Karlsruher Institut für Technologie

Automatisierte Qualitätssicherung von User Stories und Assistenz bei der Entwicklung funktionaler Tests

Diplomarbeit
von

Adrian Genaid

am Institut für Programmstrukturen und Datenorganisation
der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Walter F. Tichy
Betreuender Mitarbeiter:	Dipl.-Inform.Wirt M. Landhäußer

Bearbeitungszeit: 02.12.2011 – 01.06.2012

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 01. Juni 2012

Kurzzusammenfassung

In dieser Diplomarbeit wird der Einsatz von Ontologien als Wissensbasis für ein Assistenzsystem in einem agilen Softwareentwicklungsprozess untersucht. Die Ontologie enthält Repräsentationen von Artefakten des Softwareentwicklungsprozesses; betrachtete Artefakte umfassen Quelltext, natürlichsprachliche Anforderungsdokumente (User Stories) und ebenfalls weitgehend natürlichsprachliche Testskripte. Dazu kommt ein Glossar zum Einsatz, das aus natürlichsprachlichen Elementen extrahiert wird. Ein Glossar kann Bewusstsein für verwendete Begriffe schaffen und dadurch konsistentere Terminologie hervorbringen. Wiederverwendung von Terminologie kann für die Unterstützung von Softwaretestern und Softwareentwicklern ausgenutzt werden, indem in ähnlichem Zusammenhang verwendete Testskriptteile und Quelltextkomponenten empfohlen werden.

Inhaltsverzeichnis

Abbildungsverzeichnis **ix**

Tabellenverzeichnis **xi**

Quelltexte-Verzeichnis **xiii**

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Zielsetzung . . . 2
- 1.3 Aufbau der Arbeit . . . 2

2 Grundlagen **3**
- 2.1 Agile Softwareentwicklung und User Stories . . . 3
- 2.2 Behavior Driven Development . . . 5
- 2.3 Maschinelle Sprachverarbeitung . . . 6
- 2.4 Wissensrepräsentation durch Ontologien . . . 8
  - 2.4.1 Die Ontologie-Beschreibungssprache OWL2 . . . 8
  - 2.4.2 Zusammenführung von Ontologien . . . 10
- 2.5 Informationsbeschaffung . . . 10
  - 2.5.1 Techniken und Modelle . . . 10
  - 2.5.2 Leistungsmessung durch Precision und Recall . . . 11

3 Verwandte Arbeiten **13**
- 3.1 Qualitätsanalyse natürlichsprachlicher Anforderungsdokumente . . . 13
- 3.2 Konzeptextraktion aus natürlichsprachlichen Texten zur Glossarbildung 16
- 3.3 Ontologien in der Softwaretechnik . . . 17
- 3.4 Nachvollziehbarkeit zwischen Anforderungsdokumenten und Quelltext 18
- 3.5 Zusammenfassung . . . 19

4 Konzept **21**
- 4.1 Analyse . . . 21
  - 4.1.1 Qualitätsmängel der User Stories . . . 22
  - 4.1.2 Schwierigkeiten bei der Erstellung von Testskripten . . . 23
  - 4.1.3 Schwierigkeiten bei der Testimplementierung . . . 23
- 4.2 Entwurf . . . 23
  - 4.2.1 Aufbau der Wissensbasis . . . 24
  - 4.2.2 Qualitätssicherung der Anforderungsdokumente . . . 29
  - 4.2.3 Gewinnung relevanter Testschritte . . . 31
  - 4.2.4 API-Komponenten zu neuen Testschritten . . . 34
- 4.3 Zusammenfassung . . . 35

5 Implementierung **37**
5.1 Vermittlungsschicht 37
5.2 Ontologie-Schnittstellen 39
5.2.1 Populationsschnittstelle 39
5.2.2 Abfrage der Wissensbasis 40
5.3 Verarbeitung natürlichsprachlicher Texte 42
5.4 Population der Ontologie 45
5.4.1 Erstellung des Softwaremodelles 45
5.4.2 User Story und zugehöriges Testskript 46
5.5 Anwendungen auf Grundlage der Ontologie 48
5.5.1 Glossar 48
5.5.2 Relevante Testschritte 49
5.5.3 Relevante API-Komponenten 52
5.6 Zusammenfassung 52

6 Evaluierung **55**
6.1 Informationen zu Evaluationsdaten 55
6.1.1 1. Projektstand 56
6.1.2 2. Projektstand 56
6.2 Evaluation der Testschrittgewinnung 56
6.2.1 TF-IDF-basiertes Verfahren 59
6.2.2 Verwendungshäufigkeit in ähnlichen Testskripten 60
6.2.3 Kombination TF-IDF und Verwendungshäufigkeit 61
6.2.4 Unterschiede der Projektstände 61
6.2.5 Mögliche Einschränkungen der Aussagekraft 61
6.2.6 Nutzung von Synonymie-Beziehungen 61
6.3 Evaluation der Gewinnung relevanter API–Komponenten 65
6.4 Laufzeitmessungen zu F-TRec 67
6.5 Zusammenfassung 68

7 Zusammenfassung und Ausblick **69**

A Struktur der Ontologie **71**
A.1 Klassenhierarchie 71
A.2 Dateneigenschaften (data properties) 71
A.3 Objekteigenschaften (object properties) 71

B Leitfaden für User Stories **75**
B.1 Prinzipien 75
B.1.1 „Story Card“ 75
B.1.2 „Vereinbarung zum Gespräch“ 75
B.1.3 „INVEST“ 75
B.2 Unterstützung maschineller Auswertbarkeit 76
B.3 Test-Kriterien 76

C Verwendete Werkzeuge und Bibliotheken **77**

D Detaillierte Auswertung der Relevanz-Sortierung **79**

Literaturverzeichnis **xv**

Abbildungsverzeichnis

1.1 Entwicklungsprozess nach Scrum mit Einsatz von Verhaltensgetriebener Entwicklung bei dem kooperierenden Unternehmen 2

2.1 Gegenüberstellung Prozess nach dem V-Modell/Agiler Prozess 4

2.2 Eine User Story auf einer Karteikarte („Story Card“) 5

4.1 Gegenüberstellung einer exemplarischen User Story und eines dazugehörigen Testskriptteiles 22

4.2 Unterstützung von User Story-Autoren, Testern und Softwareentwicklern mittels F-TRec 24

4.3 Ein Überblick der Quellen zur Ontologie-Befüllung. 25

4.4 Prozess der Extraktion natürlichsprachlicher Elemente. 28

4.5 Deklaration einer Relation als transitiv im Ontologieeditor *protégé* . . 35

5.1 Skizze zur Architektur von F-TRec 38

5.2 Bereitstellung und Nutzung entfernter Objekte über Pyro4 39

5.3 Beispiel für Inferenz in Ontologien 41

5.4 Einbettung der SPARQL-DL-API in die Ontologieinfrastruktur nach [11] 42

5.5 Verarbeitung eines Beispielsatzes 43

5.6 Ein Syntaxbaum, der ein zusammengesetztes Nomen enthält. 44

5.7 Ein Beispielquelltext und seine entsprechende Darstellung in der Ontologie 46

5.8 Analyse von User Story und zugehörigem Testskript zur Anlage der Individuen und ihrer Verknüpfungen 47

5.9 Listenansicht der Glossaroberfläche 50

5.10 Detailansicht eines Eintrags 50

6.1 Beispielgraph für Präzision und Ausbeute aus Tabelle 6.2 58

6.2 Präzision und Ausbeute der Teststichprobe von Projektstand 1 62

6.3 Präzision und Ausbeute der Teststichprobe von Projektstand 2 63

6.4 Einzel-Ergebnisse Kombination Verwendungshäufigkeit und TF-IDF mit allen Heuristiken: Ausbeute, Präzision und F_1-Maß 64

6.5 Präzision und Ausbeute für vorgeschlagene Methodenaufrufe und Objekterstellungen 66

6.6 Präzision und Ausbeute für vorgeschlagene Methodenaufrufe und Objekterstellungen mit transitiver Aufrufrelation 66

A.1 Die Klassenhierarchie der Ontologie 72

D.1 Einzel-Ergebnisse TF-IDF mit allen Heuristiken: Ausbeute, Präzision und F_1-Maß 80

D.2 Einzel-Ergebnisse Verwendungshäufigkeit: Ausbeute, Präzision und F_1-Maß 81

Tabellenverzeichnis

3.1 Übersicht zu verwandten Arbeiten 14

6.1 Vergleich der Projektdaten 56

6.2 Beispiel zur Bewertung der Ergebnisse eines Verfahrens mit einer User Story 57

6.3 Präzision pro erreichte Ausbeute eines Verfahrens in Projektstand 1 . 59

6.4 Präzision pro erreichte Ausbeute eines Verfahrens in Projektstand 2 . 60

A.1 Dateneigenschaften mit Basiseigenschaft 71

A.2 Objekteigenschaften mit inverser Eigenschaft und Basiseigenschaft . . 73

Listings

1 Ein einfaches Testskript für das Hinzufügen einer Adresse. 27

2 Eine SPARQL-DL-Abfrage nach Testschritten, die mit dem natürlichsprachlichen Element *button* verknüpft sind. 32

3 Funktion zum Hinzufügen einer Klassenausprägung (Individuum). . . 40

4 Verknüpfung der durch `individual` und `related` gegebenen Individuen. 40

5 OWL/XML-Repräsentation einer Klasse mit Daten- und Objekteigenschaften. 41

6 Die vollständige Klassenhierarchie einer Ontologie in Form von direkten Unterklassen-Paaren (vgl. [11]) 42

7 Abfrage aller Paare von Testimplementierungen und Attributwerten . 48

8 Abfrage aller Glossar-Einträge mit ihrer die Wortart spezifizierenden Klasse 49

9 Abfrage der optionalen Beschreibung des NSEs „button“ 49

10 Abfrage existierender Synonyme des NSEs „button“ 49

11 Abfrage aller Testschritte, die das NSE „button“ referenzieren 51

12 Abfrage aller Testschrittimplementierungen und sie referenzierende Testskripte 52

13 Abfrage aller Testskripte, die das NSE „button“ referenzieren 52

14 Abfrage der von einer Testimplementierung verwendeten Komponenten 52

1. Einleitung

In der Softwareentwicklung spielen Anforderungsdokumente eine große Rolle. Die Qualität dieser Dokumente beeinflußt in hohem Maße die Qualität des entstehenden Softwareproduktes, ebenso wie den Aufwand zur Fertigstellung. Zunehmend ersetzen agile Prozesse traditionelle Prozess-Modelle wie Wasserfall- oder V-Modell. Bei diesen Modellen werden zunächst die Anforderungen erhoben, danach erfolgt gemäß dieser Anforderungen die Implementierung. Zuletzt werden Tests durchgeführt, die geeignet sind, die Korrektheit bezüglich der Anforderungen zu überprüfen.

Ein agiler Prozess versucht hingegen, Kommunikationsbarrieren durch ständige Interaktion der am Projekt Beteiligten aufzuheben. Diesem Ziel dient die User Story, eine kurze Beschreibung geforderter Funktionalität, die vorwiegend eine „Vereinbarung zum Gespräch“ ist (vgl. [38]). Kurze Notizen in der User Story stehen für zusätzliches Wissen (wie beispielsweise Vereinbarungen aus einem Gespräch) oder für Informationen, was gelten muss, damit die Implementierung der Story vom Kunden akzeptiert wird (Kriterien). Die Kriterien werden in automatisierte Tests überführt, um nach Akzeptanz der Implementierung kontinuierlich die Funktionalität sicherzustellen. Verhaltensgetriebene Entwicklung (Behavior Driven Development, kurz BDD) bietet eine zusätzliche Möglichkeit, Domänenwissen in automatisierte Tests und damit in die Implementierung hineinzubringen. Dazu können Beteiligte aus der Domäne, wie Tester, Tests zu Akzeptanzkriterien in natürlicher Sprache definieren (im folgenden Testskripte genannt).

Eine wichtige Erwartung von der Nutzung von User Stories und BDD-Werkzeugen ist die Angleichung bzw. Verteilung von Wissen über die am Projekt Beteiligten (siehe [38], [37]). Somit sollen auch domänenferne Beteiligte – wie Entwickler – Wissen über die Domäne erhalten. Entwicklungsspezifische Konzepte sollen den aus der fremden Domäne stammenden Beteiligten bekannt gemacht werden.

1.1 Motivation

In dem in dieser Arbeit betrachteten Projekt eines kooperierenden Unternehmens kommt ein Entwicklungsprozess nach Scrum zum Einsatz, der die Arbeit mit User Stories beinhaltet. Zusätzlich wird ein BDD-Werkzeug verwendet. Abbildung 1.1

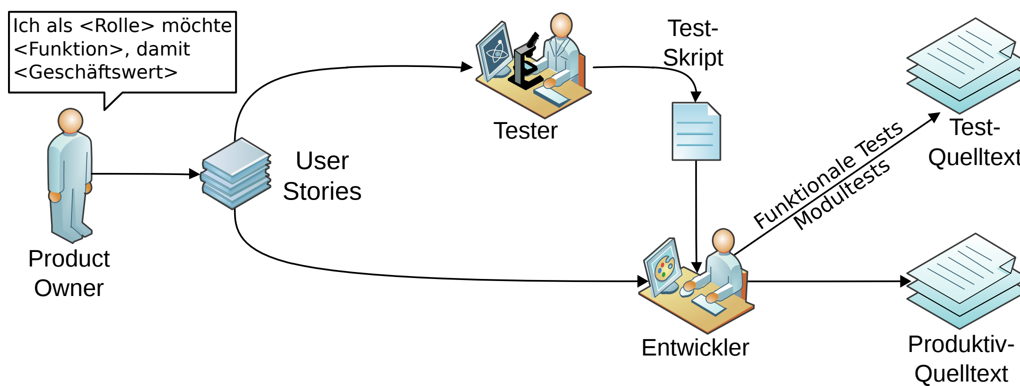


Abbildung 1.1: Entwicklungsprozess nach Scrum mit Einsatz von Verhaltensgetriebener Entwicklung bei dem kooperierenden Unternehmen

veranschaulicht den Prozess anhand der Verarbeitung vom Product Owner erstellter User Stories durch Tester und Entwickler.

Die genannten Erwartungen wurden hier nur zum Teil erfüllt. So behindern Qualitätsmängel in den User Stories, zum Teil hervorgerufen durch Wissensmangel, die Entwicklung. Wiederum aufgrund ungenügenden Wissenstransfers sind Tester häufig nicht in der Lage, das gewünschte Verhalten einer Komponente in einem Testskript zu beschreiben. Die Aufgabe der Testskripterstellung übernehmen daher oft Entwickler, die häufig Implementierungsdetails statt gewünschten Verhaltens einer Komponente prüfen. Die Formulierung von Testschritten in natürlicher Sprache lässt Mehrdeutigkeiten zu, wodurch Testschritte in bestimmtem Kontext fehlinterpretiert werden. Dies kann zu einer Fehlimplementierung führen.

1.2 Zielsetzung

Diese Arbeit setzt sich mit einem Assistenzsystem für mehrere typische Arbeitsvorgänge auseinander. Durch die Bereitstellung aufbereiteten Wissens, das sowohl aus bestehendem Quelltext von Tests und Software als auch aus User Stories und Testskripten extrahiert wird, sollen an einem Softwareprojekt Beteiligte in ihren Aufgaben unterstützt werden:

1. Autoren von User Stories sollen Hinweise auf inkonsistente Begrifflichkeiten erhalten.
2. Testern sollen häufig verwendete und thematisch passende Testschritte vorgeschlagen werden, um Wiederverwendung zu erleichtern.
3. Entwickler sollen zusätzlich Hinweise auf API-Komponenten zur Erleichterung der Implementierung eines Testschrittes erhalten.

1.3 Aufbau der Arbeit

Kapitel 2 erläutert Grundlagen des behandelten Themas. Verwandte Arbeiten werden in Kapitel 3 diskutiert. Kapitel 4 nennt die zu adressierenden Probleme und skizziert deren Lösung, während Implementierungsdetails in Kapitel 5 zu finden sind. Eine Evaluation des Ansatzes wird in Kapitel 6 durchgeführt. Kapitel 7 fasst die Erkenntnisse zusammen und erläutert den Ausblick.

2. Grundlagen

2.1 Agile Softwareentwicklung und User Stories

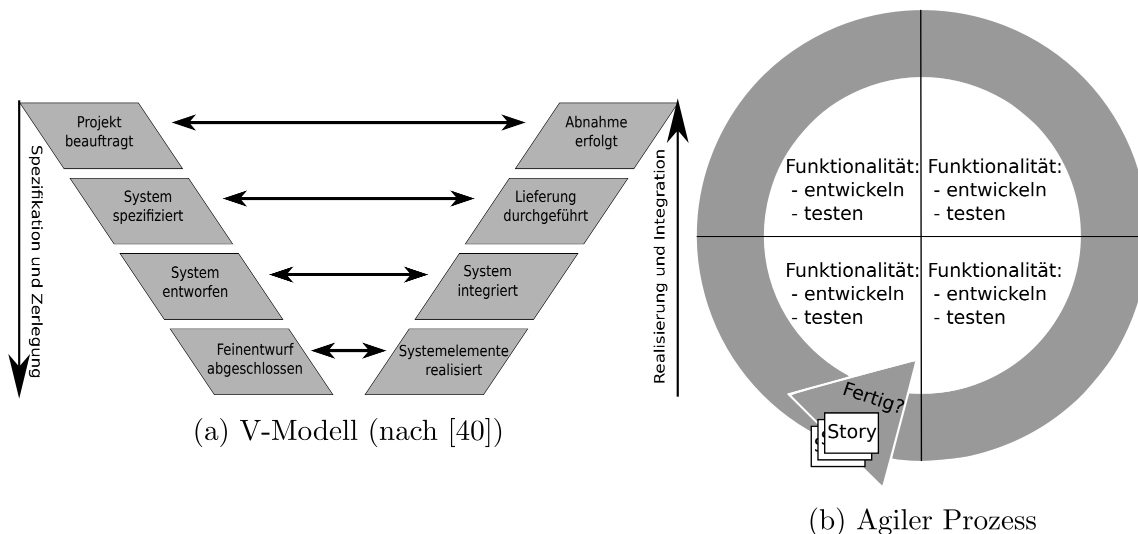
User Stories sind ein Grundbestandteil agiler Softwareentwicklung. Explizit stellt Cohn einen Prozess, der auf User Stories aufbaut, dem traditionellen V-Modell gegenüber [25]. Wie man in Abbildung 2.1 sehen kann, werden bei Prozessen nach dem V-Modell zu Beginn des Projektes alle Anforderungen erhoben. Anhand der Anforderungsdokumente wird das System implementiert. An die Implementierungsphasen schließen sich mehrere Testphasen an.

In der agilen Softwareentwicklung hingegen wird keine Anforderungsanalyse im traditionellen Sinne durchgeführt, stattdessen werden Grundzüge des geforderten Systems vom Kunden in User Stories formuliert und im Laufe der Entwicklung iterativ in weiteren User Stories verfeinert. Abb. 2.1 veranschaulicht den Unterschied zwischen dem traditionellen V-Modell und agilen Prozessmodellen.

Auch wenn User Stories in verschiedenen agilen Software-Entwicklungsprozessen verwendet werden, werden im folgenden vorwiegend Ausprägungen des Prozessmodelles Scrum beschrieben. Dies ist jedoch nicht Voraussetzung für die Erkenntnisse dieser Arbeit, sondern dient als Ansatzpunkt der Beschreibung der Arbeit mit User Stories.

Ein grundlegender Unterschied von User Stories zu traditionellen Anforderungsdokumenten ist die Betonung der mündlichen statt der schriftlichen Kommunikation (siehe [25], [38]). Ein traditionelles Anforderungsdokument soll eine Komponente so genau beschreiben, dass ein Entwickler sie ohne Rücksprache mit dem Kunden oder Projektleiter erstellen kann. Eine User Story enthält hingegen nur die Grobbeschreibung einer Funktionalität oder eines Aspektes einer Funktionalität. Über die gesamte Lebensdauer der User Story werden in Gesprächen mit dem Kunden oder Projektleiter Details erarbeitet. Währenddessen können der User Story auch Informationen hinzugefügt oder entfernt werden.

User Stories können entweder auf Karteikarten verwaltet werden („Story Cards“) oder elektronisch. Abbildung 2.2 zeigt eine User Story in Papierform. Die Papierform unterstreicht einige der Aspekte einer User Story wie Informalität und Kürze



(a) V-Modell (nach [40])

(b) Agiler Prozess

Abbildung 2.1: Gegenüberstellung Prozess nach dem V-Modell/Agiler Prozess

(aufgrund Platzmangel). Ein Vorteil der elektronischen Speicherung ist vor allem die leichte Verfügbarkeit für verteilte Teams.

Eine User Story gilt als erfüllt, wenn der Kunde die Implementierung der Funktionalität akzeptiert. Richtlinien für die Akzeptanz der Implementierung bieten die in automatisierten funktionalen Tests manifestierten Kriterien. Diese Richtlinien spezifiziert der Kunde vor und während der Implementierung in der User Story. Dadurch erhalten Entwickler zusätzliche Details der gewünschten Funktionalität. Sobald eine User Story erfüllt wurde, kann die Karteikarte zerrissen oder ihr elektronisches Pendant gelöscht werden. Auch dieses Vorgehen unterscheidet sich von traditionellen Anforderungsdokumenten, denn diese haben in der Regel keine begrenzte Lebensdauer. Der Inhalt der User Story lebt einerseits in der Implementierung weiter, andererseits in den automatisierten funktionalen Tests.

Eine User Story sollte mindestens die folgenden Komponenten enthalten [25]:

- Eine Beschreibung der Story; empfohlen ist die Form „Ich als <Rolle> möchte <Funktion>, damit <Geschäftswert>“
- Kriterien: in Geprächen herausgearbeitete Grundlagen für funktionale Tests
- Eine Abschätzung des Aufwandes
- Eine Priorität

In der agilen Softwareentwicklung mit Scrum besteht ein Release aus einer oder mehreren Iterationen mit fester Dauer (genannt Sprints) [26]. Vor jeder Iteration priorisiert der Product Owner – der die „Vision“ des Produktes verantwortet – die restlichen User Stories; die Sammlung der Stories wird in Scrum Product Backlog genannt. Anhand Priorität und Aufwandsschätzung wählt das Entwicklungsteam User Stories für einen Sprint aus. Tester und Entwickler entwerfen die funktionalen Tests während der Arbeiten zur Erfüllung einer User Story.

Wake entwarf Kriterien für gute User Stories, die er als INVEST bezeichnet (*Independent*, *Negotiable*, *Valuable*, *Estimable*, *Small*, *Testable*) [74]. Diese werden in Anhang B genauer erläutert.

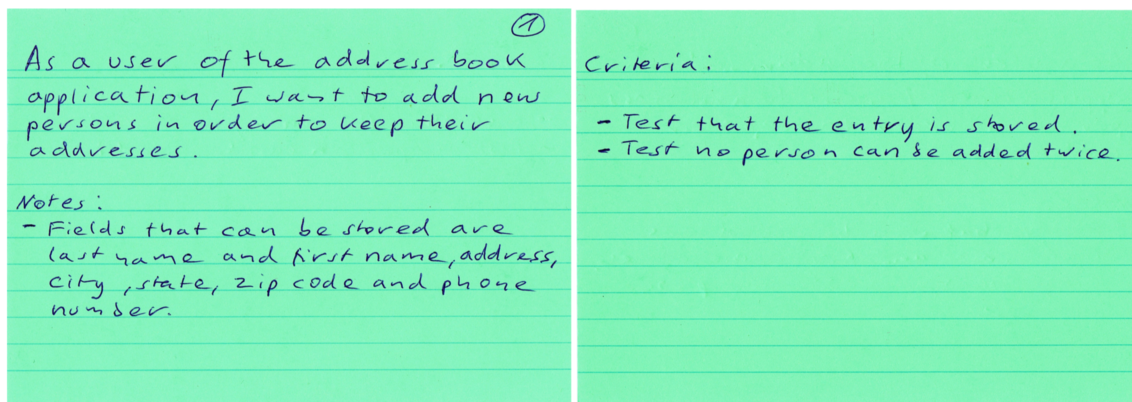


Abbildung 2.2: Eine User Story auf einer Karteikarte („Story Card“)

2.2 Behavior Driven Development

Wie bereits beschrieben werden in der agilen Softwareentwicklung Tests iterativ und parallel zur Implementierung erstellt. Dazu wird häufig die testgetriebene Entwicklung eingesetzt (Test-Driven Development, TDD) [17]. Bei dieser Methode schreiben Entwickler zunächst Komponententests für eine gewünschte Funktionalität. Hiernach implementiert man die Komponente mit möglichst geringem Aufwand im Hinblick auf Korrektheit gemäß der definierten Tests. Im Anschluß wird der entstandene Quelltext aufgeräumt, wobei die Tests noch immer erfolgreich durchlaufen müssen. Der Entwickler überlegt sich dann weitere Tests, die einerseits die Komponente genauer beschreiben, andererseits wichtige Randfälle abprüfen. Diese Schritte (Tests definieren, entwickeln, aufräumen) werden so lange ausgeführt, bis die gewünschte Funktionalität erreicht wurde.

Die bei der testgetriebenen Entwicklung entstehenden Tests sind stark an die Implementierung bzw. die interne Struktur der Komponenten gekoppelt. Daher eignen sie sich nicht als Basis für funktionale Tests. Ein Beispiel aus *The RSpec Book* ist die Verwendung eines Feldes zur Speicherung von Registrierungen und dessen späterer Änderung zu einer Liste [22]. Das Verhalten der Registrierung bleibt gleich, doch durch die Änderung der Datenstruktur wird der Test fehlschlagen. Einen Kunden interessiert die zugrunde liegende Datenstruktur eher wenig; sein Augenmerk liegt stärker auf dem Geschäftswert, der gerade in dem Verhalten der Komponente besteht.

Verhaltensgetriebene Entwicklung (Behavior-Driven Developement, BDD) – in Abgrenzung zu TDD – verwendet daher keinen komponententestbasierten Ansatz; stattdessen betrachtet BDD das Verhalten der Komponente [62]. Dazu wurden für die Namen von Testfunktionen zunächst ganze Sätze verwendet, die das gewünschte Verhalten beschreiben. Diese Sätze können auch von Kunden gelesen und verstanden werden, wodurch die definierten Tests die Rolle der in Abschnitt 2.1 genannten funktionalen Tests erhalten.

Weitergehende Einflussnahme in die Testspezifizierung durch Kunden erlaubt und bedingt die Verwendung natürlicher Sprache. Domänenspezifische Sprachen, die durch schwache Strukturen für Fachpersonal zumindest lesbar sind, sind dafür von Vorteil. Diese „Business Readable Domain Specific Languages“ [37] (wie Gherkin [2])

werden in einigen BDD-Werkzeugen verwendet. Gherkin ermöglicht die Beschreibung des Verhaltens einer Software in Testskripten ohne Verwendung von Implementierungsdetails.

Es gibt mehrere Werkzeuge, die Gherkin-Skripte in Quelltextskelette übersetzen. Beispiele hierfür sind die ursprüngliche Ruby-Implementierung [10] Cucumber [1], SpecFlow [12] für C#.NET [13] und lettuce [5] für Python [9].

Die Grammatik von Gherkin definiert nur wenige strukturierende Schlüsselwörter. Für diese Schlüsselwörter existieren Zuordnungen in mehreren Sprachen. Wir betrachten die der englischen Sprache.

Skriptstruktur

- *Feature*: „Beginnt“ das Testskript, enthält *Scenario*-Elemente; Freitext kann dahinter angegeben werden, hat keine Bedeutung für die Werkzeuge
- *Scenario*: Definiert ein Szenario durch eine Liste von Testschritten; kann ebenso Freitext erhalten
- *Background*: Wie *Scenario*, wird jedoch vor jedem Szenario des Testskriptes ausgeführt um oft benötigten Kontext herzustellen
- *Scenario Outline*: enthält Testschritte mit Platzhaltern und eine entsprechende Tabelle, aus der die Platzhalter ersetzt werden

Testschritte

Alle Arten von Testschritten werden von den Werkzeugen gleich behandelt. Nutzer sollten die Testschrittarten jedoch unterschiedlich behandeln.

- *Given*: Vorbedingungsschritt, mit dessen Hilfe das System in einen wohldefinierten Zustand versetzt wird
- *When*: Benutzerinteraktion, um die zu testende Funktionalität auszuführen
- *Then*: Beobachtung/Evaluation des Ergebnisses
- *But/And*: Erhalten die Art des vorherigen Testschrittes; dient der intuitiveren Lesbarkeit des Skriptes

2.3 Maschinelle Sprachverarbeitung

Maschinelle Sprachverarbeitung (Natural Language Processing, NLP) betrachtet die Analyse und/oder Generierung natürlicher Sprache mithilfe von Computern. Eine Definition von Liddy lautet:

„Natural Language Processing is a theoretically motivated range of computational techniques for analyzing and representing naturally occurring texts at one or more levels of linguistic analysis for the purpose of achieving human-like language processing for a range of tasks or applications.“ [59]

Natürliche Sprache kann entweder in geschriebener, maschinenlesbarer Form oder als akustisches/optisches Signal vorkommen, das zunächst in maschinenlesbaren Text transformiert werden muss. Wir beschäftigen uns in der vorliegenden Arbeit ausschließlich mit maschinenlesbaren Texten. Ein Ziel der maschinellen Sprachverarbeitung ist das Sprachverstehen (Natural Language Understanding), dass jedoch bislang nicht erreicht wurde. Dennoch gibt es einige praktische Anwendungen der maschinellen Sprachverarbeitung (siehe [59]):

- Informationsbeschaffung (Information Retrieval)
- Informationsextraktion
- Fragenbeantwortung
- Zusammenfassung
- maschinelle Übersetzung
- Dialogsysteme

Sprachverarbeitung kann auf mehreren Ebenen stattfinden, wobei für höhere Ebenen die unteren zumeist auch involviert sind. Das motiviert ein Model eines Sprachverarbeitungsfließbandes, das aber stark von der Anwendung abhängt. Dementsprechend sind die verschiedenen Ebenen und ihre Aufgaben im folgenden skizziert (siehe [59]):

1. *Phonologie*: Laute und Regeln der Lautbildung; hauptsächlich benötigt für Spracherkennung
2. *Morphologie*: Wortgrammatik: Flexion, Derivation, Komposition
3. *Lexikalische Semantik*: Bedeutung/Funktion des einzelnen Wortes; hier findet die Wortartbestimmung statt
4. *Syntax*: Grammatische Struktur eines Satzes
5. *Semantik*: Bedeutung eines Satzes bezüglich Interaktionen der Komponentenbedeutungen
6. *Diskurs*: Bedeutung über Satzgrenzen hinweg: Anaphora-Auflösung etc.
7. *Pragmatik*: Nutzung externer Quellen, um die Bedeutung des Textes zu erfassen

In der vorliegenden Arbeit verwenden wir die Stufen 2-4.

2.4 Wissensrepräsentation durch Ontologien

Der Begriff *Ontologie* hat mehrere Bedeutungen. In der Philosophie bezeichnet dieser Begriff „Das Wesen von Dingen“ (siehe [21]). Aristoteles nahm dazu in seiner Arbeit *MetaPhysics* eine Einteilung von Prädikaten vor, die geeignet sein sollen, alles in der Welt zu vergleichen (vgl. [21], Seite 3).

Davon abzugrenzen ist der Begriff einer Ontologie in der Informatik. Studer definiert eine Ontologie folgendermaßen: „An ontology is a formal, explicit specification of a shared conceptualisation“ [71]. Mit dieser Definition wird gefordert, dass die Art der aus der Realität ermittelten Konzepte und ihre Beschränkungen explizit gegeben sind. Formale Spezifikation bedingt weiterhin die Maschinenlesbarkeit der Konzeptualisierung. Schlussendlich drückt eine Ontologie das geteilte Wissen einer Gruppe aus.

Ontologien ermöglichen, die Struktur eines Systemes formal zu modellieren. Die Struktur eines Systemes umfasst alle beobachtbaren Entitäten und die Relationen zwischen ihnen, soweit sie dem Modellzweck dienlich sind (vgl. [70]). Auch wenn es sich stark unterscheidende Formalisierungen gibt, weisen alle die folgenden drei Hauptkomponenten auf (siehe [21]):

- *Klassen* repräsentieren Konzepte und sind in einer Vererbungshierarchie (*Taxonomie*) strukturiert
- *Relationen* sind zumeist binär und stellen Verknüpfungstypen dar; sie können in manchen Formalisierungen auch zur Definition von Konzepteigenschaften verwendet werden.
- *Instanzen* bezeichnen Elemente bzw. Individuen eines Konzeptes

Wir betrachten im folgenden die Formalisierung bzw. Ontologiesprache *OWL2* genauer, da wir diese in der vorliegenden Arbeit verwenden.

