUNT / SORT | VIEW PHOTO | SMS. OR COPY. | Mittelwert | Standardabweichung |
|---|---|---|---|---|---|---|---|---|---|
| #Sichere Element-Vorschläge (0.0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| #Sichere Element-Vorschläge (1.0) | 94 | 121 | 33 | 23 | 23 | 62 | 19 | 53,57 | 37,30 |
| #Sichere Element-Vorschläge | 94 | 121 | 33 | 23 | 23 | 62 | 19 | 53,57 | 37,30 |
| #Alle Element-Vorschläge | 152 | 176 | 72 | 51 | 56 | 100 | 50 | 93,86 | 47,56 |
| Automatisierungspotential | 61,84% | 68,75% | 45,83% | 45,10% | 41,07% | 62,00% | 38,00% | 51,80% | 11,20% |
| Berei. Automatisierungspotential | **66,20%** | **72,89%** | **53,23%** | **56,10%** | **50,00%** | **68,89%** | **47,50%** | **59,26%** | **9,24%** |

Tabelle 6.6: Automatisierungspotential für die Feature-Expansion in MobileMedia

In dieser Fallstudie sind keine Top-Element-Vorschläge mit dem Relevanzwert von *0* aufgetreten. Alle sicheren Element-Vorschläge hatten somit den Wert von *1*. Das bereinigte Automatisierungspotential liegt im Durchschnitt über alle Features hinweg bei rund *59%*, bei einer Standardabweichung von ca. *9%.* Für das Feature SMS TRANSFER ließen sich die meisten Entscheidungen automatisieren (rund *73%*). Die wenigsten hingegen für SMS TRANSFER OR COPY MEDIA (rund *48%*).

### Interpretation

Es ist wichtig zu betonen, dass eine absolut sichere Entscheidung nur dann getroffen werden kann, wenn alle globalen Informationen bekannt sind. In LEADT basieren die berechneten Relevanzwerte aber nur auf den jeweils aktuell verfügbaren, lokalen Informationen. Zudem besteht ein Unsicherheitsfaktor bei Vorschlägen, die textuelle Informationen verwenden, da bislang Polysemie-Effekte oder abweichende Bedeutungen in unterschiedlichen Kontexten nicht berücksichtigt werden. Ein Anwender muss sich daher bewusst sein, dass eine solche Automatisierung des Prozesses immer ein gewisses Risiko für Fehlentscheidungen birgt, auch wenn es in der Regel minimal sein wird.

In der untersuchten MobileMedia-Fallstudie haben lokal sichere Empfehlungen immer zu korrekten Entscheidungen geführt. Unter Verwendung des beschriebenen Automatisierungsansatzes müsste der Anwender im Durchschnitt von allen generierten Vorschlägen eigentlich nur knapp *40%* hinsichtlich ihrer Korrektheit manuell prüfen, um Repräsentationen mit dem selben Vollständigkeitsgrad sowie der selben Effektivität zu identifizieren. Jene Beobachtung unterstreicht, dass mit dem LEADT-Konzept ein wesentlicher Teil der Feature Mining Aufgabe auch vollautomatisch gelöst werden könnte ohne die Qualität dabei erheblich zu gefährden. Nicht zuletzt deswegen ist anzunehmen, dass LEADT einem Anwender helfen kann den Transitionsaufwand zu reduzieren.

## 6.3.4 Benötigter Zeitaufwand

Wie bereits erwähnt kann mit dieser Fallstudie nicht final beantwortet werden in wie weit LEADT dabei helfen kann den Zeitaufwand zu verringern (vgl. *Kapitel 6.2*). Hierzu bräuchte man Vergleichwerte aus Studien in denen Anwender eine identische Aufgabe manuell lösen. Im Folgenden werden zumindest aber die mit LEADT benötigten Zeiten dargestellt, um dem Leser ein ungefähres Gefühl für die erreichte Größenordnung zu vermitteln.

### Metriken

Für die Durchführung der Lokalisierung und Expansion der einzelnen Features nach dem definierten Schema aus *Kapitel 6.2.1* wurde die *benötigte Zeit* bzw. der *Zeitaufwand* für alle einzelnen Aktivitäten gestoppt, welche der Experimentator insgesamt benötigt hat. Für Teilmengen-Features musste zunächst eine Anfrage formuliert werden und hiervon der (korrekte) Treffer mit dem höchsten Relevanzwert ausgewählt werden. Für Obermengen-Features musste dieser Schritt nicht erfolgen. Der Hauptteil der Aktivitäten entfiel vor allem auf die Feature-Expansion. In dieser musste jeder einzelne Vorschlag hinsichtlich seiner Feature-Zugehörigkeit manuell geprüft werden. Als Entscheidungsgrundlage wurde hierfür die annotierte Original-Version genutzt (vgl. *Kapitel 6.2.1*). Für Top-

Element-Vorschläge von LEADT, die laut der Original-Annotierung nicht zu dem Feature gehören, wurde nach reiflicher Überlegung entschieden, ob sie aufgrund des Vollständigkeitskriteriums zusätzlich aufgenommen werden müssen (vgl. *Kapitel 6.3.1; 6.3.2*).

$$\textit{Zeitaufwand} = \textit{Benötigte Zeit}_{\textit{Lokalisierung}} + \textit{Benötigte Zeit}_{\textit{Expansion}}$$

Der *bereinigte Zeitaufwand* entspricht dem Zeitaufwand für den gesamten Prozess abzüglich der benötigten Zeit für die Überprüfung der letzten zehn Element-Vorschläge. Oder anders formuliert: Der bereinigte Zeitaufwand gibt die benötigte Zeit bis zur Prüfung des Stoppkriteriums an.

$$\textit{Bereinigter Zeitaufwand} = \textit{Benötigte Zeit}_{\textit{Lokalisierung}} + \textit{Benötigte Zeit ohne Stoppkriterium}_{\textit{Expansion}}$$

In *Kapitel 6.3.3* wurde festgestellt, dass eine beträchtliche Anzahl aller Element-Vorschläge in der Feature-Expansion automatisiert bearbeitet werden kann. Aufgrund dessen lässt sich der *(bereinigte) tatsächliche Zeitaufwand* des Anwenders wie folgt berechnen:

$$\textit{Tatsächlicher Zeitaufwand} = \textit{Zeitaufwand} * \frac{100 - \textit{Automatisierungspotential}}{100}$$

$$\textit{Bereinigter Tatsächlicher Zeitaufwand} = \textit{Bereinigter Zeitaufwand} * \frac{100 - \textit{Ber. Automatisierungspotential}}{100}$$

## Ergebnisse

*Tabelle 6.7* zeigt die für jedes einzelne Feature spezifischen Zeiten in Minuten, sowie die Summen, den Mittelwerte als auch die Standardabweichungen über alle Features hinweg.

Aktivitäten der Feature-Lokalisierung haben in der Summe über alle Features gesehen weniger als eine Minute ausgemacht, sodass dieser Einfluss im Wesentlichen nur einen Bruchteil an der gesamten Zeit darstellt. Zurückzuführen ist jene geringe Zeit vor allem auf das reglementierte Vorgehen in dieser Fallstudie, bei dem der Einfluss des Anwenders sehr stark eingeschränkt wird (vgl. *Kapitel 6.2.1*).

Insgesamt hat der Vorgang für alle Features in der Summe rund *132 min* gedauert. Subtrahiert man die jeweilige Behandlung des Stoppkriteriums, ergibt sich ein bereinigter Zeitaufwand von ca. *116 min*. Unter Ausschöpfung des Automatisierungspotentials in der Feature-Expansion verbleiben als tatsächlicher (manueller) zeitlicher Aufwand rund *57 min* bzw. in der bereinigten Form *42 min.* Im Durchschnitt beträgt der bereinigte tatsächliche Zeitaufwand pro Feature *6 min* bei einer Standardabweichung von *2,5 min*.

| | Play Music | SMS Transfer | Copy Media | Favouri-tes | Count / Sort | View Photo | SMS. or Copy. | Summe | Mittelwert | Stndard-abweich. |
|---|---|---|---|---|---|---|---|---|---|---|
| Benötigte Zeit$_{Lokalisierung}$ [min] | 0,16 | 0,16 | 0,16 | 0,16 | 0,16 | - | - | 0,8 | 0,16 | 0 |
| Benötigte Zeit$_{Expansion}$ [min] | 30 | 35 | 14 | 10 | 11 | 21 | 10 | 131 | 18,7 | 10,3 |
| Benötigte Zeit ohne Stoppk.$_{Exp.}$ [min] | 28 | 33 | 12 | 8 | 9 | 18 | 7 | 115 | 16,4 | 10,4 |
| Zeitaufwand [min] | 30,2 | 35,2 | 14,2 | 10,2 | 11,2 | 21 | 10 | 131,8 | 18,8 | 10,3 |
| Bereinigter Zeitaufwand [min] | 28,2 | 33,2 | 12,2 | 8,2 | 9,2 | 18 | 7 | 115,8 | 16,5 | 10,4 |
| Tatsächlicher Zeitaufwand [min] | 11,6 | 11,1 | 7,7 | 5,7 | 6,6 | 8 | 6,2 | 56,9 | 8,1 | 2,4 |
| Ber. Tatsächlicher Zeitaufwand [min] | **9,6** | **9,1** | **5,8** | **3,7** | **4,7** | **5,6** | **3,7** | **42,1** | **6** | **2,4** |

Tabelle 6.7: Benötigter manueller Zeitaufwand für das Feature Mining

### Interpretation

Der Zeitaufwand wird grundsätzlich durch viele unterschiedliche Rahmenfaktoren beeinflusst, wie z.B. Kenntnisstand des Anwenders zu dem Aufbau des Systems, Erfahrung im Umgang mit dem Werkzeug, etc. Zudem kommen während des Feature Mining Lerneffekte hinzu, sodass beispielsweise Vorschläge für ein Feature B schneller bewertet werden können, da entsprechende Elemente schon im Kontext der Expansion von Feature A untersucht wurden. Solche Faktoren wurden in dieser Fallstudie nicht kontrolliert. Die gemessenen Werte für die einzelnen Features verstehen sich daher nur als grobe Richtwerte.

In dieser Fallstudie konnten innerhalb von etwas mehr als zwei Stunden im Durchschnitt *83%* der gesamten Feature-Repräsentationen annotiert werden (vgl. *Kapitel 6.3.1*). Unter Verwendung des Automatisierungsansatzes, hätte der gleiche Stand sogar in weniger als einer Stunde erreicht werden können. Der Experimentator war nicht an der Entwicklung oder Wartung von MobileMedia beteiligt. Die Erfahrungen bzw. der Kenntnisstand zu dem Aufbau oder der Funktionsweise des Systems sind im Allgemeinen als gering zu bewerten. Die Vergleichsannotierungen des Original-Systems haben aber dennoch in den meisten Fällen eine zügige Entscheidung ermöglicht. Es ist davon auszugehen, dass in der Praxis ein Anwender von LEADT auch ein Experte des zu überführenden Systems ist (vgl. *Kapitel 3.1*). Aufgrund dessen sollte es ihm möglich sein die Korrektheit der Element-Vorschläge bzw. die tatsächliche Feature-Zugehörigkeit ähnlich schnell zu beurteilen, auch wenn keine „Entscheidungshilfe“ zur Verfügung steht. Unter dieser Annahme wäre daher ein ähnliches Ergebnis vermutlich auch unter Feldbedingungen beobachtbar.

Auch wenn es bislang aufgrund der fehlenden Experimente nicht belegt werden kann, so ist doch stark anzuzweifeln, dass unter gleichen Voraussetzungen mit einem rein manuellen Vorgehen eine ähnliche Zeit erreichbar ist.

### 6.3.5 Fazit

In der Auswertung der Fallstudie wurde gezeigt, dass *allein nur mit Hilfe der von LEADT generierten Element-Vorschläge* Feature-Repräsentationen nahezu vollständig (*81,5%*) und sehr effektiv (*81,6%)* identifiziert werden konnten. Jenes wurde vor allem möglich, weil der während der Feature-Expansion verwendete Algorithmus zur Bestimmung von relevanten Element-Vorschlägen Rückschlüsse auf die tatsächliche Feature-Zugehörigkeit erlaubt und somit signifikant bessere Ergebnisse als ein Zufallsgenerator erzielt. Sollte ein Anwender keinen besseren intuitiven Anhaltspunkt haben, so ist es daher stets zielführender den Vorschlag von LEADT mit dem höchsten Relevanzwert bzw. der höchsten Zugehörigkeitswahrscheinlichkeit näher zu untersuchen, als zufällig irgendein Element auszuwählen. Es ist davon auszugehen, dass die mit einer derart hohen Effektivität generierten Element-Vorschläge einem Anwender helfen können den Suchraum erheblich einzugrenzen. Im Umkehrschluss würde jenes bedeuten, dass der manuelle (Such-)Aufwand hierdurch gesenkt werden könnte.

Der wesentliche manuelle Aufwand für den Anwender bzw. den Experimentator bestand in der Prüfung der Element-Vorschläge hinsichtlich ihrer Korrektheit. Das fehlende Systemwissen eines Experten wurde mit der annotierten Original-Version als Entscheidungsgrundlage kompensiert. Alle hierbei als zunächst „nicht korrekt“ zu bewertenden Vorschläge, mussten hinsichtlich des Vollständigkeitskriteriums nachgeprüft werden, da nicht ausgeschlossen werden konnte, dass sie von den Entwicklern der Software übersehen wurden. Auch wenn die insgesamt benötigte Zeit von rund 2 Stunden und 12 Minuten (bereinigt: 1 h 56 min) nicht ohne Vorbehalt reproduzierbar und generalisiert ist, zeigt die Größenordnung dennoch das grundsätzliche Potential des LEADT-Konzepts.

Darüber hinaus wurde beobachtet, dass für über die Hälfte der generierten Element-Vorschläge eine manuelle Überprüfung hinsichtlich der Korrektheit eigentlich überflüssig ist, weil die berechnete (lokale) Zugehörigkeitswahrscheinlichkeit bei 100% liegt. Aufgrund dessen beträgt der tatsächliche manuelle Aufwand für den Anwender sogar weniger als eine Stunde. Trotz eines verbleibenden Fehlerrisikos, waren in dem Fall von MobileMedia solche lokal sicheren Empfehlungen immer korrekt. Jene Beobachtung zeigt, dass mit dem LEADT-Konzept in einigen Fällen ein wesentlicher Teil der Feature Mining Aufgabe vollautomatisch gelöst werden könnte ohne die Qualität dabei erheblich zu gefährden. Nicht zuletzt deswegen kann angenommen werden, dass LEADT einem Anwender durchaus helfen kann den Transitionsaufwand zu reduzieren.

In der MobileMedia Fallstudie wurde demonstriert, dass LEADT auch mit einem Minimum an menschlicher Intuition und Erfahrung einen wesentlichen Beitrag für das Feature Mining leisten kann. Es steht außer Frage, dass jede zusätzliche nützliche Information oder Leistung, die ein Anwender mit einbringen kann, die Ergebnisse noch einmal deutlich verbessern würde (vgl. *Kapitel 6.3.1*). Aufgrund dessen wird die Meinung vertreten,

dass das vorgestellte Konzept prinzipiell geeignet ist, um die Komplexität und den Aufwand einer Transition eines Systems hin zu einer annotierten SPL-Plattform entscheidend zu reduzieren.

## 6.4 Experimentelle Kritik

Im Folgenden werden einige Kritikpunkte hinsichtlich der *Wiederholbarkeit* und *Generalisierbarkeit* der erzielten Ergebnisse diskutiert.

Die *Wiederholbarkeit* oder auch *Reliabilität* besagt zu welchem Grad eine Untersuchung mit den selben Ergebnissen wiederholt werden kann. In dieser Fallstudie wurden alle durchgeführten Schritte und die hierbei gemessenen Daten detailliert beschrieben und dokumentiert. Äußere Rahmenbedingungen wurden weitestgehend minimiert. Es wurde zudem eine frei verfügbare Opensource-Applikation verwendet, welche als Software-Produktlinie entwickelt wurde. Alle Features wurden von den Entwicklern der Software markiert, sodass LEADT gegen eine vertrauenswürdige Basis verglichen werden konnte. Alle notwendigen Modifikationen der Software oder der Feature-Repräsentationen wurden transparent dargestellt und begründet. In der Auswertung der Fallstudie wurden Leistungsmetriken, Ergebnisse sowie dessen zugehörige Interpretationen explizit von einander abgegrenzt. Wobei sich in allen drei Bereichen um eine wertfreie und objektive Darstellung bemüht wurde. Mit Ausnahme der Untersuchung des Zeitaufwandes in *Kapitel 6.3.4,* sollte es ohne weiteres möglich sein die Ergebnisse zu reproduzieren.