2.4.1 Die Ontologie-Beschreibungssprache OWL2

Die Ontologie-Beschreibungssprache OWL2, die vom World Wide Web Consortium (W3C) spezifiziert wurde, ermöglicht die Beschreibung einer Grundstruktur mithilfe einer Klassenhierarchie. In OWL2 können Individuen, also Konzept-Instanzen, zu Klassen zugeordnet und Individuen untereinander über Relationen verknüpft werden. Desweiteren gibt es die Möglichkeit, Metadaten zu Klassen hinzuzufügen und Individuen Daten hinzuzufügen.

Jedes Element der Ontologie wird über einen eindeutigen Kenner (Internationalized Resource Identifier, IRI) angelegt bzw. referenziert. Dieser besteht aus einem Basisteil, der den URI (Uniform Resource Identifier) der Ontologie enthält, und einem Namen des Elements. Im folgenden wird basierend auf einer Anleitung von Horridge der Aufbau der Ontologie-Sprache kurz erläutert [43].

2.4.1.1 Klassenhierarchie

Grundlage der Ontologie sind analog zur obigen Beschreibung Klassen. Notwendige Informationen zur Definition einer Klasse sind der Name der Klasse und die direkten Basisklassen. Alle erstellten Klassen stammen zumindest von einer Basisklasse ab, wobei eine vordefinierte Basisklasse namens `Thing` existiert. Mehrfachvererbung ist ebenso möglich. Zu einer Klasse kann man weiterhin deren äquivalente und disjunkte Klassen angeben.

2.4.1.2 Eigenschaften

Die weiter oben genannten Relationen werden in OWL Eigenschaften (Properties) genannt. Es gibt folgende Arten von Eigenschaften in OWL:

- Objekteigenschaften (*object properties*): Relationen zwischen Individuen, beiderseitig einschränkbar auf bestimmte Klassen bzw. den Schnitt von Klassen

- Dateneigenschaften (*data properties*): Zuordnung von Datentypen zu einer Klasse, wodurch Individuen Informationen z.B. durch Zeichenketten oder Ganzzahlen erhalten können

- Annotationseigenschaften (*annotation properties*): Metadaten von Ontologie, Eigenschaften oder Klassen, im folgenden nicht näher behandelt

Objekteigenschaften erhalten einen Namen, der konventionsgemäß das Präfix „`is`“ oder „`has`“ besitzt und bei kleingeschriebenem Namensanfang aus angereihten Wörtern mit großem Anfangsbuchstaben bestehen sollte. Der Definitionsbereich der Objekteigenschaften lässt sich durch Angabe von Klassen in *Domain* und *Range* angeben. Wird mehr als eine Klasse in *Domain* oder *Range* angegeben, so gilt die Relation nur für die Schnittmenge der Individuen dieser Klassen. Weitere Eigenschaften der Relation sind die Angabe der Kardinalität der Relation durch Charakterisierung als funktionale oder invers funktionale Relation, des Weiteren Symmetrie, Transitivität und Reflexivität.

Dateneigenschaften können über die *Domain* wieder auf Individuen einer Klasse oder der Schnittmenge mehrerer Klassen eingeschränkt werden; die Option *Range* gibt den Datentyp der Dateneigenschaft an.

Sowohl Objekt- als auch Dateneigenschaften können in einer Vererbungshierarchie angelegt werden, wobei jeweils eine vordefinierte Basiseigenschaft existiert.

2.4.1.3 Individuen

Individuen sind, wie bereits in Abschnitt 2.4.1 erwähnt, die Konzept-Instanzen. Ein Individuum kann sowohl explizit durch Definition eines entsprechenden Axioms einer Klasse zuordnet sein als auch implizit über zugeordnete Eigenschaften zur Instanz einer Klasse werden.

2.4.2 Zusammenführung von Ontologien

Während zunächst versucht wurde, in Ontologien universal gültiges Wissen zu erfassen, wurde dies im späteren Verlauf der Erforschung von Ontologien weitgehend aufgegeben. Der Grund für die Aufgabe des Universalitätsanspruches ist, dass Wissen je nach der beabsichtigten Nutzung verschieden strukturiert wird. Einen weitreichenden Ansatz zur Bereitstellung von Weltwissen bietet jedoch Cyc [28].

Thematisch ähnliche oder komplementäre Ontologien kann man miteinander kombinieren; dies nennt man *ontology matching* oder *ontology alignment* [32]. So lässt sich Wissen zusammenführen, das unterschiedlich strukturiert ist.

2.5 Informationsbeschaffung

Informationsbeschaffung (Information Retrieval) beschäftigt sich mit der Repräsentation, Speicherung und Organisation von Informationselementen und dem Zugriff auf diese (vgl. [16]). Informationselemente sind dabei üblicherweise natürlichsprachliche – daher unstrukturierte – Dokumente, wie sie beispielsweise im Internet oder in Bibliotheken existieren. Verschiedene Modelle kommen für die Aufgabe der Indexerstellung und Suche zum Einsatz. Traditionell verwendet man einen Index zur Speicherung von aus Dokumenten extrahierten Termen (Schlüsselwörtern). Die Art der Indexerstellung hängt dabei von dem jeweiligen Modell ab. Im folgenden erläutern wir grundlegende Modelle und die Evaluation von Verfahren.

2.5.1 Techniken und Modelle

Zu den klassischen Modellen der Informationsbeschaffung gehören das Boolesche Modell und das Vector Space-Modell (siehe [16]).

Das Boolesche Modell ist ein einfaches, auf Mengentheorie basierendes Modell. Eine Abfrage enthält Terme, die mit booleschen Operatoren verknüpft sind. Es gewichtet die zur Anfrage zurückgelieferten Dokumente binär; somit passt ein Dokument auf die Anfrage oder nicht. Diese scharfe Trennung kann entweder zuviele Treffer oder zuwenige Treffer zurückgeben. Dabei ist durch die fehlende Gewichtung kein Anhaltspunkt der Relevanz gegeben.

Das Vector Space-Modell hingegen basiert auf nicht-binären Gewichten der Abfrage- und Indexterme, durch die auch Teilübereinstimmungen gefunden werden. Dazu wird jedes Dokument und die Abfrage jeweils als Vektor mit der Dimension der im System enthaltenen Indexterme aufgefasst. Jeder Vektor enthält dazu für jeden Indexterm das Gewicht dieses Terms bezüglich des Dokumentes. Die Ähnlichkeit eines Dokumentes zu der Abfrage wird über die Berechnung des Abstandes der entsprechenden Vektoren berechnet. Hierzu kommen verschiedene Abstandsmaße zum Einsatz, wie zum Beispiel der Cosinus des Winkels zwischen den Vektoren.

Als Generalisierung des Booleschen Modelles führen Baeza-Yates und Ribeiro-Neto schlüsselwortbasierte Abfragen auf, wozu auch natürlichsprachliche Abfragen gezählt werden [16]. Dabei wird die Abfrage als Liste von Wörtern aufgefasst, während ein weiteres, darunterliegendes Modell eine Gewichtung vornimmt.

2.5.2 Leistungsmessung durch Precision und Recall

Die Leistung eines Verfahrens zur Informationsbeschaffung kann durch die Maße Präzision (Precision) und Ausbeute (Recall) quantifiziert werden. Für diese Maße nimmt man eine Menge an relevanten Dokumenten bezüglich einer Informationsanfrage an, die das optimale Ergebnis darstellen (vgl. [16]). Diese optimale Ergebnismenge bezeichnen wir als R mit Mächtigkeit $|R|$, die Informationsanfrage als I. Bezüglich einer Informationsanfrage I wird weiterhin von dem zu evaluierenden Verfahren die Ergebnismenge A mit Mächtigkeit $|A|$ zurückgegeben. Weiterhin ist $|Ra|$ die Anzahl der Dokumente in der Schnittmenge von R und A.

Die Präzision gibt an, wie groß der Anteil relevanter Dokumente an der Ergebnismenge ist. Aus obigen Mengendefinitionen berechnet sie sich folgendermaßen:

$$Präzision = \frac{|Ra|}{|A|}$$

Die Ausbeute besagt, welcher Anteil der relevanten Dokumente in der Ergebnismenge erhalten ist:

$$Ausbeute = \frac{|Ra|}{|R|}$$

Ausbeute und Präzision können über das gewichtete harmonische Mittel kombiniert werden. Dieses Maß wird F_β-Maß genannt und wie folgt gebildet (siehe [23]):

$$F_\beta = (1 + \beta^2) * \frac{Präzision * Ausbeute}{\beta^2 * Präzision + Ausbeute}$$

Der Parameter β bestimmt die Gewichtung des harmonischen Mittels; bei F_1 werden Präzision und Ausbeute gleich gewichtet, bei $F_{0.5}$ geht die Präzision doppelt so stark in das Ergebnis ein wie die Ausbeute. Da sowohl die Präzision als auch die Ausbeute nur Werte zwischen 0 und 1 einschließlich annehmen können, liegt das Maximum des F-Maßes bei 1, was bei Präzision und Ausbeute von 1 erreicht wird. Je näher Präzision und Ausbeute aneinander liegen, desto höher ist das Ergebnis.

3. Verwandte Arbeiten

Eine Aufgabe, die in dieser Arbeit betrachtet wird, ist die Qualitätsanalyse von Anforderungsdokumenten. Existierende Arbeiten und Verfahrensweisen dazu diskutieren wir in Abschnitt 3.1.

Ein häufig in der Softwareentwicklung verwendetes und einfaches Mittel zur Qualitätsverbesserung und leichteren Verständlichkeit von Anforderungsdokumenten stellen Glossare dar. Durch Termextraktion kann dies unterstützt werden. Des Weiteren bietet die Termextraktion den Aufbau eines Index zur Informationsbeschaffung (Information Retrieval). Relevante Arbeiten dazu sind in Abschnitt 3.2 aufgeführt.

Ontologien werden in der Softwaretechnik und Anforderungsanalyse häufig eingesetzt. Ansätze hierzu befinden sich in Abschnitt 3.3.

Die Fähigkeit, Verbindungen zwischen Anforderungsdokumenten, Dokumentation und Software-Artefakten wie Quelltext herzustellen, bezeichnet man in der Softwaretechnik als Nachvollziehbarkeit (Traceability). Für die vorliegende Arbeit interessante Ansätze der Gewinnung von Nachvollziehbarkeit werden in Abschnitt 3.4 diskutiert.

Tabelle 3.1 gibt einen Überblick über die diskutierten Arbeiten und die Nutzung von Erkenntnissen oder Methoden für diese Arbeit.

3.1 Qualitätsanalyse natürlichsprachlicher Anforderungsdokumente

User Stories, ebenso wie die Agile Entwicklung, sind relativ junge Ansätze der Softwareentwicklung. Es existieren bislang wenige Arbeiten, die sich mit Qualitätsanalyse von User Stories beschäftigen. Der Hauptgrund hierfür ist sicherlich die Rolle von User Stories als Vereinbarung eines Gespräches (siehe Abschnitt 2.1). Des Weiteren hemmt die kurze Lebensdauer von User Stories die detaillierte Betrachtung. Aus diesem Grund beschäftigen sich die meisten hier angegebenen Arbeiten mit anderen, vergleichbaren Formen von Anforderungsdokumenten.

Autor(en)	Thema	genutzt?
Fantechi et. al	Use Case-Analyse	nein
Körner & Brumm	Spezifikationsanalyse	nein
Sillitti et al.	Studie Anforderungen	ja
Zave & Jackson	Anforderungsanalyse	ja
Park et al.	Glossarextraktion	ja
Kof	Ontologieextraktion aus Anforderungsdokumenten	ja
Lapshinova-Koltunski & Heid	Subkategorieextraktion aus zusammengesetzten Substantiven	ja
Breitmann & Leite	User Story/Glossarverknüpfung	ja
Calero et al.	Ontologieklassifikation	ja
Welty	AST-Anreicherung	ja
Zhang et al.	Nachvollziehbarkeit mittels Ontologien	ja
Miller	WordNet	ja
Antoniol et al.	Nachvollziehbarkeit über VSM	nein
Marcus & Maletic	Nachvollziehbarkeit über LSI	nein
Ziftci & Krueger	Nachvollziehbarkeit mittels ausführbarer Spezifikationen	nein

Tabelle 3.1: Übersicht zu verwandten Arbeiten

Fantechi et al. beschäftigten sich mit der linguistischen Analyse von Use Cases [33]. Hier werden einige Qualitätsmetriken in folgenden Bereichen der Aussagekraft erläutert und anhand eines ca. 100 Use Cases umfassenden Anforderungsdokumentes evaluiert:

- Mehrdeutigkeitsverringerung
- Verständlichkeitsverbesserung

In beiden Bereichen kommen lexikalische und syntaktische Methoden zur Anwendung. Dadurch werden Mehrdeutigkeiten und Verständlichkeitsprobleme sowohl auf Wortebene wie auch auf Satzebene addressiert. Fantechi et al. analysieren Verfahren aus drei verschiedenen Werkzeugen: *QuARS*, *ARM* und *SyTwo*. *QuARS* verwendet anpassbare Sammlungen von Indikatoren (Terme und linguistische Konstruktionen) für Schwächen in Testbarkeit, Konsistenz und Verständlichkeit, um auf Satzbasis mögliche Defekte festzustellen. Für *ARM* hat Wilson Qualitätsattribute von Anforderungsdokumenten und quantitativ messbare Äußerungen dieser Attribute zusammengestellt [76]. Eine Kategorisierung von Phrasen und Wörtern zu den Qualitätsattributen wurde dann aus bestehenden Anforderungsdokumenten extrahiert. Die qualitative Einschätzung eines natürlichsprachlichen Anforderungsdokumentes lässt sich über das Auftreten dieser Phrasen und Wörter treffen. *SyTwo* erkennt syntaktisch mehrdeutige Sätze aufgrund verschiedener Möglichkeiten des Aufbaus eines Syntaxbaumes. Auch der Wert der *Coleman-Liau*-Lesbarkeitsmetrik dient als Indikator der Qualität [27].

Körner und Brumm beleuchten sprachliche Probleme und schlagen diesbezüglich Lösungen vor , die im *Requirements Engineering Specification Improver* (RESI) Verwendung finden [53], [52], [54]. Die Probleme umfassen unter anderem Nominalisierung, Mehrdeutigkeit und Bedeutungsähnlichkeit verwendeter Begriffe. Verschiedene lexikalische Datenbanken und Ontologien wie Cyc, ConceptNet und WordNet kommen zum Einsatz, um Spezifikationsdokumente auf die beschriebenen Probleme zu prüfen.

Da User Stories, im Gegensatz zu Spezifikationen, explizit keine vollständige Beschreibung einer Funktion enthalten sollen, sind Analysen bezüglich Vollständigkeit nicht sinnvoll. Schwache Formulierungen, die in Aspekten Interpretationsspielraum lassen, stellen bei User Stories aufgrund des agilen Prozesses ein geringes Problem dar. Sie können sogar notwendig sein, um zu verdeutlichen, dass noch Klärungsbedarf besteht.

Sillitti et al. befragten sowohl Firmen, die agile Prozesse einsetzen, als auch solche, die dokumentationsgetrieben entwickeln, zu Kommunikationsproblemen mit Kunden [67]. Dokumentationsgetriebene Methoden werden in der Arbeit als solche bezeichnet, die vor Entwicklungsbeginn eine möglichst vollständige Analyse der Anforderungen durchführen. Im Gegensatz zur Arbeit mit agilen Methoden ist die (laut Studie häufig auftretende) Änderung von Anforderungen für diesen Ansatz problematisch. Der Großteil der Befragten empfand die Fähigkeit von Kunden, Anforderungen komplett, präzise und korrekt zu spezifizieren als nicht sehr zufriedenstellend. Die von Sillitti et al. identifizierten Hauptursachen dafür war für mit beiden Ansätzen arbeitende Unternehmen:

1. Mangel an Deutlichkeit bezüglich Geschäftszielen („lack of clarity in business objectives“)

2. sprachliche Missverständnisse („communication-language-communication channel“)

3. Informations-/Wissensknappheit („scarcity of information/knowledge“)

In der dokumentationsgetriebenen Entwicklung müssen diese Probleme durch hohe Qualitätsanforderungen an die Erstellung von Anforderungsdokumenten vor Projektbeginn gemindert werden. Agil operierende Unternehmen nutzen stattdessen die direkte Kommunikation mit dem Kunden während des Projektes zur Verringerung der Probleme. Allerdings ist die starke Abhängigkeit von der Kundenkommunikation auch ein mögliches Problem für agile Prozesse. So nannten alle agil arbeitenden Unternehmen Zeitmangel des Kunden als eine Ursache für Kommunikationsschwierigkeiten. Dies entspricht auch der Erkenntnis Cohns, nach der der Kunde beziehungsweise dessen Vertretung die höchste Auslastung in einem agil entwicklenden Team aufweist [25].

Wie wir feststellten, ist aus oben genannten Gründen eine umfassende qualitative Einschätzung einer User Story bezüglich Vollständigkeit und schwacher Formulierungen nicht zielführend. Andererseits ist die in User Stories verwendete Terminologie ein wichtiger Ansatzpunkt für Qualitätsbetrachtungen.

Zave und Jackson betrachten die Disziplin der Anforderungsanalyse (Requirements Engineering) und identifizieren vier Fundamente, die bislang nur schwach beziehungsweise undeutlich erscheinen [79]. Eines dieser Fundamente ist, dass die gesamte Terminologie, die in der Anforderungsanalyse verwendet wird, sich auf die tatsächliche Umgebung, für die eine Maschine[1] gebaut werden soll, gründet (vgl. [79]). Jegliche formale Repräsentation begründet sich auf primitive Terme, die keine innere formale Bedeutung besitzen. In der Anforderungsanalyse kommen die Terme aus der realen Welt, also der tatsächlichen Umgebung, aus der die Anforderungen abgeleitet werden. Die Autoren nennen die Schwierigkeit, die Bedeutung eines Begriffes aus der realen Welt klar zu treffen, wohlbekannt, dennoch seien bislang wenige Arbeiten zur Lösung dieses Problems entstanden. Ein Lösungsvorschlag für das Problem verlangt die Angabe einer Definition für jeden neu eingebrachten Term.

Mit einem Glossar, das unter anderem zur Qualitätssicherung von User Stories verwendet wird, schlagen wir eine ähnliche Lösung vor. Dazu machen wir den Autoren einer User Story auf im Glossar nicht vorhandene und nicht definierte Begriffe aufmerksam. Außerdem werden ihm Begriffe empfohlen, die als Synonyme oder genauer spezifizierende Hyponyme im Glossar verknüpft sind.

3.2 Konzeptextraktion aus natürlichsprachlichen Texten zur Glossarbildung

Mit der Glossarextraktion aus großen Textbeständen beschäftigt sich ein Artikel von Park et al. [63]. Hierfür wird eine morphologische Analyse und eine Wortartanalyse durchgeführt, die gemeinsam die Extraktion von relevanten Glossareinträgen ermöglichen. Nur Nominalphrasen werden für die Extraktion berücksichtigt. Regeln für Kombinationen von Adjektiven, Konjunktionen und Nomen steuern die Aufnahme einer Phrase in das Glossar. Die Streichung von Personen- und Ortsnamen (mittels Named Entity Recognition), spezielle Zeichen enthaltenden Wörtern und zu viele Wörter enthaltenden Phrasen wird in Filterstufen durchgeführt.

Kof nutzt in einer Arbeit eine syntaktische Analyse zur Extraktion von Phrasen [51]. Mittels eines Zerteilers können einfacher als mit dem in der vorigen Arbeit genutzten Wortartmarkierers Nominalphrasen identifiziert werden. Extrahierte Phrasen werden einem Ontologieentwickler zum semiautomatischen Aufbau einer Ontologie für die Anforderungsanalyse vorgeschlagen.

Wir verwenden in der vorliegenden Arbeit ebenfalls eine syntaktische Analyse zur Extraktion bestimmter Phrasenteile. Die Filterungstechniken, die Park et al. einsetzen, nutzen wir in ähnlicher Form in der vorliegenden Arbeit.

Lapshinova-Koltunski und Heid beschreiben verschiedene Formen von zusammengesetzten Substantiven in der deutschen Sprache und die Folgen für die automatische Extraktion von Subkategorien [57]. So ist eine allgemein anerkannte Annahme gewesen, dass außer in Ausnahmefällen der *Kopf* der zusammengesetzten Substantive auch werttragend (*Kern*) ist. Der Kopf ist in germanischen und anderen *rechtsköpfigen* Sprachen das letzte (Teil-)Wort der Kombination. So enthält beispielsweise die Komposition „Nominalphrase“ den Kopf (und Kern) „Phrase“ und den *Modifikator*

[1] Der Begriff *Maschine* bezeichnet hier das Software-Artefakt eines *Systems*, das außer automatischen auch manuelle Komponenten enthalten kann (vgl. [79])

„nominal“. Durch Extraktion der Subkategorie „Phrase“ können dieser nun die Kompositionen „Nominalphrase“ und „Verbalphrase“ zugeordnet werden. Die Annahme, dass der Kopf fast immer werttragend ist, erweist sich jedoch als nicht zutreffend, wenn sie auch für die Mehrheit zusammengesetzter Substantive gilt. Wir extrahieren auf diese Weise Hyponymie-Beziehungen aus zusammengesetzten Substantiven, die jedoch gemäß der Erkenntnisse manueller Überprüfung bedürfen.

Ein Architekturvorschlag von Breitman und Leite beschäftigt sich mit der elektronischen Verwaltung von User Stories [20]. Eine Besonderheit des Rahmenwerks ist die Verknüpfung eines Glossars mit den User Stories. Auch wenn in der vorliegenden Arbeit keine Verwaltung von User Stories angestrebt wird, stellen wir den Zusammenhang zwischen User Stories und einem Glossar her.

3.3 Ontologien in der Softwaretechnik

Durch den Einsatz von Ontologien können wir verschiedene analysierte Artefakte, die wir zur Informationsgewinnung nutzen, in einer Wissensdatenbank repräsentieren. Im Gegensatz zu relationalen Datenbankensystemen bieten Ontologien automatische Inferenz nicht explizit angegebener Beziehungen. Dadurch sind Ontologien eine geeignete Form von Wissensdatenbanken für die Softwaretechnik. Beispiele für ihre Nutzung und eine Differenzierung der Herangehensweisen zur Anwendung in dieser Domäne finden sich in einem Buch von Calero et al. [21]. Unter anderem wird der Aufbau und Einsatz von Ontologien als Glossar bzw. Thesaurus darin beschrieben, was wir – wie in Abschnitt 3.1 beschrieben – in dieser Arbeit verwenden.

Ein weiterer Ansatz zur Gewinnung von Wissen über eine Software ist die Errichtung eines Software-Modelles aus Quelltext. Auch Quelltext enthält natürlichsprachliche Elemente wie Kommentare und die Bezeichnungen von Namensräumen, Klassen und deren Elementen.

Welty beschreibt die Anreicherung eines abstrakten Syntaxbaumes, der aus Quelltext extrahiert wird [75]. Dazu werden zusätzliche, im Syntaxbaum nur implizit vorhandene, Verknüpfungen aus dem Syntaxbaum abgeleitet. Das sind unter anderem inverse Verknüpfungen zu denen, die der abstrakte Syntaxbaum enthält (verwendet beispielsweise eine Funktion eine Variable, dann wird die Variable von der Funktion verwendet). Weiterhin werden Pfad-Verknüpfungen hinzugefügt, mit deren Hilfe übliche Aufgaben eines Softwareentwicklers vereinfacht werden können. Das Software-Modell basiert auf einer einfachen Code-Taxonomie, die von Devanbu et al. entworfen wurde und in dem Software-Informationssystem LaSSIE zum Einsatz kam [31].

Zhang et al. nutzen ein ebenfalls der Struktur des Software-Modelles von Devanbu et al. ähnliches Modell zur Gewinnung von Verbindungen zwischen Quelltext einer Software und deren Dokumentation [80]. Dieses wird weiterhin in einer Arbeit von Khamis et al. zur Analyse der Kommentarqualität in Java-Quelltext verwendet [49], [48], [47]. Vorwiegend wird es darin zur Überprüfung der Übereinstimmung von Kommentar und Quelltext genutzt. Witte et al. setzen das Software-Modell ein, um Entwicklern die Software-Pflege von Java-Programmen zu erleichtern [78] [77]. Gefundene Konzeptübereinstimmungen werden in der Dokumentation über eine Erweiterung der Entwicklungsumgebung *eclipse* mit Querverweisen versehen.

Wir haben die Strukturen und Methoden des Aufbaus des Software-Modelles in der vorliegenden Arbeit in angepasster Form dem Modell der Arbeit von Zhang et al. übernommen.

WordNET ist eine lexikalische Datenbank für die englische Sprache [61]. Diese enthält unter anderem auf ihre Stammformen reduzierte und definierte Nomen, die in Synonym-Mengen in einer Hierarchie angeordnet sind. Der Oberbegriff für alle Nomen, also die Wurzel der Hierarchie, ist *entity*. Beispielsweise sind die Begriffe *dalmatian* (Dalmatiner) und *Newfoundland* (Neufundländer) unter anderem Spezialisierungen des allgemeinen Begriffs *dog* (Hund). Die von WordNET bereitgestellten Beziehungen ermöglichen uns unter anderem, Mehrdeutigkeiten von Begriffen aufzulösen. Auf das Beispiel des Dalmatiners bezogen können wir in geographischem Kontext schließen, dass keine Hunderasse gemeint war, sondern ein Bewohner der Region Dalmatien. In dieser Arbeit kommt WordNET vorwiegend zur Bestimmung der Grundform von Wörtern zum Einsatz.

3.4 Nachvollziehbarkeit zwischen Anforderungsdokumenten und Quelltext

Verbindungen zwischen einem Anforderungsdokument und auf dessen Basis entstandenem Quelltext können meist nicht eindeutig erkannt werden. Verschiedene Verfahren kommen zur Anwendung, um zu einer im Anforderungsdokument beschriebenen Funktionalität deren Implementierung zu identifizieren. Grundlage vieler dieser Verfahren ist die Informationsbeschaffung.

Antoniol et al. stellen die Nachvollziehbarkeit zwischen Quelltext und Dokumentation durch Nutzung von Modellen der Informationsbeschaffung her [14]. Dazu erstellen sie zu jeder Komponente des Quelltextes eine Abfrage, wobei alle darin vorkommenden Bezeichner in Einzelwörter aufgespaltet und normalisiert werden. Die Dokumentationstexte werden in Wörter aufgespalten, mittels einer Stoppwortliste gefiltert und zur Bildung von Stammformen einer morphologischen Analyse unterzogen. Ein Klassifizierer liefert eine gewichtete Liste von Dokumenten zu jeder Abfrage zurück. Dabei betrachten sie einerseits das probabilistische Modell und das Vector Space Model und evaluieren die Eignung für diese Aufgabe. Der Aufbau des Indexes der Abfragen und Dokumente unterscheidet sich von Modell zu Modell.

Ein weiteres Modell der Informationsbeschaffung, *Latent Sematic Indexing*, wird in der Arbeit von Marcus und Maletic betrachtet [60]. Dieses hat den Vorteil, Zusammenhänge zwischen Begriffen, wie Synonymie-Eigenschaften, automatisch zu erfassen. Nachteilig ist der hohe Aufwand vor allem beim Hinzufügen neuer Dokumente, da der gesamte Index neu erstellt werden muss. Zur Anwendung von LSI werden alle Dokumente (auch der Quelltext) in Teildokumente ähnlicher Größe aufgeteilt. Dabei ist die Granularität der Aufspaltung wichtig. Während Sätze zu wenig Informationen für LSI enthalten, sind Kapitel zu groß. Marcus und Maletic verwenden daher in der Dokumentation Absätze und – je nach Quelltext-Art (prozedural/objektorientiert) – Funktionen oder Klassen als Einzeldokumente. Diese Dokumente werden über ein semantisches Ähnlichkeitsmaß miteinander in Beziehung gesetzt. Im Vergleich zur Arbeit von Antoniol et al. erreichten sie eine leichte Verbesserung der Ausbeute und eine starke Verbesserung der Präzision.

Unser Ansatz unterscheidet sich von den erläuterten Verfahrensweisen in mehrerer Hinsicht. Einerseits verwenden wir Anforderungsdokumente, die jeweils vor der Implementierung der beschriebenen Funktionalität entstehen. Andererseits suchen wir nicht direkt die Beziehungen von den natürlichsprachlichen Dokumenten zu Quelltext, sondern zu ebenfalls natürlichsprachlichen Testskripten bzw. Testschritten. Verbindungen zum Quelltext erhalten wir durch die Implementierungen der Testschritte, die eindeutig zueinander zugeordnet werden können.

Ein Ansatz von Ziftci und Krueger nutzt ausführbare Spezifikationen, *features* genannt, zur Gewinnung von Verknüpfungen zu automatisierten Tests [81]. Mithilfe eines Profilers speichern sie besuchte Software-Komponenten sowohl während der Ausführung der Spezifikation als auch während der Testläufe. Durch Vergleich der Komponenten ordnen sie Spezifikation zu automatisierten Testfällen zu. Im Vergleich zu Verfahren der Informationsbeschaffung resultiert diese Methode in hoher Präzision bei hoher Ausbeute. Es ist unklar, ob dieses Verfahren auch zur Unterstützung von Test-Wiederverwendbarkeit nutzbar ist. Doch auch wenn das der Fall ist, so ist Aufwand zur Erstellung der ausführbaren Spezifikationen notwendig, der von Softwareentwicklern aufgebracht werden muss. Da die Definition des Testskriptes aus der User Story möglichst Tester durchführen sollen, ist dieser Ansatz in der vorliegenden Arbeit nicht anwendbar.

Nachvollziehbarkeit wird in unterschiedlicher Weise genutzt (siehe [14], [60]). Dazu zählt die Unterstützung von Wartungsarbeiten an der Software, und, eng damit verbunden, die Erleichterung des Programmverständnisses. Des Weiteren ist die Wiederverwendung von Softwarekomponenten und Bibliotheken ein Ziel der Nachvollziehbarkeit. In der vorliegenden Arbeit stellen wir Nachvollziehbarkeit her, um wiederverwendbare Testschritte innerhalb eines Projektes zu finden und vorzuschlagen. Außerdem sollen auch Programmkomponenten vorgeschlagen werden, durch die die Implementierung von Testschritten erleichtert wird. Somit verwenden wir die Nachvollziehbarkeit für beide genannten Aufgaben.

3.5 Zusammenfassung

Wie eingangs erwähnt, konnten wir keine Arbeiten finden, die in ähnlichem Umfang das Thema dieser Arbeit betreffen. Teilkomponenten des vorgeschlagenen Systemes umfassen

- die Qualitätsanalyse natürlichsprachlicher Anforderungsdokumente,
- die Extraktion von Konzepten aus natürlichsprachlichen Texten mit dem Ziel der Glossarerstellung,
- die Nutzung von Ontologien in der Softwaretechnik und
- die Gewinnung von Nachvollziehbarkeit zwischen Anforderungsdokumenten und Quelltext.

Dazu existieren jeweils Arbeiten, die Teilaspekte betrachten. Die vorliegende Arbeit hat spezifische Ausgangspunkte, namentlich einen agilen Entwicklungsprozess und verhaltensgetriebene Entwicklung. Daher unterscheiden sich die Vorhergehensweisen

teilweise erheblich von denen der genannten Arbeiten, auch wenn das Grundproblem ein ähnliches ist. Vorwiegend betrifft dies das Problem der Qualitätsanalyse natürlichsprachlicher Anforderungsdokumente und die Gewinnung von Nachvollziehbarkeit.

4. Konzept

Wie bereits in Kapitel 1 motiviert, liegen die Ansatzpunkte zur Analyse der Problemstellung in der Formulierung der Anforderungsdokumente in Form von User Stories und der Testskripte. Diese Artefakte analysieren wir in Abschnitt 4.1. Die vorgeschlagene Lösung erläutern wir in Abschnitt 4.2.

4.1 Analyse

Die in der Einleitung genannten Kommunikationsschwierigkeiten führen zu vielseitigen Problemen im Verlauf des Entwicklungsprozesses. Dabei sind User Stories, die Unklarheiten enthalten, nicht zwingend problematisch für den Entwicklungsprozess, denn deren Charakterisierung als eine „Vereinbarung zum Gespräch" erlaubt und bedingt sogar die Aussparung von Details. Erst bei der Berücksichtigung einer User Story für eine Entwicklungsphase (Sprint) und während der Entwicklung der mit der User Story beschriebenen Funktionalität werden Details erarbeitet. Doch auch dann sollte man der User Story nur solche Vereinbarungen hinzufügen, die sich die Entwickler schlecht merken können oder die für funktionale Tests interessant sind.

Der in dieser Arbeit betrachtete Aspekt umfasst User Stories und deren funktionale Tests. Ein funktionaler Test, in der Literatur häufig auch als *Akzeptanztest* bezeichnet, basiert auf Kriterien, die in der User Story oft in Kurzform angegeben werden. Werden alle zu einer User Story gehörenden funktionalen Tests erfolgreich absolviert, kann der Kunde bzw. der Product Owner die User Story abnehmen.

Notwendige Voraussetzung für das Funktionieren dieses Prozesses ist rege Kommunikation. Die Etablierung einer konsistenten Sprache im Team, die man sich durch die erhöhte Kommunikation erhofft, tritt jedoch nicht immer ein. Wenn die Terminologie in den User Stories nicht konsistent ist, hat dies Hindernisse bei der effizienten Entwicklung von Funktionalität und deren Tests zur Folge.

Auch wenn sowohl User Story als auch Testskripte (mit strukturellen Einschränkungen) in natürlicher Sprache verfasst sind, ist die Entwicklung eines Testskriptes zu einer User Story aufwändig. Einerseits werden nicht alle Kriterien explizit genannt, andererseits sind auch explizit genannte Kriterien in einem geringen Detaillierungsgrad gegeben. Der Aufwand der Formulierung lässt sich erahnen, wenn man die

As a user of the address book application, I want to add new persons in order to store their addresses. Notes: - Fields that can be stored are last name and first name, address, city, state, zip code and phone number. Criteria: - Test that the entry is stored.	**Feature:** As a user of the address book application, I want to add persons in order to store their addresses. **Scenario:** Add person to empty list **Given** the address book is empty **When** I add the person "John Smith" **Then** the address book has 1 entry **And** the name of the 1. entry is "John Smith"

Abbildung 4.1: Gegenüberstellung einer exemplarischen User Story und eines dazugehörigen Testskriptteiles

Gegenüberstellung einer möglichen User Story und des dazugehörigen Testskriptes in Abb. 4.1 betrachtet.

Ein Testskript besteht, wie in Abschnitt 2.2 beschrieben, aus Szenarien, die wiederum Testschritte enthalten. Jeder Testschritt besteht aus einer logischen Aktion, wie zum Beispiel einer Benutzeraktion oder einer Ergebnisprüfung. Die User Story beschreibt die Funktionalität allgemein, während der Tester konkrete Testfälle konstruieren muss, um das Verhalten der Funktionalität zu überprüfen. Daher kann man nicht von einer Übersetzung der User Story in ein Testskript sprechen. Auch wird aus diesem Grund eine maschinelle Erstellung eines Testskriptes zu einer User Story in dieser Arbeit nicht angestrebt.

4.1.1 Qualitätsmängel der User Stories

Die im Rahmen dieser Arbeit betrachteten User Stories haben teilweise deutliche Mängel. In vielen Fällen sind Notizen und Kriterien in problematischer Reihenfolge und Form; zum Beispiel dient in Einzelfällen auch die Historie eines E-Mail-Wechsels zur Detaillierung eines Aspektes der betreffenden User Story.

Wie in Abschnitt 2.1 erläutert, besteht ein Vorteil von User Stories in Kartenform in ihrer Platzbeschränkung. Dadurch werden Autoren dazu gezwungen, nur notwendige Informationen aufzunehmen. Werden mehr Informationen benötigt, kann das auch ein Hinweis auf eine sinnvolle Aufspaltung der User Story in mehrere Stories sein. Die elektronisch gespeicherten User Stories des betrachteten Projektes enthalten in Einzelfällen viel Text, der teilweise auf wenige Detailinformationen gekürzt werden könnte.

Die Strukturierung der User Stories ohne Konvention, wie verschiedenartige Aufzählungspunkte und vielfältige Überschriftsvariationen, erschweren die Lesbarkeit und maschinelle Auswertbarkeit.

Häufiges Auftreten spezieller Zeichen des Unicodezeichensatzes, beispielsweise Ellipsen („. . .“) und Aufzählungspunkte, sind weiterhin problematisch für maschinelle Auswertungen, da viele Bibliotheken diese nicht unterstützen.

Ausgehend von diesen Mängeln haben wir einen Leitfaden zur Erstellung von User Stories entworfen, der sich in Anhang B befindet. Dieser fasst die Grundsätze der Arbeit mit User Stories zusammen und gibt Möglichkeiten der Erleichterung maschineller Auswertung an. Darüber hinaus enthält der Leitfaden Kriterien, anhand derer eine User Story von ihrem Ersteller bewertet werden kann.

4.1.2 Schwierigkeiten bei der Erstellung von Testskripten

BDD-Testprogramme wie Cucumber und SpecFlow vereinfachen die Aufspaltung eines Tests in kleine Einheiten, die Testschritte. Diese kann man in anderen Testskripten wiederverwenden, wodurch die Implementierungsarbeit entfällt. Des Weiteren kann der Testschritt parametrisierbar entworfen werden, um die Wiederverwendbarkeit weiter zu steigern. Ein Tester sollte daher auch auf Testschritte achten, die möglicherweise wiederverwendet werden können, um redundante Implementierungen zu vermeiden. Hierzu ist jedoch entweder umfassendes Wissen über die existierenden Tests notwendig oder aber aufwändige Recherche in den Testquelltexten.

Erschwert werden Verständnis und Wiederverwendung eines Testschrittes weiter, wenn das Testskript statt von einem Tester von einem Softwareentwickler geschrieben wird, der unnötige Implementierungsdetails verwendet. So ist der Testschritt „`Wenn ich eine Adresse in das Adressbuch einfüge`“ leichter verständlich als ein solcher: „`Wenn ich in die Tabelle "Adresses" einen Datensatz einfüge`“.

Die Formulierung der Testskripte hängt dabei von dem Hintergrund bzw. der Domäne des Erstellers ab. Testschritte, die von einem Entwickler bezüglich einer zu testenden Softwarebibliothek geschrieben werden, sehen sicherlich anders aus als Testschritte, die ein Immobilienkaufmann zum testen einer Objektverwaltung formuliert. Testskripte stellen eine persistierte Form der (vergänglichen) Anforderungsdokumente dar. Daher gilt für sie die Feststellung von Zave und Jackson, dass sich die gesamte Terminologie auf die tatsächliche Umgebung gründet, in ähnlichem Maße [79].

4.1.3 Schwierigkeiten bei der Testimplementierung

Muss ein Testschritt implementiert werden, verwendet der Entwickler Softwarekomponenten, die die gewünschte Aktion oder Prüfung ausführen. Die Funktionalität ist oft jedoch nur über weitere Komponenten erreichbar, die dem Entwickler unter Umständen nicht bekannt sind. Auch ist der Entwickler, der die Testimplementierung durchführt, nicht zwingend derjenige, der die Implementierung der Funktionalität vornimmt.

4.2 Entwurf

Das Grundelement dieser Arbeit ist eine Wissensbasis, die wir sowohl aus Quelltextelementen als auch aus Testskript-Teilen und natürlichsprachlichen Elementen (im folgenden NSEs genannt) aus Anforderungsdokumenten aufbauen. Diese Wissensbasis ermöglicht die Herausbildung und Nutzung von Zusammenhängen zwischen NSEs und Programmteilen. Gleichzeitig beinhaltet sie ein Glossar, das der Etablierung konsistenter Begrifflichkeiten dienen soll. Ihr Aufbau ist in Abschnitt 4.2.1 beschrieben.

Die hier beschriebenen Lösungsansätze bauen weitgehend auf der Auswertung dieser Zusammenhänge auf. Dies ist der Fall in der Qualitätssicherung der Anforderungsdokumente in Abschnitt 4.2.2 und den Vorschlagssystemen für Testschritte in Abschnitt 4.2.3 und API-Komponenten in Abschnitt 4.2.4. Die beschriebenen Verfahren wurden prototypisch in F-TRec (*Functional Test Recommender*) implementiert. Kapitel 6 evaluiert das vorgestellte Verfahren und F-TRec.

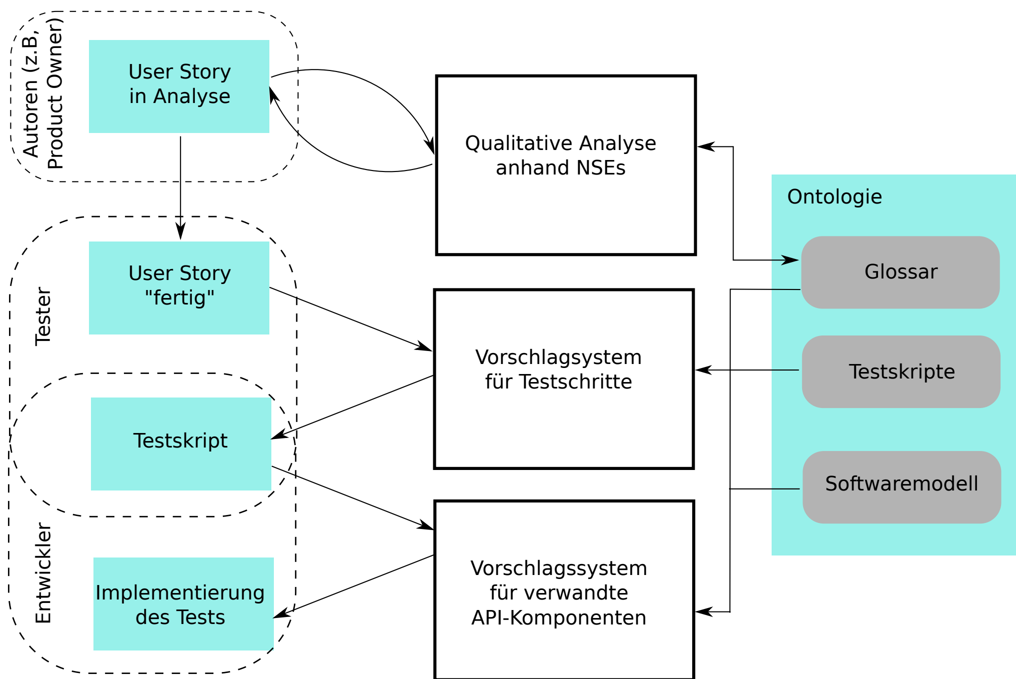


Abbildung 4.2: Unterstützung von User Story-Autoren, Testern und Softwareentwicklern mittels F-TRec

Abb. 4.2 zeigt den Weg von der Erarbeitung der User Story bis zur Definition von Tests mit der Unterstützung von F-TRec. Über eine qualitative Analyse, die aus der Betrachtung der in der User Story verwendeten Terminologie hervorgeht, sollen Autoren von User Stories Hinweise auf Schwächen erhalten. Zu einer von Testern und Entwicklern bearbeiteten User Story sollen diese Vorschläge erhalten, die die Erstellung der Testskripte und Testschritt-Implementierungen erleichtern. Auch dazu dient die Terminologie der User Story.

Wir nehmen für die nachfolgenden Betrachtungen an, dass die Analysegrundlage ein laufendes Projekt ist und F-TRec zur Formulierung neuer User Stories und Testskripte eingesetzt wird. Es existiert daher einerseits Quelltext, der anhand von User Stories entwickelt wurde. Andererseits wurden funktionale Tests entwickelt, die geeignet sind, die Erfüllung von User Stories zu prüfen. Für ein neu beginnendes Projekt sind selbstverständlich weder Testschritte noch Quelltext vorhanden, die zur Analyse mit F-TRec herangezogen werden können.

Bei Fortschreiten des Projektes wird die Datenbasis jedoch zunehmen. Darüber hinaus werden später hinzugefügte Komponenten mit höherer Wahrscheinlichkeit auf existierenden Komponenten aufbauen. Dadurch sollte F-TRec immer präzisere Ergebnisse liefern.

4.2.1 Aufbau der Wissensbasis

Wir verwenden eine Ontologiestruktur, die Konzepte für Quelltextelemente, Testskripte und NSEs nebst auf Konzepten definierten Relationen bereitstellt. Den Aufbau dieser Struktur zeigen wir in Abschnitt 4.2.1.1.

In diese werden Quelltextelemente und Teile der Testskripte eingefügt. Desweiteren fügen wir der Ontologie aus Testskripten und Anforderungsdokumenten extrahierte

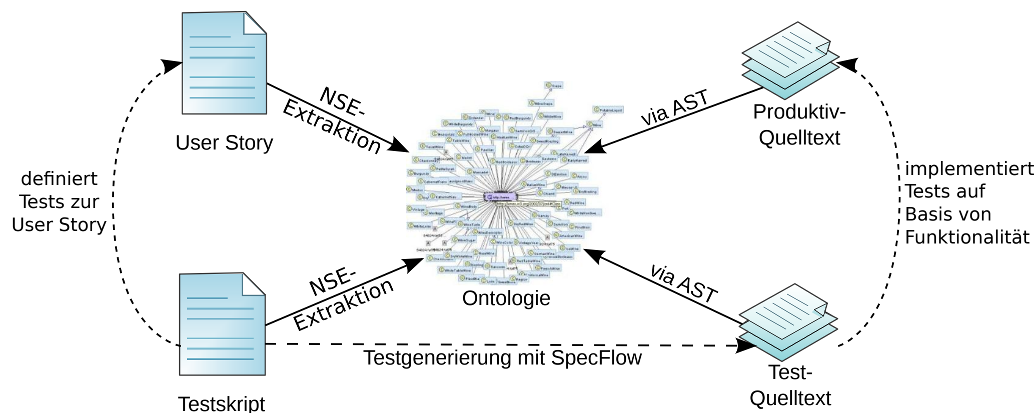


Abbildung 4.3: Ein Überblick der Quellen zur Ontologie-Befüllung.

NSEs hinzu. Abb. 4.3 veranschaulicht die Quellen, die zur Extraktion von Konzepten herangezogen werden, und ihre Zusammenhänge. Nach der Befüllung werden NSEs mit Implementierungsdetails verbunden. Diese Schritte erläutern wir in Abschnitt 4.2.1.2.

4.2.1.1 Aufbau der Ontologiestruktur

Die Ontologiestruktur enthält eine Hierarchie von Grundkonzepten, denen extrahierte Informationen bei der Befüllung als Instanzen zugeordnet werden können. Zusätzlich enthält sie vordefinierte Relationen, die wir für die Verknüpfungen der Instanzen benötigen. Dadurch erhalten wir eine Schablone für die Befüllung und die Nutzung der Informationen der resultierenden Ontologie. Zur Erstellung des Softwaremodelles wurde der Ontologieeditor *protégé* eingesetzt (siehe [8]).

Softwaremodell

In der Ontologie wird die gesamte Struktur eines Programmes abgebildet. Dies erreichen wir, indem für die strukturellen Sprachelemente wie Namensräume, Typen und deren Eigenschaften und Methoden Konzepte in der Ontologie hinterlegt werden. Basierend auf Zhangs Quelltext-Ontologie (siehe [80]) haben wir die in Abbildung A.1 unterhalb des Knotens *CSharpThing* dargestellte Struktur entworfen, die an die Elemente der im Projekt verwendeten Sprache angepasst ist. Relationen, die wir für das Softwaremodell entworfen haben, umfassen strukturelle Beziehungen wie die Zugehörigkeit von Typen zu einem Namensraum oder Eigenschaftsbeziehungen. Des Weiteren werden Nutzungsbeziehungen wie Aufrufe oder Instantiierungen abgebildet.

Testskripte

Auch für Testskripte gibt es eine Konzeptstruktur, die sich aus den Teilen eines Testskriptes – analog zu der in Abschnitt 2.2 angegebenen Beschreibung – zusammensetzt:

- *FeatureHead* ist die strukturelle Wurzel eines Testskriptes, die einen Identifikator und die Beschreibung der zugehörigen User Story erhält

- *Scenario* dient als Zusammenfassung eines logischen Testablaufes; auch hier kann eine Beschreibung angegeben werden
- *ScriptStep* repräsentiert einen Testschritt, wobei sowohl der Text als auch die Reihenfolge des Auftretens innerhalb eines Szenarios angegeben werden können

Die zugehörigen Relationen stellen strukturelle Beziehungen dar; so kann darüber eine *Scenario*-Instanz als Teil einer *Feature*-Instanz deklariert werden. Dem *Scenario*-Element weitgehend entsprechende *Background*- und *ScenarioOutline*-Elemente behandeln wir ebenso wie *Scenario*-Elemente.

Natürlichsprachliche Elemente/Glossar

Für die NSEs existieren Konzepte für Verben und Nomen. Relationen, die die Erstellung des Glossars ermöglichen, umfassen Beziehungen zwischen NSE-Instanzen, wie Synonym-Beziehungen für Nomen. Die Vergabe einer Beschreibung des NSEs ist außerdem durch eine auf den Konzepten für Verben und Nomen vergebene Dateneigenschaft möglich.

Relationen zur Verknüpfung dieser Bestandteile

Weitere Relationen ermöglichen die Verknüpfung von Instanzen der genannten Bereiche. So können Glossareinträge mit Testskripten verknüpft werden, und Testschritte mit entsprechendem Quelltext.

4.2.1.2 Hinzufügen und Verknüpfen von Quelltext-Elementen

Zur Befüllung der Ontologie analysieren wir den Projekt-Quelltext (sowohl Produktiv- als auch Testquelltext) mithilfe eines Zerteilers. Wir laufen den entstandenen abstrakten Syntaxbaum (AST) ab und generieren für alle strukturellen Elemente eine Konzeptinstanz in der Ontologie (z.B. Klassen, Methoden usw.). Diese Instanzen verknüpfen wir über die dafür vorgesehenen Relationen gemäß der Quelltextstruktur. Wir extrahieren des Weiteren Vererbungshierarchien und verbinden die entsprechenden Instanzen miteinander.

Die Inhalte von Methoden und ähnlichen Elementen stellen eine weitere Quelle für Verknüpfungen dar; so nehmen wir Methodenaufrufe und Objekterstellungen in die Ontologie auf.

Eine detaillierte Darstellung der Implementierung und der hierfür verwendeten Komponenten ist in Abschnitt 5.4.1 gegeben.

4.2.1.3 Einbindung der Inhalte von Testskripten

Für jede User Story existiert ein Testskript. Die im Testskript verwendeten Testschritte wiederum haben eine Implementierung, die Komponenten der Software verwendet. Dadurch sind Testskripte ein ideale Brücke zwischen User Stories und Quelltext.

Die Testskripte zerlegen wir mit einem einfachen Zerteiler in ihre Einzelbestandteile. Ein Testskript besteht aus einem Titel und Testszenarien. Ein Testszenario wird

```
1 Feature: As a user of the address book application,
2          I want to add persons in order to store
3          their addresses.
4
5     Scenario: Add person to empty list
6         Given the address book is empty
7         When I add the person "John Smith"
8         Then the address book has 1 entry
9         And the name of the 1. entry is "John Smith"
```

Listing 1: Ein einfaches Testskript für das Hinzufügen einer Adresse.

wiederum durch einen Titel und Testschritte beschrieben; Ein Testschritt ist ein Satz in natürlicher Sprache beginnend mit *Given*, *When* oder *Then*. Eine ausführlichere Beschreibung des Aufbaus von Testskripten befindet sich in Abschnitt 2.2. Ein Beispiel des Tests einer Adressbuchapplikation wird in Listing 1 gezeigt.

Alle Testskriptelemente fügen wir der Ontologie hinzu, wobei die Reihenfolge der Schritte in Relationen festgehalten wird.

4.2.1.4 Extraktion natürlichsprachlicher Elemente

Die letzte Zutat für unsere Ontologie sind NSEs, die wir aus User Stories und Testschritten extrahieren. Wir fügen der Ontologie nur (zusammengesetzte) Substantive und Verben hinzu, da wir vorwiegend diese Wortarten im Kontext funktionaler Verhaltensbeschreibungen als bedeutungstragend erachten.

Hierzu verwenden wir einen Zerteiler für natürliche Sprache, der einen Syntaxbaum zu einem betrachteten Satz liefert. In diesem Syntaxbaum sind einerseits die Wortarten der einzelnen Wörter angegeben. Andererseits wird auch die grammatische Struktur ersichtlich, die wir zur Extraktion zusammengesetzter Substantive (*Substantivkomposita*) nutzen. In der englischen Sprache werden, anders als beispielsweise in der deutschen Sprache, Komposita zumeist ohne besondere Kennzeichnung der Zusammengehörigkeit der Wörter verwendet. Die Wörter, die das Kompositum bilden, werden isoliert hintereinander geschrieben; auch Bindestriche werden selten verwendet.

Durch Lemmatisierung werden extrahierte Begriffe normalisiert. Außerdem sorgen Filterstufen für eine Einschränkung des in das Glossar aufgenommenen Vokabulars. Diese umfassen die Entfernung von Personennamen und für das Projekt unwichtigen Begriffen, die in einer Stoppwortliste gepflegt werden. Die Stufen der Verarbeitung natürlicher Sprache von der Eingabe der Einzelsätze bis zu der Speicherung in der Ontologie sind in Abb. 4.4 schematisch dargestellt. Abschnitt 5.3 erläutert den Prozess der Extraktion detailliert.

Komposition, also die Verbindung mehrerer Wörter zu einem neuen, ist eine Hauptquelle für die Erweiterung des Wortschatzes (vgl. [56], [58]). Das grammatische Hauptwort (*Kopf*) eines Substantivkompositums steht in germanischen Sprachen an letzter Stelle und bestimmt damit Kasus, Numerus und Genus des Kompositums. Dies wird als *Rechtsköpfigkeit* bezeichnet. Des Weiteren enthält ein Kompositum auch ein bedeutungsbestimmendes Wort, hier *Kern* genannt (siehe [35]). Wie in Abschnitt 3.2 beschrieben, ist in der Mehrzahl der Fälle der Kopf identisch mit dem Kern. Dies können wir zur Extraktion von Subkategorien, bzw. Hyponym-Beziehungen zwischen dem Kopf und dem Kompositum heranziehen. Da jedoch ein bedeutender Teil der

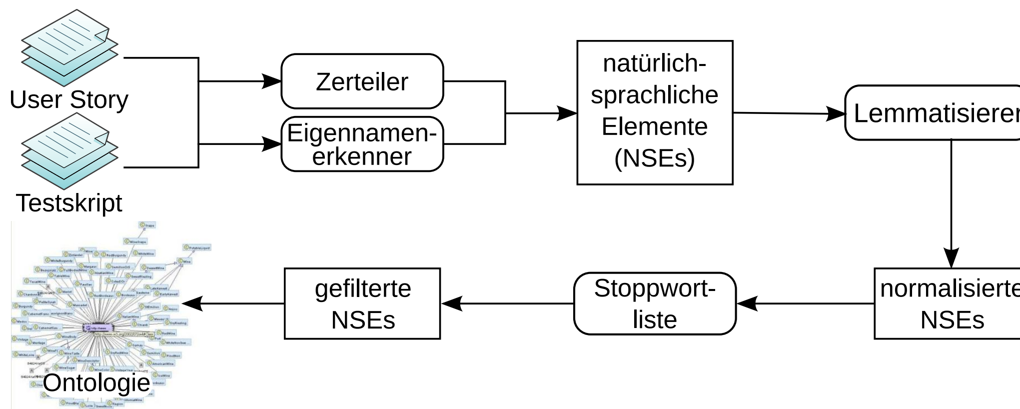


Abbildung 4.4: Prozess der Extraktion natürlichsprachlicher Elemente.

Komposita nicht diesem Muster folgt, bedarf die Nutzung dieses Zusammenhangs einer manuellen Überprüfung (siehe [57], [35]). Daher kann der Ansatz zur Extraktion dieser Beziehungen nur semi-automatisch erfolgen. Die Trennung zusammengeschriebener Komposita ist aufwändig und fehleranfällig, somit empfiehlt sich, nur solche Komposita dieser Analyse zu unterziehen, die entweder nicht verbunden sind, oder anhand eines Bindestriches leicht getrennt werden können.

Weitere semantische Verknüpfungen können im Kontext dieser Arbeit nicht mithilfe maschineller Auswertung gebildet werden. Einerseits ist der Umfang eines aus User Stories gebildeten Korpus zu gering für statistische Auswertung. Andererseits sind existierende Korpora in den meisten Fällen nicht nutzbar aufgrund der spezialisierten Sprache der Domänenexperten und anderer am Entwicklungsprozess Beteiligter. Daher müssen solche Verknüpfungen manuell gepflegt werden, was durch das Glossar in Abschnitt 4.2.2.1 ermöglicht wird.

Direkte anaphorische Verbindungen über Pronomen oder Proformen können über zusätzliche Werkzeuge wie JavaRAP aufgelöst werden [65]. In den betrachteten User Stories wird Anaphorik jedoch kaum genutzt, daher erfolgt in dieser Arbeit keine Behandlung.

4.2.1.5 Verknüpfung der Ontologie-Individuen

Für jede erfüllte User Story existiert ein Testskript, das das in der User Story beschriebene Verhalten testen soll. Der Zusammenhang zwischen einer User Story und einem Testskript ist nicht notwendigerweise explizit. Er kann jedoch durch Konventionen, wie die Angabe desselben Identifikators in User Story und Testskript explizit hergestellt werden. In dem betrachteten Projekt ist dies der Fall, wodurch wir den Zusammenhang automatisch ableiten können.

User Stories werden als Anforderungsdokumente mit begrenzter Lebensdauer nicht reproduzierbar in der Ontologie abgespeichert. Jedoch werden die aus ihnen extrahierten NSEs mit der entsprechenden Repräsentation des Testskriptes in der Ontologie verknüpft. Das dazugehörige Testskript wird im Kontext der User Story betrachtet. Alle aus einem Testschritt extrahierten NSEs verbinden wir mit diesem Testschritt. Also ist ein NSE in der Ontologie entweder mit einem Testskript verbunden oder mit einem Testschritt, oder mit beidem. NSEs, die mit einem Testskript verknüpft sind, sind transitiv mit allen Testschritten des Testskriptes verbunden.

Da wir nur erfüllte User Stories zur Befüllung der Ontologie berücksichtigen, können wir von einer fertigen Implementierung der Testschritte ausgehen. Zur Verdeutlichung der Zusammenhänge zwischen Testskripten und deren Testquelltext betrachten wir im folgenden die Arbeitsweise von BDD-Werkzeugen wie SpecFlow [12].

Legt ein Tester ein Testskript für eine User Story an, so wird einerseits das Testskript in ausführbare Testläufe in der Zielsprache überführt. Die ausführbaren Testläufe lösen anhand des Textes der Testschritte die entsprechenden Testschritt-Implementierungen auf und führen diese aus. Sie beinhalten jedoch – außer dem Text des Testschrittes – keine inhaltlich für den Test relevanten Details.

Die Implementierung eines Testschrittes nimmt der Entwickler in einer Methode vor, die er per Konvention gemäß des Testschrittes benennt. Dabei verwendet man z.B. CamelCase; die Wörter des Testschritttextes werden also jeweils mit großem Anfangsbuchstaben aneinandergereiht. Der Entwickler implementiert in der Methode das durch den Text des Testschrittes ausgedrückte Verhalten. Die Implementierung des Testschrittes „`When I click on the button Open`“ könnte beispielsweise die Funktionalität ausführen, die mit der entsprechenden Schaltfläche verbunden ist.

Der Zusammenhang zwischen einem Testschritt und der diesen Testschritt implementierenden Methode wird durch Quelltextannotation bzw. -attributierung hergestellt. Die Annotation für eine zu Zeile zwei des Listing 1 gegebene Implementierung könnte beispielsweise `When("I add the person\"John Smith\"")` lauten. Wie in Abschnitt 4.1.2 beschrieben, erlaubt SpecFlow parametrisierte Testschritt-Implementierungen. Die von mehreren Testschritten genutzte Implementierung des genannten Schrittes kann so beliebige Zeichenketteninitialisierungen ermöglichen. Die Annotation für eine wiederverwendbare Testschritt-Implementierung ist durch `When("I add the person\"(.*)\"")` gegeben. Da die parametrisierte Variante des Testschrittes in Form eines regulären Ausdrucks gegeben ist, können sowohl SpecFlow als auch F-TRec einen Testschritt zu seiner Implementierung auflösen. Wir verknüpfen Testschritte und diese implementierende Testmethoden in der Ontologie während der gleichzeitigen Betrachtung von User Story und dazugehörigem Testskript.

So entstehen indirekt Verbindungen zwischen NSEs und Quelltextbereichen, die von den Testschritt-Implementierungen referenziert werden.

4.2.2 Qualitätssicherung der Anforderungsdokumente

Als wichtigen Ansatzpunkt der Qualitätssicherung von User Stories identifizierten wir bereits die darin verwendete Terminologie. Zu diesem Zweck nutzen wir ein semiautomatisch aufgebautes Glossar, das integraler Bestandteil der Ontologie ist. Das Glossar selbst wird in Abschnitt 4.2.2.1 erläutert, die Nutzung zur Analyse der User Story in Abschnitt 4.2.2.2.

Andere Qualitätsmängel, wie wir sie in Abschnitt 4.1 erörterten, können vor allem durch ein tieferes Verständnis des Sinnes von User Stories erreicht werden. Dazu ist ein Leitfaden entstanden, der kurz in Abschnitt 4.2.2.3 beschrieben wird.

4.2.2.1 Glossarentwurf

Glossare dienen in der Softwaretechnik dazu, dass sich unter den an einem Softwareentwicklungsprozess Beteiligten eine gemeinsame Sprache entwickelt (vgl. [41]).

Dies umfasst sowohl Begriffe aus der Anwendungsdomäne der Software (beispielsweise der Begriff *Fraktur* im Kontext medizinischer Software), als auch solche aus der Softwareentwicklung (zum Beispiel der Begriff *Panel* für ein bestimmtes Element der graphischen Oberfläche). Ein Glossar muss die Möglichkeit bieten, Begriffe zu beschreiben, sodass Beteiligte die richtige Verwendung nachschauen können.

In einem hochkommunikativen Entwicklungsprozess, wie er in agilen Entwicklungsprozessen und durch die Arbeit mit User Stories gefordert wird, soll sich das Wissen im Team angleichen. Beteiligte verwenden trotzdem ungenaue Begrifflichkeiten, was durch den informellen Charakter der User Stories verstärkt wird. Dies kann im Entwicklungsprozess eine höhere Arbeitsintensität hervorrufen oder sogar zu Fehlern führen (siehe [51]). Präzise Begriffe sind jedoch vor allem für die automatische Assistenz bei der Entwicklung der Testskripte und der Testimplementierung wichtig.

NSEs, die wir aus User Stories und Testschritten extrahieren, können solche Begriffe sein. Nur Domänenexperten sind in der Lage, Beschreibungen der Begriffe zu liefern. Es existieren Verfahren, mit deren Hilfe semantisch verwandte oder gleiche Terme identifiziert werden können. Hierfür benötigt man jedoch große Korpora, die im Allgemeinen für die hochspezialisierte Sprache, die Projektbeteiligte nutzen, nicht existiert. Daher können wiederum nur Domänenexperten Verbindungen zwischen synonym oder hyponym verwendeten Termen liefern. Unterstützung können wir über die in Abschnitt 4.2.1.4 erläuterte Extraktion von Hyponymiebeziehungen aus Komposita anbieten.

So wird das Glossar semiautomatisch während der Projektlaufzeit befüllt. Das Ergebnis kann in fachlich ähnlichen Projekten wiederverwendet werden, sodass gleich zu Beginn eine Sammlung von Basisbegriffen festgelegt ist. Die Integration des Glossars in die Ontologie bietet einerseits den am Projekt Beteiligten die Bildung eines konsistenten Vokabulars, andererseits erleichtert sie die maschinelle Auswertbarkeit zur Unterstützung des Teams.

4.2.2.2 Glossargestützte Analyse der User Stories

Ersteller von User Stories sollen mit einer Prüfung auf inkonsistentes Vokabular unterstützt werden. Dazu ermitteln wir die in der User Story genutzten NSEs auf dieselbe Weise wie zur Speicherung in der Ontologie. Für jedes der NSEs prüfen wir, ob es im Glossar vorhanden ist. Ist dies nicht der Fall, so kann der Ersteller entscheiden, ob dieses NSE aufzunehmen ist oder es sich um einen inkonsistenten Term handelt. Bei der Aufnahme eines neuen NSEs in das Glossar kann der Ersteller sogleich eine Beschreibung und semantische Verknüpfungen zu anderen NSEs angeben.

Weiterhin können wir für jedes im Glossar vorhandene NSE semantisch verwandte NSEs ermitteln, falls solche Verknüpfungen existieren. Speziellere Begriffe (Hyperonyme) und Synonyme können wir dann zu einem Begriff angeben. Vor allem die Existenz von Hyperonymen ist ein Anzeichen von Mehrdeutigkeit.

User Stories beschreiben – je später in der Entwicklungsphase, desto mehr – oft Funktionalität, die auf bestehender Funktionalität aufbaut. Daher ist anzunehmen, dass neue User Stories viele Begriffe verwenden, die bereits im Glossar enthalten sind. Das Verhältnis von unbekannten NSEs zu solchen, die im Glossar existieren, können wir daher als Indikator für fehlenden Kontext heranziehen.

Wie in Abschnitt 2.1 erläutert, ist ein Nachteil der elektronischen Speicherung die fehlende Platzbeschränkung. Dadurch schreiben Autoren schnell zu viele Informationen in eine User Story, was sowohl die Verständlichkeit als auch die maschinelle Auswertbarkeit erschwert. Über auf Erfahrungswerten basierende Schwellwerte für die Länge einer User Story können weitere Indikatoren gewonnen werden. Dabei können wir entweder die Gesamtanzahl von Wörtern bzw. NSEs oder die Anzahl unterschiedlicher NSEs betrachten.