Die Messung der benötigten Zeit bzw. des Zeitaufwands hängt von zahlreichen Rahmenfaktoren und Kenntnissen des Experimentators ab, welche im Rahmen der Fallstudie nicht kontrolliert wurden (vgl. *Kapitel 6.3.4*). Hierbei wurde allerdings auch nicht das Ziel verfolgt gesicherte Erkenntnisse abzuleiten, sondern vielmehr eine ungefähre Größenordnung für den benötigten Zeitaufwand zu vermitteln.

Der Hauptkritikpunkt, hinsichtlich der *Generalisierbarkeit* der Ergebnisse auf andere Kontexte, ergibt sich aufgrund der Verwendung einer Fallstudie als Evaluationsmethode. Mit MobileMedia wurde nur ein einziges System aus einer spezifischen Domäne gewählt, bei dem die Quellcodegröße sowie die Anzahl der Features als klein bis mittelgroß einzustufen ist. Aus theoretischen Gesichtspunkten spricht nichts dagegen, dass das grundsätzliche LEADT-Konzept ähnlich gute Ergebnisse auch in anderen Systemen und Domänen erzielen könnte. Allerdings können nur weitere, unterschiedliche Fallbeispiele jene Vermutung erhärten. Gegeben der zahlreichen, unterschiedlichen Arten von auftretenden Software-Systemen wird es aber vermutlich nie gelingen zweifelsfrei die Anwendbarkeit für *alle* Fälle zu belegen.

Ein weiterer wichtiger Kritikpunkt begründet sich in dem Schema nach dem die Suchanfragen in der Feature Lokalisierung formuliert wurden. Es kann nicht pauschal generalisiert werden, dass Feature-Namen allein immer zu korrekten Treffern führen (vgl. *Kapitel 3.1*). Solche Startelemente werden aber unbedingt benötigt, damit die Feature-Expansion erst überhaupt ansetzen kann. In dieser Fallstudie musste ein automatisierbares Schema vereinbart werden, welches frei von der Erfahrung oder Intuition eines Anwenders ist. Feature-Namen stellen hierbei eine Informationseinheit dar, welche immer verfügbar und daher unabhängig von einem Anwender sind.

## 6.5 Zusammenfassung

Einleitend werden in diesem Kapitel drei unterschiedliche Evaluationsmethoden diskutiert, mit denen prinzipiell eine Aussage zur Tauglichkeit und Leistungsfähigkeit von LEADT gemacht werden könnte. Es wird festgestellt, dass Aspekte der Nutzerfreundlichkeit und Bedienbarkeit bislang in der Entwicklung des Prototypen eine untergeordnete Rolle gespielt haben. Qualitative und quantitative Methoden mit Experten und Anwendern wären in diesem frühen Stadium daher noch stark von Bedienungshürden geprägt. Für die exemplarische Überprüfung bzw. Demonstration des grundsätzlichen Potentials wurde sich daher für eine Fallstudie entschieden.

Die primäre Zielsetzung von Feature Mining Werkzeugen begründet sich in der Aufwandsreduktion für den Anwender, bei gleich bleibender oder höheren Qualität der Plattform. In der Fallstudie wird die Erreichung dieses Ziels indirekt überprüft. Hierbei wird als Grundidee angenommen, dass der manuelle Anteil der Arbeit abnehmen wird, wenn Teile der Feature Mining Aufgabe mit einer hohen Qualität automatisiert bearbeitet werden können. Daher wird im Rahmen des Untersuchungsaufbaus ein Feature Mining Vorgehen definiert, welches rein auf (automatisch) generierten Feature-Element-Vorschlägen von LEADT basiert. Der Anwender bzw. in dem Fall der Experimentator darf keine Elemente zu der Feature-Repräsentation zuordnen, die nicht vorher explizit durch den Prototypen vorgeschlagen wurden. Dessen Aufgabe beschränkt sich lediglich auf die Bewertung der Korrektheit der einzelnen Vorschläge sowie gegebenenfalls die anknüpfende Annotierung der zugehörigen Elemente im Quellcode.

Als zu überführendes Testsystem wurde die von der Lancaster Universität entwickelte MobileMedia Applikation (7. Release) verwendet. Der wesentliche Vorteil dieser mittelgroßen Anwendung ist, dass sie als Software-Produktlinie entwickelt wurde und daher alle Feature-Repräsentationen im Quellcode explizit ausweist. Durch die Verfügbarkeit jener Vergleichsannotierungen konnte die erzielte Leistung von LEADT daher zuverlässig und glaubwürdig in einem Soll-Ist-Vergleich beurteilt werden.

Die Datenerhebung als solches wurde in diesem Kapitel nicht näher diskutiert. Der Leser wurde hierzu für eine umfassende Darstellung auf den *Anhang A* verwiesen. Der Fokus lag vielmehr in der Auswertung der Ergebnisse, die aus den gemessenen Daten abgeleitet werden konnten. Jenes erfolgte in den vier unterschiedlichen Bewertungsaspekten: Erzielte Vollständigkeit der Features, Effektivität der Element-Vorschläge, Automatisierungspotential sowie benötigter Zeitaufwand. Die spezifischen Untersuchungen untergliederten sich jeweils in Motivation für die Betrachtung, Metriken, Ergebnisse und Interpretation.

In *Tabelle 6.8* sind die Ergebnisse für die wichtigsten betrachteten Metriken zusammengefasst dargestellt. Hieraus lässt sich ablesen, dass die sieben untersuchten Features im Mittel mit einer Vollständigkeit von rund 82% und einer bereinigten Vorschlagseffektivität von ebenfalls rund 82% identifiziert wurden. Pro Feature betrug der manuelle bereinigte Zeitaufwand im Durchschnitt ca. 19 Minuten. Unter Ausschöpfung des Automatisierungspotentials in der Feature-Expansion ließe sich diese Zeit sogar auf 6 Minuten pro Feature verringern.

| | PLAY MUSIC | SMS TRANSFER | COPY MEDIA | FAVOURITES | COUNT / SORT | VIEW PHOTO | SMS. OR COPY. | Mittelwert | Standardabweichung |
|---|---|---|---|---|---|---|---|---|---|
| Vollständigkeitsgrad$_{Gesamt}$ | 86,22% | 98,63% | 86,69% | 41,67% | 86,21% | 74,36% | 96,60% | 81,48% | 17,85% |
| Bereinigte Vorschlagseffektivität | 84,51% | 93,37% | 66,13% | 80,49% | 86,96% | 94,44% | 65,00% | 81,56% | 11,06% |
| Ber. Automatisierungspotential | 66,20% | 72,89% | 53,23% | 56,10% | 50,00% | 68,89% | 47,50% | 59,26% | 9,24% |
| | | | | | | | | | |
| Bereinigter Zeitaufwand [min] | 28,2 | 33,2 | 12,2 | 8,2 | 9,2 | 18 | 7 | 18,8 | 10,3 |
| Ber. Tats. Zeitaufwand [min] | 9,6 | 9,1 | 5,8 | 3,7 | 4,7 | 5,6 | 3,7 | 6 | 2,4 |

Tabelle 6.8: Wesentliche Ergebnisse im Überblick

In dem Fazit der Auswertung wurde resümiert, dass LEADT auch mit einem Minimum an menschlicher Intuition und Erfahrung einen wesentlichen Beitrag für das Feature Mining einbringen kann. Aufgrund dessen wurde die Meinung angenommen, dass dieses Konzept prinzipiell geeignet ist, um die Komplexität und den Aufwand einer Transition eines Systems hin zu einer annotierten SPL-Plattform entscheidend zu reduzieren.

Die gemessenen Ergebnisse bzw. erzielten Erkenntnisse wurden abschließend im Rahmen einer experimentellen Kritik hinsichtlich ihrer Wiederholbarkeit und Generalisierbarkeit kurz diskutiert. Festzuhalten hiervon ist vor allem, dass es mit Ausnahme der Untersuchung des Zeitaufwandes ohne weiteres möglich sein sollte die Ergebnisse zu reproduzieren. Ein wesentlicher Kritikpunkt hinsichtlich der Generalisierbarkeit ergibt sich, weil nur ein einziges System aus einer spezifischen Domäne betrachtet wurde. Daher müssen weitere Fallbeispiele folgen, um die Ergebnisse auch unter anderen Rahmenbedingungen zu bestätigen.

# 7 Zusammenfassung und Ausblick

Mit Hilfe von Konzepten wie Subroutinen in den 1960ern, Modulen in den 1970ern, Objekten in den 1980ern, Komponenten in den 1990ern und schließlich Services in den 2000ern wurde im Software-Engineering immer wieder versucht das Leistungsvermögen und die Wirtschaftlichkeit von Software zu erhöhen. Diese Entwicklung wurde nicht zuletzt durch die immer wieder ansteigende Komplexität der zu entwickelnden und zu wartenden Anwendungen angetrieben [CN02, Vorwort]. Seit der Jahrtausendwende hat die *Software-Produktlinienentwicklung* einen enormen Auftrieb erfahren und etabliert sich allmählich als nächstes bedeutendes Paradigma [BKPS04, S. 265; LSR07, S. 3; SPK06].

Die *Software-Produktlinienentwicklung* verlangt ein Vorgehen mit organisierter Wiederverwendung für eine Gruppe von Softwaresystemen. Hierfür müssen alle nachgefragten *Features* eines ganzen Marktes bzw. einer *Domäne* im Voraus in einer strategisch geplanten *Plattform* realisiert werden. Aus jener können dann, je nach Kundenwunsch durch die Selektion von einzelnen Features aus einem *Feature-Modell*, spezifische *Software-Produktvarianten* bzw. ganze *-Produktlinien* mit geringem Aufwand erzeugt werden [AK09; BKPS04, S. 277ff.; CN02, S. 5ff.; LSR07, S. 314ff.; PBL05, S. 15; SEI10].

Die Modellierung und Umsetzung einer solchen vorausschauenden Produktlinie-Plattform ist nahezu unmöglich, wenn das Unternehmen noch keine Kenntnisse zu Markt, Kunden und technischen Realisierungsmöglichkeiten hat. In der Praxis bietet sich dieses neue Paradigma daher vor allem für Unternehmen an, welche bereits über entsprechende Erfahrungen durch die Entwicklung von eigenen Produkten verfügen [BKPS04, S. 137; BOS00; Joh06; PBL05, S. 394]. Eine vollständige Neuentwicklung einer Produktlinien-Plattform mit einem „modernen" Kompositionsansatz erfordert enorme Investitionen und birgt zahlreiche Risiken. Eine Realisierung der Plattform mit einem leichtgewichtigen Annotationsansatz besticht demgegenüber mit einer deutlich geringeren Adoptionshürde. Trotzt gewissen Abstrichen hinsichtlich der Softwarequalität und Architekturstabilität, bietet sich die letzt genannte Alternative vor allem als kurz- und mittelfristige (Zwischen-) Lösung an [BKPS04, S. 8; CK02; ES01; Joh06; Käs10; Kru01; MTW93; SE02; SEI10].

Auch wenn eine Überführung der bestehenden Systeme zu einer annotationsbasierten Plattform die vermeintlich einfachere Lösung darstellt, bleibt sie dennoch komplex und aufwändig. Es ist davon auszugehen, dass in dem Altsystem-Quellcode die zu annotierenden Features nur selten konsistent kommentiert oder modular repräsentiert sind

[LAL+10; MR05; MRB+05, OS02; SE02; WBP+03]. Aktuell muss die Zuordnung zwischen Features und ihren Repräsentationen meist manuell hergestellt werden. Gerade für umfangreiche Systeme, mit tausenden Quellcodezeilen sowie verstreuten und vermischten Feature-Repräsentationen, ist die Bewältigung jener Aufgabe nicht nur sehr zeitaufwändig, sondern auch fehlerbehaftet [WR07].

Mit dem Ziel der weiteren Senkung der Adoptionshürde wurde in dieser Arbeit daher vorgeschlagen jene Zuordnungs- bzw. Annotierungsaufgabe (semi-)automatisch zu lösen. Hierfür wurden zunächst wesentliche Teilaufgaben, Herausforderungen und Annahmen analysiert bzw. definiert. So wurde beispielsweise der gesamte Prozess in die drei Teilaktivitäten *Lokalisierung*, *Expansion* und *Dokumentation* aufgegliedert. Darüber hinaus wurde festgelegt, dass Feature-Repräsentationen *vollständig* erfasst werden müssen. Das heißt, nach dem Entfernen der Repräsentationen, müssen alle gültigen Programmvarianten fehlerfrei ausführbar sein, dürfen keine Rückschlüsse auf die Existenz des Features zulassen und auch keinen ungenutzten Quellcode hinterlassen. Jene Forderung impliziert in der Regel auch, dass Features im Quellcode *feingranular* abgebildet und annotiert werden müssen – sprich, bis auf der Ebene von lokalen Variablen und Ausdrücken innerhalb von Methoden. Zudem wurde vereinbart, dass die unterstützenden Werkzeuge eine *inkrementelle Transition* von einzelnen Features unterstützen und selbstverständlich hierbei einen *hohen Grad der Automatisierung* anstreben müssen.

Es wurde festgestellt, dass viele Forschungsgebiete existieren, welche sich einer verwandten Fragestellung annehmen. Nach dem aktuellen Kenntnisstand des Autors, verfolgt allerdings keiner dieser Themenbereiche das Ziel eine annotationsbasierte Software-Produktlinien-Plattform aufzubauen. So wird beispielsweise nie die Vollständigkeitsanforderung adressiert. Vor jenem Hintergrund wurde gemäß der herausgearbeiteten Kriterien ein neues Themen- bzw. Forschungsgebiet Namens *Feature Mining from Source Code,* oder abgekürzt *Feature Mining,* begründet.

Aus Schnittmengen mit verwandten Forschungsgebieten wurden anschließend zahlreiche Ansätze vorgestellt, welche für die Entwicklung von Feature Mining Lösungen wertvolle Erkenntnisse einbringen könnten. So wurden für die Feature-Lokalisierung, -Expansion und -Dokumentation jeweils unterschiedliche Publikationen zu Konzeptkategorien gruppiert und hinsichtlich ihrer aktuellen Eignung für das Feature Mining bewertet. Diese *Darstellung des aktuellen Stands der Technik* bietet interessierten Forschern einen facettenreichen Einstiegspunkt für mögliche eigene Arbeiten in diesem Bereich.

Aufbauend auf der Untersuchung der diversen Ansätze und Werkzeuge aus den verwandten Forschungsgebieten, wurde das *Location, Expansion and Documentation Tool – LEADT* vorgeschlagen und prototypisch als Eclipse Plug-in für die Programmiersprache Java implementiert. Mit diesem Lösungskonzept wird erstmalig die Möglichkeit geboten *alle Aufgaben des Feature Mining strukturiert und integriert zu bearbeiten*. Innerhalb einer ein-

heitlichen Umgebung können hierbei der Quellcode und das Feature-Modell verwaltet, zugehörige Features semiautomatisch in dem (Alt-)System annotiert und gültige Produktvarianten generiert werden.

In LEADT wird das Aufgabenspektrum der Lokalisierung und Dokumentation durch die Verwendung von bereits verfügbaren Werkzeugen abgedeckt. Für die Feature-Expansion konnte allerdings nicht vollständig auf fertige Werkzeuge von anderen zurückgegriffen werden, welche die Anforderungen des Feature Mining erfüllen. Infolgedessen wurde ein eigenes *vollständigkeitsfokussiertes, automatisiertes Expansionskonzept* entwickelt. Zu den wesentlichen Neuerungen zählt in diesem Zusammenhang die Verwendung von Domäneninformationen aus dem Feature-Modell, die Abstraktion des Quellcodes in einem feingranularen System-Modell sowie die Kombination von Feature-Element-Vorschlagsansätzen mit komplementären Stärken.