4.2.2.3 Richtlinien zur Erstellung von User Stories

Ausgehend von der Analyse in Abschnitt 4.1.1 ist ein Leitfaden entstanden. Dieser befindet sich in Anhang B dieser Arbeit.

Dieser berücksichtigt einerseits, dass die Beteiligten in dem betrachteten Projekt vorher mit traditionellen Prozessmodellen arbeiteten. Dementsprechend ist das Verständnis einer User Story gemäß Abschnitt 2.1 oft nicht gegeben. Daher werden Grundsätze im Leitfaden angegeben.

Ein weiterer Teil des Leitfadens erläutert, wie die genannten Probleme bei der maschinellen Auswertung einer User Story umgangen werden können.

Zuletzt haben wir eine kurze Liste klarer Fehlerkriterien entworfen, auf die Autoren User Stories prüfen können.

4.2.3 Gewinnung relevanter Testschritte

Ist eine User Story geschrieben worden, so sollen relevante existierende Testschritte für die Erstellung eines Testskriptes gefunden werden.

Eine Annahme des hier beschrieben Ansatzes ist, das die Verwendung von NSEs in der User Story eine Eingrenzung existierender Testschritte auf potentiell relevante Testschritte erlaubt. Dazu können wir solche Testschritte abfragen, die mit den verwendeten NSEs verknüpft sind.

Die Ergebnismenge, die man durch Abfrage solcher Schritte erhält, ist im Allgemeinen dennoch für die effektive Nutzung zu umfangreich. Ein Großteil der Schritte ist dann für die betrachtete User Story irrelevant. Daher kommen Sortiermechanismen zum Einsatz, durch die eine Einschränkung der Ergebnisse ermöglicht wird. In dieser Arbeit verwendete Sortiermethoden beschreiben wir in Abschnitt 4.2.3.1.

Desweiteren ist auch die Betrachtung von Schritten notwendig, die in sehr vielen oder ähnlichen User Stories benötigten Kontext herstellen. Ein Beispiel für einen solchen Basisschritt ist Schritt 1 in Listing 1, der in vielen Testskripten zum Testen der Addressbuchapplikation verwendet werden könnte. Diese Vorgehensweise ist in Abschnitt 4.2.3.2 erläutert.

4.2.3.1 Begrifflich verwandte Testschritte

Da wir verwendete Konzepte als Grundlage der Verwandtschaft einer User Story zu Testschritten betrachten, extrahieren wir zunächst die NSEs einer User Story. Diese Extraktion wird auf dem gleichen Wege durchgeführt wie während der Befüllung der Ontologie mit bereits erfüllten User Stories. Die resultierenden NSEs bezeichnen

```
1  PREFIX o: <http://ontoserv/project1#>
2  SELECT ?step ?steptype ?steptext WHERE {
3     Type(?nle, o:StoryConcept),
4     PropertyValue(?nle, o:hasName, "button"),
5     PropertyValue(?nle, o:hasStepReference, ?step),
6     PropertyValue(?step, o:hasTestImplementation, ?impl),
7     PropertyValue(?impl, o:hasAttribute, ?attr),
8     PropertyValue(?attr, o:hasAttributeValue, ?steptext),
9     PropertyValue(?attr, o:hasName, ?steptype),
10 }
```

Listing 2: Eine SPARQL-DL-Abfrage nach Testschritten, die mit dem natürlichsprachlichen Element *button* verknüpft sind.

wir im folgenden als Abfrage-NSEs. Dann fragen wir die Ontologie nach allen Testschritten ab, die mit den Abfrage-NSEs verknüpft sind. Listing 2 zeigt eine Abfrage nach Schritten, die mit dem Abfrage-NSE *button* verbunden sind.

Die folgenden Ansätze sollen die gefundenen Testschritte sortieren, um eine sinnvolle Eingrenzung der gefundenen Testschritte zu ermöglichen.

Termfrequenz und inverse Dokumentfrequenz

Ein Ansatz zur Sortierung der Schritte liegt in der Annahme begründet, dass die Relevanz eines Testschrittes mit der Anzahl an Verbindungen mit Abfrage-NSEs zusammenhängt. Somit ist ein Testschritt mit mehreren Verbindungen zu Abfrage-NSEs relevanter als ein mit nur einem Abfrage-NSE verknüpfter. Allerdings haben die NSEs einen unterschiedlichen Informationsgehalt, da ihre Nutzungshäufigkeit und -verteilung über die existierenden Testschritte nicht gleich ist.

Dies führt zu der folgenden Gewichtungsfunktion, die die Anzahl der Testschritte berücksichtigt, die zu den Abfrage-NSEs gefundenen werden. Dazu verwenden wir die Termfrequenz $tf_{i,j}$ eines NSE i und eines Testschrittes j gemeinsam mit der inversen Dokumentfrequenz idf_i zur Gewichtung der NSEs: Die idf_i wird über $idf_i = log\frac{N}{n_i}$ berechnet, wobei N die Anzahl aller im Abfrageergebnis befindlichen Testschritte ist und n_i die Anzahl der Testschritte, die NSE i enthalten. Zusammen mit der Termfrequenz wird das Gewicht für ein NSE und einen Testschritt über $w_{i,j} = tf_{i,j} * idf_i$ berechnet (vgl. [46]).

Ein Testschritt, der aus einem zumeist kurzen natürlichsprachlichen Satz besteht, enthält ein NSE selten mehrfach. Außerdem kann die Häufigkeit eines NSEs in der betrachteten User Story ein Indikator für die Relevanz des NSEs sein. Daher multiplizieren wir das entstehende Gewicht mit einem Frequenzrang des NSE bezüglich der User Story. Die Bildung des Frequenzranges ist in Abschnitt 5.5.2 genauer beschrieben. Wir benutzen den Frequenzrang statt der tatsächlichen Frequenz, um das Gewicht häufig verwendeter NSEs zu reduzieren.

Anschließend bilden wir einen Relevanzwert für jeden Testschritt, indem wir die Gewichte aller NSEs, die mit dem Testschritt verbunden sind, aufsummieren. Die Testschritte werden dann nach dem Relevanzwert sortiert.

Verhältnis von Abfrage-NSEs und nicht abgefragten NSEs im Testschritt

Ein weiterer Ansatzpunkt ist das Verhältnis von Abfrage-NSEs und NSEs, die nicht in der Abfrage enthalten sind. Wir nehmen an, dass Testschritte, die viele nicht

in der Abfrage enthaltene NSEs referenzieren, geringere Relevanz haben als solche, die nur Abfrage-NSEs referenzieren. Dazu werden zu jedem Testschritt alle mit ihm verknüpften NSEs von der Ontologie abgefragt. Die Anzahl referenzierter Abfrage-NSEs nennen wir QN, die Anzahl der nicht in der Abfrage enthaltenen NSEs wird mit FN bezeichnet. Der bezüglich des Verhältnisses gebildete Relevanzwert W ist dann durch $W = QN - FN$ gegeben.

Zum isolierten Einsatz dieses Verfahrens sollten statt der Anzahlen der NSE-Referenzen der Informationsgehalt jedes NSEs über alle abgefragten Dokumente herangezogen werden. Wir verwenden dieses Verfahren jedoch zur Erweiterung des IDF-basierten Verfahrens und addieren den Verhältniswert W zum Relevanzwert.

Minimaler Abstand referenzierter NSEs in der User Story

Auch der minimale Abstand von NSEs in der User Story kann herangezogen werden. Testschritte, die in der User Story direkt benachbarte NSEs (zum Beispiel NSEs in einem Satz) enthalten, beziehen sich eher auf diesen Satz als solche, die mit großem Abstand erwähnte NSEs enthalten. Dazu bestimmen wir eine Liste der aus der User Story extrahierten NSEs, in der die Reihenfolge des Auftretens beibehalten wird. Allerdings eignet sich dieses Verfahren nicht zur isolierten Bewertung von Testschritten. Ein Testschritt, der nur ein NSE enthält, erhält einen Abstandswert von 0, während einer, der viele Abfrage-NSEs referenziert, notwendigerweise einen hohen Abstandswert aufweist. Eine Normalisierung dieses Wertes ist somit unerlässlich. Dazu dividieren wir das erhaltene Abstandsmaß durch die Anzahl der NSE-Referenzen.

In Kombination mit einem anderen Verfahren sollen so Testschritte mit NLE-Kombinationen, die in der User Story keinen Zusammenhang haben, abgewertet werden. Der normalisierte Abstandswert wird hier von dem Relevanzwert des IDF-Ansatzes subtrahiert.

4.2.3.2 Häufig referenzierte Schritte

Gerade Testschritte, die in vielen Testskripten verwendet werden, sind potentiell relevant zur Entwicklung eines Testskriptes für eine neue User Story. Ein häufig wiederkehrender Schritt könnte beispielsweise das Laden von Daten sein, der als Ausgangsbasis für viele Tests benötigt wird. Andere Testschritte werden für alle Testschritte einer Funktionalitätsgruppe verwendet, wie beispielsweise die Aktivierung einer Ansicht.

Einerseits haben solche Testschritte oft keinen oder nur geringen sprachlichen Zusammenhang zu der in der User Story beschriebenen Funktionalität. Andererseits treten darin NSEs auf, die insgesamt häufig verwendet werden, und somit aufgrund geringer inverser Dokumentfrequenz gering gewichtet werden.

Anders als in Abschnitt 4.2.3.1 muss eine Bewertung daher vorwiegend auf der Verwendungshäufigkeit der Testschritte beruhen. Da Testschritte einer Funktionalitätsgruppe jedoch gegenüber global verwendeten Testschritten eine geringere Verwendungshäufigkeit aufweisen, müssen auch hier sprachliche Verwandtschaftsbeziehungen herangezogen werden.

Ähnlich der Suche nach thematisch ähnlichen Testschritten können wir auch Testskripte suchen und diese nach der Anzahl referenzierter Anfrage-NSEs gewichten.

Des Weiteren berechnen wir zu jeder Testschrittimplementierung die Anzahl von Testskripten, die diese Implementierung verwenden. Wir fragen zu den 5 am höchsten bewerteten Testskripten alle referenzierten Testschrittimplementierungen ab und sortieren sie über die Verwendungshäufigkeit.

4.2.3.3 Kombination und Auswahl für Vorschläge

Die Verfahren aus den Abschnitten 4.2.3.1 und 4.2.3.2 finden unterschiedliche Testschritte und gewichten nach verschiedenen Kriterien. Die Kombination orthogonaler Verfahren kann in der Informationsrückgewinnung eingesetzt werden, um die Ausbeute und Präzision (Recall und Precision) zu erhöhen. Das bedeutet für einen Benutzer des Systemes, dass er weniger und relevantere Ergebnisse erhält. Verschiedene Verfahren können angewendet werden, um gute Gewichtungen von Kombinationen zu erhalten. Gethers et al. verwenden dazu die Principal Component Analysis, mit der orthogonale Vektoren der Verfahren bestimmt werden können [39]. Eine Arbeit von Scheuer beschreibt die Anwendung von neuronalen Netzen zu diesem Zweck [66]. Die Bestimmung einer optimalen Gewichtung der Verfahren ist jedoch nicht Teil dieser Arbeit.

Experimente zeigten, dass die einfache Kombination der Ergebnisse bereits eine deutliche Verbesserung im Vergleich zu den isoliert verwendeten Verfahren bringt. Mittels der Verfahren werden dazu die Testschritte nach ihrem Relevanzwert absteigend sortiert: $R = \{r_1, \dots, r_n\}$. Die kombinierte Ergebnisliste ist dann durch $K = \{r_1, s_1, r_2, s_2, \dots\}$ gegeben, wobei r_i mit dem Verfahren aus Abschnitt 4.2.3.1 und s_i mit dem Verfahren aus Abschnitt 4.2.3.2 gewonnen wurde. Wird ein Testschritt von beiden Verfahren gefunden, wird dieser nur mit dem geringsten Index in die kombinierte Ergebnisliste übernommen.

In der Evaluation hat sich gezeigt, dass die Auswahl von zwischen 20 und 30 der höchstbewerteten Ergebnisse einen guten Kompromiss zwischen Ausbeute und Präzision darstellen.

4.2.4 API-Komponenten zu neuen Testschritten

Zum testen neuer Funktionalität müssen neue Testschritte entstehen, die Funktionalität aus der Software verwenden.

Das Softwaremodell enthält hierzu Aufruf- und Objekterstellungsbeziehungen. Neue Funktionalität, auch wenn sie bereits implementiert wurde, wird in existierenden Test nicht referenziert. Doch durch die iterative Vorgehensweise in einem agilen Prozess können wir davon ausgehen, dass neue Funktionalität selten isolierte Funktionalität darstellt. So wird sie in den meisten Fällen eine Erweiterung der bestehenden Funktionalität darstellen.

Thematisch verwandte Testschritte, deren Gewinnung in Abschnitt 4.2.3.1 beschrieben ist, können daher als Grundlage zur Gewinnung von API-Komponenten zur Implementierung des Testschrittes dienen.

Statt jedoch eine User Story zur Gewinnung der Suchparameter zu nutzen, verwenden wir dazu die im Testskript angegebenen, nicht implementierten Testschritte. Mit dem gleichen Verfahren, mit dem wir NSEs aus User Stories extrahieren, ziehen wir die Abfrage-NSEs aus dem Testschritt. Die Abfrage mit einem NSE, die

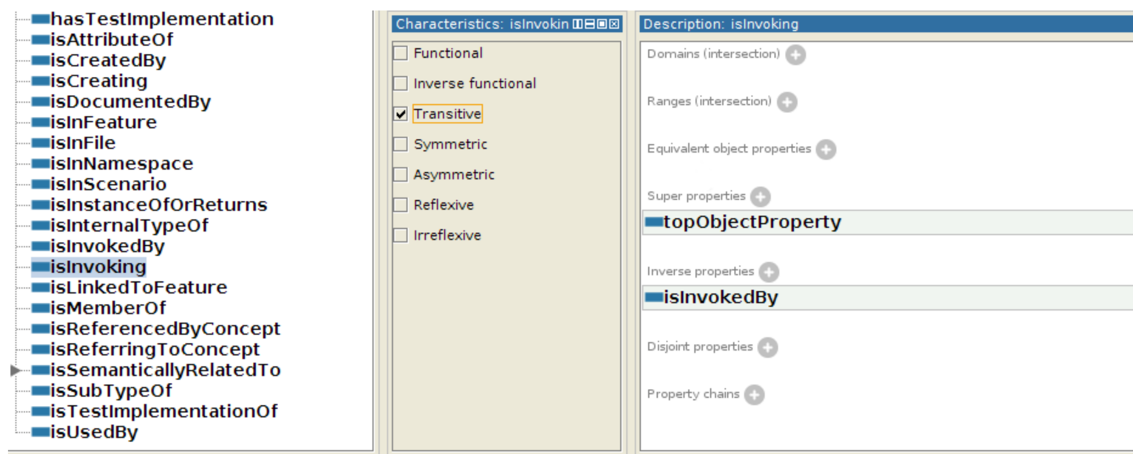


Abbildung 4.5: Deklaration einer Relation als transitiv im Ontologieeditor *protégé*

in 4.2.3.1 beschrieben wird, erweitern wir so, dass sie statt Testschritten die in der Testschritt-Implementierung verwendeten Komponenten zurückgibt.

Wir bestimmen die Häufigkeit der Nutzung einer Komponente unter den gefundenen Testschritten zur Gewichtung ihrer Relevanz. Wenn man die Aufrufrelation transitiv auffasst, also auch die Aufrufpfade mitberücksichtigt, hat dies Konsequenzen für die Ergebnismenge. Dadurch werden einerseits Komponenten gefunden, die sich direkter auf getestete Funktionalität beziehen. Andererseits kann sich die Anzahl gefundener Komponenten enorm erhöhen. Abbildung 4.5 zeigt die Einstellung der Transitivität auf einer Relation. In der Evaluation in Kapitel 6 zeigen wir, inwieweit die Nutzung einer transitiven Relation mit dem vorgeschlagenen Verfahren sinnvoll ist.

4.3 Zusammenfassung

Zunächst analysierten wir die Ausgangssituation eines betrachteten Software-Entwicklungsprojektes. Dabei haben wir dessen agilen Prozess nach Scrum, die Verwendung von User Stories und den Einsatz Verhaltensgetriebener Entwicklung betrachtet. Wir identifizierten Probleme in folgenden Bereichen:

1. Erstellung der User Stories
2. Erstellung von natürlichsprachlichen Testskripten
3. Implementierung der Testfunktionalität

Danach erläuterten wir den Entwurf eines Ontologie-basierten Systemes zur Lösung der genannten Probleme. Die Ontologie wird dazu aus verschiedenen Artefakten des Entwicklungsprozesses aufgebaut:

- Quelltext der Software (sowohl der Test- als auch der Produktivquelltext)
- Natürlichsprachliche Testskripte
- Begriffe (NSEs) aus User Stories und Testskripten

Über die NSEs wird ein Glossar erstellt, das auf einem Eintrag die Angabe von Beschreibungen und semantischen Beziehungen zu anderen NSEs erlaubt. Durch Nutzung des Glossars können wir inkonsistente Begrifflichkeit einer User Story transparent machen.

Zur Assistenz von Testern schlagen wir auf Basis aus der User Story extrahierter Begriffe existierende Testschritte vor. Mithilfe dieser Testschritte kann dem Tester die Erstellung eines Testskriptes erleichtert werden.

Entwickler unterstützen wir durch die Suche nach Quelltextkomponenten, die mit einem neuen Testschritt in Verbindung stehen.

5. Implementierung

Die entworfene Architektur besteht einerseits aus Ontologie-Schnittstellen, die sowohl die Befüllung und Manipulation als auch die Abfrage ermöglichen. Weitere Komponenten stellen Dienste zur Extraktion und Verknüpfung von Elementen aus Quelltext, User Stories und Testskripten bereit.

Über die Ontologie-Schnittstellen realisieren wir die Anwendungen, die in den Abschnitten 4.2.2, 4.2.3 und 4.2.4 beschrieben sind.

Aufgrund unterschiedlicher Plattformabhängigkeiten der Komponenten dient ein plattformunabhängiges Protokoll zur Kommunikation. Die hier beschriebene Implementierung wurde vorwiegend in der Programmiersprache Python entwickelt [9].

Abbildung 5.1 skizziert das hier beschriebene System, das wir weiterhin als F-TRec, Functional Test Recommender, bezeichnen. Dabei nutzen sowohl die Extraktionskomponenten als auch die Anwendungen die Schnittstellen zur Verarbeitung natürlichsprachlicher Texte sowie für den Zugriff auf die Ontologie.

Abschnitt 5.1 motiviert und beschreibt den Einsatz eines netzwerkbasierten Vermittlungsprotokolles als Schnittstelle zwischen den plattformabhängigen Komponenten. Die Ontologie-Schnittstellen und den Aufbau der Basis-Ontologie erläutern wir in Abschnitt 5.2. Die Extraktion natürlichsprachlicher Elemente zeigt Abschnitt 5.3. Abschnitt 5.4 beschreibt die Befüllung der Ontologie aus Artefakten des Softwareprojektes. In Abschnitt 5.5 beschreiben wir die Nutzung Ontologie zur Lösung der in dieser Arbeit gestellten Aufgaben.

5.1 Vermittlungsschicht

Die Bibliotheken, die wir in F-TRec verwenden, sind für unterschiedliche Plattformen entworfen. Es werden .NET, die Java Virtual Machine (JVM) und CPython als Plattformen genutzt. Auch für .NET und die JVM existieren Python-Implementierungen, wodurch für die Implementierung weitgehend nur eine Sprache verwendet wurde.

Es gibt mehrere Bibliotheken für Python, die die Nutzung von Objekten in verschiedenen Pythonprozessen, auch über ein Netzwerk verteilt, ermöglichen sollen. Zu

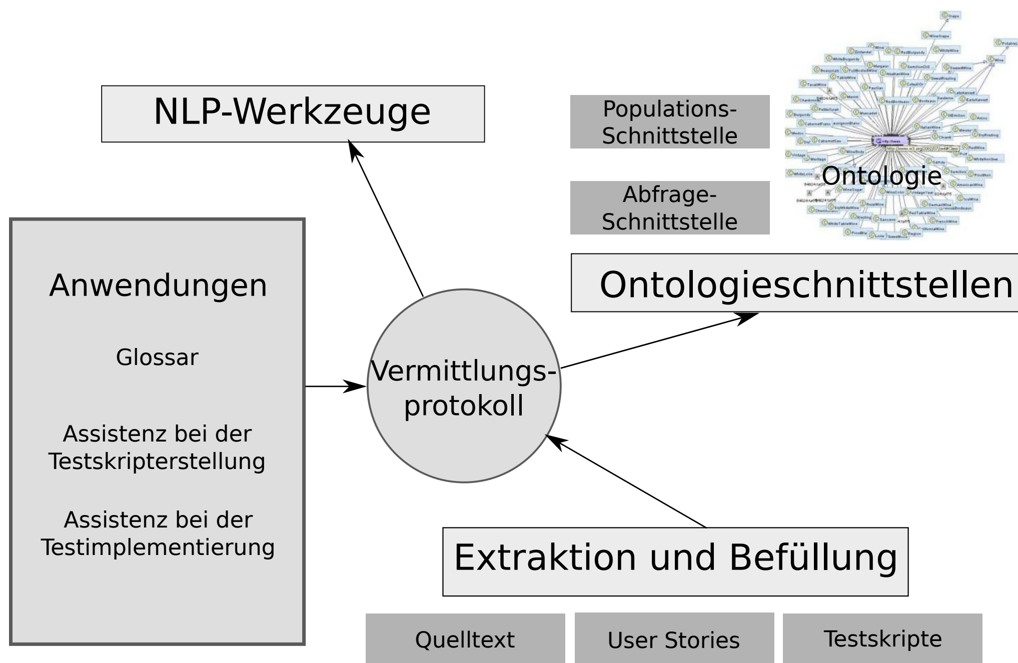


Abbildung 5.1: Skizze zur Architektur von F-TRec

nennen sind hier Pyro (Python Remote Objects [30]) und RPyC (Remote Python Call [34]).

Allerdings können diese Bibliotheken aufgrund eines Implementierungsfehlers nicht mit dem verwendeten IronPython in Version 2.7 auf der .NET-Plattform Mono eingesetzt werden. Für Pyro gibt es jedoch unter dem Namen PyroLite Client-Bibliotheken für .NET und die JVM, wodurch sich dieser Fehler umgehen lässt. Aus diesem Grund wurde Pyro als Grundlage für die Vermittlung zwischen CPython, IronPython auf Mono und Jython auf der JVM gewählt.

Mit Pyro werden beliebige Objekte für die entfernte Nutzung freigegeben, die über eine URL für Dienstnehmer erreichbar sind. Hierzu werden Stellvertreter-Objekte im benutzenden Prozess angelegt, die wie lokale Objekte verwendet werden können. Ein Nameserver ermöglicht außerdem die Suche nach freigegebenen Objekten, sodass die Objekt-URL nutzenden Prozessen nicht bekannt sein muss. Abbildung 5.2 skizziert, wie wir mittels Pyro von einem Dienstnehmer (Client) aus ein Objekt eines Dienstgebers verwenden. Dienstnehmer und Dienstgeber sind unterschiedliche Python-Prozesse, die auf unterschiedlichen Rechnern laufen können.

Der Einsatz von Pyro hat – außer der Lösung von Plattformabhängigkeiten – den Vorteil, dass wir freigegebene Objekte auf mehrere Knoten in einem Rechnernetzwerk verteilen können. So kann ein Rechner die Ontologiekomponenten bereitstellen, während die Quelltextextraktion von einem anderen Rechner aus erfolgt und wieder andere Rechner die Anwendungen zur Assistenz bei der Erstellung von Testskripten und Testimplementierungen anbieten.

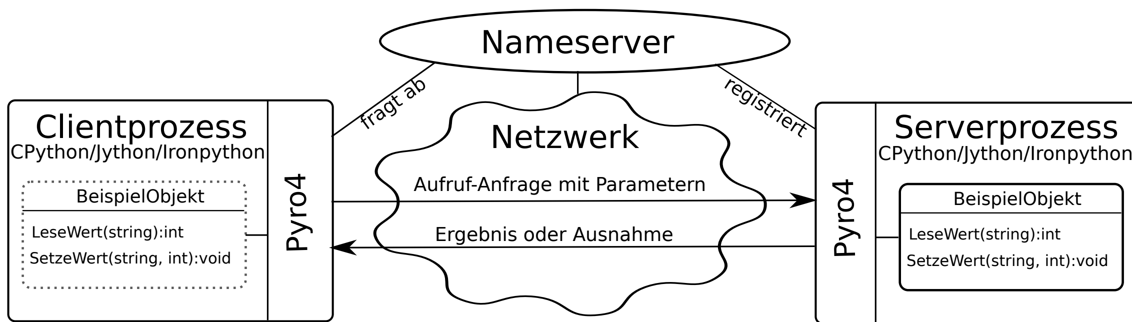


Abbildung 5.2: Bereitstellung und Nutzung entfernter Objekte über Pyro4

5.2 Ontologie-Schnittstellen

Die Datenbasis für alle Aufgaben, die in dieser Arbeit behandelt werden, ist eine Ontologie. In dieser speichern und verknüpfen wir Daten aus verschiedenen Artefakten, die im Software-Entwicklungsprozess entstehen.

Es existieren verschiedene Sprachen zur Ontologiebeschreibung. Eine Entscheidung für eine bestimmte Ontologiesprache sollte man, soweit die Mächtigkeit der Sprache für die Aufgabe genügt, von der Infrastruktur wie Editoren und Inferenzwerkzeugen abhängig machen (siehe Goméz-Pérez und Corcho, [42]). OWL, die Web Ontology Language, die vom W3C (World Wide Web Consortium) spezifiziert wird, ist eine der am weitesten verbreiteten Ontologiesprachen. Es baut auf dem von Goméz-Pérez und Corcho erläuterten DAML+OIL auf (siehe [15]). Ein Vorteil von OWL gegenüber den anderen in der Arbeit von Goméz-Pérez und Corcho betrachteten Sprachen ist, außer einer starken Infrastruktur, die Möglichkeit, Inkonsistenzen in der Ontologie festzustellen.

Wir verwenden Version 2 von OWL [72]. Grundlegende Konzepte der Sprache haben wir in Abschnitt 2.4.1 zusammengefasst.

Wir erläutern in Abschnitt 5.2.1 die Schnittstelle zum anlegen und verknüpfen von Konzept-Instanzen. Abschnitt 5.2.2 beschreibt die Schnittstelle zum Zugriff auf eine erstellte Ontologie.

5.2.1 Populationsschnittstelle

Sowohl zum schreibenden als auch zum lesenden Zugriff auf eine Ontologie verwenden wir die OWL-API, eine Bibliothek, die Elementarfunktionalität zur Arbeit mit OWL-Ontologien anbietet [44]. Auf dieser aufbauend haben wir drei Grundoperationen in der abstrahierten Schnittstelle realisiert. Diese Operationen bestehen im Anlegen eines Individuums und dessen Verknüpfung mit anderen Individuen und mit zusätzlichen Dateneigenschaften wie Zeichenketten oder Ganzzahlen.

Anlegen von Klassen und Relationen wird für die Implementierung nicht benötigt, da wir ein festes Klassenschema aus entsprechenden Elementen befüllen. Abschnitt 4.2.1.1 beschreibt dieses Schema; In Anhang A sind die Klassenhierarchie und die Relationen aufgeführt.

Listing 3 zeigt die Implementierung der API-Funktion zum Anlegen eines Individuums und dessen Verknüpfung mit einer Klasse über ein `ClassAssertionAxiom`,

```
def add_individual(manager, ontology, class_name, individual_name):
    pm = DefaultPrefixManager(str(
        ontology.getOntologyID().getOntologyIRI()) + '#')
    datafactory = ontology.getOWLDataFactory()

    owlclass = datafactory.getOWLClass(class_name, pm)
    individual = datafactory.getOWLNamedIndividual(individual_name, pm)
    classAssertion = datafactory.getOWLClassAssertionAxiom(owlclass, individual)
    manager.addAxiom(ontology, classAssertion)
```

Listing 3: Funktion zum Hinzufügen einer Klassenausprägung (Individuum).

wie in Abschnitt 2.4.1.3 beschrieben. Die Referenzen `manager` und `ontology` gewinnt man durch das Laden der Ontologie über die OWL-API. Durch die Nutzung des `DefaultPrefixManager` muss der Benutzer nicht die volle IRI des anzulegenden Elementes und seiner Klasse angeben.

Die Implementierungen für die Verknüpfung von Individuen ist in gekürzter Form in Listing 4 zu sehen. Analog dazu ist die Verknüpfung eines Individuums mit Dateneigenschaften implementiert; hier wird statt eines `ObjectPropertyAssertionAxioms` das `DataPropertyAssertionAxiom` verwendet.

Die OWL-API transformiert bei der Speicherung in einer Datei auf diese Weise angelegte Individuen in das OWL/XML-Format. Ein Beispiel für die XML-Repräsentation eines Individuums zeigt Listing 5 für eine Klasse.

Die beschriebenen Funktionen werden, gekapselt als Methoden eines Objektes, über die Vermittlungsschicht bereitgestellt, wie in Abschnitt 5.1 gezeigt. Jede Komponente, die Elemente zur Ontologie hinzufügt, nutzt diese Populationsschnittstelle.

5.2.2 Abfrage der Wissensbasis

Wir benötigen sowohl für die Anwendungen als auch zur Verknüpfung der Information während der Befüllung der Ontologie eine Möglichkeit, die in der Ontologie enthaltenen Informationen abzurufen. Von relationalen Datenbanksystemen sind verschiedene SQL-Varianten bekannt, die einem solchen Zweck dienen. Auch für Ontologien sind entsprechende Sprachen entstanden, die sich teilweise an SQL anlehnen. Für den Zugriff benötigen wir zunächst ein Inferenzwerkzeug, das in der Lage ist, die logischen Verknüpfungen der Ontologie auszuwerten. Auf diesem basierend können wir Abfragebibliotheken verwenden, die zu einer in bestimmter Sprache gegebenen Anfrage alle möglichen Antworten zurückliefert.

5.2.2.1 Inferenzwerkzeuge (Reasoner)

Eine Besonderheit von Ontologien im Vergleich zu anderen Datenspeichern wie relationalen Datenbanken ist, dass man basierend auf explizit gegebenen Informationen

```
def add_objectproperty(manager, ontology, individualname, propname, relatedname):
    [...]
    obj_prop = datafactory.getOWLObjectProperty(propname, pm)
    indiv = datafactory.getOWLNamedIndividual(individualname, pm)
    related = datafactory.getOWLNamedIndividual(relatedname, pm)

    opa = datafactory.getOWLObjectPropertyAssertionAxiom(obj_prop, indiv, related)
    manager.addAxiom(ontology, opa)
```

Listing 4: Verknüpfung der durch `individual` und `related` gegebenen Individuen.

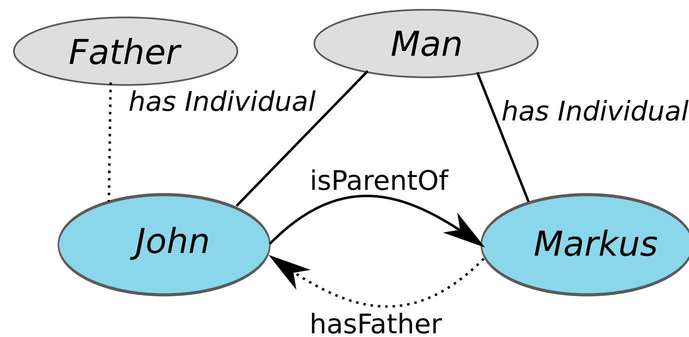


Abbildung 5.3: Beispiel für Inferenz in Ontologien

(den Axiomen) implizite Informationen ableiten kann. Zum Beispiel ist die implizite Verknüpfung „`Markus hasParent John`“ nicht ohne Berechnung bekannt, wenn die explizite Verknüpfung „`John isParentOf Markus`“ mit der zur Relation `hasParent` inversen Relation `isParentOf` gegeben ist. In dem Beispiel könnte außerdem die Instanz `John`, die explizit dem Konzept `Man` zugeordnet ist, implizit zu einer Instanz des Konzeptes `Father` werden. Diese Konstellation wird in Abbildung 5.3 dargestellt.

Für solche Inferenzaufgaben setzt man einen Reasoner ein, der logische Schlüsse aus den expliziten Information zieht. Vor der Nutzung bzw. Abfrage expliziter und impliziter Informationen muss der Inferenzschritt durchgeführt werden. Für OWL2 sind einige Reasoner implementiert worden; [73] listet diese auf. Wir verwenden den Reasoner Pellet, da dieser sich für unsere Ontologie als zuverlässig und hinreichend schnell erwies [69].

Alle Reasoner, die mit der OWL-API nutzbar sind, bieten eine Schnittstelle zur Abfrage von Zusammenhängen, die von verschiedenen Abfragesystemen genutzt werden kann.

5.2.2.2 Abfragesprachen

Auch zur Abfrage der Daten über einen Reasoner gibt es einige Sprachen und Implementierungen. Allerdings ist in der Version 2 von OWL die Verfügbarkeit und Stabilität von Lösungen noch gering. So war zum Zeitpunkt dieser Arbeit für die

```
1  <ClassAssertion>
2      <Class IRI="#Class"/>
3      <NamedIndividual IRI="#Application.AddressBook"/>
4  </ClassAssertion>
5  <ObjectPropertyAssertion>
6      <ObjectProperty IRI="#isInNamespace"/>
7      <NamedIndividual IRI="#Application.AddressBook"/>
8      <NamedIndividual IRI="#Application"/>
9  </ObjectPropertyAssertion>
10 <DataPropertyAssertion>
11     <DataProperty IRI="#hasName"/>
12     <NamedIndividual IRI="#Application.AddressBook"/>
13     <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#string">
14         AddressBook
15     </Literal>
16 </DataPropertyAssertion>
```

Listing 5: OWL/XML-Repräsentation einer Klasse mit Daten- und Objekteigenschaften.

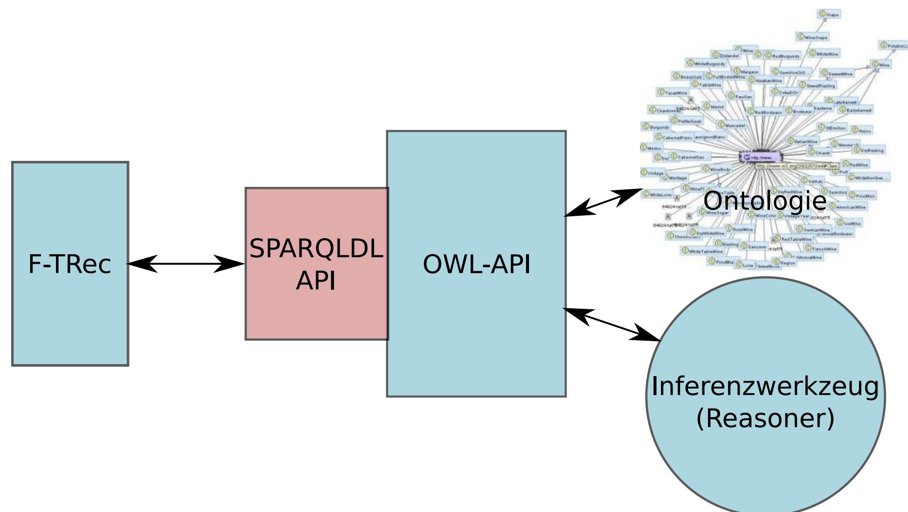


Abbildung 5.4: Einbettung der SPARQL-DL-API in die Ontologieinfrastruktur nach [11]

Sprache SPARQL, die vom W3C standardisiert wird, keine zufriedenstellende Implementierung verfügbar. Diese Sprache, vor allem in der im Standardisierungsprozess befindlichen Version 1.1, wäre besser geeignet als die in dieser Arbeit verwendete Implementierung von SPARQL-DL, die SPARQL-DL-API [68],[11]. So sollen mit SPARQL in Version 1.1 beispielsweise verschachtelte Abfragen gestellt werden können, was eine leichtere und effizientere Implementierung der Abfragen erlauben würde. Die Bibliothek SPARQL-DL-API stellte sich dennoch als geeignet heraus.

Die Abfragekomponente lädt zunächst eine Ontologie über die OWL-API. Der Inferenzschritt wird daraufhin von dem Reasoner Pellet durchgeführt. Zuletzt wird eine Abfrageinstanz über die SPARQL-DL-API initialisiert. Diese wird über die Vermittlungsschicht anderen Komponenten bereitgestellt. Abbildung 5.4 skizziert diese Konstellation.

Ein Beispiel für die Abfrage der Basisklasse/Unterklasse-Tupel aller Klassen einer Ontologie über SPARQL-DL ist in Listing 6 zu sehen.

```
1 SELECT ?parentclass ?subclass WHERE {
2     DirectSubclassOf(?subclass, ?parentclass)
3 }
```

Listing 6: Die vollständige Klassenhierarchie einer Ontologie in Form von direkten Unterklassen-Paaren (vgl. [11])

5.3 Verarbeitung natürlichsprachlicher Texte

Im Gegensatz zu maschinenlesbaren Sprachen ist die Verarbeitung natürlicher Sprachen immer mit unsicherem Schließen verbunden. In vielen Fällen basieren Einzelkomponenten, die in der Verarbeitung natürlicher Sprache zur Anwendung kommen, auf statistischen Verfahren. Hierzu werden Klassifikatoren auf umfangreichen Korpora trainiert.

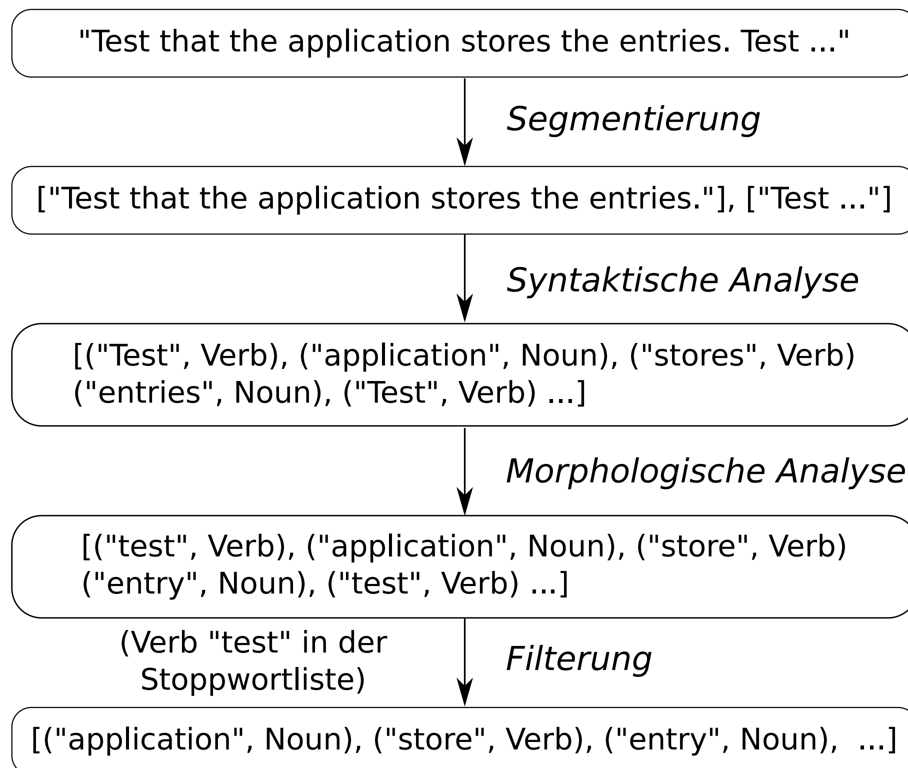


Abbildung 5.5: Verarbeitung eines Beispielsatzes

Wir verwenden NLTK, ein Python-Rahmenwerk zur Verarbeitung natürlicher Sprache, das weiterhin den Zugriff auf WordNet bietet [19]. Des Weiteren kommen mehrere Bibliotheken der Stanford NLP Gruppe zum Einsatz:

- *Stanford factored Parser*, ein Zerteiler für natürliche Sprache (siehe [50])
- *Stanford Named Entity Recognizer*, ein Erkenner von Namensreferenzen wie Orten, Personen und Organisationen (siehe [36])

In der vorliegenden Arbeit verwenden wir die NLP-Werkzeuge zur Extraktion von Begriffen. Dies wird in den folgenden Schritten erläutert. Abbildung 5.5 zeigt die Verarbeitung eines Textteiles für jeden Teilschritt.

Segmentierung

In diesem Schritt wird ein Text in Sätze zerlegt. Da oft Sätze in den betrachteten User Stories nicht durch einen Punkt abgeschlossen werden, nutzen wir aus, dass sich ein Satz nie über mehrere Zeilen erstreckt. Also wird der Text zunächst in einzelne Zeilen zerlegt. Darauf wenden wir den Punkt-Segmentierer von NLTK an, der bei Vorkommen mehrerer Sätze in einer Zeile diese in Einzelsätze auftrennt.

Eine weitere Aufspaltung der Sätze in Wörter führen wir hier nicht durch, da der im nächsten Schritt erläuterte Zerteiler dies übernimmt.

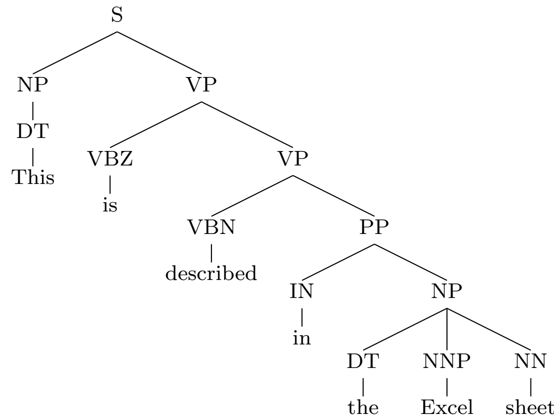


Abbildung 5.6: Ein Syntaxbaum, der ein zusammengesetztes Nomen enthält.

Syntaktische Analyse

Wir analysieren die aus der Segmentierung gewonnenen Sätze mit einem Zerteiler (Stanford factored Parser). Das ermöglicht die Erkennung der Satzstruktur und der Rollen der einzelnen Wörter für den Satz. Die Ausgabe des Zerteilers ist ein Syntaxbaum, der die Satzstruktur repräsentiert. Den Syntaxbaum ziehen wir zur Extraktion der Begriffe heran.

Der Syntaxbaum für die Phrase *„This is described in the Excel sheet"* ist in Abb. 5.6 gezeigt. Der Wurzelknoten `S` repräsentiert die gesamte Phrase (Satz), während die Blätter die Wörter repräsentieren. Die inneren Knoten beschreiben die Struktur der Phrase; ein innerer Knoten direkt vor einem Blatt kennzeichnet die Wortart des damit verbundenen Wort-Knotens. Verben sind durch einen Wortartbezeichner gekennzeichnet, der mit `VB` beginnt, während Nomen an einem Wortartbezeichner, der mit *NN* beginnt, erkennbar sind.

Zusammengesetzte Substantive werden im englischen meist nicht zusammengeschrieben; die Bestandteile stehen daher im Syntaxbaum innerhalb einer Nominalphrase als einzelne Blätter nebeneinander. Im Beispielbaum aus Abb. 5.6 ist dies der Fall für das Kompositum *Excel sheet*. Verben und isoliert stehende Nomen extrahieren wir ohne weitere Betrachtung des Kontextes.

Morphologische Analyse

In einem weiteren Schritt ermitteln wir, soweit möglich, die Grundform der extrahierten Verben und Nomen. Die Grundformermittlung nennt man Stemming oder – bei wörterbuchgestütztem Verfahren – Lemmatisierung. Beim Stemming werden übliche Flexionen von Wörtern entfernt oder ersetzt. Beispielsweise wird bei dem Wort „`cats`" die Endung „`-s`" zur Kennzeichnung des Plural für Nomen entfernt. Ein bekanntes Verfahren dafür ist der Porter-Algorithmus (siehe [64]). Das Problem solcher Verfahren ist jedoch einerseits, dass der zu einem Wort ermittelte Stamm nicht in jedem Fall ein gültiges Wort ergibt. Zum Beispiel wird „`university`" auf „`univers`" reduziert. Andererseits können Ausnahmen von Flexionsregeln nicht berücksichtigt werden. Ein Beispiel hierfür ist das Verb „`broke`", dass mit dem Porter-Algorithmus nicht auf „`break`" zurückgeführt werden kann. Beide Schwächen des rein regelbasierten Verfahrens machen es für den Aufbau eines Glossars ungeeignet.

Lemmatisierung hingegen ermöglicht die Behandlung dieser Ausnahmen, wenn die Wörter im Wörterbuch enthalten sind. Mit einem auf WordNet basierenden Lemmatisierer wird „`broke`“ auf „`break`“ zurückgeführt. Auch die Wortart wird bei der Lemmatisierung berücksichtigt. Falls ein Wort nicht auf sein Lemma zurückgeführt werden kann, wird es ohne Änderung zurückgegeben. Dadurch ist sichergestellt, dass wir entweder auf gültige Begriffe reduzieren, oder zumindest den Originalbegriff erhalten. Dieses Verfahren nutzen wir zur Normalisierung der extrahierten Begriffe.

Filterung

Manche der extrahierten NSEs enthalten keine nutzbaren Informationen, daher wenden wir mehrere Filter an. So verwenden wir den Stanford Named Entity Recognizer, um Personennamen in den User Stories zu identifizieren und herauszufiltern. Andere unerwünschte Wörter, beispielsweise Substantive wie *February*, Verben wie *to be* oder Pronomen, müssen wir manuell in einer Stopwortliste pflegen, da ihre Bedeutung projektspezifisch sein kann.

5.4 Population der Ontologie

Im folgenden beschreiben wir, wie die im Softwareentwicklungsprozess entstehenden Artefakte in der Ontologie repräsentiert werden. Abschnitt 5.4.1 führt die Befüllung der Ontologiestruktur aus dem Quelltext des Projektes aus. In Abschnitt 5.4.2 erläutern wir die Extraktion und Speicherung von NSEs aus User Stories und Testschritten. Außerdem erläutern wir die Verknüpfung der Komponenten.

5.4.1 Erstellung des Softwaremodelles

Grundlegend für die Abfrage von Testquelltext und API-Komponenten ist die Erstellung eines Softwaremodelles aus dem Quelltext dieser Software. Dazu müssen wir die für die Ontologie interessanten Elemente aus dem Quelltext ziehen. Ein Zerteiler erstellt zu diesem Zweck aus Quelltext einen Syntaxbaum (Abstract Syntax Tree). Beim Ablaufen des Syntaxbaumes wird das Ontologieschema, dass vorher keinerlei Individuen bzw. Instanzen besitzt, mit Elementen des Syntaxbaumes befüllt.

Da der Quelltext des im Rahmen dieser Arbeit betrachteten Softwareprojektes in der .NET-Sprache C# verfasst ist, benötigen wir einen Zerteiler für diese Sprache. Wir verwenden NRefactory5 für diese Aufgabe, da aktuelle Sprachelemente berücksichtigt werden und die Auflösung von Typen, Eigenschaften, Aufrufen etc. aus einer Kollektion von Quellcodedateien möglich ist [7]. Wir sprechen die Bibliothek mithilfe von IronPython an, einer Python-Implementierung für die .NET-Plattform.

Für jeden Knotentyp des Syntaxbaumes, der für das Softwaremodell relevant ist, existiert eine zur Informationsextraktion genutzte Klasse. Die betrachteten Knotentypen sind Deklarationen von Namensräumen, Typen (Klassen, Strukturen/Enumerationen und Schnittstellen) und deren Eigenschaften und Methoden. Zusätzlich werden Quelltextdateinamen, Kommentare und Attribute extrahiert. Beim Durchlaufen des Syntaxbaumes jeder analysierten Quelltextdatei werden die voll qualifizierten Namen der Komponenten in die Ontologie übernommen. Für Methoden hängen wir weiterhin die Namen der Parametertypen an den vollqualifizierten Namen an, um einen eindeutigen Namen für das Individuum zu erhalten.

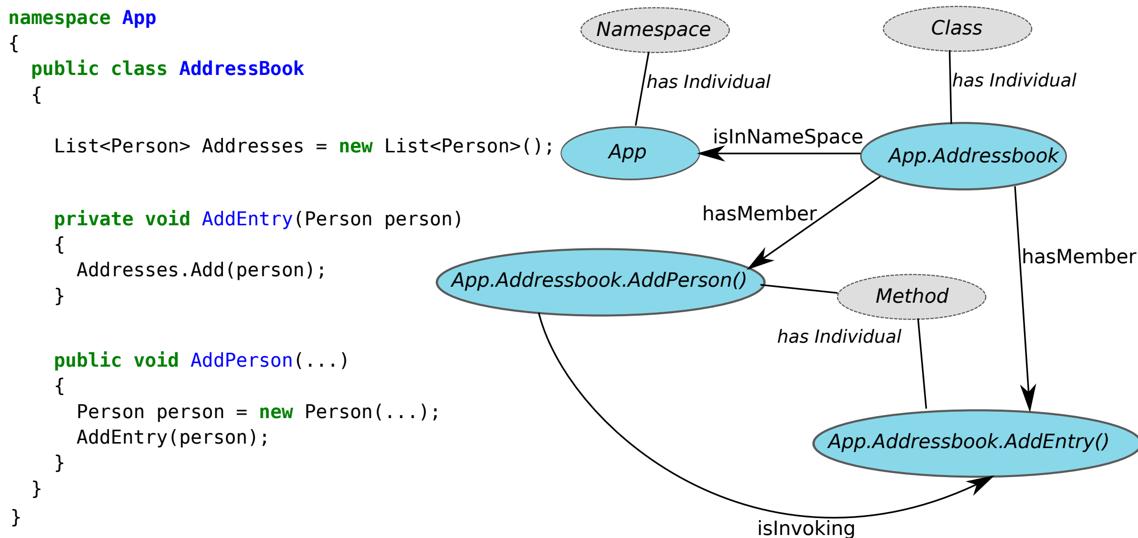


Abbildung 5.7: Ein Beispielquelltext und seine entsprechende Darstellung in der Ontologie

Wir extrahieren des Weiteren Beziehungen zwischen den Komponenten: eine Klasse ordnen wir ihrem Namensraum zu, eine Eigenschaft der dazugehörigen Klasse. NRefactory verknüpft Attribute direkt mit den betreffenden Elementen, sodass wir diese Relationen in der Ontologie direkt speichern können. Kommentare können jedoch nur durch ihre Position im Syntaxbaum mit den durch sie beschriebenen Elementen assoziiert werden. Kommentare in der Implementierung einer Methode unterscheiden wir in der Ontologie als Individuen der Klasse *InnerComment* von solchen, die außerhalb der Deklaration stehen (*OuterComment*). Die Zuordnung von Kommentaren zu Elementen über die Position im Syntaxbaum hängt von konventionsgemäßer Kommentierung durch Entwickler ab. Abbildung 5.7 zeigt Beispielquelltext und einen Auszug der aus der Extraktion resultierenden Elemente und Verknüpfungen.

Ein zweiter Schritt dient der Auflösung referenzierter Typen und Eigenschaften zur Extraktion von Basisklassen, Feld- und Rückgabetypen. Wir betrachten darüber hinaus die Implementierung von Methoden, Konstruktoren und Destruktoren zur Extraktion von Methodenaufrufen und Objekterstellungen. Die Referenzauflösung von NRefactory5 nutzend, können wir so über alle betrachteten Quelltextdateien Verknüpfungen zwischen referenzierenden und referenzierten Komponenten anlegen. Dadurch werden Vererbungshierarchien und Aufrufbäume in der Ontologie repräsentiert.

5.4.2 User Story und zugehöriges Testskript

Wir betrachten eine abgeschlossene User Story gemeinsam mit dem dazu entworfenen Testskript. Abbildung 5.8 zeigt die Extraktion von NSEs aus User Story und dem dazugehörigen Testskript. Weiterhin werden darin die Erstellung von Individuen für Teile des Testskriptes und deren Verknüpfung mit NSEs und Testmethoden dargestellt.

Ein einfacher Zerteiler, den wir für die Sprache Gherkin entworfen haben, ermöglicht die Aufspaltung von Testskripten in ihre Einzelkomponenten.

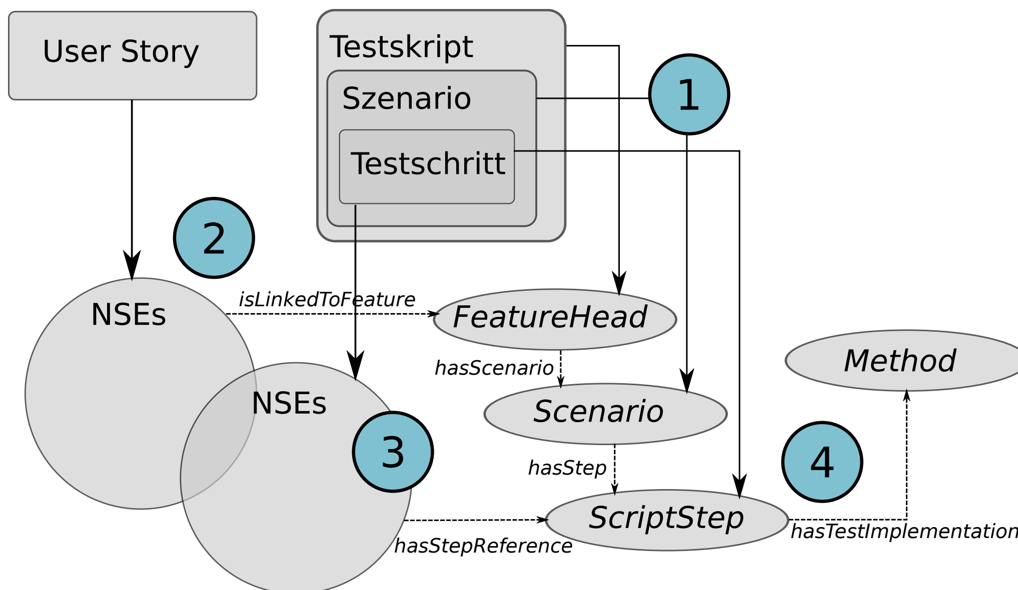


Abbildung 5.8: Analyse von User Story und zugehörigem Testskript zur Anlage der Individuen und ihrer Verknüpfungen

Gemäß der Strukturelemente der Testskripte enthält das Ontologieschema Konzepte für die einzelnen Bestandteile (siehe Abschnitte 2.2 und 4.2.1.1).

Wir erstellen für jedes extrahierte Element ein Individuum, das dem entsprechenden Konzept der Ontologie hinzugefügt wird. Für das Wurzelelement des Testskriptes erhält das zugehörige Individuum den Identifikator der User Story. Zusätzlich wird ihm der Beschreibungstext hinzugefügt, der identisch mit der Kurzbeschreibung der User Story ist. In Abbildung 5.8 ist dies unter Punkt 1 dargestellt.

Aus der User Story extrahieren wir NSEs gemäß Abschnitt 5.3. Jedes NSE fügen wir – der Wortart entsprechend – als Individuen einer Unterklasse von *StoryConcept* hinzu, also entweder als *VerbConcept* oder als *NounConcept*. Zusätzlich verknüpfen wir jedes dadurch erstellte Individuum mit dem obigen Individuum des Testskript-Wurzelelementes über die Relation *isLinkedToFeature*. Punkt 2 von Abbildung 5.8 zeigt die Extraktion von NSEs aus der User Story und ihre Verknüpfung in der Ontologie mit dem Testskript-Wurzelelement.

Für jedes Szenario-ähnliche Element legen wir ein Individuum des Konzeptes *Scenario* an, das über die Relation *hasScenario* als Teil des Wurzelelementes markiert wird.

Für jeden Testschritt, der in dem angelegten Szenario-Element enthalten ist, legen wir ein Individuum des Konzeptes *ScriptStep* an. Den Text des Testschrittes teilen wir auf, sodass der Typ des Testschrittes und der restliche Text getrennt vorliegen. Diese Teile fügen wir dem Individuum hinzu. Zusätzlich speichern wir die Position des Testschrittes innerhalb des Szenarios und verknüpfen die Individuen für Testschritt und Szenario über die Relation *hasStep*.

Den Text des Testschrittes unterziehen wir der gleichen Analyse wie den der User Story und extrahieren so die NSEs des Testschrittes. Auch für diese erstellen wir Individuen, verknüpfen sie jedoch nicht mit dem Individuum des Testskript-Wurzel-

```
PREFIX ontl: <http://ontoserv/project1#>
SELECT ?testimpl ?steptype ?steptext WHERE {
    DirectType(?testattr, ontl:Attribute),
    PropertyValue(?testattr, ontl:hasName, "Given"),
    PropertyValue(?testattr, ontl:isAttributeOf, ?testimpl),
    PropertyValue(?testattr, ontl:hasName, ?steptype),
    PropertyValue(?testattr, ontl:hasAttributeValue, ?steptext)
} OR WHERE {
    [...]
    PropertyValue(?testattr, ontl:hasName, "When"),
    [...]
} OR WHERE {
    [...]
    PropertyValue(?testattr, ontl:hasName, "Then"),
    [...]
}
```

Listing 7: Abfrage aller Paare von Testimplementierungen und Attributwerten

elementes, sondern mit dem Individuum des Testschrittes. Dafür verwenden wir die Relation *hasStepReference*, wie in Punkt 3 von Abbildung 5.8 dargestellt ist.

Die Testschritte beinhalten des Weiteren durch ihren Text die Informationen zur Verbindung mit ihren Testschrittimplementierungen. Wie wir in Abschnitt 4.2.1.5 erläuterten, müssen wir diese Verbindung jedoch auflösen. Zu diesem Zweck fragen wir die Ontologie nach allen Funktionen, die Testschrittimplementierungen darstellen, und ihren dazugehörigen Attributen/Attributwerten ab. Testimplementierungen werden in der Ontologie nicht explizit gekennzeichnet. Wie in Abschnitt 4.2.1.5 beschrieben, zeichnet ein Attribut eine Methode als Testimplementierung aus. Daher nutzen wir als Indikator einer Testschrittimplementierung damit verbundene Attribute mit dem Text „`Given`“, „`When`“ oder „`Then`“. Die gekürzte Abfrage nach allen Testimplementierungen ist in Listing 7 angegeben.

Weiterhin legten wir in Abschnitt 4.2.1.5 dar, dass der Wert des Attributes einen regulären Ausdruck darstellt. Daher müssen wir für jeden Testschritt prüfen, auf welches Attribut er sich bezieht. Ist der Zusammenhang gefunden, so können wir den Testschritt direkt mit der Implementierung verknüpfen. Wie Punkt 4 von Abbildung 5.8 zeigt, dient dazu die Relation *hasTestImplementation*.

5.5 Anwendungen auf Grundlage der Ontologie

In den vorigen Abschnitten haben wir die Implementierung zum Aufbau der Ontologie beschrieben. Hier erläutern wir, wie die Informationen genutzt werden, um die in den Abschnitten 4.2.2, 4.2.3 und 4.2.4 entworfene Funktionalität zu ermöglichen.

Zunächst betrachten wir dazu das Glossar in Abschnitt 5.5.1. In Abschnitt 5.5.2 erläutern wir, wie wir Testschritte aus der Ontologie erhalten und wie wir deren Relevanz bewerten. Abschnitt 5.5.3 befasst sich mit der Gewinnung relevanter API-Komponenten.

5.5.1 Glossar

Die Konzepte, die zur Gruppierung von NSEs dienen, sind *StoryConcept* und dessen wortartbasierte Spezialisierungen *VerbConcept* und *NounConcept*. Alle Einträge des Glossars erhalten wir somit über die Abfrage aller Individuen des Konzeptes StoryConcept. Für einen Glossareintrag sind jedoch vor allem dessen Eigenschaften

```
1 PREFIX ontl: <http://ontoserv/project1#>
2 SELECT ?nle_id ?nle_text ?nle_type WHERE {
3     Type(?nle_id, ontl:StoryConcept),
4     PropertyValue(?nle_id, ontl:hasName, ?nletext),
5     DirectType(?nle_id, ?nletype)
6 }
```

Listing 8: Abfrage aller Glossar-Einträge mit ihrer die Wortart spezifizierenden Klasse

```
1 PREFIX ontl: <http://ontoserv/project1#>
2 SELECT ?desc WHERE {
3     PropertyValue(ontl:noun_story_concept-button, ontl:hasDescription, ?desc)
4 }
```

Listing 9: Abfrage der optionalen Beschreibung des NSEs „button“

von Relevanz für einen Benutzer. Eine Dateneigenschaft, die jeder Glossareintrag zugewiesen bekommt, ist die Verknüpfung der Zeichenkette über die Relation *hasName*. Über eine weitere Dateneigenschaft können wir einem Glossareintrag optional eine Beschreibung zuweisen. Das gleiche gilt auch für die Objekteigenschaften, die Synonymie-Beziehungen realisieren.

In Listing 8 ist die Abfrage nach allen Glossareinträgen angegeben. Die Variable `?nle_id` gibt uns die IRI des Glossareintrages in der Ontologie zurück. Mithilfe dieser IRI können wir optionale Parameter wie semantische Verbindungen zu anderen Einträgen und die Beschreibung abrufen, was wir in den Listings 9 und 10 zeigen. Die mit Präfix `ontl` versehene IRI `noun_story_concept-button` ist der Lesbarkeit halber in gekürzter Form verwendet. Die tatsächliche Synonymie-Relation (Synonym, Hyponym oder Hyperonym) kann man aus `?stype` schließen.

Für den Zugriff auf das Glossar haben wir eine webbasierte graphische Benutzerschnittstelle entwickelt. Diese nutzt die obigen Abfragen. Abbildung 5.9 zeigt die Ansicht einer Liste von Einträgen im Glossar. Die Detailansicht wird in Abbildung 5.10 dargestellt.

Da die Ontologieschnittstelle bislang nur Operationen zum hinzufügen enthält, können Einträge nicht über F-TRec geändert werden. Dazu kann ein Ontologieeditor wie *protégé* dienen.

5.5.2 Relevante Testschritte

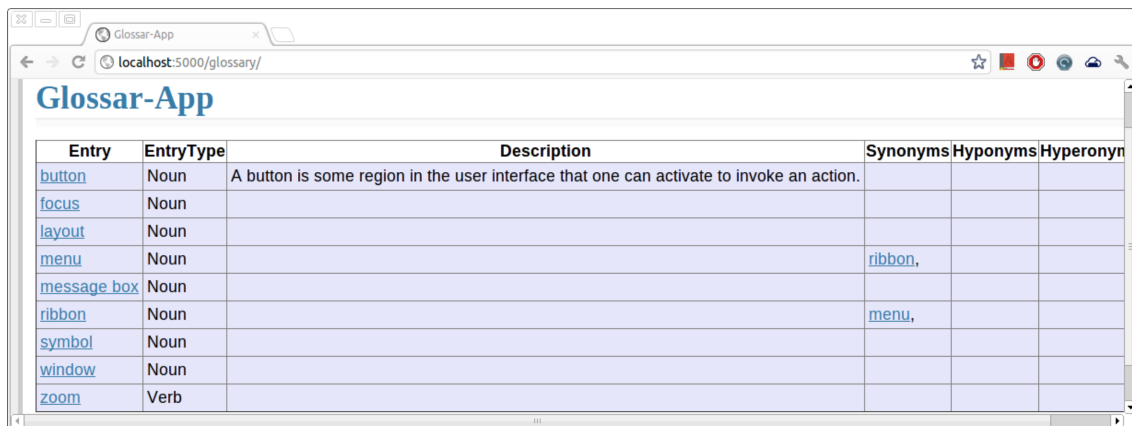
Auf zwei verschiedene Arten fragen wir die Ontologie bei der Suche nach relevanten Testschritten ab.