Anhand einer Fallstudie wurde die *Leistungsfähigkeit des implementierten Prototyps* exemplarisch *evaluiert*. In dieser wurden mit Hilfe von LEADT insgesamt sieben Features in der mittelgroßen MobileMedia Applikation lokalisiert, expandiert und dokumentiert. Hierbei wurden die Bewertungsaspekte *Vollständigkeit der Features*, *Effektivität der Element-Vorschläge*, *Automatisierungspotential* sowie *benötigter Zeitaufwand* betrachtet. Für jeden dieser Aspekte wurde kurz motiviert weshalb dieser zu untersuchen ist. Im Anschluss daran wurden konkrete Metriken definiert, erzielte Ergebnisse quantifiziert und interpretiert. Abschließend wurde resümiert, dass LEADT grundsätzlich in der Lage ist die Komplexität und den Aufwand einer Transition eines Systems hin zu einer annotationsbasierten Plattform entscheidend zu reduzieren. Jenes Ergebnis lässt sich allerdings bislang nicht ohne Einschränkungen generalisieren, da nur ein einziges System aus einer spezifischen Domäne betrachtet wurde.

## Ausblick

Im Rahmen dieser Diplomarbeit wurde der Forschungsbereich Feature Mining in erster Annäherung weitestgehend in der Breite durchgedrungen, um ein möglichst umfassendes Bild zu vermitteln. In zukünftigen Arbeiten könnten neue Einsichten oder Verbesserungen durch punktuelle Vertiefungen erzielt werden. Folgend werden einige solcher möglichen Ansatzpunkte kurz beschrieben.

Eine feinmaschigere Taxonomie, für die Bewertung von Ansätzen aus verwandten Problembereichen, könnte dabei helfen den aktuellen Stand der Technik genauer abzubilden. Hierdurch könnten beispielsweise weitere komplementäre Stärken von unterschiedlichen Konzepten leichter erkannt werden.

Die neu entwickelten Komponenten in LEADT sind aktuell nur prototypisch umgesetzt. Verbesserungen könnten in jedem Fall erzielt werden, wenn unter anderem folgende Bereiche weiter ausgebaut würden:

- Wie die Fallstudie gezeigt hat, wäre es lohnenswert in dem System-Modell zusätzlich Kontrollflussbeziehungen innerhalb von Fehlerbehandlungen sowie die Strukturierungseinheit Block zu erfassen. Für umfangreiche Systeme sollte es vor allem auch möglich sein das System-Modell in einer externen Datenbank zu verwalten.

- Es könnten weitere Beziehungen aus dem Feature-Modell identifiziert und verarbeitet werden, wie beispielsweise Notwendigkeit- und Ausschlussbeziehungen zwischen Gruppen von Features.

- Die aktuelle Implementierung des Textmuster-Erkennungsansatzes benutzt einen einfachen Algorithmus. Stattdessen wäre es ratsam, verstärkt auf fundierte Erkenntnisse aus dem Bereich der Computerlinguistik zurückzugreifen.

- Der Typsystem-Vorschlagsansatz setzt in der jetzigen Version nur zwei zentrale Prüfregeln um. Hier besteht, nicht zuletzt aufgrund der Vollständigkeitsforderung, Erweiterungsbedarf. Aktuell fehlen Regeln, welche z.B. Vererbungsbeziehungen, die Initialisierung von Konstanten oder auch Rückgabe-Anweisungen in Methoden betreffen.

- Im Rahmen der Feature-Expansion und -Dokumentation werden Verhaltensfehler bislang nicht behandelt. Daher werden Techniken gebraucht mit denen solche Fehlertypen zuverlässig erkannt und behoben werden können.

- Es sind Fälle vorstellbar in denen der Suchraum für ein Feature durch ein Ausschlussverfahren eingeschränkt werden kann. Aufgrund dessen sollte ebenso die Expansion von zugehörigen „Nicht-Features“ unterstützt werden.

Ein Problem, welches in dieser Arbeit nicht direkt behandelt wurde, ist die Bestimmung des Stoppkriteriums für die Expansion. In der Fallstudie erfolgte ein Abbruch wenn zehn Vorschläge hintereinander falsch waren. Bei einem solchen Vorgehen kann nicht ausgeschlossen werden, dass ein korrekter Vorschlag auf elfter oder zwölfter Position übersehen wird. Es werden daher Techniken benötigt, die einem Anwender helfen zu entscheiden, ab wann es nicht mehr lohnenswert ist weitere Element-Vorschläge zu prüfen.

Damit die Leistungsfähigkeit der einzelnen Feature Mining Lösungen sinnvoll eingeschätzt und untereinander verglichen werden kann, müssen aussagekräftige Benchmark-Systeme benannt werden. Forschungsarbeiten müssen hierbei zunächst genau prüfen, welche Anforderungen jene überhaupt erfüllen sollten. Darüber hinaus wird ein einheitliches Kennzahlensystem benötigt, mit dem nicht nur die Leistungsfähigkeit quantifiziert, sondern auch eine Aussage zur Vollständigkeit von Feature-Repräsentationen getroffen werden kann. Hierfür sind zwei unterschiedliche Fälle zu behandeln. Zum einen, das (Benchmark-)System ist bereits vollständig annotiert, sodass eine Soll-Ist Abweichung ermittelt werden kann. Zum anderen, das System ist nicht annotiert. Gerade der Letztge-

nannte Fall ist für die Praxis besonders relevant, da zuverlässige Qualitätsaussagen auch dann benötigt werden, wenn die tatsächlichen Feature-Repräsentationen unbekannt sind.

In dieser Arbeit wurde im Rahmen der MobileMedia-Fallstudie nur ein einziges System aus einer spezifischen Domäne betrachtet. Es müssen unbedingt weitere Fallbeispiele folgen, um die bislang erzielten Ergebnisse auch unter anderen Rahmenbedingungen zu bestätigen.

In *Kapitel 2.3.2* wurde festgestellt, dass langfristig gesehen eine Transformation der annotierten Features in physische Module vorteilhaft sein kann. Im Zusammenhang mit dem Feature Mining sollte daher perspektivisch auch über Lösungen nachgedacht werden, mit denen eine solche Restrukturierung bzw. Transformation (semi-)automatisch vollzogen werden kann. Als zentrale An- und Herausforderung gilt hierbei, dass trotz der internen Strukturveränderung das externe Verhalten bzw. die Semantik des Systems erhalten bleiben muss. Liu et al. [LBL06] bezeichnen einen solchen Vorgang als *Feature-oriented Refactoring* und definieren einige Grundlagen auf denen man zukünftig aufbauen könnte.

Diese Arbeit spannt ein neues Forschungsfeld auf und gibt viele mögliche Richtungen vor, an denen sich zukünftige Arbeiten orientieren können. Es bleibt zu hoffen, dass andere Forscher inspiriert durch diesen Beitrag einen Zugang zu der Feature Mining Thematik finden und verbesserte Lösungsansätze einbringen werden. Ein solches Engagement würde in jedem Fall dazu beitragen die Software-Produktlinienentwicklung ein Schritt näher an die industrielle Praxis heranzuführen.

# A Erhobene Daten und Messwerte

In diesem Anhang werden erhobene Daten und Messwerte aus der Feature Mining Fallstudie aus dem *6. Kapitel* dargestellt. Hierdurch wird die Möglichkeit geboten das durchgeführte Vorgehen aus *Kapitel 6.2.1* besser nachvollziehen zu können. Jene Werte dienen zudem als Grundlage für die Berechnung der Metriken, welche in *Kapitel 6.3* im Rahmen der Fallstudien-Auswertung verwendet werden.

Gemäß des 1. Schritts des festgelegten Vorgehens in *Kapitel 6.2.1* wurden die Teilmengen- und Obermengen-Features identifiziert, um festzulegen für welche Features Startfragmente aus der Lokalisierung benötigt werden. Jene Analyse hat ergeben, dass PLAY MUSIC, SMS TRANSFER, COPY MEDIA, FAVOURITES sowie COUNT/SORT *Teilmengen-Features* sind. VIEW PHOTO und SMS TRANSFER OR COPY MEDIA gehören hingegen zu den *Obermengen-Features*. Für VIEW PHOTO können Startfragmente aus der Repräsentation von SMS TRANSFER abgeleitet werden. SMS TRANSFER OR COPY MEDIA wird wiederum durch bekannte Elemente aus SMS TRANSFER sowie COPY MEDIA gestützt. Gemäß der hier genannten Reihenfolge wurden dann die übrigen Schritte des Feature Mining Vorgangs durchgeführt. Im Folgenden werden die dabei erhobenen Daten und Messwerte dargestellt.

## A.1 Teilmengen-Features

In diesem Kapitel wird einleitend für jedes Teilmengen-Feature angegeben, welches *Startelement* im Rahmen der Feature-Lokalisierung identifiziert und ausgewählt wurde (vgl. *Kapitel 6.2.1, Schritt 2a*).

Anschließend wird jeweils ein so genanntes *Expansionsprotokoll* dargestellt. Jenes spiegelt das Vorgehen von *Schritt 3* in *Kapitel 6.2.1* wider. Entlang der x-Achse sind die einzelnen (Top-)Element-Vorschläge abgetragen, die hinsichtlich ihrer Feature-Zugehörigkeit geprüft wurden. Die Farbe eines jeden Datenpunkts drückt hierbei aus, ob der Element-Vorschlag korrekt (Original-Feature-Element bzw. Zusatz-Feature-Element) oder nicht korrekt (Nicht-Feature-Element) war. Die Einordnung der Datenpunkte entlang der y-Achse bildet den Relevanzwert ab, mit dem das jeweilige Element vorgeschlagen wurde. Darüber hinaus lässt sich in dem Expansionsprotokoll die Zunahme der insgesamt identifizierten Feature-Elemente entnehmen. Ein Element-Vorschlag kann z.B. auch eine ganze

Methode, Klasse oder sogar eine Kompilierungseinheit sein, die weitere Schlüssel-Elemente umschließt. Daher ist die Anzahl der identifizierten Feature-Elemente in der Regel deutlich größer als die Anzahl der Element-Vorschläge. Die Expansionsprotokolle sind derart skaliert, dass die obere, rechte Ecke das Maximum an identifizierbaren Feature-Elementen repräsentiert.

Während der durchgeführten Feature-Expansion wurden in einigen Fällen Elemente identifiziert, welche laut des geforderten Vollständigkeitskriteriums zu einem Feature zugeordnet werden müssten (vgl. *Kapitel 3.1*), aber in der originalen MobileMedia Applikation nicht als solche markiert sind. In eindeutig begründbaren Fällen wurden entsprechende Elemente *zusätzlich* mit in die Feature-Repräsentation aufgenommen. Zur Wahrung der Nachvollziehbarkeit und Reproduzierbarkeit der erzielten Ergebnisse werden alle solche *Zusatz-Feature-Elemente* sowie die zugehörigen Begründungen für jedes Feature explizit ausgewiesen.

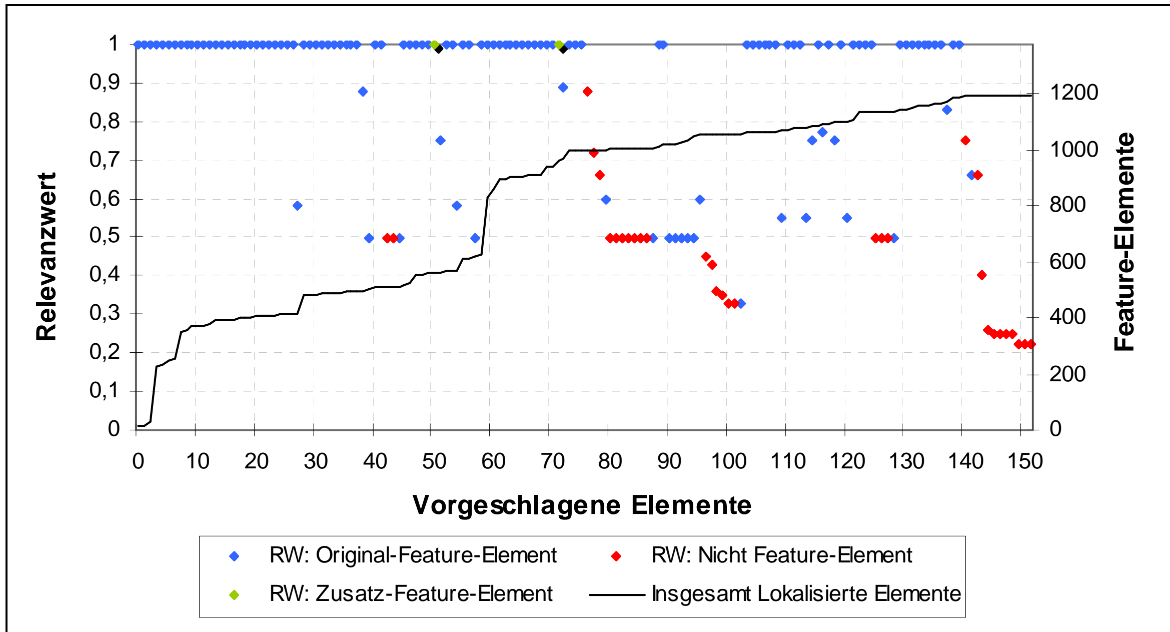
Abschließend sind für jedes Feature *Messwerte* zusammengefasst, welche für die Auswertung in *Kapitel 6.3* benötigt werden. Zum einen ist hier, in der Granularität der Schlüssel-Element-Typen erkennbar (vgl. *Kapitel 5.2.1*), wie viele Elemente zu der Feature-Repräsentation gehören (bestehend aus Original- und Zusatz-Elementen) und wie viele hiervon lokalisiert, expandiert bzw. in der Summe identifiziert wurden. Zum anderen lässt sich die Anzahl von korrekten (Original/Zusatz-)Element-Vorschlägen, die Anzahl der nicht korrekten Vorschläge sowie die Summe aus all diesen ablesen. Darüber hinaus ist abgebildet, wie viele Element-Vorschläge einen (extremen) Relevanzwert von 0 oder 1 hatten. In dem jeweils letzten Block sind die Zeiten dargestellt, welche der Experimentator für die Bearbeitung der Feature-Lokalisierung und -Expansion (mit und ohne Behandlung des Stoppkriteriums bzw. der letzten zehn falschen Vorschläge) benötigt hat.