Begrifflich verwandte Testschritte

Einerseits verwenden wir dazu ein Verfahren, das im wesentlichen dem Booleschen Modell bei Verwendung einer *OR*-Verknüpfung entspricht (siehe Abschnitt 2.5). Dies

```
1 PREFIX ontl: <http://ontoserv/project1#>
2 SELECT ?synonym ?synonym_text ?stype WHERE {
3     DirectSubPropertyOf(?stype, ontl:isSemanticallyRelatedTo),
4     PropertyValue(ontl:noun_story_concept-button, ?stype, ?synonym),
5     PropertyValue(?synonym, ontl:hasName, synonym_text)
6 }
```

Listing 10: Abfrage existierender Synonyme des NSEs „button“



Entry	EntryType	Description	Synonyms	Hyponyms	Hyperonyn
button	Noun	A button is some region in the user interface that one can activate to invoke an action.			
focus	Noun				
layout	Noun				
menu	Noun		ribbon,		
message box	Noun				
ribbon	Noun		menu,		
symbol	Noun				
window	Noun				
zoom	Verb				

Abbildung 5.9: Listenansicht der Glossaroberfläche

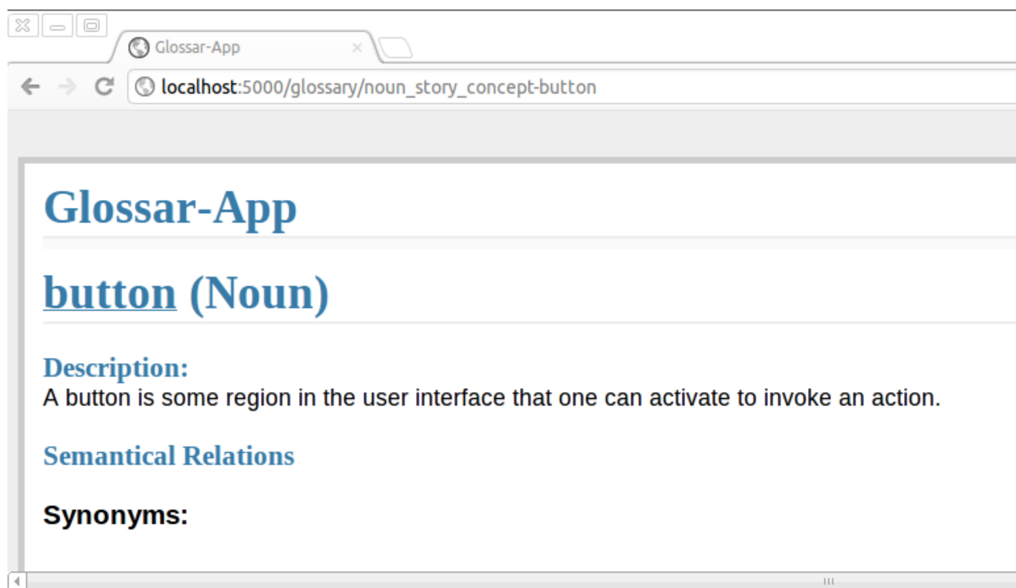


Abbildung 5.10: Detailansicht eines Eintrags

```
PREFIX ontl: <http://ontoserv/project1#>
SELECT DISTINCT ?step ?steptype ?steptext ?nle_type ?nle_text WHERE {
   Type(?concept, ontl:NounConcept),
   PropertyValue(?concept, ontl:hasName, "button"),
   PropertyValue(?concept, ontl:hasStepReference, ?step),
   PropertyValue(?step, ontl:hasTestImplementation, ?step_impl),
   PropertyValue(?step, ontl:isReferringToConcept, ?referred_concept),
   PropertyValue(?referred_concept, ontl:hasName, ?nle_text),
   DirectType(?referred_concept, ?nle_type),
   PropertyValue(?step_impl, ontl:hasAttribute, ?testattr),
   PropertyValue(?testattr, ontl:hasAttributeValue, ?steptext),
   PropertyValue(?testattr, ontl:hasName, ?steptype)
}
```

Listing 11: Abfrage aller Testschritte, die das NSE „button“ referenzieren

zeigen wir in Listing 11. Darin ist auch die Einschränkung nach der Wortart enthalten (`ontl:NounConcept`). Zur Realisierung der in Abschnitt 4.2.3.1 beschriebenen Funktionalität führen wir diese Abfrage für alle in einer User Story vorhandenen NSEs aus. Die Ergebnisse gruppieren wir, sodass wir zu jedem Testschritt alle referenzierten NSEs erhalten bzw. umgekehrt. Außerdem benötigen wir zur Berechnung der inversen Dokumentfrequenz das Verhältnis der Anzahl an referenzierten Schritten eines NSEs zur Gesamtzahl der Schritte. Beides erhalten wir jeweils aus den Gruppierungen.

Bei der Extraktion der NSEs aus der User Story geben wir eine Liste der NSEs zurück. Dies ermöglicht die Bestimmung der Termfrequenz. Wie wir in Abschnitt 4.2.3.1 erwähnten, normalisieren wir die Frequenz durch Bildung eines Frequenzranges. Dazu sei N die Anzahl der auftretenden Termfrequenzen. Für jede Termfrequenz bestimmen wir einen Wert, der zwischen 1 und $N + 1$ liegt. Terme mit der höchsten Termfrequenz erhalten also die normalisierte Termfrequenz $N + 1$, während solche mit der geringsten Termfrequenz die normalisierte Termfrequenz 1 erhalten.

In der Liste bleibt die Reihenfolge der NSEs entsprechend ihres Vorkommens in der User Story intakt, was wir für die Berechnung der Kookkurrenz-Distanz verwenden.

Häufig verwendete Testschritte

Die herangezogene Verwendungshäufigkeit bezieht sich auf die Testskripte. Somit müssen wir zu der Testimplementierung jedes Testschrittes bestimmen, aus wie vielen Testskripten heraus sie referenziert wird.

Grundlage dafür ist die Abfrage in Listing 12. Zusätzlich zu der Implementierung gibt diese auch das Attribut der Implementierung zurück. Die Verwendungshäufigkeit ergibt sich aus der Anzahl von Paaren, die zu einem Testschritt zurückgeliefert werden.

Wir erhalten also alle Testschritte zurück. Diese Anzahl begrenzen wir durch die Auswahl von Testskripten, die begrifflich relevant sind. Ähnlich zu der booleschen Abfrage aus dem vorigen Abschnitt erhalten wir diese über Listing 13. Wir betrachten dann nur die 5 ähnlichsten Testskripte, wobei die Ähnlichkeit über die Anzahl referenzierender NSEs bestimmt wird.

Wir können nun alle Testschritte bestimmen, die von diesen Testskripten verwendet werden und sortieren diese Testschritte nach ihrer Häufigkeit.

```
PREFIX ontl: <http://ontoserv/project1#>
SELECT DISTINCT ?test_script ?impl ?step_type ?step_template WHERE {
    Type(?step, ontl:ScriptStep),
    PropertyValue(?step, ontl:hasTestImplementation, ?impl),
    PropertyValue(?step, ontl:isInScenario, ?scenario),
    PropertyValue(?scenario, ontl:isInFeature, ?test_script),
    PropertyValue(?impl, ontl:hasAttribute, ?test_attr),
    PropertyValue(?test_attr, ontl:hasName, ?step_type),
    PropertyValue(?test_attr, ontl:hasAttributeValue, ?step_template)
}
```

Listing 12: Abfrage aller Testschrittimplementierungen und sie referenzierende Testskripte

```
PREFIX ontl: <http://ontoserv/project1#>
SELECT ?test_script WHERE {
    Type(?concept, ontl:NounConcept),
    PropertyValue(?concept, ontl:hasName, "button"),
    PropertyValue(?concept, ontl:isLinkedToFeature, ?test_script)
}
```

Listing 13: Abfrage aller Testskripte, die das NSE „button“ referenzieren

5.5.3 Relevante API-Komponenten

Für die Gewinnung relevanter API-Komponenten nutzen wir die Verbindungen der Testimplementierungen zu weiteren Komponenten des Quelltextes. Wie in Abschnitt 4.2.4 erläutert, erhalten wir die Abfrage-NSEs aus der Analyse eines Testschrittes. Mit diesen fragen wir die Ontologie nach Testschritten über die Abfrage in Listing 11 ab. Zu jedem Testschritt erhalten wir über die in Listing 14 angegebene Abfrage die von der Implementierung referenzierten Komponenten.

Da wir die Komponenten nach ihrer Verwendungshäufigkeit gewichten wollen, bestimmen wir nun, wie oft eine Komponente in den Ergebnissen zurückgeliefert wird. Wir geben die Komponenten dann nach ihrer Verwendungshäufigkeit sortiert zurück.

5.6 Zusammenfassung

In diesem Kapitel motivierten und beschrieben wir die Architektur von F-TRec.

Dazu erläuterten wir den Einsatz einer netzwerkbasierten und plattformtransparenten Vermittlungsschicht, durch die Plattformabhängigkeiten der interagierenden Komponenten gelöst werden.

```
PREFIX ontl: <http://ontoserv/project1#>
SELECT ?component ?component_type WHERE {
    DirectType(?attr, ontl:Attribute),
    PropertyValue(?attr, ontl:hasName, "' + step_type + '"),
    PropertyValue(?attr, ontl:hasAttributeValue, "' + step_text + '"),
    PropertyValue(?attr, ontl:isAttributeOf, ?meth),
    PropertyValue(?meth, ontl:isInvoking, ?component),
    DirectType(?component, ?component_type)
} OR WHERE {
    DirectType(?attr, ontl:Attribute)
    PropertyValue(?attr, ontl:hasAttributeValue, "' + step_text + '")
    PropertyValue(?attr, ontl:hasName, "' + step_type + '"),
    PropertyValue(?attr, ontl:isAttributeOf, ?meth),
    PropertyValue(?meth, ontl:isCreating, ?component),
    DirectType(?component, ?component_type)
}
```

Listing 14: Abfrage der von einer Testimplementierung verwendeten Komponenten

Ontologieschnittstellen ermöglichen den schreibenden und lesenden Zugriff auf die als Wissensbasis genutzte Ontologie. Für den schreibenden Zugriff sind elementare Funktionen entstanden, durch die die Erstellung von Individuen, Verknüpfung zwischen Individuen und Hinzufügen von Daten zu einem Individuum realisiert werden kann. Lesender Zugriff auf die Ontologie wird durch die Nutzung einer Abfragebibliothek über eine Abfragesprache namens SPARQL-DL bereitgestellt.

Des Weiteren beschrieben wir, wie die Befüllung der Ontologie aus Quelltext, User Stories und Testskripten implementiert wurde. Auch die Herstellung des Zusammenhangs wird hier detailliert erläutert.

Die in den Abschnitten 4.2.2, 4.2.3 und 4.2.4 behandelten Anwendungen, die auf der Wissensbasis operieren, betrachteten wir hier unter dem Gesichtspunkt ihrer Implementierung. Diese besteht zu wesentlichen Teilen in der Abfrage der in der Ontologie vorhandenen Informationen.

6. Evaluierung

In diesem Kapitel evaluieren wir das Konzept bzw. die Implementierung. Auswertbare Ergebnisse können wir zur Gewinnung relevanter Testschritte und Quelltextfragmente darstellen. Wie in Kapitel 1 beschrieben, wurde die in dieser Arbeit beschriebene Lösung anhand der Analyse eines laufenden Projektes eines kooperierenden Unternehmens erarbeitet. Daher stand uns zur Evaluation dieses Projekt vollständig zur Verfügung.

In Abschnitt 6.1 geben wir Informationen zu den Projektdaten an. Die Gewinnung relevanter Testschritte zur Assistenz von Testern evaluieren wir in Abschnitt 6.2. Abschnitt 6.3 zeigt die Bewertung der Gewinnung relevanter Quelltextkomponenten zur Assistenz von Entwicklern. Weiterhin haben wir in Abschnitt 6.4 Laufzeitcharakteristiken der Komponenten von F-TRec aufgeführt.

6.1 Informationen zu Evaluationsdaten

Die Projektdaten die wir für diese Arbeit verwendeten, bestanden aus

- User Stories, die zum Betrachtungszeitpunkt bereits erfüllt waren,
- Quelltext der funktionalen Tests mit den dazugehörigen Testskripten und
- aus dem Quelltext extrahierten Modellen der Software.

Wir erhoben die Daten an jeweils zwei Terminen, so dass zwei Projektstände erfasst sind. Der Quelltext des Projektes stand jeweils nur für die Extraktion des Softwaremodelles zur Verfügung. Die Maße für die Größe des Quelltextes wurden (soweit genau angegeben) mit *ohcount* ermittelt. Dabei haben wir nur die Angabe für C#.NET-Quelltext verwendet; XML-Quelltext und anderes blieb ohne Beachtung.

Artefakte	Größe/Menge	
	Projektstand 1	Projektstand 2
Quelltext Gesamt	ca. 200K Zeilen (*LoC*)	223808 Zeilen (*LoC*)
Quelltext funktionaler Tests	9979 Zeilen (*LoC*)	16722 Zeilen (*LoC*)
User Story/Testskript-Paare	42 Stck	63 Stck
User Story Ø	104 Wörter	114 Wörter
User Story min.	42 Wörter	24 Wörter
User Story max.	417 Wörter	417 Wörter
Testschritte	702 Stck.	985 Stck.
Testschrittimplementierungen	196 Stck.	262 Stck.
Testschritte/Testskript	ca. 15,1 Ø	ca. 14,8 Ø
Implementierungen/Testskript	ca. 6,5 Ø	ca. 7,5 Ø

Tabelle 6.1: Vergleich der Projektdaten

6.1.1 1. Projektstand

Bei der Datenerfassung des ersten Projektstandes haben wir keine Informationen zur Quelltextgröße notiert. Daher ist für diesen Stand keine genauere Zahl angegeben. Des Weiteren konnte eine veraltete Version des Zerteilers für C#.NET viele Aufrufe und Objekterstellungen nicht auflösen. Dies war erst mit Einsatz der neueren Version im zweiten Projektstand möglich.

Wie in Tabelle 6.1 ersichtlich ist, enthalten die User Stories durchschnittlich 104 Wörter. Testschrittimplementierungen werden häufig wiederverwendet, wie sich an der Anzahl von Testschritten und Testschrittimplementierungen ablesen lässt. Durchschnittlich enthält ein Testskript 15,1 Testschritte, wobei nur 6,5 Testschrittimplementierungen verwendet werden.

6.1.2 2. Projektstand

Bei der Extraktion des 2. Projektstandes konnten wir eine genaue Quelltextgrößenbestimmung vornehmen. Tabelle 6.1 führt die Daten detailliert auf. Die Größe des gesamten Quelltextes liegt bei ca. 224T Zeilen, wovon der Quelltext funktionaler Tests ca. 17T Zeilen einnimmt. Die Anzahl der User Story/Testskript-Paare hat deutlich zugenommen im Vergleich zum 1. Projektstand. Auch der Testquelltext ist deutlich gewachsen.

Das Verhältnis von Testschritten zu deren Implementierungen ist leicht gestiegen (ca. 3,58 beim 1. Projektstand, ca. 3,76 beim 2. Projektstand). User Stories enthalten im 2. Projektstand durchschnittlich etwas mehr Wörter (114 statt 104). Durchschnittlich enthält ein Testskript mit 14,8 Testschritten etwas weniger, als es beim 1. Projektstand der Fall war. Allerdings werden hier 7,5 unterschiedliche Testschrittimplementierungen verwendet, was eine geringere Wiederverwendung im 2. Projektstand bedeutet.

6.2 Evaluation der Testschrittgewinnung

Um die Testschrittgewinnung bewerten zu können, benötigen wir zunächst User Stories, für die wir die tatsächlich relevanten Ergebnisse kennen. Dies erreichen wir

#Ergebnisse	#Gefunden	#Relevant	#Extra	Ausbeute	Präzision
5	3	7	3	0.429	0.6
10	6	7	3	0.857	0.6
15	7	7	3	1.0	0.467
20	7	7	3	1.0	0.35
25	7	7	3	1.0	0.28
30	7	7	3	1.0	0.233
35	7	7	3	1.0	0.2
36	7	7	3	1.0	0.193

Tabelle 6.2: Beispiel zur Bewertung der Ergebnisse eines Verfahrens mit einer User Story

durch die Verwendung einer Teststichprobe, die aus den User Story-/Testskript-Paaren der jeweils neuesten 2 Sprints besteht. Die Anzahl von User Stories in einem Sprint war unterschiedlich, des Weiteren existierte nicht für jede User Story ein Testskript. Aus diesem Grund war in dem jeweils neuesten Sprint die Anzahl an User Stories zu gering für eine repräsentative Teststichprobe.

Bei den User Stories, die für einen Sprint ausgewählt wurden, können wir von weitgehender Unabhängigkeit ausgehen, da diese häufig parallel entwickelt werden. Dies ist bei User Stories aus zwei verschiedenen Sprints nicht der Fall. Zur Überprüfung der von F-TRec für eine User Story gelieferten Ergebnisse lassen wir diese User Story und das entsprechende Testskript bei der Befüllung der Ontologie unberücksichtigt. Die zum gleichen Sprint gehörenden User Stories werden jedoch in die Ontologie übernommen. Der jüngere der beiden Sprints referenziert unter Umständen NSEs und Testimplementierungen, die im älteren Sprint noch nicht bekannt sein können. Daher müssen wir zur Betrachtung der User Stories des älteren der zwei Sprints zusätzlich alle User Stories des jüngeren Sprints bei der Befüllung auslassen.

Wir verwenden die mit dem Projektquelltext befüllte Ontologie gemäß Abschnitt 4.2.1.2 und führen weiterhin die in Abschnitten 4.2.1.3, 4.2.1.4 und 4.2.1.5 beschriebenen Schritte durch. Dabei lassen wir jedoch wie beschrieben jeweils die User Story unberücksichtigt, für die die Evaluation durchgeführt werden soll. Zusätzlich bleiben für User Stories des älteren Sprints solche des jüngeren außen vor. Es entsteht so für jeden Testschritt der Teststichprobe eine Ontologie.

Die zur Evaluation bei der Befüllung ausgelassene User Story aus der Teststichprobe hat ein Testskript, dessen Testschritte wir als optimale Ergebnismenge auffassen können. Einschränkend dazu kann man jedoch nur solche Testschritte finden, die bereits von anderen Testskripten verwendet wurden. Alle anderen Testschritte wurden ausschließlich zum testen der betrachteten User Story implementiert. Wir können solche Testschritte von den anderen trennen und erhalten so die optimale Menge findbarer Testschritte.

Bezüglich der in Abschnitt 2.5.2 beschriebenen Ausbeute und Präzision ist diese optimale Menge an Testschritten gleichbedeutend mit der Menge relevanter Dokumente (R). Für eine Ergebnismenge, die wir mit einem der in den Abschnitten 4.2.3 und 5.5.2 beschriebenen Verfahren gewinnen, können wir die Ausbeute und Präzision mithilfe der aus dem Testskript erhaltenen optimalen Ergebnismenge errechnen.

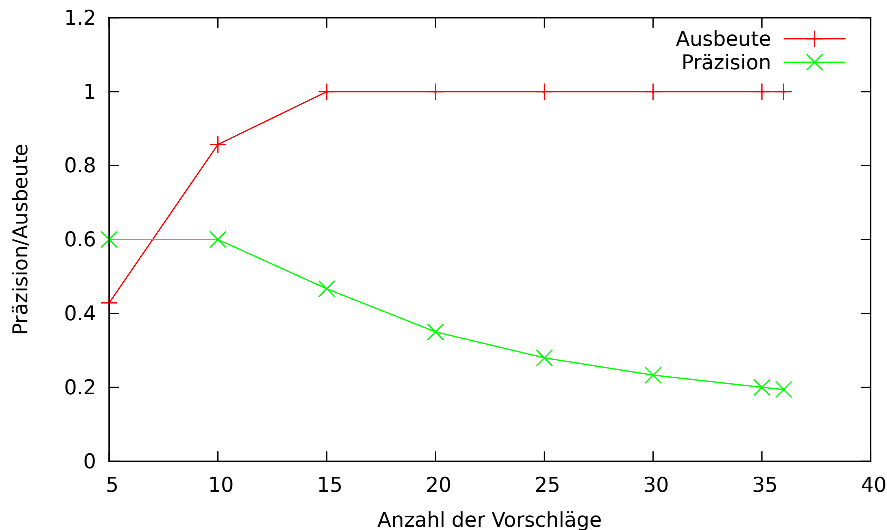


Abbildung 6.1: Beispielgraph für Präzision und Ausbeute aus Tabelle 6.2

Die beschriebenen Verfahren liefern jeweils eine unterschiedliche Anzahl an Ergebnissen. Wir verwenden die vollständigen Ergebnislisten. Um eine genauere Betrachtung der Qualität der Ergebnisse zu erhalten, berechnen wir Präzision und Ausbeute für mehrere Abschnitte der Menge. Wir haben eine Schrittweite von 5 gewählt, es werden also die $5, 10, \ldots, n$ höchstbewerteten Schritte zurückgegeben, wobei n die Anzahl von Testschritten ist, die ein Verfahren bezüglich einer User Story liefert.

Dadurch ergibt sich pro Verfahren und User Story der Teststichprobe ein Bewertungsergebnis wie in Tabelle 6.2. *#Ergebnisse* gibt die Anzahl an zurückgegebenen Ergebnissen an. Die Anzahl gefundener Testschritte ist in der Spalte *#Gefunden* angegeben, während in der Spalte *#Relevant* die Anzahl findbarer Testschritte und unter *#Extra* die Anzahl ausschließlich für dieses Testskript eingeführter Testschritte angezeigt ist. *Ausbeute* und *Präzision* lassen sich aus den Spalten *#Ergebnisse*, *#Gefunden* und *#Relevant* berechnen. Mit diesen Informationen können wir einen Graph für Präzision und Ausbeute zeichnen. Für das Beispiel aus Tabelle 6.2 ergibt sich der in Abbildung 6.1 gezeigte Graph.

Die Ergebnisse aller User Stories eines Projektstandes bzgl. eines Verfahrens können wir nun durch Addition der Einträge von einander entsprechender Zeilen zusammenführen. Das sind Zeilen mit gleicher Ergebnisanzahl bzw. der gleichen Zeilennummer: Sind die Ergebnisanzahlen z.B $20, 20, 18$, was jeweils der 4. Zeile entspricht, werden diese auf 58 addiert und ebenso die anderen Angaben dieser Zeilen. Ist die entsprechende Zeile in einem Ergebnis nicht enthalten, wird die letzte Zeile verwendet. Die Werte für Ausbeute und Präzision summieren wir jedoch nicht auf. Für die entstehenden Summen können diese wiederum berechnet werden und bilden so den Ergebnisdurchschnitt.

Für den 1. Projektstand sind 7 User Stories mit Testskripten in den jüngsten zwei Sprints bearbeitet worden. Die zwei jüngsten Sprints des 2. Projektstandes enthalten 9 User Stories mit Testskripten. Diese User Stories wurden in die Teststichprobe aufgenommen.

Verfahren \ Ausbeute	10%	20%	30%	40%	50%
TF-IDF	14,3%	6,3%	6,7%	7,6%	6,9%
TF-IDF mit Kookkurrenzheuristik	14,3%	6,3%	6,7%	6,7%	6,8%
TF-IDF mit NSE-Differenzheuristik	17,1%	7,6%	9,1%	9,1%	7,2%
TF-IDF mit beiden Heuristiken	14,3%	7,9%	9,7%	9,7%	7,5%
Verwendungshäufigkeit	22,9%	22,9%	24,3%	24,3%	20,0%
TF-IDF und Verwendungshäufigkeit	17,1%	18,6%	18,6%	15,7%	15,7%

Tabelle 6.3: Präzision pro erreichte Ausbeute eines Verfahrens in Projektstand 1

Die Graphen für Präzision und Ausbeute sind in den Abbildungen 6.2 und 6.3 dargestellt. Zu beachten ist in den Graphen, dass die Ergebnisanzahl (in den Abbildungen als Anzahl der Vorschläge bezeichnet) jeweils die Summe der Ergebnisanzahlen für jede User Story der jeweiligen Teststichprobe darstellt. 90 Ergebnisse stehen also für jeweils zehn Vorschläge der neun User Stories. Die mit steigender kumulierter Ergebnisanzahl geringeren Abstände der Messungen kommen durch die unterschiedlichen Ergebnisanzahlen zustande.

Zum einfacheren Vergleich der Verfahren haben wir in den Tabellen 6.3 und 6.4 Werte für die Ausbeute und die mit dieser Ausbeute auftretende Präzision aufgetragen. Da wir nicht zu jedem Verfahren Messpunkte mit der angegebenen Ausbeute haben, geben wir in einem solchen Fall die Präzision der nächsthöheren Ausbeute an. Bei mehrfachen Messpunkten mit derselben Ausbeute verwenden wir die höhere Präzision.

Abbildung 6.4 zeigt Graphen für Ausbeute, Präzision und das F_1-Maß der Ergebnisse für die einzelnen User Stories beider Projektstände. Hier ist nur die Kombination aus TF-IDF-basiertem Verfahren und der Verwendungshäufigkeit dargestellt, in Anhang D sind diese Verfahren getrennt voneinander aufgeführt. Deutlich ist für Projektstand 1 an der Präzision (Abbildung 6.4c) und dem F_1-Maß (Abbildung 6.4e) zu erkennen, dass das Verfahren für manche User Stories eine bessere Sortierung ermöglicht als für andere. Auch lässt sich anhand des Graphen für das F_1-Maß eine Anzahl zurückgelieferter Ergebnisse bestimmen, für die möglichst viele relevante Ergebnisse in einer möglichst kleinen Ergebnismenge liegen. Bezüglich Abbildung 6.4e ist ein Wert zwischen 20 und 25 sinnvoll, da in diesem Intervall die Werte des F_1-Maßes abnehmen. Im zweiten Projektstand liegen die einzelnen Graphen weitaus enger aneinander. Ein sinnvoller Wert zur Begrenzung der Ergebnismenge liegt hier eher zwischen 30 und 40, wie man in Abbildung 6.4f erkennen kann.

6.2.1 TF-IDF-basiertes Verfahren

Hier betrachten wir die in Abschnitt 4.2.3.1 erläuterten Verfahren. Da diese Verfahren dieselbe Abfrage von Testschritten verwenden und sich nur durch Sortierungsheuristiken unterscheiden, erhalten wir hier jeweils dieselbe Anzahl an Testschritten. Abbildung 6.2 zeigt die Ergebnisse für den ersten Projektstand, Abbildung 6.3 die für den zweiten Projektstand.

In den Graphiken zu dem TF-IDF-Verfahren ohne zusätzliche Heuristiken in den Abbildungen 6.2a und 6.3a sind die Kurven für Präzision und Ausbeute aufgetragen.

Verfahren \ Ausbeute	10%	20%	30%	40%	50%
TF-IDF	7,4%	6,2%	6,7%	7,2%	6,1%
TF-IDF mit Kookkurrenzheuristik	5,2%	5,8%	6,9%	6,7%	6,2%
TF-IDF mit NSE-Differenzheuristik	7,4%	7,1%	6,4%	6,2%	6,2%
TF-IDF mit beiden Heuristiken	5,9%	5,8%	6,0%	6,5%	6,1%
Verwendungshäufigkeit	15,6%	14,1%	14,1%	15,0%	15,1%
TF-IDF und Verwendungshäufigkeit	15,6%	8,9%	8,9%	10,2%	11,4%

Tabelle 6.4: Präzision pro erreichte Ausbeute eines Verfahrens in Projektstand 2

Maximal wird in Projektstand 1 eine Ausbeute von ca. 57,5% bei einer Präzision von 6,5% und 335 Ergebnissen erreicht. Das entspricht ca. 48 Ergebnissen je User Story. Die Präzision liegt maximal bei einem Wert von 14,3%, wobei die Ausbeute 12,5% und die Anzahl der Ergebnisse 35 beträgt, also 5 Ergebnisse je User Story.

In Projektstand 2 hingegen erreicht die Ausbeute ihr Maximum bei 67,2% bei einer Präzision von 5,5% und 740 Ergebnissen, was ca. 82 Ergebnissen je User Story entspricht. Das Maximum der Präzision liegt bei 7,4% bei einer Ausbeute von 16,4% und 135 Schritten, also ca. 15 Ergebnissen je User Story.

Ganz ähnliche Charakteristiken weisen die Kombinationen des TF-IDF-Verfahrens mit den verschiedenen Heuristiken auf. In den Abbildungen 6.2b und 6.3b sind Präzision und Ausbeute für die Kombination des TF-IDF-Verfahrens mit der kookkurrenzbasierten Heuristik aufgetragen. Graphen für die Kombination des TF-IDF-Verfahrens mit der Differenz abgefragter und nicht abgefragter NSE-Anzahlen befinden sich in den Abbildungen 6.2c und 6.3c. Die Abbildungen 6.2d und 6.3d zeigen die Ergebnisse für die Kombination des TF-IDF-Verfahrens mit beiden o.g. Heuristiken.

Die maximale Ausbeute liegt hier jeweils – da die Ergebnismenge gleich ist – auf dem selben Wert. Doch die Sortierung der Testschritte wird von den Heuristiken nicht positiv beeinflusst, was man an den Werten der Präzision und Ausbeute für geringere Ergebnisanzahlen erkennen kann. Die NSE-Differenzheuristik weist, wie in Tabelle 6.3 ersichtlich, eine leicht höhere Präzision als die anderen TF-IDF-basierten Verfahren für Ausbeuten von 30% und 40% auf. Im zweiten Projektstand (siehe Tabelle 6.4) sehen wir jedoch, dass dies nicht stabil ist.

6.2.2 Verwendungshäufigkeit in ähnlichen Testskripten

Deutlich höhere Präzision weisen die Graphen zu dem auf Verwendungshäufigkeit basierenden Verfahren auf. Diese befinden sich in den Abbildungen 6.2e und 6.3e. Für Projektstand 1 ergibt sich ein Maximalwert für die Ausbeute von 87,5% bei einer Präzision von 15,2% und einer Ergebnisanzahl von 231 (33 Vorschläge pro User Story). Die Präzision beträgt maximal 24,3%, wobei die Ausbeute dann bei 42,5% und die Ergebnisanzahl bei 70 liegt, also 10 Ergebnissen je User Story.

In Projektstand 2 erreichen wir eine maximale Ausbeute von 67,2% bei einer Präzision von 9,5% und 430 Ergebnissen (48 Ergebnissen pro User Story). Die maximale Präzision liegt in Projektstand 2 bei einem Wert von 15,6% mit einer Ausbeute von 11,5% und 45 Ergebnissen (5 Vorschlägen pro User Story).

6.2.3 Kombination TF-IDF und Verwendungshäufigkeit

In der Kombination eines IDF-Verfahrens – hier mit allen Heuristiken – und des auf Verwendungshäufigkeit basierenden Verfahrens ist vorwiegend eine Verbesserung der Ausbeute im Vergleich zu den einzeln betrachteten Verfahren zu beobachten. So erreichen wir in beiden Projektständen eine Ausbeute von ca. 95% (siehe Abbildungen 6.2f und 6.3f). Die Präzision liegt dabei in Projektstand 1 bei 12,1% und 315 Ergebnissen (45 je User Story); in Projektstand 2 beträgt die Präzision bei maximaler Ausbeute 6,0% mit 968 Ergebnissen (108 Ergebnisse je User Story). Die Erhöhung der Ausbeute untermauert die in Abschnitt 4.2.3.3 getroffene Annahme, dass diese Verfahren unterschiedliche Testschritte liefern.

Die maximale Präzision liegt in Projektstand 1 bei 18,6% bei einer Ausbeute von 32,5% und 70 Ergebnissen (10 je User Story). In Projektstand 2 beträgt die maximale Präzision 15,6% bei einer Ausbeute von 11,5% und 45 Ergebnissen (5 je User Story).

6.2.4 Unterschiede der Projektstände

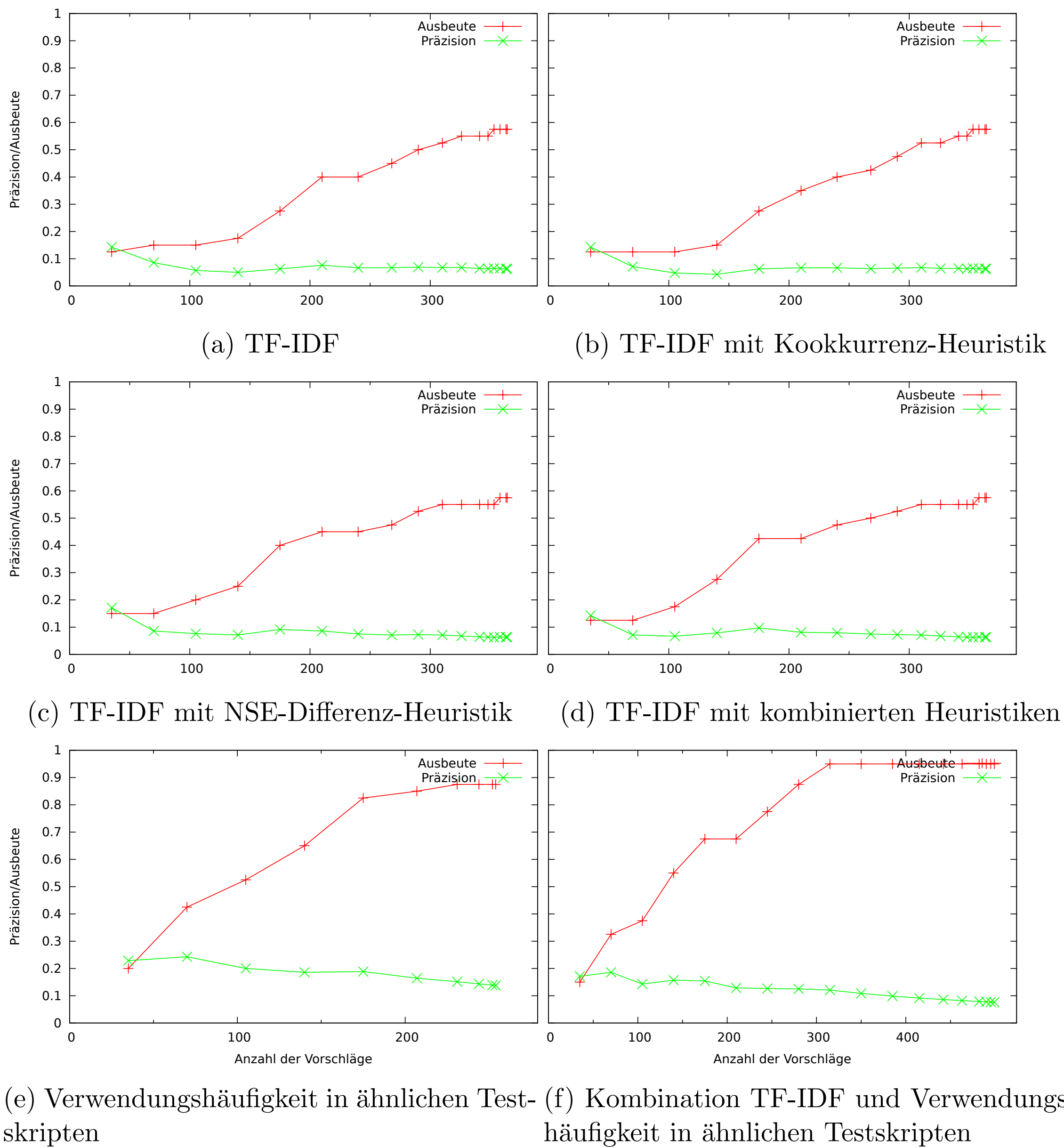
In Abschnitt 4.2 stellten wir die Hypothese auf, dass F-TRec bei wachsender Datenbasis des Projektes eine höhere Präzision bei der Suche von Komponenten wie Testschritten erreicht. Diese Aussage kann jedoch durch die Ergebnisse für die zwei Projektstände nicht gestützt werden. Die Präzision nimmt im Gegenteil ab; auch die Ausbeute erhöht sich nicht in allen Verfahren. Der Grund hierfür ist, dass mehr Testschritte für eine Anfrage geliefert werden und die Sortierung durch die verschiedenen Verfahren dies nicht kompensieren kann.