### A.1.1 Play Music

#### Startelement aus der Feature-Lokalisierung

| **Ort** | Klasse: MusicPlayController | **Typ** | METHODE |
|---|---|---|---|
| **Element** | **public** MusicPlayController (MainUIMidlet midlet, AlbumData albumData, List albumListScreen, PlayMediaScreen pmscreen) {<br>**super**(midlet, albumData, albumListScreen);<br>**this**.pmscreen = pmscreen;<br>} | | |

## Expansionsprotokoll




## Zusatz-Feature-Elemente

| **Nummer** | 51 | **Ort** | Klasse - MediaController |
|---|---|---|---|
| **Typ** | IMPORT | **Element** | java.io.InputStream |
| **Grund**: | nur Zugriffe aus PLAY MUSIC | | |

| **Nummer** | 72 | **Ort** | Klasse - AlbumData |
|---|---|---|---|
| **Typ** | METHODE | **Element** | MediaData[] loadMediaDataFromRMS(String recordName) {…} |
| **Grund**: | nur Zugriffe aus PLAY MUSIC | | |

## Messwerte

| | **#Originale Feature-Elemente** | **#Zusätzliche Feature-Elemente** | #Feature-Elemente | **#Lokalisierte Feature-Elemente** | **#Expandierte Feature-Elemente** | #Identifizierte Feature-Elemente |
|---|---|---|---|---|---|---|
| **KOMPILIERUNGSEINH.** | 8 | | 8 | | 7 | 7 |
| **FELD** | **27** | | **27** | | **23** | **23** |
| FELD REFERENZ | 115 | 1 | 116 | 1 | 98 | 99 |
| IMPORT | 66 | 1 | 67 | | 61 | 61 |
| **LOKALE VARIABLE** | **98** | **1** | **99** | **4** | **84** | **88** |
| LOK. VARIABLEN REF. | 112 | | 112 | 1 | 102 | 103 |
| **METHODE** | **34** | **1** | **35** | **1** | **26** | **27** |
| METHODEN REFERENZ | 246 | 1 | 247 | 1 | 209 | 210 |
| PARAMETER REF. | 328 | 1 | 329 | 3 | 276 | 279 |
| **TYP** | **8** | | **8** | | **7** | **7** |
| TYP REFERENZ | 333 | 5 | 338 | 4 | 287 | 291 |
| | **1375** | **11** | **1386** | **15** | **1180** | **1195** |

| #Korrekte Original-Element-Vorschläge | 118 |
|---|---|
| #Korrekte Zusatz-Element-Vorschläge | 2 |
| **#Korrekte Element-Vorschläge** | **120** |
| #Nicht-Korrekte Element-Vorschläge | 32 |
| **#Alle Element-Vorschläge** | **152** |

| **#Sichere Element-Vorschläge (0.0)** | **0** |
|---|---|
| **#Sichere Element-Vorschläge (1.0)** | **94** |

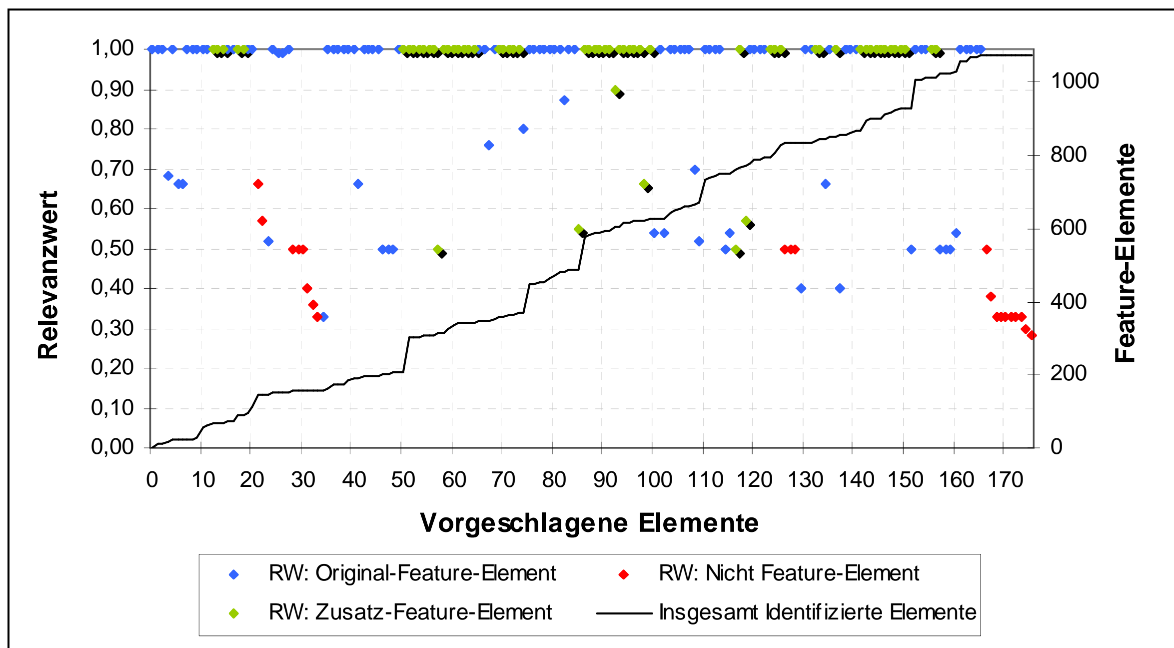
| **Benötigte Zeit für Feature-Lokalisierung (in min)** | **0,16** |
|---|---|
| **Benötigte Zeit für Feature-Expansion (in min)** | **30** |
| **Benötigte Zeit für Feature-Expansion ohne Stoppkriterium (in min)** | **28** |

## A.1.2 SMS Transfer

### Startelement aus der Feature-Lokalisierung

| **Ort** | Klasse: PhotoViewScreen | **Typ** | METHODE |
|---|---|---|---|
| **Element** | **public boolean** isFromSMS() {<br>return fromSMS;<br>} | | |

### Expansionsprotokoll



## Zusatz-Feature-Elemente

| **Nummer** | 12 | **Ort** | Klasse - AddMediaToAlbum |
|---|---|---|---|
| **Typ** | IMPORT | **Element** | javax.microedition.lcdui.Image |
| **Grund**: | nur Zugriffe aus SMS TRANSFER | | |

| **Nummer** | 13 | **Ort** | Klasse - PhotoViewController |
|---|---|---|---|
| **Typ** | IMPORT | **Element** | javax.microedition.lcdui.Image |
| **Grund**: | nur Zugriffe aus SMS TRANSFER | | |

| **Nummer** | 14 | **Ort** | Klasse - PhotoViewController; Methode - boolean handleCommand(Command c) |
|---|---|---|---|
| **Typ** | LOK. VAR. | **Element** | String photoname |
| **Grund**: | nur Zugriffe aus SMS TRANSFER | | |

| **Nummer** | 17 | **Ort** | Klasse - AlbumData |
|---|---|---|---|
| **Typ** | IMPORT | **Element** | javax.microedition.lcdui.Image |
| **Grund**: | nur Zugriffe aus SMS TRANSFER | | |

| **Nummer** | 18 | **Ort** | Klasse - AlbumData |
|---|---|---|---|
| **Typ** | METHODE | **Element** | byte[] loadMediaBytesFromRMS(String recordName, int recordId) {…} |
| **Grund**: | nur Zugriffe aus SMS TRANSFER | | |

| **Bereich** | 51-92 | **Ort** | Kompilierungseinheit - SMSMessaging.java |
|---|---|---|---|
| **Typ** | diverse | **Element** | Elemente innerhalb von SMSMessaging, zum Schluss: SMSMessaging |
| **Grund**: | nur Zugriffe aus SMS TRANSFER; In SourceForge Version gehört SMSMessaging zu dem Feature | | |

| **Bereich** | 93-100 | **Ort** | Kompilierungseinheit - BaseMessaging.java |
|---|---|---|---|
| **Typ** | diverse | **Element** | Elemente innerhalb von BaseMessaging, zum Schluss: BaseMessaging |
| **Grund**: | nur Zugriffe aus SMS TRANSFER; In SourceForge Version gehört BaseMessaging zu dem Feature | | |

| **Bereich** | 116-125 | **Ort** | Kompilierungseinheit - SmsSenderThread.java |
|---|---|---|---|
| **Typ** | diverse | **Element** | Elemente innerhalb von SmsSenderThread, zum Schluss: SmsSenderThre. |
| **Grund**: | nur Zugriffe aus SMS TRANSFER; In SourceForge Version gehört SmsSenderThread zu dem Feature | | |

| **Bereich** | 133-156 | **Ort** | Kompilierungseinheit - NetworkScreen.java |
|---|---|---|---|
| **Typ** | diverse | **Element** | Elemente innerhalb von NetworkScreen, zum Schluss: NetworkScreen |
| **Grund**: | nur Zugriffe aus SMS TRANSFER; In SourceForge Version gehört NetworkScreen zu dem Feature | | |

## Messwerte

| | **#Originale Feature-Elemente** | **#Zusätzliche Feature-Elemente** | #Feature-Elemente | **#Lokalisierte Feature-Elemente** | **#Expandierte Feature-Elemente** | #Identifizierte Feature-Elemente |
|---|---|---|---|---|---|---|
| **KOMPILIERUNGSEINH.** | 3 | 4 | 7 | | 7 | 7 |
| **FELD** | **15** | **15** | **30** | | **29** | **29** |
| FELD REFERENZ | 44 | 71 | 115 | 1 | 112 | 113 |
| IMPORT | 32 | 19 | 51 | | 51 | 51 |
| **LOKALE VARIABLE** | **60** | **36** | **96** | | **96** | **96** |
| LOK. VARIABLEN REF. | 80 | 41 | 121 | | 121 | 121 |
| **METHODE** | **15** | **27** | **42** | **1** | **41** | **42** |

| | | | | | | |
|---|---|---|---|---|---|---|
| METHODEN REFERENZ | 117 | 54 | 171 | | 168 | 168 |
| PARAMETER REF. | 132 | 47 | 179 | | 174 | 174 |
| **TYP** | **3** | **4** | **7** | | **7** | **7** |
| TYP REFERENZ | 159 | 113 | 272 | | 268 | 268 |
| | **660** | **431** | **1091** | **2** | 1074 | 1076 |

| | |
|---|---|
| #Korrekte Original-Element-Vorschläge | 94 |
| #Korrekte Zusatz-Element-Vorschläge | 61 |
| **#Korrekte Element-Vorschläge** | **155** |
| #Nicht-Korrekte Element-Vorschläge | 21 |
| **#Alle Element-Vorschläge** | **176** |

| | |
|---|---|
| **#Sichere Element-Vorschläge (0.0)** | **0** |
| **#Sichere Element-Vorschläge (1.0)** | **121** |

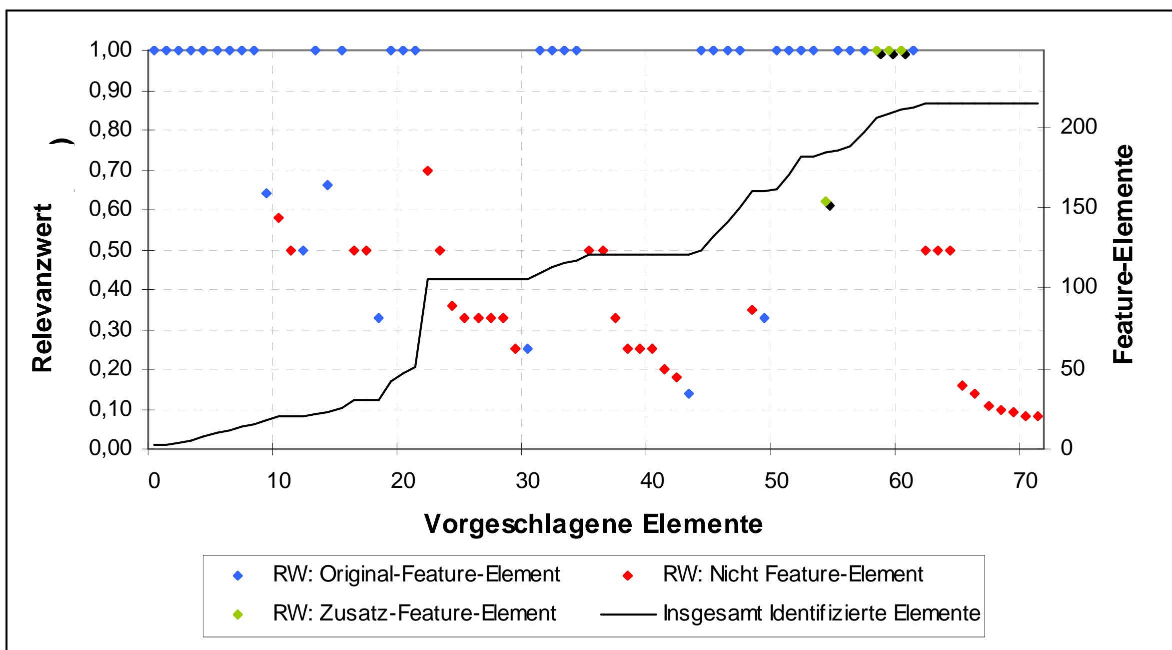
| | |
|---|---|
| **Benötigte Zeit für Feature-Lokalisierung (in min)** | **0,16** |
| **Benötigte Zeit für Feature-Expansion (in min)** | **35** |
| **Benötigte Zeit für Feature-Expansion ohne Stoppkriterium (in min)** | **33** |

## A.1.3 Copy Media

### Startelement aus der Feature-Lokalisierung

| **Ort** | Klasse: MusicPlayController | **Typ** | FELD |
|---|---|---|---|
| **Element** | **private String** mediaName; | | |

### Expansionsprotokoll



## Zusatz-Feature-Elemente

| Nummer | 54 | Ort | Klasse - MusicPlayController |
|---|---|---|---|
| Typ | IMPORT | Element | javax.microedition.lcdui.Alert |
| Grund: | nur Zugriffe aus COPY MEDIA | | |

| Nummer | 58 | Ort | Klasse - MusicPlayController |
|---|---|---|---|
| Typ | IMPORT | Element | javax.microedition.lcdui.AlertType |
| Grund: | nur Zugriffe aus COPY MEDIA | | |

| Nummer | 59 | Ort | Klasse - MusicPlayController |
|---|---|---|---|
| Typ | IMPORT | Element | javax.microedition.rms.RecordStoreFullException |
| Grund: | nur Zugriffe aus COPY MEDIA | | |

| Nummer | 60 | Ort | Klasse - MusicPlayController |
|---|---|---|---|
| Typ | IMPORT | Element | javax.microedition.lcdui.Display |
| Grund: | nur Zugriffe aus COPY MEDIA | | |

## Messwerte

| | **#Originale Feature-Elemente** | **#Zusätzliche Feature-Elemente** | #Feature-Elemente | **#Lokalisierte Feature-Elemente** | **#Expandierte Feature-Elemente** | #Identifizierte Feature-Elemente |
|---|---|---|---|---|---|---|
| **KOMPILIERUNGSEINH.** | | | | | | |
| **FELD** | **2** | | **2** | **1** | | **1** |
| FELD REFERENZ | 20 | | 20 | | 17 | 17 |
| IMPORT | | 4 | 4 | | 4 | 4 |
| **LOKALE VARIABLE** | **19** | | **19** | | **18** | **18** |
| LOK. VARIABLEN REF. | 22 | | 22 | | 20 | 20 |
| **METHODE** | **4** | | **4** | | **4** | **4** |
| METHODEN REFERENZ | 54 | | 54 | | 44 | 44 |
| PARAMETER REF. | 66 | | 66 | | 57 | 57 |
| **TYP** | | | | | | |
| TYP REFERENZ | 53 | 4 | 57 | 1 | 49 | 50 |
| | **240** | **8** | **248** | **2** | **213** | **215** |

| | |
|---|---|
| #Korrekte Original-Element-Vorschläge | 37 |
| #Korrekte Zusatz-Element-Vorschläge | 4 |
| **#Korrekte Element-Vorschläge** | **41** |
| #Nicht-Korrekte Element-Vorschläge | 31 |
| **#Alle Element-Vorschläge** | **72** |

| | |
|---|---|
| **#Sichere Element-Vorschläge (0.0)** | **0** |
| **#Sichere Element-Vorschläge (1.0)** | **33** |

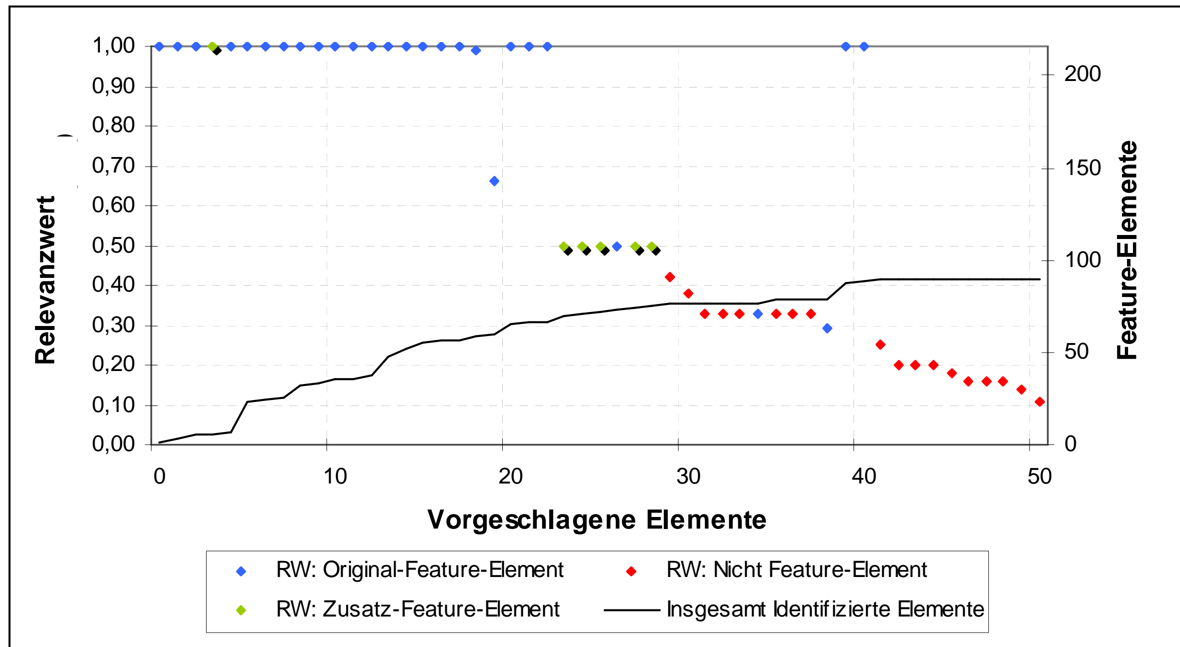
| | |
|---|---|
| **Benötigte Zeit für Feature-Lokalisierung (in min)** | **0,16** |
| **Benötigte Zeit für Feature-Expansion (in min)** | **14** |
| **Benötigte Zeit für Feature-Expansion ohne Stoppkriterium (in min)** | **12** |

## A.1.4 Favourites

### Startelement aus der Feature-Lokalisierung

| **Ort** | Klasse: MediaData | **Typ** | FELD |
|---|---|---|---|
| **Element** | **private boolean** favorite = **false**; | | |

### Expansionsprotokoll




### Zusatz-Feature-Elemente

| **Nummer** | 23 | **Ort** | Klasse – MediaListController; Methode - void showMediaList(String recordName, boolean sort, boolean favorite) |
|---|---|---|---|
| **Typ** | LOK. VAR. | **Element** | boolean favorite |
| **Grund**: | nur Zugriffe aus FAVOURITES; Namensübereinstimmung | | |

| **Bereich** | 24-25 | **Ort** | Klasse – MediaController; Methode - boolean goToPreviousScreen() |
|---|---|---|---|
| **Typ** | PARAM. REF. | **Element** | showMediaList(getCurrentStoreName(), false, false); |
| **Grund**: | nur Zugriffe aus FAVOURITES | | |

| **Nummer** | 27 | **Ort** | Klasse – MediaController; boolean handleCommand(Command command) |
|---|---|---|---|
| **Typ** | PARAM. REF. | **Element** | showMediaList(getCurrentStoreName(), false, false) |
| **Grund**: | nur Zugriffe aus FAVOURITES | | |

| **Nummer** | 28 | **Ort** | Klasse – MediaListController; boolean handleCommand(Command command) |
|---|---|---|---|
| **Typ** | PARAM. REF. | **Element** | showMediaList(getCurrentStoreName(), false, false) |
| **Grund**: | nur Zugriffe aus FAVOURITES | | |

### Messwerte

| | #Originale Feature-Elemente | #Zusätzliche Feature-Elemente | #Feature-Elemente | #Lokalisierte Feature-Elemente | #Expandierte Feature-Elemente | #Identifizierte Feature-Elemente |
|---|---|---|---|---|---|---|
| **KOMPILIERUNGSEINH.** | | | | | | |
| **FELD** | **3** | | **3** | **1** | **1** | **2** |
| FELD REFERENZ | 17 | | 17 | | 6 | 6 |
| IMPORT | | | | | | |
| **LOKALE VARIABLE** | **10** | **1** | **11** | | **5** | **5** |
| LOK. VARIABLEN REF. | 35 | | 35 | | 22 | 22 |
| **METHODE** | **3** | | **3** | | **3** | **3** |
| METHODEN REFERENZ | 44 | | 44 | | 20 | 20 |
| PARAMETER REF. | 63 | 5 | 68 | | 26 | 26 |
| **TYP** | | | | | | |
| TYP REFERENZ | 35 | | 35 | | 6 | 6 |
| | **210** | **6** | **216** | **1** | **89** | **90** |

| | |
|---|---|
| #Korrekte Original-Element-Vorschläge | 28 |
| #Korrekte Zusatz-Element-Vorschläge | 5 |
| **#Korrekte Element-Vorschläge** | **33** |
| #Nicht-Korrekte Element-Vorschläge | 18 |
| **#Alle Element-Vorschläge** | **51** |

| | |
|---|---|
| **#Sichere Element-Vorschläge (0.0)** | **0** |
| **#Sichere Element-Vorschläge (1.0)** | **23** |

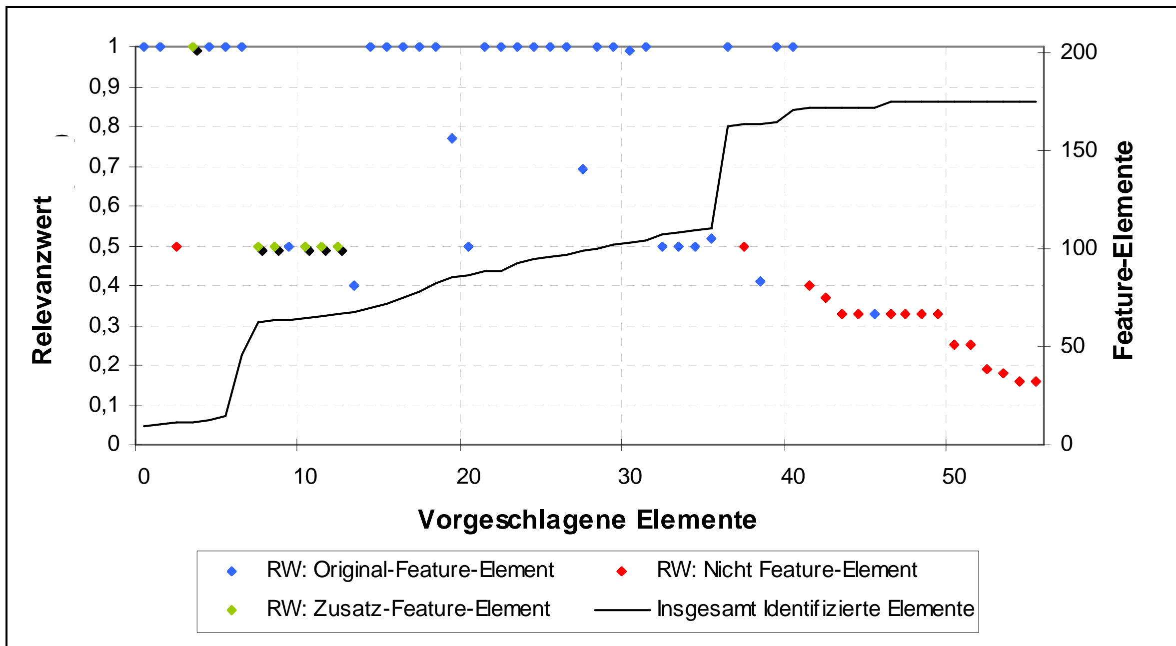
| | |
|---|---|
| **Benötigte Zeit für Feature-Lokalisierung (in min)** | **0,16** |
| **Benötigte Zeit für Feature-Expansion (in min)** | **10** |
| **Benötigte Zeit für Feature-Expansion ohne Stoppkriterium (in min)** | **8** |

## A.1.5 Count/Sort

### Startelement aus der Feature-Lokalisierung

| **Ort** | Klasse: MediaListScreen | **Typ** | FELD |
|---|---|---|---|
| **Element** | **public static final** Command sortCommand = **new** Command("Sort by Views", Command.ITEM, 1); | | |

## Expansionsprotokoll




## Zusatz-Feature-Elemente

| **Nummer** | 4 | **Ort** | Klasse – MediaListController; Methode - void showMediaList(String record-Name, boolean sort, boolean favorite) |
|---|---|---|---|
| **Typ** | LOK. VAR. | **Element** | boolean sort |
| **Grund**: | nur Zugriffe aus COUNT/SORT; Namensübereinstimmung | | |

| **Bereich** | 8-9 | **Ort** | Klasse – MediaController; Methode - boolean goToPreviousScreen() |
|---|---|---|---|
| **Typ** | PARAM. REF. | **Element** | showMediaList(getCurrentStoreName(), false, false); |
| **Grund**: | nur Zugriffe aus COUNT/SORT | | |

| **Nummer** | 11 | **Ort** | Klasse – MediaController; boolean handleCommand(Command command) |
|---|---|---|---|
| **Typ** | PARAM. REF. | **Element** | showMediaList(getCurrentStoreName(), false, false) |
| **Grund**: | nur Zugriffe aus COUNT/SORT | | |

| **Nummer** | 12 | **Ort** | Klasse – MediaListController; boolean handleCommand(Command com-mand) |
|---|---|---|---|
| **Typ** | PARAM. REF. | **Element** | showMediaList(getCurrentStoreName(), false, false) |
| **Grund**: | nur Zugriffe aus COUNT/SORT | | |

| **Nummer** | 13 | **Ort** | Klasse – MediaController; boolean handleCommand(Command command) |
|---|---|---|---|
| **Typ** | PARAM. REF. | **Element** | showMediaList(getCurrentStoreName(), true, false) |
| **Grund**: | nur Zugriffe aus COUNT/SORT | | |

### Messwerte

| | #Originale Feature-Elemente | #Zusätzliche Feature-Elemente | #Feature-Elemente | #Lokalisierte Feature-Elemente | #Expandierte Feature-Elemente | #Identifizierte Feature-Elemente |
|---|---|---|---|---|---|---|
| **KOMPILIERUNGSEINH.** | | | | | | |
| **FELD** | **2** | | **2** | **1** | **1** | **2** |
| FELD REFERENZ | 16 | | 16 | 1 | 10 | 11 |
| IMPORT | | | | | | |
| **LOKALE VARIABLE** | **18** | **1** | **19** | | **18** | **18** |
| LOK. VARIABLEN REF. | 42 | | 42 | | 35 | 35 |
| **METHODE** | **6** | | **6** | | **6** | **6** |
| METHODEN REFERENZ | 37 | | 37 | 1 | 29 | 30 |
| PARAMETER REF. | 47 | 5 | 52 | 3 | 44 | 47 |
| **TYP** | | | | | | |
| TYP REFERENZ | 29 | | 29 | 3 | 23 | 26 |
| | **197** | **6** | **203** | **9** | **166** | **175** |

| | |
|---|---|
| #Korrekte Original-Element-Vorschläge | 34 |
| #Korrekte Zusatz-Element-Vorschläge | 6 |
| **#Korrekte Element-Vorschläge** | **40** |
| #Nicht-Korrekte Element-Vorschläge | 16 |
| **#Alle Element-Vorschläge** | **56** |

| | |
|---|---|
| **#Sichere Element-Vorschläge (0.0)** | **0** |
| **#Sichere Element-Vorschläge (1.0)** | **23** |

| | |
|---|---|
| **Benötigte Zeit für Feature-Lokalisierung (in min)** | **0,16** |
| **Benötigte Zeit für Feature-Expansion (in min)** | **11** |
| **Benötigte Zeit für Feature-Expansion ohne Stoppkriterium (in min)** | **9** |

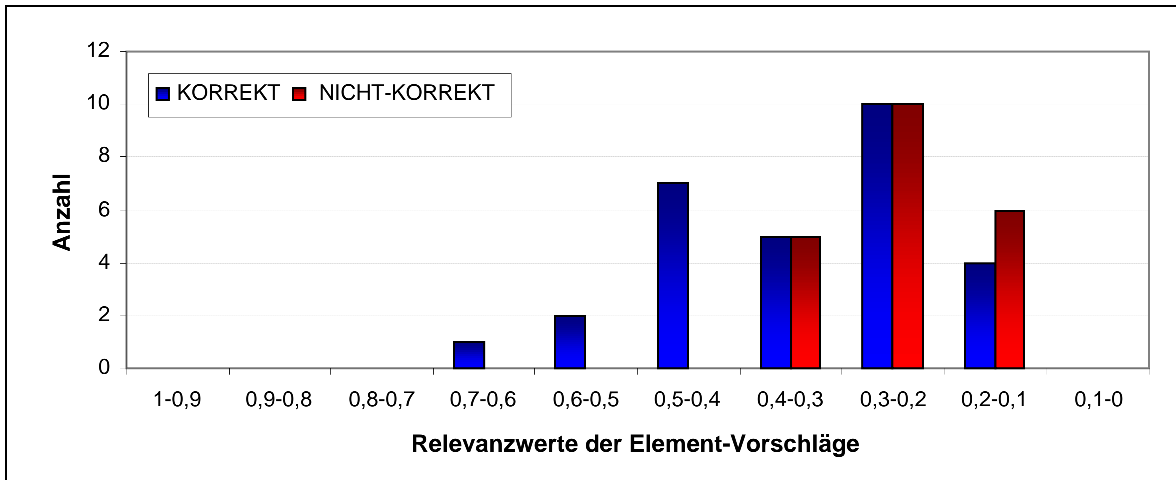
## A.2 Obermengen-Features

Für die in diesem Kapitel dargestellten Obermengen-Features musste keine Lokalisierung von Start-Elementen mit Hilfe der IR-Suche erfolgen. Aufgrund von Abhängigkeiten zu bereits identifizierten Teilmengen-Features, wurden Element-Vorschläge direkt von LEADT generiert (vgl. *Kapitel 6.2.1, Schritt 2b*). Im Folgenden wird für jedes Obermengen-Feature die Verteilung der Element-Vorschläge dargestellt, mit der die Feature-Expansion begonnen wurde.

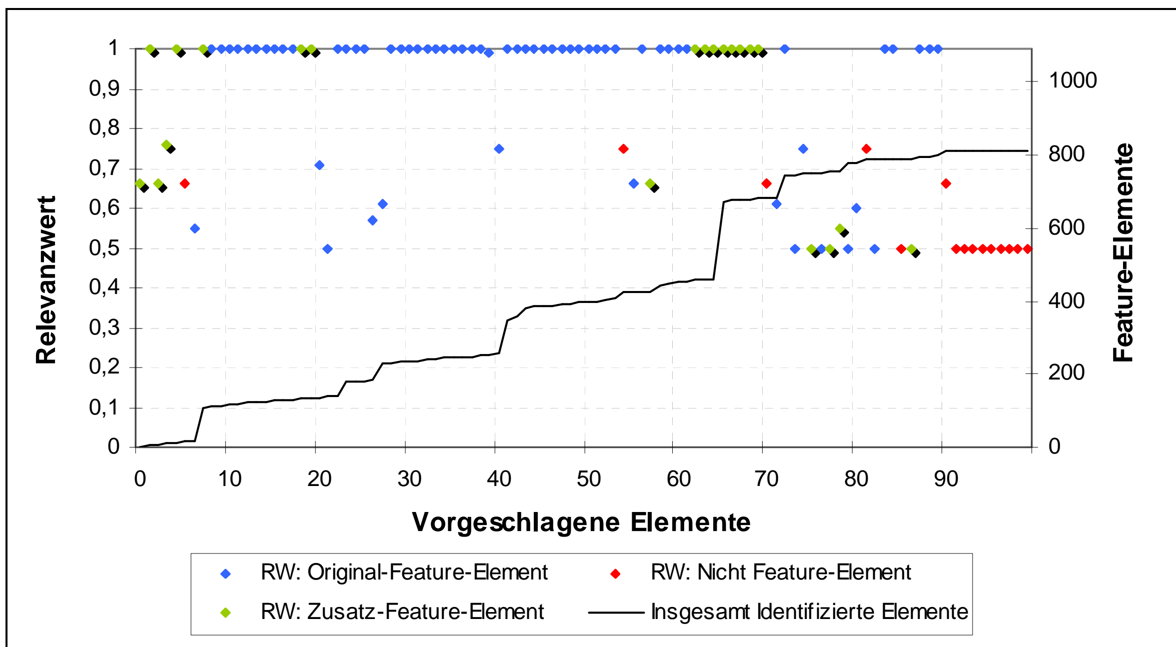
Analog zu den Teilmengen-Features werden im Anschluss jeweils die *Expansionsprotokolle*, *Zusatz-Feature-Elemente* sowie die für die Auswertung notwendigen *Messwerte* dargestellt (vgl. *Kapitel A.1*).