6.2.5 Mögliche Einschränkungen der Aussagekraft

Wir analysieren nicht tatsächlich den Quelltext des Projektes zu verschiedenen Zeitpunkten, sondern lassen bei der Befüllung die jeweiligen Testskripte und User Stories außen vor. Existierende Testschrittimplementierungen werden bei der Befüllung der Ontologie mit dem Quelltext der funktionalen Tests in der Ontologie repräsentiert. Wenn sie von keinem Testskript außer dem außen vor gelassenen referenziert werden, haben sie jedoch keine Verknüpfungen zu Testschritten. Somit ist aus der von Testschritten ausgehenden Sicht der Suche dieser Testschritt nicht existent. Unterschiede zu einem auf verschiedenen Quelltextständen operierenden Verfahren existieren dennoch. So können Testschrittimplementierungen beispielsweise bei der Entwicklung des Testskriptes eine andere Parametrierung erhalten. Die Aussagekraft der Ergebnisse wird dadurch jedoch nicht stark eingeschränkt, da wir auch solche Testschritte vorschlagen wollen, die unter Umständen durch Änderung der Parametrisierung ein weiteres Einsatzfeld erhalten.

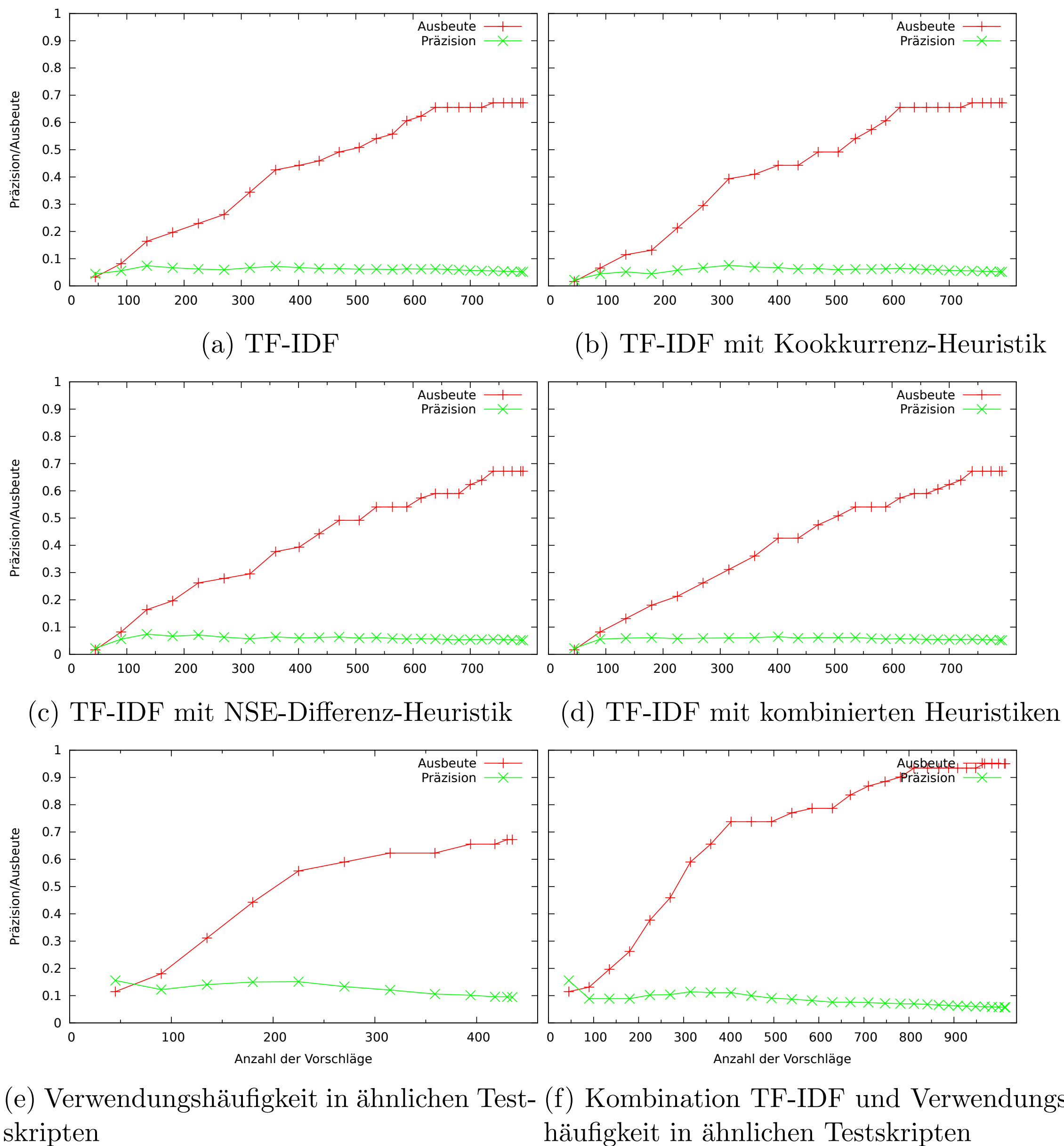
6.2.6 Nutzung von Synonymie-Beziehungen

In Abschnitt 4.2.2.1 haben wir erläutert, dass das Glossar auch in der Assistenz von Testern und Entwicklern genutzt werden kann. Durch Ausnutzung von Synonymiebeziehungen konnten wir Verbesserungen der Ergebnisse feststellen. Dies haben wir in einem Artikel, der im Rahmen der vorliegenden Arbeit entstanden ist, anhand einzelner, leicht identifizierbarer Synonyme gezeigt [55]. Bei der Evaluation dieser Arbeit haben wir dies nicht weiter analysiert, auch da uns das notwendige Domänenwissen nicht zur Verfügung stand.



(a) TF-IDF
(b) TF-IDF mit Kookkurrenz-Heuristik
(c) TF-IDF mit NSE-Differenz-Heuristik
(d) TF-IDF mit kombinierten Heuristiken
(e) Verwendungshäufigkeit in ähnlichen Testskripten
(f) Kombination TF-IDF und Verwendungshäufigkeit in ähnlichen Testskripten

Abbildung 6.2: Präzision und Ausbeute der Teststichprobe von Projektstand 1



(a) TF-IDF

(b) TF-IDF mit Kookkurrenz-Heuristik

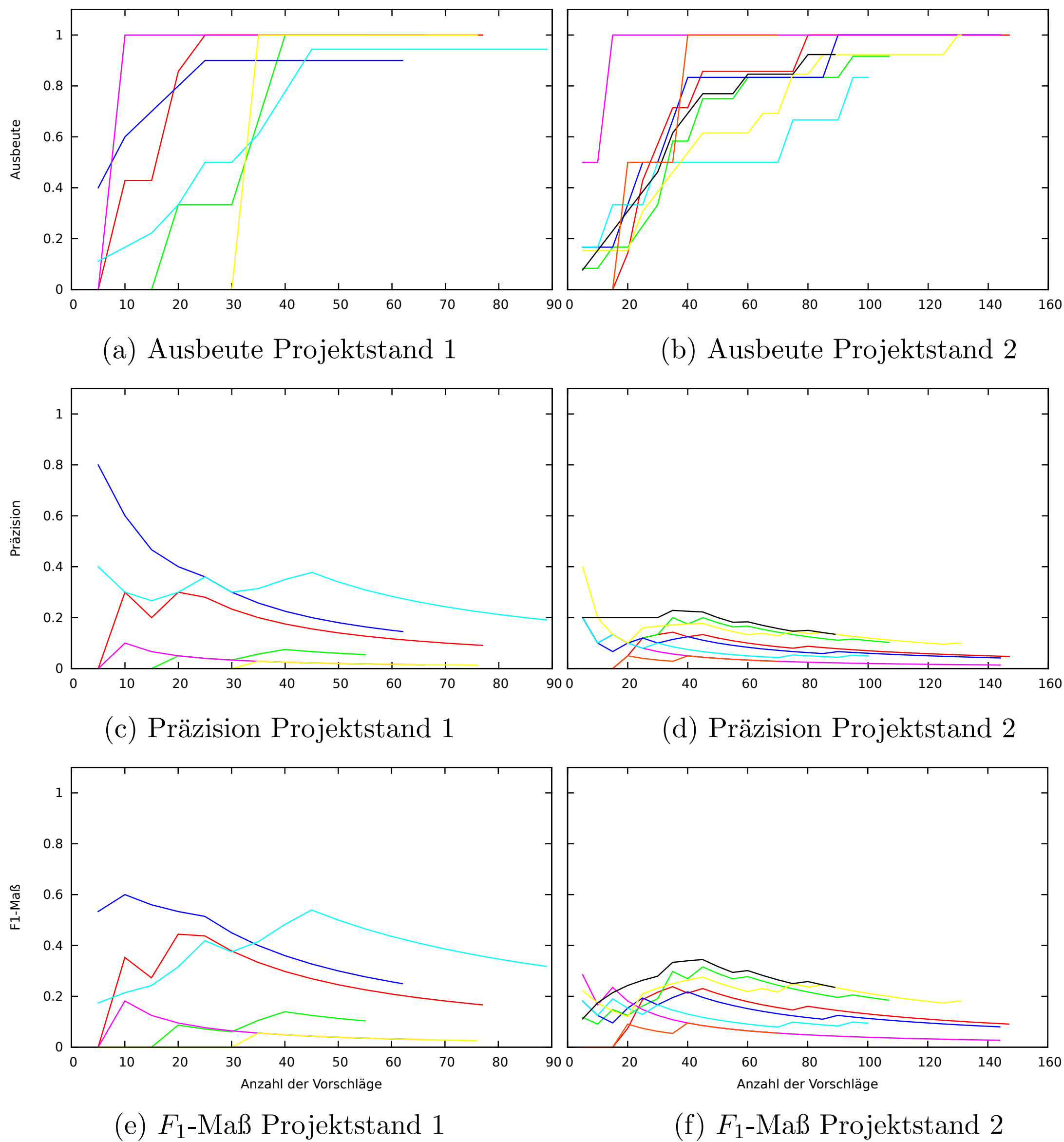
(c) TF-IDF mit NSE-Differenz-Heuristik

(d) TF-IDF mit kombinierten Heuristiken

(e) Verwendungshäufigkeit in ähnlichen Testskripten

(f) Kombination TF-IDF und Verwendungshäufigkeit in ähnlichen Testskripten

Abbildung 6.3: Präzision und Ausbeute der Teststichprobe von Projektstand 2



(a) Ausbeute Projektstand 1

(b) Ausbeute Projektstand 2

(c) Präzision Projektstand 1

(d) Präzision Projektstand 2

(e) F_1-Maß Projektstand 1

(f) F_1-Maß Projektstand 2

Abbildung 6.4: Einzel-Ergebnisse Kombination Verwendungshäufigkeit und TF-IDF mit allen Heuristiken: Ausbeute, Präzision und F_1-Maß

6.3 Evaluation der Gewinnung relevanter API–Komponenten

Wie in Abschnitt 6.1 beschrieben, konnten für den 1. Projektstand kaum Aufruf- und Objekterstellungsverknüpfungen gewonnen werden. Aus diesem Grund betrachten wir die Gewinnung von für die Testimplementierung relevanten Programmkomponenten nur bezüglich des 2. Projektstandes.

Auch im 2. Projektstand entspricht das Quelltextmodell in Bezug auf Aufruf- und Objekterstellungsbeziehungen nicht dem tatsächlichen Quelltext. Grundsätzlich leidet der Ansatz, den wir mit NRefactory gewählt haben, an typischen Problemen statischer Quelltextanalyse. Wenn wir die Klasse eines Objektes nicht auflösen können, werden Aufrufe von Methoden des Objektes nicht aufgezeichnet. Ein häufig vorkommendes Beispiel ist der Aufruf einer Methode, die ein Objekt einer unbestimmten Klasse aus einem Zwischenspeicher zurückliefert. Der erste Aufruf wird aufgelöst. Da das zurückgelieferte Objekt jedoch nicht aufgelöst werden kann, können auch auf dem Objekt ausgeführte Operationen nicht verfolgt werden. Weiterhin enthält das Quelltextmodell keine Aufrufe in Bibliotheken, da wir nur Individuen erstellen und verknüpfen, die im vorliegenden Quelltext vorhanden sind. Daher sind nicht alle Aufrufe, die im Quelltext existieren, auch in dem Quelltextmodell vorhanden.

Das Verfahren zur Bestimmung der Ausbeute und Präzision ist dem in Abschnitt 6.2 angelehnt. Wir verwenden dazu dieselben Ontologien. Für jedes Testskript der Teststichprobe bestimmen wir die in der dazugehörigen Ontologie nicht verknüpften Testschritte. Das sind ebensolche, die einzig von dem Testskript referenziert werden.

Wir können die Implementierung des Testschrittes mit dem gleichen Verfahren auflösen, dass wir in Abschnitt 5.4.2 zur Verknüpfung eines Testschrittes mit seiner Implementierung nutzen. Die aus dieser Implementierung gewonnene Menge von Aufrufen und Objekterstellungen gleichen wir mit derjenigen ab, die wir durch das in Abschnitt 4.2.4 beschriebene Verfahren gewannen.

In genanntem Abschnitt wurde beschrieben, dass wir die Relation `isInvoking` intransitiv oder transitiv auffassen können. Durch die transitive Aufrufrelation ist zu erwarten, dass wir weitaus größere Ergebnismengen erhalten. Dies gilt sowohl für die Menge der tatsächlichen Aufrufe als auch für Vorgeschlagene.

Der Graph in Abbildung 6.5 zeigt Präzision und Ausbeute für die Vorschläge bei intransitiver Relation für Aufrufe. Maximal werden hier 54,8% Ausbeute erreicht mit einer Präzision von 10,3% und 330 Ergebnissen, also 11 Ergebnissen pro Testschritt. Die Präzision liegt bei maximal 39,3% bei einer Ausbeute von 35,5% und 56 Ergebnissen (ca. 2 Ergebnisse je Testschritt).

In Abbildung 6.6 sind die Werte für Präzision und Ausbeute bei Verwendung der transitiven Relation aufgetragen. Maximal erreicht das Verfahren eine Ausbeute von 66,4%, allerdings beträgt die Präzision dann nur noch 5,3% und die Anzahl der Ergebnisse liegt bei 2846, was 95 Ergebnissen pro Testschritt entspricht. Die maximale Präzision beträgt 57,9% bei einer Ausbeute von 14,6% bei 57 Ergebnissen (ca. 2 pro Testschritt). Durch die transitive Relation erhalten wir sehr viele Ergebnisse, die Gesamtanzahl beträgt 2951. Für jeden der 30 Testschritte fallen somit durchschnittlich 98,4 Ergebnisse an. Fraglich ist vorwiegend, inwieweit die hier als relevant eingeschätzten Aufrufe tatsächlich relevant sind. Die Relevanz der Vorschläge

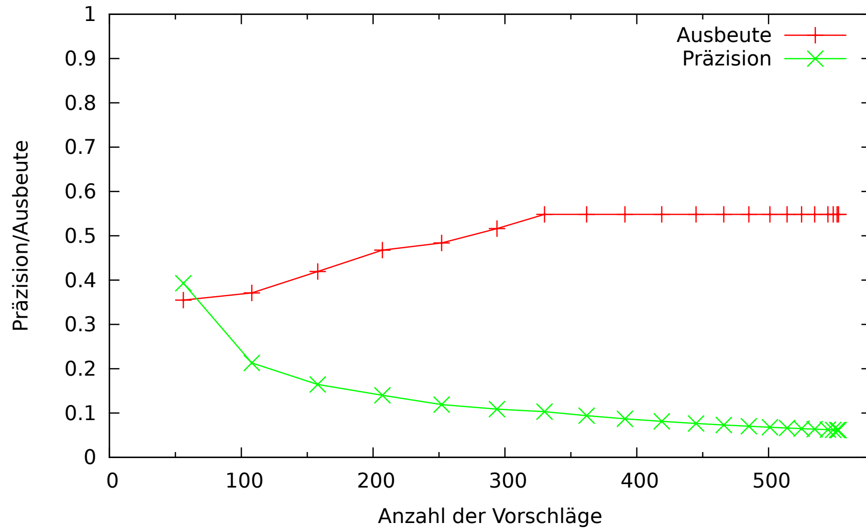


Abbildung 6.5: Präzision und Ausbeute für vorgeschlagene Methodenaufrufe und Objekterstellungen

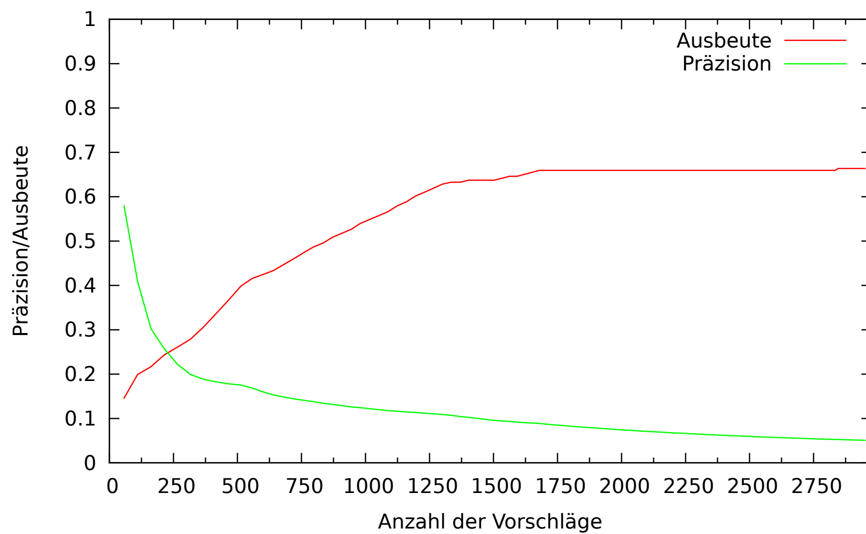


Abbildung 6.6: Präzision und Ausbeute für vorgeschlagene Methodenaufrufe und Objekterstellungen mit transitiver Aufrufrelation

bei intransitiver Relation beruht auf der tatsächlichen Verwendung in Testschritten. Diese explizite Verwendung in der Testimplementierung ist bei Aufrufen, die über die transitive Relation gefunden werden, nicht vorhanden.

6.4 Laufzeitmessungen zu F-TRec

Für die folgenden Messungen kam ein Rechner mit auf 2,2 GHz getaktetem Zweikern-Prozessor (Intel core i3-2330M) und 8GB Ram zum Einsatz.

Wir nahmen keine Laufzeitmessungen bei der Erstellung des Softwaremodelles für das Projekt des kooperierenden Unternehmens vor. Daher verwenden wir hier die Laufzeiten bei der Erstellung eines Softwaremodelles für ein Projekt vergleichbarer Größenordnung.

Ähnliche Ausmaße weist das Addin-Quelltextverzeichnis der Entwicklungsumgebung *Monodevelop* in Version 2.8.5 mit 216970 Zeilen Quelltext auf (siehe [6]). Die Befüllung der Ontologie mit diesen Quelltexten dauerte durchschnittlich 8:10 Minuten bei 5 Testläufen. Der Speichernutzung bei der Befüllung betrug 471944KB für die Ontologiebefüllungskomponente und 623148KB für Quelltextextraktionskomponente. Zusammen ergibt sich somit eine Speichernutzung von 1095092KB, also ca. 1GB.

Die entstandene OWL/XML-Datei umfasst 66066KB. Im Vergleich dazu ist die OWL/XML-Datei des ersten Projekstandes 69580KB groß und die des zweiten Projektstandes 114260KB. Ein Grund für die weitaus größere Ergebnisdatei des zweiten Projektstandes ist vermutlich die stärkere Verknüpfung der Komponenten im Projekt, da die Addins des Monodevelop-Projektes geringeren Zusammenhang haben, während wir für das Evaluationsprojekt den gesamten Quelltext betrachteten. Der Größenunterschied zwischen den OWL/XML-Dateien des ersten und zweiten Projektstandes ist mit der Korrektur der Methode zur Extraktion der Aufruf- und Objekterstellungsverknüpfungen zu erklären.

Das Hinzufügen eines Paares aus User Story und Testskript dauert durchschnittlich 9,5 Sekunden für den zweiten Projektstand (63 Paare). Darin ist die Zeit zur Initialisierung der NLP-Werkzeuge und der Ontologie-Komponenten für Befüllung und Abfrage nicht einberechnet.

Die Befüllungsschritte könnten automatisch durchgeführt werden, nachdem ein Entwickler Änderungen in ein Versionsverwaltungssystem eingebracht hat. Für den Einsatz durch jeden Entwickler bzw. Tester sind die Laufzeiten zu lang.

Im zweiten Projektstand dauerte die Extraktion der NSEs einer Story, Abfrage von Testschritten und deren Sortierung über alle Verfahren durchschnittlich 10,5 Sekunden (für die 9 User Stories der Teststichprobe). Die Extraktion der NSEs einer User Story dauert durchschnittlich 6,5 Sekunden. Die Abfrage von Testschritten, die mit diesen NSEs in Verbindung stehen, benötigt durchschnittlich 2,0 Sekunden. Nur das auf Verwendungshäufigkeit basierende Verfahren nutzt weitere Abfragen. Dementsprechend entfällt der Großteil der Restzeit mit 1,9 Sekunden auf dieses Verfahren.

Die durchschnittliche Dauer einer Abfrage ist für praktischen interaktiven Einsatz zu lang. Hier müssten Optimierungen eingesetzt werden, um dem Benutzer die Vorschläge schnell liefern zu können. Doch ist die Laufzeit der Abfrage sicherlich geringer als die, die ohne den Einsatz von F-TRec mit der Recherche in existierenden Testskripten verbracht werden würde.

Die Abfrage von API-Komponenten dauert im zweiten Projektstand durchschnittlich 6,3 Sekunden. Den Großteil dieser Zeit nimmt dabei mit 5,6 Sekunden die Abfrage von von Testschrittimplementierungen referenzierten Komponenten ein. Da Testschritte im Vergleich zu User Stories kurz sind, dauert die Extraktion von NSEs durchschnittlich nur 0,4 Sekunden. Eine weitere Abfrage der Ontologie wird bei der Abfrage ähnlicher Testschritte benötigt; die durchschnittliche Dauer dafür beträgt 0,3 Sekunden

Auch hier ist die Dauer einer Abfrage für den interaktiven Einsatz zu optimieren. Wiederum kann jedoch ein Teil der Quelltextrecherche durch das Vorschlagssystem überflüssig werden, weshalb die Laufzeit nicht zu hoch für den Einsatz ist.

6.5 Zusammenfassung

Zunächst haben wir in diesem Kapitel die Evaluationsdaten beleuchtet, die aus Quelltext, User Stories und Testskripten von zwei verschiedenen Projektständen bestehen.

Zur Evaluation der Gewinnung relevanter Testschritte haben wir zu jedem erläuterten Verfahren die Präzision und Ausbeute bezüglich jeweils einer Teststichprobe je Projektstand ermittelt. Die Teststichprobe bestand dabei aus Paaren von User Stories und Testskripten der letzten zwei Sprints, wodurch wir im ersten Projektstand 7 und im zweiten Projektstand 9 Testpaare betrachteten. Eine Ontologie, die alle User Story/Testskript-Paare außer der zur Evaluation gewählten enthielt, fragten wir nach dazu relevanten Testschritten mit jedem Verfahren und Verfahrenskombinationen ab und glichen die Ergebnisse zur Gewinnung der Ausbeute und Präzision mit dem tatsächlichen Testskript ab. Die höchste Präzision ergab sich für das Verfahren, dass aufgrund der Verwendungshäufigkeit von Testschritten, die in ähnlichen Testskripten vorkommen, diese Testschritte sortiert. Die höchste Ausbeute ergab sich jedoch für die Kombination dieses Verfahrens mit einem TF-IDF-basierten Verfahren.

In ähnlicher Weise evaluierten wir auch die Gewinnung von Methoden und Klassen, die für die Implementierung von Testschritten relevant sind. Dazu fragen wir die Ontologie für die nicht verknüpften Testschritte ab und erhielten Vorschläge, die wir mit den in der Ontologie verknüpften Methoden und Klassen abglichen, für die Aufruf- und Objekterstellungsverknüpfungen aus der Implementierung des Testschrittes existieren. Hier weist ein auf transitiver Aufruf-Relation basierendes Verfahren zwar eine höhere maximale Präzision und Ausbeute auf, allerdings ist die Aussagekraft der Relevanzeigenschaft zu gering und die tatsächliche Anzahl an Ergebnissen zu hoch für eine Empfehlung der Ausnutzung der Transitivität. Fassen wir die Relation intransitiv auf, liegt vor allem die maximale Ausbeute auf zu geringem Niveau. Dennoch ist der Ansatz vielversprechend, da oft bereits unter den ersten zwei höchstbewerteten Komponenten Treffer zu finden sind.

Zuletzt führten wir Laufzeitmessungen für mehrere Abläufe durch, die dem Aufbau der Ontologie oder ihrer Nutzung dienen.

7. Zusammenfassung und Ausblick

Wir haben aus den Artefakten, die in einem realen Projekt entstanden sind, eine Wissenbasis aufgebaut, die Unterstützung für mehrere Abläufe im Entwicklungsprozess ermöglichen soll. Die in dieser Arbeit betrachteten Hilfestellungen betreffen einerseits die Arbeit mit Anforderungsdokumenten, andererseits die Überprüfung der Anforderungen durch automatisierte Tests. Grundlage für gute Anforderungsdokumente ist eine konsistente und aus der Umwelt des beabsichtigten Einsatzes stammende Terminologie. Ein Glossar, das wir aus natürlichsprachlichen Anforderungsdokumenten aufbauen, soll die Konsistenz erhöhen und Bewusstsein für die verwendete Terminologie schaffen. Ein Assistenzsystem, das einerseits Testern wiederverwendbare Testschritte vorschlägt und Entwicklern Hilfestellung bei der Implementierung neuer Testschritte gibt, soll die Entwicklung automatisierter Tests vereinfachen.

Wie die Evaluation gezeigt hat, haben wir über die in dieser Arbeit aufgebaute Wissensbasis eine Grundlage geschaffen, Testern und Entwicklern brauchbare Ergebnisse zu Assistenzzwecken zu liefern. Zwar liegt die Präzision der Ergebnisse auf geringerem Niveau, dennoch können zumeist unter den ersten 10-20 Vorschlägen relevante Testschritte gefunden werden. Dies kann für Tester und Entwickler bereits eine deutliche Zeitersparnis bedeuten.

Inwieweit das Glossar die Erstellung von User Stories unterstützen kann, konnte in dieser Arbeit nicht ermittelt werden. Dazu ist die Beobachtung eines Projektes notwendig, in dem das Glossar verwendet wird.

Als Ausgangspunkt für weitere Forschungen bietet das Konzept des Ansatzes durch die Nutzung einer Ontologie eine leicht erweiterbare Plattform, mit der auch andere Probleme in Softwareprojekten angegangen werden können.

Weitere Verfahren sollten entworfen und erprobt werden, mit deren Hilfe weitere Leistungsverbesserung möglich ist. Ein Ansatz, der genauer beleuchtet werden kann, umfasst die Nutzung im Glossar vorhandener Synonymie-Beziehungen zur Erweiterung von Abfragen. Der hauptsächliche Hinderungsgrund für die Evaluation mit Synonymie-Beziehungen ist die spezialisierte Sprache in einem solchen Projekt. Dadurch kann ein außerhalb des Projektes Stehender diese Beziehungen nicht korrekt bestimmen. Daher ist eine Evaluation nur möglich, wenn in einem Projekt aktiv

mit dem Glossar gearbeitet wird. Zusätzliche Verknüpfungen zwischen Glossareinträgen – beispielsweise gemeinsame Nennung in einem Satz – könnten Ansätze für eine Verbesserung der Abfrageleistung bieten.

Der Quelltext ist in unserem Ansatz nur über Testskripte bzw. deren Testschritte mit dem Glossar verbunden. Wenn wir Glossareinträge jedoch direkt mit entsprechenden Teilen des Quelltextes verbinden, ermöglicht dies Unterstützung der Entwickler bei vielfältigen Aufgaben. Beispielsweise könnten von Glossareinträgen ausgehend diesbezügliche Implementierungsteile zugänglich gemacht werden. Zusätzlich kann die Einbindung von Informationen einer Versionsverwaltung die Entstehung des Quelltextes mitberücksichtigen. Ein darauf basierender iterativer Aufbau des Quelltextmodelles hätte gegenüber des in dieser Arbeit verwendeten Ansatzes den Vorteil, dass nicht für jede Änderung ein neues Softwaremodell erstellt werden muss.

Wir betrachteten speziell User Stories in dieser Arbeit. Durch den Grundsatz der Verhandelbarkeit und damit Veränderlichkeit sind sie für agile Prozesse gut geeignet. Vor allem Use Cases weisen jedoch deutliche Parallelen zu User Stories auf. Ein jüngerer Entwurf zu Use Cases – bezeichnet als Use-Case 2.0 – berücksichtigt Grundsätze, die bislang von User Stories abgedeckt werden (Jackobson et al. [45]):

- Testbarkeit durch Angabe von Testfällen
- Fokus auf den Wert für Kunden
- „Scheibenweiser“ Aufbau des Systems
- inkrementelle Lieferung

Auf solchen Anforderungsdokumenten lässt sich der beschriebene Ansatz voraussichtlich ebensogut anwenden.

Für Use Cases existieren in der Literatur Anhaltspunkte für Qualitätsindikatoren (Beispiele hierfür siehe [24], [18]). Bezüglich solcher quantifizierbarer Indikatoren existieren für User Stories zum Zeitpunkt dieser Arbeit keine Anhaltspunkte. Entwurf und experimentelle Verifizierung der in Abschnitt 4.2.2.2 erläuterten Ansätze glossarbasierter Qualitätsmetriken für User Stories könnte für zukünftige Arbeiten interessant sein. Weiterhin können die in Anhang B genannten Kriterien auf maschinelle Auswertbarkeit hin geprüft werden.

A. Struktur der Ontologie

A.1 Klassenhierarchie

Die Klassenhierarchie ist in Abbildung A.1 dargestellt.

A.2 Dateneigenschaften (data properties)

Tabelle A.1 zeigt die Dateneigenschaften mit Basiseigenschaften.

A.3 Objekteigenschaften (object properties)

Tabelle A.2 zeigt die Objekteigenschaften mit inversen Eigenschaften und Basiseigenschaften.