### A.2.1 View Photos

#### Verfügbare Element-Vorschläge – abgeleitet aus SMS TRANSFER




#### Expansionsprotokoll




#### Zusatz-Feature-Elemente

| **Bereich** | 1-7; 57-78 | **Ort** | Kompilierungseinheit - PhotoViewController.java |
|---|---|---|---|
| **Typ** | diverse | **Element** | Elemente innerhalb von PhotoViewController, zum Schluss: PhotoView-Controller |
| **Grund**: | Namensübereinstimmung; Der PhotoViewController ist mit SMS TRANSFER \|\| COPY MEDIA annotiert, wird aber nur aus SMS TRANSFER und VIEW PHOTO referenziert. In Varianten die COPY MEDIA enthalten aber nicht SMS TRANSFER bzw. VIEW PHOTO wäre die gesamte Klasse ungenutzter Quellcode. Die enthaltene Importanweisung *ubc.midp.mobilephoto.core.ui.screens.PhotoViewScreen* würde in diesem Fall sogar einen Typfehler verursachen. In der SourceForge Version gehört jene Importanweisung zu VIEW PHOTO. | | |

| **Nummer** | 18 | **Ort** | Klasse - Constants |
|---|---|---|---|
| **Typ** | FELD | **Element** | **public static final int** SCREEN_WIDTH = 176 |
| **Grund**: | nur Zugriffe aus VIEW PHOTO | | |

| **Nummer** | 19 | **Ort** | Klasse - Constants |
|---|---|---|---|
| **Typ** | FELD | **Element** | **public static final int** SCREEN_HEIGHT = 205 |
| **Grund**: | nur Zugriffe aus VIEW PHOTO | | |

| **Nummer** | 86 | **Ort** | Klasse - MediaListScreen |
|---|---|---|---|
| **Typ** | FELD | **Element** | **public static final int** SHOWPHOTO = 1 |
| **Grund**: | nur Zugriffe aus VIEW PHOTO; Namensübereinstimmung | | |

## Messwerte

| | **#Originale Feature-Elemente** | **#Zusätzliche Feature-Elemente** | #Feature-Elemente | **#Lokalisierte Feature-Elemente** | **#Expandierte Feature-Elemente** | #Identifizierte Feature-Elemente |
|---|---|---|---|---|---|---|
| **KOMPILIERUNGSEINH.** | 5 | 1 | 6 | | 4 | 4 |
| **FELD** | **17** | **4** | **21** | | **12** | **12** |
| FELD REFERENZ | 66 | 10 | 76 | | 56 | 56 |
| IMPORT | 35 | 17 | 52 | | 40 | 40 |
| **LOKALE VARIABLE** | **68** | **19** | **87** | | **65** | **65** |
| LOK. VARIABLEN REF. | 81 | 22 | 103 | | 82 | 82 |
| **METHODE** | **26** | **2** | **28** | | **18** | **18** |
| METHODEN REFERENZ | 126 | 64 | 190 | | 141 | 141 |
| PARAMETER REF. | 182 | 70 | 252 | | 186 | 186 |
| **TYP** | **5** | **1** | **6** | | **4** | **4** |
| TYP REFERENZ | 197 | 74 | 271 | | 204 | 204 |
| | **808** | **284** | **1092** | | **812** | **812** |

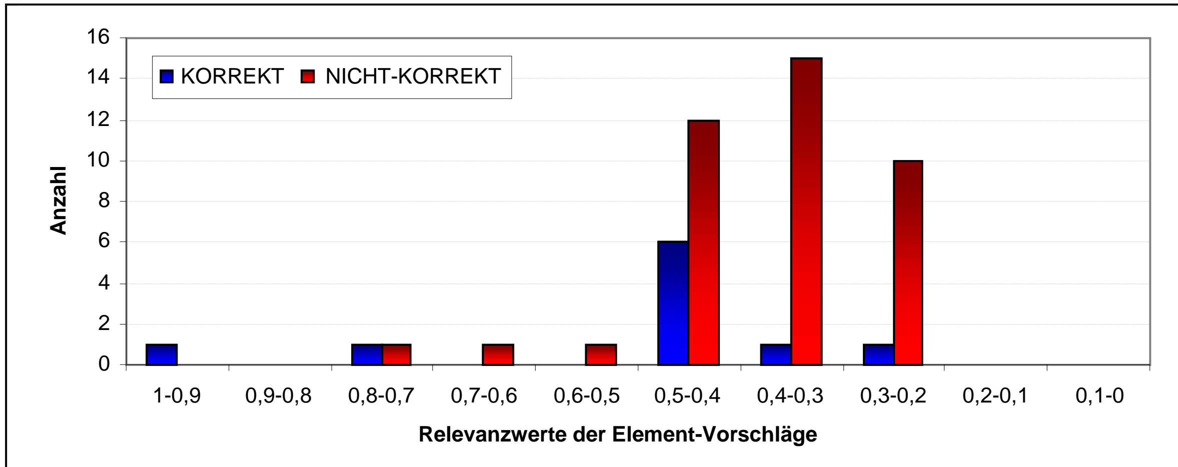
| | |
|---|---|
| #Korrekte Original-Element-Vorschläge | 64 |
| #Korrekte Zusatz-Element-Vorschläge | 21 |
| **#Korrekte Element-Vorschläge** | **85** |
| #Nicht-Korrekte Element-Vorschläge | 15 |
| **#Alle Element-Vorschläge** | **100** |

| | |
|---|---|
| **#Sichere Element-Vorschläge (0.0)** | **0** |
| **#Sichere Element-Vorschläge (1.0)** | **62** |

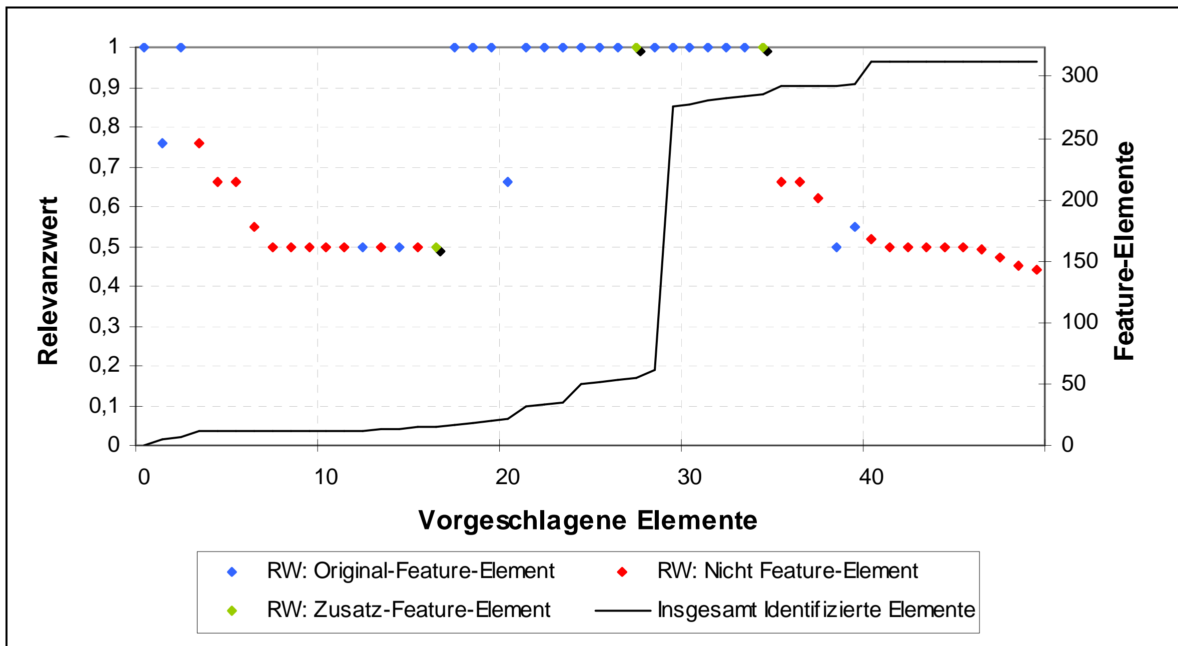
| | |
|---|---|
| **Benötigte Zeit für Feature-Lokalisierung (in min)** | **-** |
| **Benötigte Zeit für Feature-Expansion (in min)** | **21** |
| **Benötigte Zeit für Feature-Expansion ohne Stoppkriterium (in min)** | **18** |

## A.2.2 SMS Transfer OR Copy Media

### Verfügbare Element-Vorschläge – abgeleitet aus SMS Transfer sowie Copy Media




### Expansionsprotokoll




### Zusatz-Feature-Elemente

| Nummer | 16 | Ort | Klasse – MediaController; Methode - void showImage(String name) |
|---|---|---|---|
| Typ | LOK. VAR. | Element | AbstractController nextcontroller = **this** |
| Grund: | nur Zugriffe aus SMS Transfer oder Copy Media | | |

| **Nummer** | 27 | **Ort** | Klasse - AddMediaToAlbum |
|---|---|---|---|
| **Typ** | METHODE | **Element** | **void** setItemName(**String** itemName) {…} |
| **Grund**: | nur Zugriffe aus SMS TRANSFER oder COPY MEDIA | | |

| **Nummer** | 34 | **Ort** | Klasse - AddMediaToAlbum |
|---|---|---|---|
| **Typ** | METHODE | **Element** | **void** setLabelPath(**String** label) {…} |
| **Grund**: | nur Zugriffe aus SMS TRANSFER oder COPY MEDIA | | |

## Messwerte

| | **#Originale Feature-Elemente** | **#Zusätzliche Feature-Elemente** | #Feature-Elemente | **#Lokalisierte Feature-Elemente** | **#Expandierte Feature-Elemente** | #Identifizierte Feature-Elemente |
|---|---|---|---|---|---|---|
| **KOMPILIERUNGSEINH.** | 1 | | 1 | | 1 | 1 |
| **FELD** | **2** | | **2** | | **1** | **1** |
| FELD REFERENZ | 11 | 2 | 13 | | 12 | 12 |
| IMPORT | 17 | | 17 | | 17 | 17 |
| **LOKALE VARIABLE** | **20** | **3** | **23** | | **23** | **23** |
| LOK. VARIABLEN REF. | 26 | | 26 | | 26 | 26 |
| **METHODE** | **2** | **2** | **4** | | **4** | **4** |
| METHODEN REFERENZ | 70 | 2 | 72 | | 70 | 70 |
| PARAMETER REF. | 80 | 2 | 82 | | 78 | 78 |
| **TYP** | **1** | | **1** | | **1** | **1** |
| TYP REFERENZ | 80 | 3 | 83 | | 80 | 80 |
| | **310** | **14** | **324** | | **313** | **313** |

| | |
|---|---|
| #Korrekte Original-Element-Vorschläge | 23 |
| #Korrekte Zusatz-Element-Vorschläge | 3 |
| **#Korrekte Element-Vorschläge** | **26** |
| #Nicht-Korrekte Element-Vorschläge | 24 |
| **#Alle Element-Vorschläge** | **50** |

| | |
|---|---|
| **#Sichere Element-Vorschläge (0.0)** | **0** |
| **#Sichere Element-Vorschläge (1.0)** | **19** |

| | |
|---|---|
| **Benötigte Zeit für Feature-Lokaliserung (in min)** | **-** |
| **Benötigte Zeit für Feature-Expansion (in min)** | **10** |
| **Benötigte Zeit für Feature-Expansion ohne Stoppkriterium (in min)** | **7** |

# Literaturverzeichnis


[ACC+02] Antoniol, G.; Canfora, G.; Casazza, G.; De Lucia, A.; Merlo, E. *Recovering traceability links between code and documentation*. IEEE Transactions on Software Engineering, #28(10), S. 970-983, 2002.

[ACCD00] Antoniol, G.; Canfora, G.; Casazza, G.; De Lucia, A. *Identifying the starting impact set of a maintenance request: A case study*. In: European Conference on Software Maintenance and Reengineering, S. 227-230, 2000.

[ACDM99] Antoniol, G.; Canfora, G.; De Lucia, A.; Merlo, E. *Recovering code to documentation links in OO systems.* In: Working Conference on Reverse Engineering, S. 136-144, 1999.

[AG05] Antoniol, G.; Guéhéneuc, Y.G. *Feature identification: A novel approach and a case study*. In: IEEE International Conference on Software Maintenance, S. 357-366, 2005.

[AG06] Antoniol, G.; Guéhéneuc, Y.G. *Feature identification: An epidemiological metaphor.* IEEE Transactions on Software Engineering, #32(9), S. 627-641, 2006.

[AH90] Agrawal, H.; Horgan, J.R. *Dynamic program slicing*. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, S. 246-256, 1990.

[AHLW95] Agrawal, H.; Horgan, J.R.; London, S.; Wong, W. *Fault localization using execution slices and dataflow tests*. In: IEEE International Symposium on Software Reliability Engineering, S. 143-151, 1995.

[AK09] Apel, S.; Kästner, C. *An overview of feature-oriented software development.* Journal of Object Technology, #8(5), S. 49-84, 2009.

[AKL09a] Apel, S.; Kästner, C.; Lengauer, C. *Vergleich und Integration von Komposition und Annotation zur Implementierung von Produktlinien*. In: Software Engineering 2009 - Fachtagung des GI-Fachbereichs Softwaretechnik, S. 101-112, 2009.

[AKL09b] Apel, S.; Kästner, C.; Lengauer, C. *FeatureHouse: Language independent, automated software composition*. In: ACM/IEEE International Conference on Software Engineering, S. 221-231, 2009.

[ALS08] Apel, S.; Leich, T.; Saake, G. *Aspectual feature modules*. IEEE Transactions on Software Engineering, #34(2), S. 162-180, 2008.

[AMGS05] Antoniol, G.; Merlo, E.; Guéhéneuc, Y.G.; Sahraoui, H. *Feature traceability in object oriented software.* In: Workshop on Traceability in Emerging Forms of Software Engineering, IEEE/ACM International Conference on Automated Software Engineering, S. 73-78, 2005.

[AMR07] Alwis, B.d.; Murphy, G.C.; Robillard, M.P. *A comparative study of three program exploration tools*. In: IEEE International Conference on Program Comprehension, S. 103-112, 2007.

[Ape07] Apel, S. *The role of features and aspects in software development.* Dissertation, Otto-von-Guericke-Universität Magdeburg, 2007.

[BA96] Bohner, S.A.; Arnold, R.S. *Software change impact analysis*. Los Alamitos: IEEE Computer Society Press, 1996.

[Bas90] Basili, V.R. *Viewing maintenance as reuse-oriented software development*. IEEE Software, #7(1), S. 19-25, 1990.

[Bat05] Batory, D. *Feature models, grammars, and propositional formulas*. In: International Software Product Lines Conference, S. 7-20, 2005.

[BBPR05] Buckner, J.; Buchta, J.; Petrenko, M.; Rajlich, V. *JRipples: A tool for program comprehension during incremental change*. In: IEEE International Workshop on Program Comprehension, S. 149-152, 2005.

[BD06a] Bohnet, J.; Döllner, J. *Analyzing feature implementation by visual exploration of architecturally-embedded call-graphs.* In: International Workshop on Dynamic Systems Analysis, S. 41-48, 2006.

[BD06b] Bortz, J.; Döring, N. *Forschungsmethoden und Evaluation für Human- und Sozialwissenschaftler*. 4. Aufl., Heidelberg: Springer Medizin Verlag, 2006.

[BDET04] Bruntink, M.; Deursen, A.v.; Engelen, R.v., Tourwe, T. *An evaluation of clone detection techniques for identifying crosscutting concerns.* In: IEEE International Conference on Software Maintenance, S. 200-209, 2004.

[BGW+99] Bayer, J.; Girard, J.F.; Würthner, M.; DeBaud, J.M.: Apel, M. *Transitioning legacy assets to a product line architecture.* In: European Software Engineering Conference, S. 446-463, 1999.

[BKPS04] Böckle, G.; Knauber, P.; Pohl, K.; Schmid, K. *Software-Produktlinien: Methoden, Einführung und Praxis*. Heidelberg: dpunkt.verlag, 2004.

[BL08] Bortz, J.; Lienert, G. *Kurzgefasste Statistik für die Klinische Forschung – Leitfaden für die verteilungsfreie Analyse kleiner Stichproben*. 3. Aufl., Heidelberg: Springer Medizin Verlag, 2008.

[BMW93] Biggerstaff, T.J.; Mitbander, B.G.; Webster, D. *The concept assignment problem in program understanding.* In: ACM/IEEE International Conference on Software Engineering, S. 482-498, 1993.

[BMW94] Biggerstaff, T.J.; Mitbander, B.G.; Webster, D. *Program understanding and the concept assignment problem.* Communications of the ACM, #37(5), S. 72-83, 1994.

[BN99] Baeza-Yates, R.; Ribeiro-Neto, B. *Modern information retrieval*. Harlow: Addison-Wesley, 1999.

[BOS00] Bergey, J.; O'Brien, L.; Smith, D. *Mining existing assets for software product lines.* Technical Report, Software Engineering Institute, Carnegie Mellon University, 2000.

[BOS01] Bergey, J.; O'Brien, L.; Smith, D. *Options analysis for reengineering (OAR): A method for mining legacy assets*. Technical Report, Software Engineering Institute, Carnegie Mellon University, 2001.

[BPS04] Beuche, D.; Papajewski, H.; Schröder-Preikschat, W. *Variability management with feature models*. Science of Computer Programming, #53(3), S. 333-352, 2004.

[BSR04] Batory, D.; Sarvela, J.N.; Rauschmayer, A. *Scaling step-wise refinement*. IEEE Transactions on Software Engineering, #30(6), S. 355-371, 2004.

[CBA09] Chen, L.; Babar, M.A.; Ali, N. *Variability management in software product lines: A systematic review.* In: International Software Product Line Conference, S. 81-90, 2009.

[CC87] Collofello, J.; Cousin, L. *Towards automatic software fault location through decision-to-decision path analysis*. In: National Computer Conference, S. 539-544, 1987.

[CC90] Chikofsky, E.; Cross, J. *Reverse engineering and design recovery: A taxonomy*. IEEE Software, #7(1), S. 13-17, 1990.

[CE00] Czarnecki, K.; Eisenecker, U. *Generative programming: Methods, tools, and applications*. Boston: Addison-Wesley, 2000.

[CFG+09] Conejero, J.; Figueiredo, E.; Garcia, A.; Hernandez, J.; Jurado, E. *Early crosscutting metrics as predictors of software instability.* In: International Conference Objects, Models, Components, Patterns, S. 136-156, 2009.

[CHS08] Classen, A.; Heymans, P.; Schobbens, P. *What's in a feature: A requirements engineering perspective.* In: International Conference on Fundamental Approaches to Software Engineering, S. 16-30, 2008.

[CK02] Clements, P.; Krueger, C.W. *Point/Counterpoint: Being proactive pays off/eliminating the adoption barrier*. IEEE Software, #19(4), S. 28-31, 2002.

[CN02] Clements, P.; Northrop, L. *Software product lines - practices and patterns*. Boston: Addison-Wesley, 2002.

[CR00] Chen, K.; Rajlich, V. *Case study of feature location using dependence graph*. In: IEEE International Workshop on Program Comprehension, S. 241-247, 2000.

[CRB04] Colyer, A.; Rashid, A.; Blair, G. *On the separation of concerns in program families*. Technical Report, Computing Department, Lancaster University, 2004.

[CZD+08] Cornelissen, B.; Zaidman, A.; Deursen, A.v.; Moonen, L.; Koschke, R. *A systematic survey of program comprehension through dynamic analysis.* Technical Report, Delft University of Technology, 2008.

[Dij76] Dijkstra, E.W. *A discipline of programming*. Englewood Cliffs: Prentice-Hall, 1976.

[DL00] Deprez, J.; Lakhotia, A. *A formalism to automate mapping from program features to code*. In: IEEE International Workshop on Program Comprehension, S. 69-78, 2000.

[DL05] Ducasse, S.; Lanza, M. *The class blueprint: Visually supporting the understanding of classes*. IEEE Transactions on Software Engineering, #31(1), S. 75-90, 2005.

[DP09] Ducasse, S.; Pollet, D. *Software Architecture Reconstruction: A process-oriented taxonomy*. IEEE Transactions on Software Engineering, #35(4), S. 573-591, 2009.

[Dre09] Dreiling, A. *Aktueller Stand und zukünftige Herausforderungen im Aspect Mining*. In: Student Conference on Software Engineering and Database Systems, S. 41-46, 2009.

[DRD99] Ducasse, S.; Rieger, M.; Demeyer, S. *A language independent approach for detecting duplicated code*. In: IEEE International Conference on Software Maintenance, S. 109-118, 1999.

[EAAG08] Eaddy, M.; Aho, A.V.; Antoniol, G.; Guéhéneuc, Y.G. *Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis*. In: IEEE International Conference on Program Comprehension, S. 53-62, 2008.

[EBG07] Egyed, A.; Binder, G.; Grunbacher, P. *STRADA: A tool for scenario-based feature-to-code trace detection and analysis*. In: ACM/IEEE International Conference on Software Engineering, S. 41-42, 2007.

[EHM06] Eichberg, M.; Haupt, M.; Mezini, M. *The SEXTANT software exploration tool*. In: IEEE Transactions on Software Engineering, #32(9), S. 753-768, 2006.

[EKS01a] Eisenbarth, T.; Koschke, R.; Simon, D. *Derivation of feature component maps by means of concept analysis.* In: European Conference on Software Maintenance and Reengineering, S. 176-179, 2001.

[EKS01b] Eisenbarth, T.; Koschke, R.; Simon, D. *Feature-driven program understanding using concept analysis of execution traces.* In: IEEE International Workshop on Program Comprehension, S. 300-309, 2001.

[EKS03] Eisenbarth, T.; Koschke, R.; Simon, D. *Locating features in source code*. IEEE Transactions on Software Engineering, #29(3), S. 210-224, 2003.

[ES01] Eisenbarth, T.; Simon, D. *Guiding feature asset mining for software product line development.* In: International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing, S. 1-4, 2001.

[ESW04] Edwards, D.; Simmons, S.; Wilde, N. *An approach to feature location in distributed systems*. Technical Report, Software Engineering Research Center, 2004.

[EV05] Eisenberg, A.D.; Volder, K.d. *Dynamic feature traces: Finding features in unfamiliar code*. In: IEEE International Conference on Software Maintenance, S. 337-346, 2005.

[FCS+08] Figueiredo, E.; Cacho, N.; Sant'Anna, C.; Monteiro, M.; Kulesza, U.; Garcia, A.; Soares, S.; Ferrari, F.; Khan, S.; Filho, F.; Dantas, F. *Evolving software product lines with aspects: An empirical study on design stability*. In: ACM/IEEE International Conference on Software Engineering, S. 261-270, 2008.

[Fei09] Feigenspan, J. *Empirical comparison of FOSD approaches regarding program comprehension - A feasibility study*. Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, 2009.

[FK05] Frakes, W.B.; Kang, K.C. *Software reuse research: Status and future*. IEEE Transactions on Software Engineering, #31(7), S. 529-536, 2005.

[FKAL09] Feigenspan, J.; Kästner, C.; Apel, S.; Leich, T. *How to compare program comprehension in FOSD empirically - An experience report*. In: Workshop on Feature-Oriented Software Development, S. 55-62, 2009.

[FKBA07] Frenzel, P.; Koschke, R.; Breu, A.P.J.; Angstmann, K. *Extending the reflection method for consolidating software variants into product lines*. In: Working Conference on Reverse Engineering, S. 160-169, 2007.

[Fly06] Flyvbjerg, B. *Five misunderstandings about case-study research*. Qualitative Inquiry, #12(2), S. 219-245, 2006.

[Fow99] Fowler, M. Refactoring: *Improving the design of existing code*. Boston: AddisonWesley, 1999.

[FSSU96] Fayyad, U.M.; Piatesky-Shapiro, G.; Smyth, P.; Uthurusamy, R. *Advances in knowledge discovery and data mining*. Boston: MA: AAAI/MIT Press, 1996.

[GKY99] Griswold, W.G.; Kato, Y.; Yuan, J.J. *AspectBrowser: Tool support for managing dispersed aspects*. Technical report, Department of Computer Science and Engineering, University of California, 1999.

[GT03] Grubb, P.; Takang, A.A. *Software maintenance concepts and practice.* 2. Aufl., New Jersey: World Scientific, 2003.

[HC01] Heineman, G.T.; Councill, W.T. *Component-based software engineering: Putting the pieces together*. Boston: Addison Wesley, 2001.

[HDS06] Hayes, J. H.; Dekhtyar, A.; Sundaram, S. K. *Advancing candidate link generation for requirements tracing: The study of methods*. IEEE Transactions on Software Engineering, #32(1), S. 4-19, 2006.

[HK01] Hannemann, J.; Kiczales, G. *Overcoming the prevalent decomposition of legacy code*. In: Workshop on Advanced Separation of Concerns, ACM/IEEE International Conference on Software Engineering, 2001.

[HK06] Han, J.; Kamber, M. *Data mining. Concepts and techniques*. 2. Aufl., Oxford: Elsevier, 2006.

[HO93] Harrison W.; Ossher, H. *Subject-oriented programming: A critique of pure objects.* In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, S. 411-428, 1993.

[HPS07] Hill, E.; Pollock, L.; Vijay-Shanker, K. *Exploring the neighbourhood with dora to expedite software maintenance.* In: IEEE/ACM International Conference on Automated Software Engineering, S. 14-23, 2007.

[HVMV05] Hajiyev, E.; Verbaere, M.; Moor, O.d.; Volder, K.d. *CodeQuest: Querying source code with DataLog*. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, S.102-103, 2005.

[IBM09] IBM CORPORATION. *Hyperspaces*. URL: http://www.research.ibm.com/-hyperspace, Stand: 01.11.2009.

[IEEE90] IEEE Standards Association. *IEEE Standard Glossary for Software Engineering Terminology*. 1990.

[Jac95] Jacobs, R. *Methods for combining experts' probability assessments*. Neural Computation, #7(5), S. 867-888, 1995.

[JBZZ03] Jarzabek, S.; Bassett, P.; Zhang, H.; Zhang, W. *XVCL: XMLbased variant configuration language.* In: ACM/IEEE International Conference on Software Engineering, S. 810-811, 2003.

[JF88] Johnson, R.E.; Foote, B. *Designing reusable classes*. Journal of Object-Oriented Programming, #1(2), S. 22-35, 1988.

[JHS01] Jones, J.; Harrold, M.: Stasko, J. *Visualization for fault localization.* In: Workshop on Software Visualization, ACM/IEEE International Conference on Software Engineering, S. 71-75, 2001.

[Joh06] John, I. *Capturing product line information form legacy user documentation*. In: Käkölä, T.; Dueñas, J.C. Software product lines - research issues in engineering and management. Berlin: Springer, 2006.

[JV03] Janzen, D.; Volder, K.d. *Navigating and querying code without getting lost*. In: International Conference on Aspect-Oriented Software Development, S. 178-187, 2003.

[Käs10] Kästner, C. *Virtual separation of concerns: preprocessors 2.0*. Dissertation, Otto-von-Guericke-Universität Magdeburg, 2010.

[KA08] Kästner, C.; Apel, S. *Type-checking software product lines – A formal approach.* In: IEEE/ACM International Conference on Automated Software Engineering, S. 258-267, 2008.

[KAK08] Kästner, C.; Apel, S.; Kuhlemann, M. *Granularity in software product lines.* In: ACM/IEEE International Conference on Software Engineering, S. 311-320, 2008.

[KAK09] Kästner, C.; Apel, S.; Kuhlemann, M. *A model of refactoring physically and virtually separated features*. In: International Conference on Generative Programming and Component Engineering, S. 157-166, 2009.

[KAM05] Ko, A.J.; Aung, H.H.; Myers, B.A. *Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks*. In: ACM/IEEE International Conference on Software Engineering, S. 126-135, 2005.

[KAT+08] Kästner, C.; Apel, S.; Trujillo, S.; Kuhlemann, M.; Batory, D. *Language-independent safe decomposition of legacy applications into features*. Technical Report, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2008.

[KCH+90] Kang, K.; Cohen, S.; Hess, J.; Novak, W.; Peterson, *A. Feature-oriented domain analysis (FODA) feasibility study*. Technical Report, Software Engineering Institute, Carnegie Mellon University, 1990.

[KHR07] Kniesel, G.; Hannemann, J.; Rho, T. *A comparison of logic-based infrastructures for concern detection and extraction*. In: Workshop on Linking aspect technology and evolution, International Conference on Aspect-Oriented Software Development, Art. 6, 2007.

[KLM+97] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopez, C.; Loingtier, J-M.; Irwin, J. *Aspect-oriented programming*. In: European Conference on Object-Oriented Programming, S. 220-242, 1997.

[KMT07] Kellens, A.; Mens, K.; Tonella, P. *A survey of automated code-level aspect mining techniques.* Transactions on Aspect-Oriented Software Development, #4, S. 143-162, 2007.

[Kos07] Koschke, R. *Survey of research on software clones*. In: Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software, S. 1-24, 2007.

[KQ05] Koschke, K.; Quante, J. *On dynamic feature location.* In: IEEE/ACM International Conference on Automated Software Engineering, S. 420-432, 2005.

[Kri01] Krinke, J. *Identifying similar code with program dependence graphs.* In: Working Conference on Reverse Engineering, S.301-309, 2001.

[Kru01] Krueger, C.W. *Easing the transition to software mass customization*. In: International Workshop on Product Family Engineering, S. 282-293, 2001.

[Kru92] Krueger, C.W. *Software Reuse*. ACM Computing Surveys, #24(2), S. 131-183, 1992.

[KTA08] Kästner, C.; Trujillo, S.; Apel, S. *Visualizing software product line variabilities in source code.* In: Workshop on Visualization in Software Product Line Engineering, Software Product Line Conference, S. 303-313, 2008.

[KTS+09] Kästner, C.; Thüm, T.; Saake, G.; Feigenspan, J.; Leich, T.; Wielgorz, F.; Apel, S. *FeatureIDE: Tool framework for feature-oriented software development*. In: ACM/IEEE International Conference on Software Engineering, S. 611-614, 2009.

[LAL+10] Liebig, J.; Apel, S.; Lengauer, C.; Kästner, C.; Schulze, M. *An analysis of the variability in forty preprocessor-based software product lines*. In: ACM/IEEE International Conference on Software Engineering, S. 105-114, 2010.

[LBL06] Liu, J.; Batory, D.; Lengauer, C. *Feature oriented refactoring of legacy applications*. In: ACM/IEEE International Conference on Software Engineering, S. 112-121, 2006.

[LFL98] Landauer, T.K.; Foltz, P.W.; Laham, D. *An introduction to latent semantic analysis*. Discourse Processes, #25(2&3), S. 259-284, 1998.

[LL07] Ludewig, J.; Lichter, H. *Software Engineering – Grundlagen, Menschen, Prozesse, Techniken*. Heidelberg: dpunkt.verlag, 2007.

[LMPR07] Liu, D.; Marcus, A.; Poshyvanyk, D.; Rajlich, V. *Feature location via information retrieval based filtering of a single scenario execution trace*. In: IEEE/ACM International Conference on Automated Software Engineering, S. 234-243, 2007.

[LSR07] Linden, F.v.d.; Schmid, K.; Rommes, E. *Software product lines in action: The best industrial practice in product line engineering*. New York: Springer-Verlag, 2007.

[LWS00] Lukoit, K.; Wilde, N.; Stowell, S.; Hennessey, T. *Trace-Graph: Immediate visual location of software features.* In: IEEE International Conference on Software Maintenance, S. 33-39, 2000.

[Man02] Mannion, M. *Using first-order logic for product line model validation.* In: Software Product Line Conference, S. 176-187, 2002.

[McF96] McFeeley, R. *IDEAL: A User's Guide for Software Process Improvement*. Technical Report, Software Engineering Institute, Carnegie Mellon University, 1996.

[McI69] McIlroy, M. *Mass produced software components: Software engineering concepts and techniques.* In: NATO Conference on Software Engineering, S. 88-98, 1969.

[MH03] Massol, V.; Husted, T. *JUnit in action.* Greenwich: Manning Publications Co., 2003.