Dateneigenschaft	Basiseigenschaft
hasAttributeValue	topDataProperty
hasCommentText	topDataProperty
hasDescription	topDataProperty
hasName	topDataProperty
hasPosition	topDataProperty
hasStepText	topDataProperty
hasStepType	topDataProperty
hasStoryID	topDataProperty
topDataProperty	

Tabelle A.1: Dateneigenschaften mit Basiseigenschaft

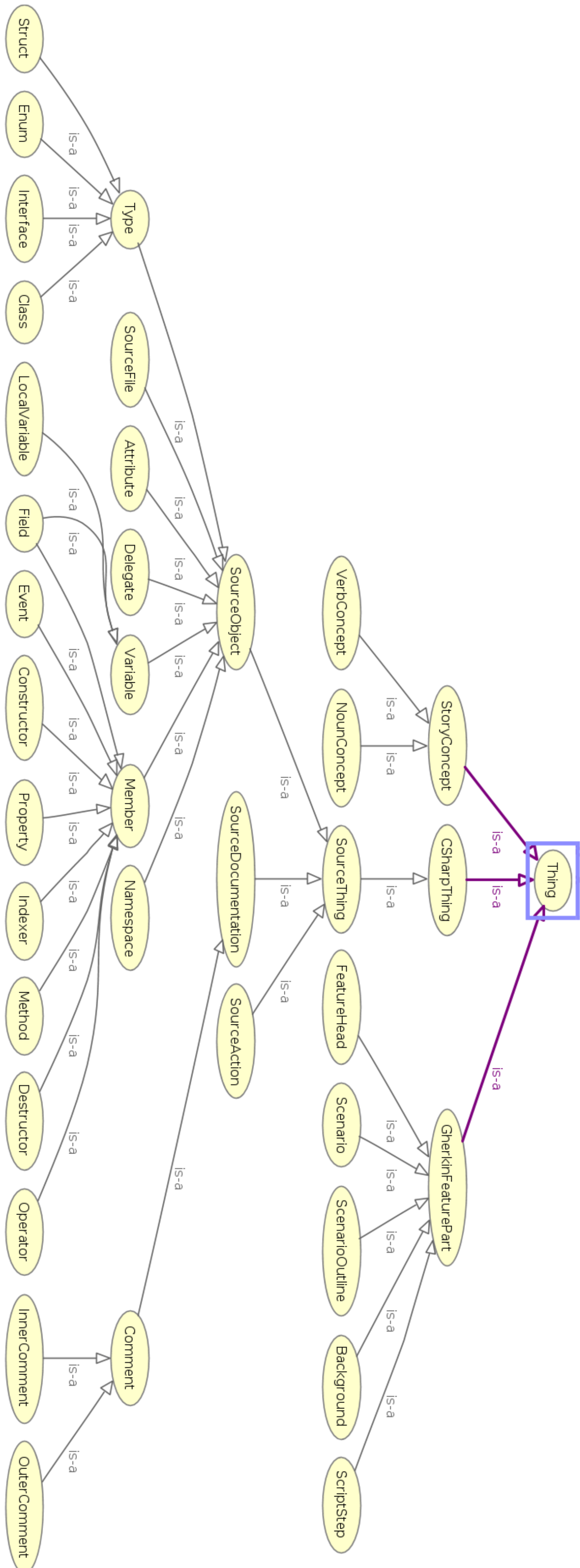


Abbildung A.1: Die Klassenhierarchie der Ontologie

Objekteigenschaft	Inverse Eigenschaft	Basiseigenschaft
hasAttribute	isAttributeOf	topObjectProperty
hasContent	isInFile	topObjectProperty
hasDocumentationFor	isDocumentedBy	topObjectProperty
hasHyperonym	hasHyponym	isSemanticallyRelatedTo
hasHyponym	hasHyperonym	isSemanticallyRelatedTo
hasInternalType	isInternalTypeOf	topObjectProperty
hasMember	isMemberOf	topObjectProperty
hasScenario	isInFeature	topObjectProperty
hasStep	isInScenario	topObjectProperty
hasStepReference	isReferringToConcept	topObjectProperty
hasSubType	isSubTypeOf	topObjectProperty
hasSynonym		isSemanticallyRelatedTo
hasTestImplementation	isTestImplementationOf	topObjectProperty
isAttributeOf	hasAttribute	topObjectProperty
isCreatedBy	isCreating	topObjectProperty
isCreating	isCreatedBy	topObjectProperty
isDocumentedBy	hasDocumentationFor	topObjectProperty
isInFeature	hasScenario	topObjectProperty
isInFile	hasContent	topObjectProperty
isInNamespace		topObjectProperty
isInScenario	hasStep	topObjectProperty
isInstanceOfOrReturns	isUsedBy	topObjectProperty
isInternalTypeOf	hasInternalType	topObjectProperty
isInvokedBy	isInvoking	topObjectProperty
isInvoking	isInvokedBy	topObjectProperty
isLinkedToFeature	isReferencedByConcept	topObjectProperty
isMemberOf	hasMember	topObjectProperty
isReferencedByConcept	isLinkedToFeature	topObjectProperty
isReferringToConcept	hasStepReference	topObjectProperty
isSemanticallyRelatedTo		topObjectProperty
isSubTypeOf	hasSubType	topObjectProperty
isTestImplementationOf	hasTestImplementation	topObjectProperty
isUsedBy	isInstanceOfOrReturns	topObjectProperty
topObjectProperty		

Tabelle A.2: Objekteigenschaften mit inverser Eigenschaft und Basiseigenschaft

B. Leitfaden für User Stories

B.1 Prinzipien

B.1.1 „Story Card“

- Platzbeschränkung
  - wichtige Details aufschreiben
  - unwichtiges wegstreichen
  - zusammengehörige Informationen zusammenführen
- Struktur
  - Vorderseite: Kurzbeschreibung und Notizen
  - Rückseite: Storytests (Kriterien)

B.1.2 „Vereinbarung zum Gespräch“

- User Story soll/kann die Kommunikation nicht ersetzen
- Im Gespräch festgelegte Details kondensiert aufschreiben

B.1.3 „INVEST“

Empfehlungen für User Stories nach Wake [74]:

- *Independant:* möglichst hohe Unabhängigkeit von anderen User Stories
- *Negotiable:* Verhandelbarkeit/Änderbarkeit bis zur Berücksichtigung für eine Iteration
- *Valuable:* Wert für den Kunden steht im Vordergrund
- *Estimable:* Überschaubarkeit und Schätzbarkeit des Aufwands
- *Small:* Geringer Aufwand (z.B. 0,5 – 10 Tage)
- *Testable:* Überprüfbarkeit

B.2 Unterstützung maschineller Auswertbarkeit

- Strukturkonvention:
  - Kurzbeschreibung
  - Notizen
  - Kriterien („Story Tests“)
- Formulierung: kurze, aber vollständige Sätze
- Rechtschreibung: z.B. über automatische Rechtschreibprüfung
- Zeichensatzbeschränkung
- Unterlassen von Kopieren/Einfügen

B.3 Test-Kriterien

1. Verwende konsistente Terminologie!
2. Strukturierung nach der Konvention Kurzbeschreibung, Notizen, Kriterien!
3. Email-Texte, Chat-Logs oder Kopien von Onlinetexten weglassen!
4. Keine unvollständigen Sätze!
5. Rechtschreibfehler korrigieren!
6. Keine Sonderzeichen wie Ellipsen („…“) oder Ligaturen („ﬀ“ statt „ff“)!

C. Verwendete Werkzeuge und Bibliotheken

- Python, Version 2.7.2+ (Ubuntu 11.10 Repository) [9]
- IronPython, Version 2.7.1 [3]
- Jython, Version 2.5.2 [4]
- PyroLite, Version 1.4 (SVN-Revision 684) [29]
- Pyro Version 4.12 [30]
- Mono 2.10.5-1 (Ubuntu 11.10 Repository)
- OpenJDK (Java) Version 1.6.0_23
- SPARQLDL-API Version 1.0.0 [11]
- Pellet Version 2.3.0
- Protégé Version 4.2 [8]
- Nrefactory Git-Revision 54b3b [7]
- Flask Version 0.7.2-1 (Ubuntu 11.10 Repository)
- Stanford Parser Version 2.0.0
- Stanford NER Tagger Version 1.2.3
- NLTK Version 2.0b9 [19]
- OWL-API Version 3.2.5.1912

D. Detaillierte Auswertung der Relevanz-Sortierung

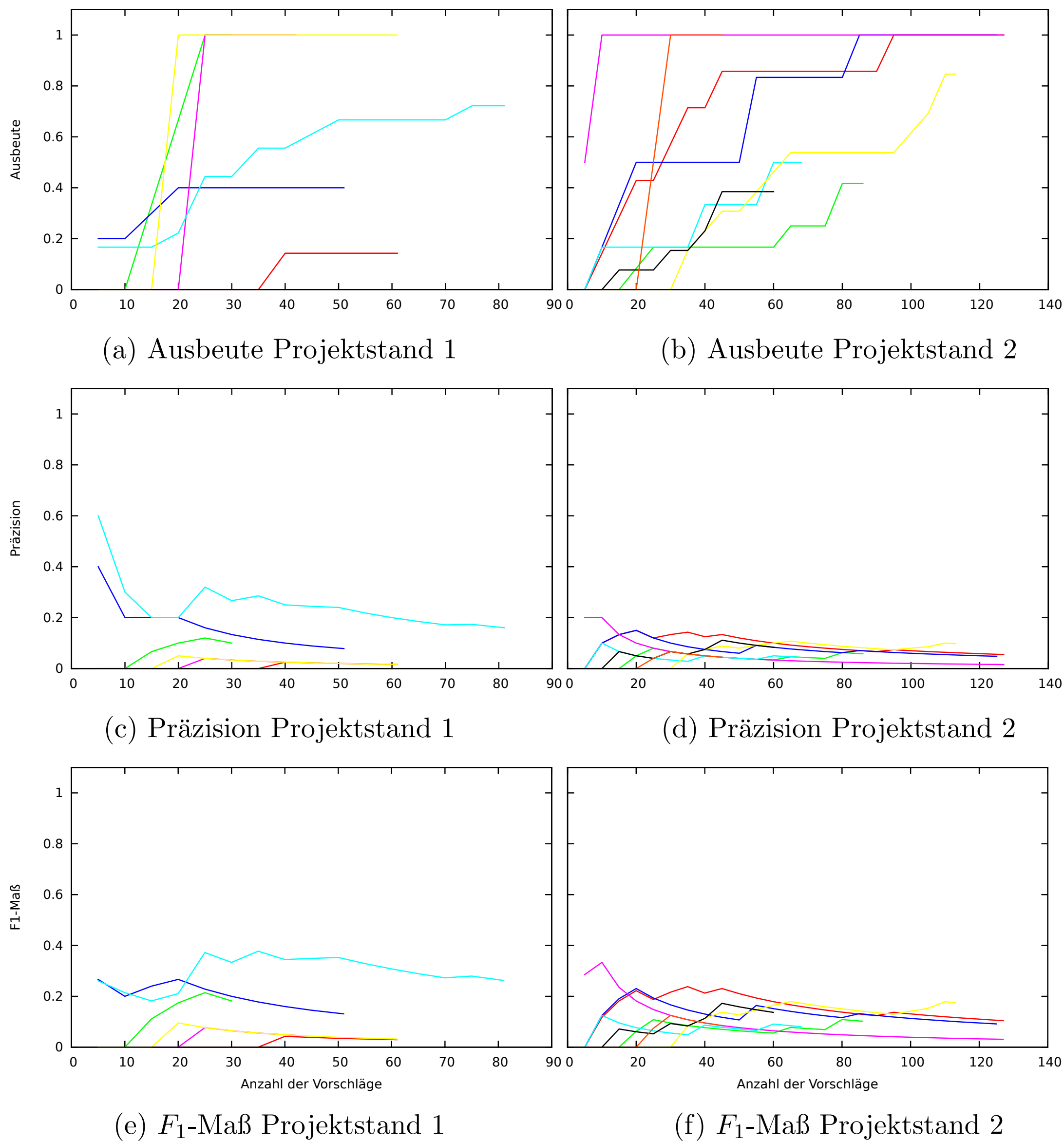
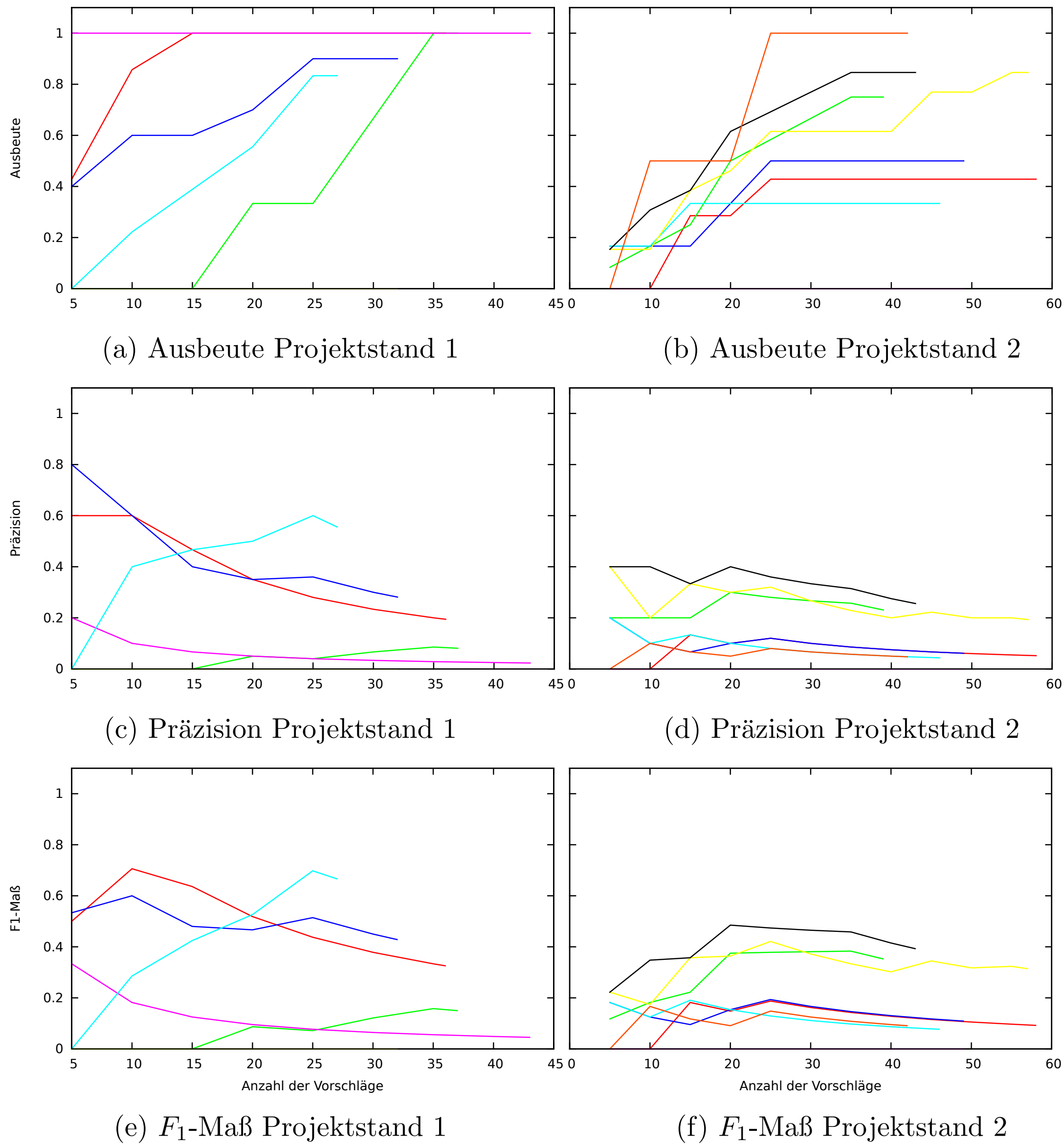


Abbildung D.1: Einzel-Ergebnisse TF-IDF mit allen Heuristiken: Ausbeute, Präzision und F_1-Maß



(a) Ausbeute Projektstand 1

(b) Ausbeute Projektstand 2

(c) Präzision Projektstand 1

(d) Präzision Projektstand 2

(e) F_1-Maß Projektstand 1

(f) F_1-Maß Projektstand 2

Abbildung D.2: Einzel-Ergebnisse Verwendungshäufigkeit: Ausbeute, Präzision und F_1-Maß

Literaturverzeichnis

[1] *Cucumber*. online. http://www.cukes.info/. – zuletzt abgerufen am 28.05.2012

[2] *Gherkin*. online. https://github.com/cucumber/cucumber/wiki/Gherkin/2060db. – zuletzt abgerufen am 29.05.2012

[3] *IronPython - the Python programming language for the .NET Framework*. online. http://ironpython.net/. – zuletzt abgerufen am 27.05.2012

[4] *Jython: Python for the Java Platform*. online. http://www.jython.org/. – zuletzt abgerufen am 27.05.2012

[5] *Lettuce documentation*. online. http://packages.python.org/lettuce/. – zuletzt abgerufen am 28.05.2012

[6] *Monodevelop Website*. online. http://monodevelop.com/. – zuletzt abgerufen am 24.05.2012

[7] *NRefactory 5*. online. https://github.com/icsharpcode/NRefactory/. – zuletzt abgerufen am 13.05.2012

[8] *protégé*. online. http://protege.stanford.edu/. – zuletzt abgerufen am 28.05.2012

[9] *Python Programming Language – Official Website*. online. http://python.org/. – zuletzt abgerufen am 28.05.2012

[10] *Ruby*. online. http://www.ruby-lang.org/de/. – zuletzt abgerufen am 28.05.2012

[11] *SPARQL-DL API*. online. http://www.derivo.de/ressourcen/sparql-dl-api.html. – zuletzt abgerufen am 07.05.2012

[12] *SpecFlow - pragmatic BDD for .NET*. online. http://specflow.org/home.aspx. – zuletzt abgerufen am 28.05.2012

[13] *C# Language Specification*. http://www.ecma-international.org/publications/standards/Ecma-334.htm. Version: Juni 2006

[14] Antoniol, G. ; Canfora, G. ; Casazza, G. ; De Lucia, A. ; Merlo, E.: Recovering traceability links between code and documentation. In: *Software Engineering, IEEE Transactions on* 28 (2002), Oktober, Nr. 10, S. 970 – 983. http://dx.doi.org/10.1109/TSE.2002.1041053. – DOI 10.1109/TSE.2002.1041053. – ISSN 0098–5589

[15] Antoniou, Grigoris ; Franconi, Enrico ; Harmelen, Frank V.: Introduction to Semantic Web Ontology Languages. In: *Reasoning Web, Proceedings of the Summer School, Malta, 2005. Number 3564 in Lecture Notes in Computer Science*, Springer, 2005

[16] Baeza-Yates, Berthier de Araújo N. Ricardo ; Ribeiro R. Ricardo ; Ribeiro: *Modern information retrieval.* 1. print. Harlow, England [u.a.] : Addison-Wesley Longman, 1999 (ACM Press books). – ISBN 0–201–39829–X

[17] Beck, Kent: *Test-driven development : by example.* Boston, Mass. : Addison-Wesley, 2003 (The Addison Wesley Signature Series). – ISBN 0–321–14653–0

[18] Bernárdez, Beatriz ; Durán, Amador ; Genero, Marcela: Empirical Evaluation and Review of a Metrics-Based Approach for Use Case Verification. In: *Journal of Research and Practice in Information Technology* 36 (2004), Nr. 4, S. 247–258

[19] Bird, Steven ; Klein, Ewan ; Loper, Edward: *Natural language processing with Python : [analyzing text with the natural language toolkit].* 1. ed. Beijing : OReilly, 2009 http://www.nltk.org/book. – ISBN 978–0–596–51649–9 ; 0–596–51649–5

[20] Breitman, Karin K. ; Prado Leite, Julio Cesar S.: Managing User Stories. In: Eberlein, Armin (Hrsg.) ; Prado Leite, Julio Cesar S. (Hrsg.): *Proceedings of the International Workshop on Time Constrained Requirements Engineering.* Essen, Germany, September 2002

[21] Calero, Coral ; Ruiz, Francisco ; Piattini, Mario[Hrsg.] ; Calero, Coral (Hrsg.) ; Ruiz, Francisco (Hrsg.) ; Piattini, Mario[Hrsg.] (Hrsg.): *Ontologies for software engineering and software technology.* Berlin : Springer, 2006. – ISBN 3–540–34517–5 ; 978–3–540–34517–6

[22] Chelimsky, David: *The RSpec book : behaviour-driven development with RSpec, Cucumber, and Friends.* 1. Raleigh, NC : The Pragmatic Bookshelf, 2010. – ISBN 978–1–93435–637–1 ; 1–93435–637–9

[23] Chinchor, Nancy: MUC-4 Evaluation Metrics. In: *Proceedings of the Fourth Message Understanding Conference*, 1992, S. 22–29

[24] Ciemniewska, Alicja ; Jurkiewicz, Jakub ; Olek, Lukasz ; Nawrocki, Jerzy R.: Supporting Use-Case Reviews. In: Abramowicz, Witold (Hrsg.): *BIS* Bd. 4439, Springer, 2007 (Lecture Notes in Computer Science). – ISBN 978–3–540–72034–8, S. 424–437

[25] Cohn, Mike: *User Stories Applied: For Agile Software Development.* Redwood City, CA, USA : Addison Wesley Longman Publishing Co., Inc., 2004. – ISBN 0321205685

[26] Cohn, Mike: *Succeeding with Agile : Software Development Using Scrum.* Upper Saddle River, EUA : Addison-Wesley, 2010

[27] Coleman, Meri ; Liau, T. L.: A computer readability formula designed for machine scoring. In: *Journal of Applied Psychology* 60(2) (1975), Apr., S. 283–284. http://dx.doi.org/10.1037/h0076540. – DOI 10.1037/h0076540

[28] Cycorp Inc.: *http://www.opencyc.org/*. online, 2012. – zuletzt abgerufen am 16.05.2012

[29] de Jong, Irmen: *Pyrolite - client library for Java and .NET*. online. http://packages.python.org/Pyro4/pyrolite.html. – zuletzt abgerufen am 27.05.2012

[30] de Jong, Irmen: *Python Remote Objects 4.x*. online. http://packages.python.org/Pyro4/. – zuletzt abgerufen am 24.05.2012

[31] Devanbu, Prem ; Brachman, Ron ; Selfridge, Peter G.: LaSSIE: a knowledge-based software information system. In: *Commun. ACM* 34 (1991), Mai, S. 34–49. http://dx.doi.org/10.1145/103167.103172. – DOI 10.1145/103167.103172. – ISSN 0001–0782

[32] Euzenat, Jérôme ; Shvaiko, Pavel: *Ontology matching*. 1. Springer-Verlag, 2007. – ISBN 3–540–49611–4

[33] Fantechi, Alessandro ; Gnesi, Stefania ; Lami, Giuseppe ; Maccari, Alessandro: Applications of linguistic techniques for use case analysis. In: *Requir. Eng* 8 (2003), Nr. 3, S. 161–170. http://dx.doi.org/10.1007/s00766-003-0174-0. – DOI 10.1007/s00766–003–0174–0

[34] Filiba, Tomer: *Remote Python Call*. online. http://rpyc.sourceforge.net/. – zuletzt abgerufen am 06.05.2012

[35] Finin, Timothy W.: The Semantic Interpretation of Nominal Compounds. In: *Proceedings of the first national conference on artificial intelligence* Association for the Advancement of Artificial Intelligence, The AAAI Press, Menlo Park, California, 1980, 310-312

[36] Finkel, Jenny R. ; Grenager, Trond ; Manning, Christopher: Incorporating non-local information into information extraction systems by gibbs sampling. In: *In ACL*, 2005, S. 363–370

[37] Fowler, M.: *BusinessReadableDSL*. online. http://martinfowler.com/bliki/BusinessReadableDSL.html. Version: Dezember 2008. – zuletzt abgerufen am 18.05.2012

[38] Fowler, Martin: *ConversationalStories*. online. http://martinfowler.com/bliki/ConversationalStories.html. Version: Februar 2010. – zuletzt abgerufen am 18.05.2012

[39] Gethers, Malcom ; Oliveto, Rocco ; Poshyvanyk, Denys ; Lucia, Andrea D.: On integrating orthogonal information retrieval methods to improve traceability recovery. In: *ICSM*, 2011, 133-142

[40] Golesny, Daniel: *Diplomarbeit Ausbaufähige Client/Server-Architektur für Sekundarstufe II auf Basis objektorientierter Programmierung*. online. http://www.golesny.de/diplom/5_2_3.html. – zuletzt abgerufen am 31.05.2012

[41] GRECHENIG, Thomas ; BERNHART, Mario (Hrsg.) ; BREITENEDER, Roland (Hrsg.) ; KAPPEL, Karin (Hrsg.): *Softwaretechnik : Mit Fallbeispielen aus realen Entwicklungsprojekten*. München : Pearson Studium, 2010 (Informatik). – ISBN 978–3–86894–007–7

[42] GÓMEZ-PÉREZ, Asunción ; CORCHO, Oscar: Ontology languages for the semantic web. In: *IEEE Intelligent Systems* 17 (2002), S. 54–60

[43] HORRIDGE, Matthew: *A Practical Guide To Building OWL Ontologies Using Protégée 4 and CO-ODE Tools*. online. http://owl.cs.manchester.ac.uk/tutorials/protegeowltutorial/. – Edition 1.3

[44] HORRIDGE, Matthew ; BECHHOFER, Sean: The OWL API: A Java API for Working with OWL 2 Ontologies. In: *OWLED*, 2009

[45] JACOBSON, Ivar ; SPENCE, Ian ; BITTNER, Kurt: *Use-Case 2.0*. Ivar Jacobson International, 2011 http://www.ivarjacobson.com/Use_Case2.0_ebook/

[46] JURAFSKY, Dan ; MARTIN, James H.: *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. 2. ed., [Pearson International Edition]. Englewood Cliffs, NJ : Prentice Hall, Pearson Education International, 2009 (Prentice Hall series in artificial intelligence). – ISBN 0–13–504196–1 ; 978–0–13–504196–3

[47] KHAMIS, Ninus: *Assessing The Quality Factors Found in In-Line Documentation Written in Natural Language: The JavadocMiner*, Concordia University Montreal, Québec, Canada, NonPeerReviewed, April 2011. http://spectrum.library.concordia.ca/7300/1/Khamis_MASc_S2011.pdf

[48] KHAMIS, Ninus ; RILLING, Juergen ; WITTE, René: Generating an NLP Corpus from Java Source Code: The SSL Javadoc Doclet. In: *New Challenges for NLP Frameworks*. Valletta, Malta : ELRA, Mai 2010, S. 41–45

[49] KHAMIS, Ninus ; WITTE, René ; RILLING, Juergen: Automatic Quality Assessment of Source Code Comments: The JavadocMiner. In: *15th Int. Conference on Applications of Natural Language to Information Systems (NLDB 2010)*. Cardiff, UK : Springer, Juni 2010 (LNCS 6177), S. 68–79

[50] KLEIN, Dan ; MANNING, Christopher D.: Fast Exact Inference with a Factored Model for Natural Language Parsing. In: *In Advances in Neural Information Processing Systems 15 (NIPS)*, MIT Press, 2003, S. 3–10

[51] KOF, Leonid: Natural Language Processing: Mature Enough for Requirements Documents Analysis. In: *10th Intl. Conf. on Applications of Natural Language to Information Systems (NLDB)*, Springer–Verlag, 2005, 91–102

[52] KÖRNER, Sven J. ; BRUMM, Torben: Improving Natural Language Specifications with Ontologies SEKE 2009 Conference, Boston, 2009

[53] KÖRNER, Sven J. ; BRUMM, Torben: RESI - A Natural Language Specification Improver. In: *Proceedings of the IEEE ICSC 2009* IEEE, 2009

[54] Körner, Sven J. ; Brumm, Torben: Natural Language Specification Improvement with Ontologies. In: *International Journal of Semantic Computing (IJSC)* 03 (2010), Nr. 04, 445-470. http://dx.doi.org/10.1142/S1793351X09000872. – DOI 10.1142/S1793351X09000872

[55] Landhäusser, Mathias ; Genaid, Adrian: Connecting User Stories and Code for Test Development. In: *Proc. of the Third International Workshop on Recommendation Systems for Software Engineering (RSSE 2012)*, 2012

[56] Langer, Stefan: Zur Morphologie und Semantik von Nominalkomposita. In: *Tagungsband der 4. Konferenz zur Verarbeitung natürlicher Sprache*, 1998

[57] Lapshinova-Koltunski, Ekaterina ; Heid, Ulrich: Head or Non-head? Semi-automatic Procedures for Extracting and Classifying Subcategorisation Properties of Compounds. In: *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, 2008. – ISBN 2–9517408–4–0

[58] Lauer, Mark: *Corpus Statistics Meet the Noun Compound: Some Empirical Results*. http://acl.ldc.upenn.edu/P/P95/P95-1007.pdf. Version: 1995

[59] *Kapitel* Natural Language Processing. In: Liddy, Elizabeth D.: *Encyclopedia of Library and Information Science*. 2. Marcel Decker Inc., 2001

[60] Marcus, A. ; Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: *Software Engineering, 2003. Proceedings. 25th International Conference on*, 2003. – ISSN 0270–5257, S. 125 – 135

[61] Miller, George A.: WordNet: a lexical database for English. In: *Commun. ACM* 38 (1995), November, Nr. 11, 39–41. http://dx.doi.org/10.1145/219717.219748. – DOI 10.1145/219717.219748. – ISSN 0001–0782

[62] North, Dan: *Introducing BDD*. Published in Better Software magazine. http://dannorth.net/introducing-bdd/. Version: März 2006

[63] Park, Youngja ; Byrd, Roy J. ; Boguraev, Branimir: Automatic Glossary Extraction: Beyond Terminology Identification. In: *COLING*, 2002

[64] Porter, M.F.: An algorithm for suffix stripping, 2006, S. 211–218

[65] Qiu, Long ; Kan, Min yen ; Chua, Tat seng: A public reference implementation of the rap anaphora resolution algorithm. In: *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC 2004)*, 2004, S. 291–294

[66] Scheuer, Klaus-Martin: *Anwendung eines Metasuchansatzes für die Vorhersage von relevanten Codeartefakten*, Karlsruhe Institut für Technologie, Diplomarbeit, Juli 2011

[67] Sillitti, A. ; Ceschi, M. ; Russo, B. ; Succi, G.: Managing uncertainty in requirements: a survey in documentation-driven and agile companies. In: *Software Metrics, 2005. 11th IEEE International Symposium*, 2005. – ISSN 1530–1435, S. 10 pp. –17

[68] Sirin, Evren ; Parsia, Bijan: SPARQL-DL: SPARQL Query for OWL-DL. In: *In 3rd OWL Experiences and Directions Workshop (OWLED-2007)*, 2007

[69] Sirin, Evren ; Parsia, Bijan ; Grau, Bernardo C. ; Kalyanpur, Aditya ; Katz, Yarden ; Katz, Yarden: Pellet: A practical OWL-DL reasoner. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 5 (2007), Nr. 2, S. 51–53. http://dx.doi.org/10.1016/j.websem.2007.03.004. – DOI 10.1016/j.websem.2007.03.004. – ISSN 1570–8268

[70] Staab, Steffen ; Studer, Rudi ; Studer, Rudi (Hrsg.): *Handbook on Ontologies*. Berlin, Heidelberg : Springer-Verlag Berlin Heidelberg, 2009 (International Handbooks on Information Systems). http://dx.doi.org/10.1007/978-3-540-92673-3. – ISBN 978–3–540–92673–3

[71] Studer, Rudi ; Benjamins, V.Richard ; Fensel, Dieter: Knowledge engineering: Principles and methods. In: *Data & Knowledge Engineering* 25 (1998), Nr. 1–2, 161 - 197. http://dx.doi.org/10.1016/S0169-023X(97)00056-6. – DOI 10.1016/S0169–023X(97)00056–6. – ISSN 0169–023X

[72] W3C: *OWL 2 Web Ontology Language*. http://www.w3.org/TR/2009/REC-owl2-overview-20091027/. Version: Oktober 2009

[73] W3C owl working group: *List of OWL Reasoners*. http://www.w3.org/2007/OWL/wiki/Implementations&oldid=26281. Version: November 2011

[74] Wake, Bill: *INVEST in Good Stories, and SMART Tasks*. online. http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/. Version: August 2003. – zuletzt abgerufen am 18.05.2012

[75] Welty, C. A.: Augmenting abstract syntax trees for program understanding. In: *Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*. Washington, DC, USA : IEEE Computer Society, 1997 (ASE '97). – ISBN 0–8186–7961–1, S. 126–

[76] Wilson, William M. ; Rosenberg, Linda H. ; Hyatt, Lawrence E.: Automated analysis of requirement specifications. In: *Proceedings of the 19th international conference on Software engineering*. New York, NY, USA : ACM, 1997 (ICSE '97). – ISBN 0–89791–914–9, S. 161–171

[77] Witte, R. ; Li, Q. ; Zhang, Y. ; Rilling, J.: Text mining and software engineering: an integrated source code and document analysis approach. In: *Software, IET* 2 (2008), Februar, Nr. 1, S. 3 –16. http://dx.doi.org/10.1049/iet-sen:20070110. – DOI 10.1049/iet–sen:20070110. – ISSN 1751–8806

[78] Witte, René ; Zhang, Yonggang ; Rilling, Juergen: Empowering Software Maintainers with Semantic Web Technologies. In: *ESWC* Bd. 4519, Springer, 2007 (Lecture Notes in Computer Science). – ISBN 978–3–540–72666–1, S. 37–52

[79] Zave, Pamela ; Jackson, Michael: Four Dark Corners of Requirements Engineering. In: *ACM Transactions on Software Engineering and Methodology* 6 (1997), S. 1–30

[80] Zhang, Yonggang ; Witte, René ; Rilling, Juergen ; Haarslev, Volker: An Ontology-based Approach for Traceability Recovery. In: *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006)*, 2006, S. 36–43

[81] Ziftci, Celal ; Krueger, Ingolf: Tracing requirements to tests with high precision and recall. In: *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, 2011, S. 472 –475