[MJS+00] Müller, H.A.; Jahnke, J.; Smith, D.; Storey, M.; Tilley, S.; Wong, K. *Reverse engineering: A roadmap.* In: ACM/IEEE International Conference on Software Engineering, Conference on Future of Software Engineering, S. 49-60, 2000.

[MLWR01] Murphy, G.C.; Lai, A.; Walker, R.J.; Robillard, M.P. *Separating features in source code: An exploratory study.* In: ACM/IEEE International Conference on Software Engineering, S. 275-284, 2001.

[MMS05] Marcus, A.; Maletic, J.I.; Sergeyev, A. *Recovery of traceability links between software documentation and source code.* In: International Journal of Software Engineering and Knowledge Engineering, #15(4), S. 811-836, 2005.

[MMW02] Mens, K.; Mens, T.; Wermelinger, M. *Maintaining software through intentional source-code views.* In: International Conference on Software Engineering and Knowledge Engineering, S. 289-296, 2002.

[MN96] Murphy, G.C.; Notkin, D. *Lightweight lexical source model extraction.* In: ACM Transactions on Software Engineering and Methodology, #5(3), S. 262-292, 1996.

[MPG03] Mens, K.; Poll, B.; Gonz'alez, S. *Using intentional source-code views to aid software maintenance.* In: IEEE International Conference on Software Maintenance, S. 169-178, 2003.

[MPR09] McMillan, C.; Poshyvanyk, D.; Revelle, M. *Combining textual and structural analysis of software artifacts for traceability link recovery*. In: Workshop on Traceability in Emerging Forms of Software Engineering, ACM/IEEE International Conference on Software Engineering, S. 41-48, 2009.

[MR05] Marcus, A.; Rajlich, V. *Panel: Identifications of concepts, features, and concerns in source code.* In: IEEE International Conference on Software Maintenance, S. 718, 2005.

[MRB+05] Marcus, A.; Rajlich, V.; Buchta, J.; Petrenko, M.; Sergeyev, A. *Static techniques for concept location in object-oriented code.* In: IEEE International Workshop on Program Comprehension, S. 33-42, 2005.

[MSRM04] Marcus, A.; Sergeyev, A.; Rajlich, V.; Maletic, J. *An information retrieval approach to concept location in source code.* In: Working Conference on Reverse Engineering, S. 214-223, 2004.

[MT05] Mens, K.; Tourwe, T. *Delving source-code with formal concept analysis*. Computer Languages, Systems & Structures, #31(3&4), S. 183-197, 2005.

[MTW93] Müller, H.A.; Tilley, S.R.; Wong, K. *Understanding software systems using reverse engineering technology perspectives from the Rigi project*. In: Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering, S. 217-226, 1993.

[MV08] Markle, L.; Volder, K.d. *JQueryScapes: Customizable Java code perspectives*. In: International Conference on Aspect-Oriented Software Development, Art. 8, 2008.

[MVH+07] Moor, O.d.; Verbaere, M.; Hajiyev, E.; Avgustinov, P.; Ekman, T.; Ongkingco, N.; Sereni, D.; Tibble, J. *Keynote address: .ql for source code analysis*. In: IEEE International Working Conference on Source Code Analysis and Manipulation, S. 3-16, 2007.

[NR69] Naur, P.; Randell, B.; *Software engineering: Report on a conference sponsored by the NATO Science Committee*. Scientific Affairs Division, NATO, 1969.

[OS02] O'Brien, L.; Smith, D. *MAP and OAR methods: Techniques for developing core assets for software product lines from existing assets*. Technical Report, Software Engineering Institute, Carnegie Mellon University, 2002.

[Par72] Parnas, D.L. *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, #15(12), S. 1053-1058, 1972.

[Par76] Parnas, D.L. *On the design and development of program families*. IEEE Transactions on Software Engineering, #2(1), S. 1-9, 1976.

[PBL05] Pohl, K.; Böckle, G.; Linden, F.v.d. *Software product line engineering: Foundations, principles, and techniques*. Berlin: Springer, 2005.

[PGM+07] Poshyvanyk, D.; Guéhéneuc, Y.G.; Marcus, A.; Antoniol, G.; Rajlich, V. *Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval*. IEEE Transactions on Software Engineering, #33(6), S. 420-432, 2007.

[PM07] Poshyvanyk, D.; Marcus, A. *Combining formal concept analysis with information retrieval for concept location in source code*. In: IEEE International Conference on Program Comprehension, S. 37-48, 2007.

[PMD06] Poshyvanyk, D.; Marcus, A.; Dong, Y. *JIRiSS - an eclipse plug-in for source code exploration.* In: IEEE International Conference on Program Comprehension, S. 252-255, 2006.

[PMDS05] Poshyvanyk, D.; Marcus, A.; Dong, Y.; Sergeyev, A. *IRiSS - A source code exploration Tool.* In: IEEE International Conference on Software Maintenance, S. 69-72, 2005.

[PMFG09] Poshyvanyk, D.; Marcus, A.; Ferenc, R.; Gyimthy, T. *Using information retrieval based coupling measures for impact analysis.* Empirical Software Engineering, #14(1), S.5-32, 2009.

[PR09] Petrenko, M; Rajlich, V. *Variable granularity for improving precision of impact analysis*. In: IEEE International Conference on Program Comprehension, S. 10-19, 2009.

[Pre97] Prehofer, C. *Feature-oriented programming: A fresh look at objects.* In: European Conference on Object-Oriented Programming, S. 419-443, 1997.

[QVWM94] Queille, J.P.; Voidrot, J.F.; Wilde, N.; Munro, M. *The impact analysis task in software maintenance: A model and a case study*. In: IEEE International Conference on Software Maintenance, S. 234-242, 1994.

[RBDL97] Reps, T.; Ball, T.; Das, M.; Larus, J. *The use of program profiling for software maintenance with applications to the year 2000 problem*. In: European Software Engineering Conference, S. 432-449, 1997.

[RCM04] Robillard, M.P.; Coelho, W.; Murphy, G.C. *How effective developers investigate source code: An exploratory study*. IEEE Transactions on Software Engineering, #30(12), S. 889-903, 2004.

[RG04] Rajlich, V.; Gosavi, P. *Incremental change in object-oriented programming*. IEEE Software, #21(4), S. 62-69, 2004.

[RM02a] Robillard, M.P.; Murphy, G.C. *Concern graphs: Finding and describing concerns using structural program dependencies*. In: ACM/IEEE International Conference on Software Engineering, S. 406-416, 2002.

[RM02b] Robillard, M.P.; Murphy, G.C. *Capturing concern descriptions during program navigation*. In: Workshop on Tool Support for Aspect-Oriented Software Development, ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, 2002.

[RM03] Robillard, M.P.; Murphy, G.C. *Automatically iInferring concern code from program investigation activities.* In: IEEE/ACM International Conference on Automated Software Engineering, S. 225-234, 2003.

[RM07] Robillard, M.P.; Murphy, G.C. *Representing concerns in source code*. ACM Transactions on Software Engineering and Methodology, #16(1), Art. 3, 2007.

[Rob05] Robillard, M.P. *Automatic generation of suggestions for program investigation*. In: European Software Engineering Conference, S. 11-20, 2005.

[Rob08] Robillard, M.P. *Topology analysis of software dependencies*. ACM Transactions on Software Engineering and Methodology, #17(4), Art. 18, 2008.

[RP09] Revelle, M.; Poshyvanyk, D. *An exploratory study on assessing feature location techniques*. In: IEEE International Conference on Program Comprehension, S. 218-222, 2009.

[RSH+07] Robillard, M.P.; Shepherd, D.; Hill, E.; Vijay-Shanker, K.; Pollock, L. *An empirical study of the concept assignment problem.* Technical Report, School of Computer Science, McGill University, 2007.

[RW02] Rajlich, V.; Wilde, N. *The role of concepts in program comprehension*. In: IEEE International Workshop Program Comprehension, S. 271-278, 2002.

[RW05] Robillard, M.P.; Weigand-Warr, F. *ConcernMapper: simple view-based separation of scattered concerns*. In: Workshop on Eclipse Technology Exchange, ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, S. 65-69, 2005.

[SBM01] Storey, M.A.D.; Best, C.; Michaud, J. *SHriMP views: An interactive environment for exploring Java programs.* In: IEEE International Workshop on Program Comprehension, S. 111-112, 2001.

[SE02] Simon, D.; Eisenbarth, T. *Evolutionary introduction of software product lines*. In: Software Product Line Conference, S. 272-282, 2002.

[SEI10] Software Engineering Institute, Carnegie Mellon University. *Software Product Lines*. URL: http://www.sei.cmu.edu/productlines/index.cfm, Stand: 07.02.2010.

[SEM05] Schäfer, T.; Eichberg, M.; Mezini, M. *Towards exploring crosscutting concerns*. In: Workshop on Linking Aspect Technology and Evolution, International Conference on Aspect-Oriented Software Development, 2005.

[SEW+06] Simmons, S.; Edwards, D.; Wilde, N.; Homan, J.; Groble, M. *Industrial tools for the feature location problem: An exploratory study*. Journal of Software Maintenance and Evolution, #18(6), S. 457-474, 2006.

[SGP04] Shepherd, D.; Gibson, E.; Pollock, L. *Design and evaluation of an automated aspect mining tool.* In: International Conference on Software Engineering Research and Practice, S. 601-607, 2004.

[SM83] Salton, G; McGill, M. *Introduction to Modem Information Retrieval*. New York: McGraw-Hill, 1983.

[SM04] Salah, M.; Mancoridis, S. *A hierarchy of dynamic software views: From object-interactions to feature interactions.* In: IEEE International Conference on Software Maintenance, S. 72-81, 2004.

[SMAP05] Salah, M.; Mancoridis, S.; Antoniol, G.; Penta, M.D. *Towards employing use-cases and dynamic analysis to comprehend mozilla.* In: IEEE International Conference on Software Maintenance, S. 639-642, 2005.

[SMAP06] Salah, M.; Mancoridis, S.; Antoniol, G.; Penta, M.D. *Scenario-driven dynamic analysis for comprehending large software system*. In: European Conference on Software Maintenance and Reengineering, S. 10-19, 2006.

[Sne98] Snelting, G. *Concept analysis – A new framework for program understanding*. In: ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, S. 1-10, 1998.

[SP03] Shepherd, D.; Pollock, L. *Ophir: A framework for automatic mining and refactoring of aspects*. Technical Report, Department of Computer and Information Sciences, University of Delaware, 2003.

[SP05] Shepherd, D.; Pollock, L. *Interfaces, aspects and views*. In: Workshop on Linking Aspect Technology and Evolution, International Conference on Aspect-Oriented Software Development, 2005.

[Spi99] Spinellis, D. *Explore, excogitate, exploit: Component mining*. IEEE Computer, #32(9), S. 114-116, 1999.

[SPK06] Sugumaran, V.; Park, S.; Kang, K. *Software product line engineering*. Communications of the ACM, #49(12), S. 28-32, 2006.

[SPLC09] Software Product Line Conference. *Product Line Hall of Fame*. URL: http://splc.net/fame.html, Stand: 19.01.2010.

[SPPC05] D. Shepherd, D.; Palm, J.; Pollock, L.; Chu-Carroll, M. *Timna: a framework for automatically combining aspect mining analyses*. In: IEEE/ACM International Conference on Automated Software Engineering, S. 184-193, 2005.

[SRP10] Savage, T.; Revelle, M.; Poshyvanyk, D. *FLAT^3: Feature location and textual tracing tool.* In: ACM/IEEE International Conference on Software Engineering, S. 255-258, 2010.

[Sto05] Storey, M.A.; *Theories, methods and tools in program comprehension: Past, present and future.* In: IEEE International Workshop on Program Comprehension, S. 181-191, 2005.

[STP05] Shepherd, D.; Tourwe, T.; Pollock, L. *Using language clues to discover crosscutting concerns.* In: Workshop on the Modeling and Analysis of Concerns, ACM/IEEE International Conference on Software Engineering, S. 1-6, 2005.

[SWM97] Storey, M.A.; Wong, K.; Müller, H.A. *"Rigi - A visualization environment for reverse engineering (research demonstration summary).* In: ACM/IEEE International Conference on Software Engineering, S. 606-607, 1997.

[TBK09] Thüm, T.; Batory, D.; Kästner, C. *Reasoning about edits to feature models*. In: ACM/IEEE International Conference on Software Engineering, S. 254-264, 2009.

[TC04] Tonella, P.; Ceccato, M. *Aspect mining through the formal concept analysis of execution traces*, In: Working Conference on Reverse Engineering, S. 112-121, 2004.

[TO01] Tarr, P.; Ossher, H. *Hyper/J: Multi-dimensional separation of concerns for Java.* In: ACM/IEEE International Conference on Software Engineering, S. 729-730, 2001.

[TOHS99] Tarr, P.; Ossher, H.; Harrison, W.; Sutton, S.M. Jr. *N degrees of separation: Multi-dimensional separation of concerns*. In: ACM/IEEE International Conference on Software Engineering, S. 107-119, 1999.

[TSK06] Tan, P.N.; Steinbach, M.; Kumar, V. *Introduction to data mining.* Boston: Addison-Wesley, 2006.

[Vol98] Volder, K.d. *Type-oriented logic meta programming*. Dissertation, Programming Technology Laboratory, Vrije Universiteit Brussel, 1998.

[WBP+03] Wilde, N.; Buckellew, M.; Page, H.; Rajlich, V.; Pounds, L. *A comparison of methods for locating features in legacy software.* Journal of Systems and Software, #65(2), S. 105-114, 2003.

[WBPR01] Wilde, N.; Buckellew, M.; Page, H.; Rajlich, V. *A case study of feature location in unstructured legacy fortran code.* In: European Conference on Software Maintenance and Reengineering, S. 68-76, 2001.

[WC96] Wilde, N.; Casey, C. *Early field experiences with the software reconnaissance technique for software comprehension*. In: IEEE International Conference on Software Maintenance, S. 312-318, 1996.

[Wei84] Weiser, M. *Program slicing.* IEEE Transactions on Software Engineering, #10(4), S. 352-357, 1984.

[WGGS92] Wilde, N.; Gomez, J.A.; Gust, T.; Strasburg, D. *Locating user functionality in old code*. In: IEEE International Conference on Software Maintenance, S. 200-205, 1992.

[WGH00] Wong, W.E.; Gokhale, S.S.; Hogan, J.R. *Quantifying the closeness between program components and features*. Journal of Systems and Software, #54(2), S. 87-98, 2000.

[WGHT99] Wong, W.E; Gokahle, S.S.; Horgang, J.R.; Trivedi, K.S. *Locating program features using execution slices*. In: IEEE Symposium on Application-Specific Systems and Software Engineering and Technology, S. 194-203, 1999.

[WL99] Weiss, D.M.; Lai, C.T.R. *Software product-line engineering: A family-based software development process.* Boston: Addison Wesley, 1999.

[WP03] Wolf, B.; Priebe, M. *Wissenschaftstheoretische Richtungen (Forschung, Statistik und Methoden, Bd. 8)*. Landau: Verlag Empirische Pädagogik, 2003.

[WR07] Weigand Warr, F.; Robillard, M.P. *Suade: Topology-based searches for software investigation.* In: ACM/IEEE International Conference on Software Engineering, S.780-783, 2007.

[WS95] Wilde, N.; Scully, M. *Software reconnaissance: Mapping program features to code*. Journal of Software Maintenance: Research and Practice, #7(1), S. 49-62, 1995.

[ZJ05] Zhang, C.; Jacobsen, H-A. *Prism query language: A crosscutting concern investigation language*. In: Software Demonstration, International Conference on Aspect-Oriented Software Development, 2005.

[ZZL+06] Zhao, W.; Zhang, L.; Liu, Y.; Sun, J.; Yang, F. *SNIAFL: Towards a static non-interactive approach to feature location*. ACM Transactions on Software Engineering and Methodology, #15(2), S. 195-226, 2006.

[ZZWD05] Zimmermann, T.; Zeller, A.; Weissgerber, P.; and Diehl, S. *Mining version histories to guide software changes.* IEEE Transactions on Software Engineering, #31(6), S. 429-445, 2005.

# Selbstständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig, ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Magdeburg, den 2. Juli 2010

Alexander Dreiling