Entwurf und Implementierung einer Werkzeugunterstützung zur sprachlichen Analyse und automatisierten Transformation von Projektlastenheften im Kontext der Automobilindustrie

An der Fachhochschule Dortmund

im Fachbereich Informatik

Studiengang Informatik

Vertiefung Praktische Informatik

erstellte Thesis

zur Erlangung des akademischen Grades

Bachelor of Science

B. Sc.

von Aaron Schul,

geboren am 24.06.1997

und Felix Ritter

geboren am 31.08.1997

Betreuung durch:

Prof. Dr. Sebastian Bab und Prof. Dr. Steffen Helke

Dortmund, 28.02.2019

Zusammenfassung

Ein weitläufiges Problem in der Autoindustrie ist die effiziente Verarbeitung von Lastenheften. Einer der Gründe dafür ist, dass viele verschiedene Bereiche bei der Entwicklung der Fahrzeuge und Einzelteile beteiligt sind. So müssen zu Beginn einer Produktentwicklung etwa Betriebswirtschaftler, Designer und Ingenieure zusammen ein Dokument entwerfen, das die Produktmerkmale widerspiegelt. Darin müssen die Anforderungen an das gewünschte Produkt so genau beschrieben sein, sodass es anhand dieses Dokuments entwickelt werden kann. Das entstandene Dokument kann dabei mehrere Hundert Seiten lang werden. Die manuelle Verarbeitung von Lastenheften ist aufgrund der verschiedenen Domänen und des großen Umfangs mit einem enormen Arbeitsaufwand verbunden. Das Resultat können Fehler in der Entwicklung sein, welche zu zeitlicher Verzögerung, zusätzlichen Kosten oder dem Abbruch des Projekts führen können. Methoden aus dem Natural Language Processing (NLP) können dabei helfen, die Überprüfung der Lastenhefte stark zu vereinfachen oder sogar teilweise zu automatisieren. Dafür kann beispielsweise der Text aus den Lastenheften analysiert und dessen Inhalt formalisiert und weiterverwendet werden. In dieser Arbeit wird daher die Entwicklung zweier NLP-basierter Werkzeuge zur Vereinfachung bzw. Lösung dieser Probleme vorgestellt. Zum einen der *Requirements-to-Boilerplate-Converter* (R2BC), welcher dem Nutzer helfen soll, das Lastenheft eines Auftraggebers in die betriebsinternen Richtlinien und Standards zu überführen und dadurch zu formalisieren. Zum anderen wird der *Delta-Analyser* (DA) vorgestellt, welcher auf dem R2BC aufbaut, indem er automatisch zwei formalisierte Lastenhefte vergleicht und dadurch im Kontext der gesamten Lastenhefte Gemeinsamkeiten und Unterschiede (Deltas) erkennt. Ziel ist es dabei, jeweils einen Prototypen als Machbarkeitsstudie zu implementieren. Nach dieser Vorstellung wird zusätzlich das weitere Potential der Programme erläutert, welches sich bei der Entwicklung der Prototypen gezeigt hat. Dieses bietet Vorschläge zur Weiterentwicklung der Programme.

Schlagworte: Anforderungen, Anforderungsmanagement, Automatisierung, Lastenheft, Werkzeugunterstützung, Automobilindustrie, Sprachschablone, Sprachanalyse, Requirements-to-Boilerplate-Converter, Delta-Analyser

Abstract

A major problem in the auto industry is the efficient processing of specifications. One of the reasons for this is that many different areas are involved in the development of vehicles and individual parts. For example, at the beginning of product development, business administrators, designers, and engineers all need to design a document that reflects the product features. The requirements for the desired product must be described in detail so that it can be developed using this document. The resulting document can be several hundred pages long. The manual processing of specifications is involved in an enormous amount of work due to the different domains and the large amount of text. In addition, this effort often has to be handled by only one person for reasons of clarity. Furthermore, the processing of the specifications by the people is very error-prone, since it is almost impossible to keep track of the entire document. The result may be developmental errors that can lead to delays, additional costs or the demolition of the project. Methods from Natural Language Processing (NLP) can help simplify or even partially automate the review of specifications. For example, the text from the specifications can be analyzed and its content formalized and reused. In this work, therefore, the development of two NLP-based tools to simplify or solve these problems will be presented. On the one hand, a *Requirements-to-Boilerplate Converter* (R2BC), which should help the user to transfer the specifications of a client into the internal guidelines and standards and thereby formalize them. On the other hand, the *Delta Analyzer* (DA) is introduced, which builds on the R2BC by automatically comparing two formalized specifications and thereby recognizing similarities and differences (deltas) in the context of the entire specification. The aim is to implement one prototype each as "*proof of concept*", ie as a feasibility study. After this presentation, the further potential of the programs, which has been discovered during the development of the prototypes, will be explained. This offers suggestions for the further development of the programs.

Keywords: Requirements, Requirements-Engineering, Automation, CRS, Tool-Support, Automotive Industry, Natural-Language-Processing, Requirements-to-Boilerplate-Converter, Delta-Analyser

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

BP	Boilerplate
CRS	Customer Requirement Specification
DA	Delta-Analyser
E-AE	Electronics-Advanced Engineering
NLP	Natural Language-Processing
OEM	Original Equipment Manufacturer
OWL	Web-Ontology-Language
PMT	Projects, Methods and Tools
POS	Part-Of-Speech
R2BC	Requirements-To-Boilerplate-Converter
RE	Requirements-Engineering

Danksagung

An dieser Stelle möchten wir uns bei allen Personen bedanken, die uns bei unserer Bachlorarbeit und während des Studiums begleitet und unterstützt haben.

Der Dank gebührt unseren Familien und Freunden aus der Heimat und aus dem Studium für ihre Unterstützung.

Wir danken auch allen Kollegen aus der Abteilung E-AE der Firma HELLA in Lippstadt für ihr Interesse und die großartige Zusammenarbeit. Die einzigartigen Erfahrungen, die wir während unseres Praktikums und später während der Bachelorarbeit in der Firma sammeln konnten, haben uns wirklich motiviert.

Besonderer Dank geht dabei an Konstantin Zichler, der weit mehr als nur Betreuer in der Firma, sondern auch Mentor und Ideengeber unserer Arbeit war.

Schlussendlich danken wir unseren Betreuern Prof. Dr. Sebastian Bab und Prof. Dr. Steffen Helke für die zahlreichen Tipps und Anregungen während der Bachelorarbeit.

Kapitel 1

Einführung

1.1 Motivation und thematische Grundlagen

In einer Vielzahl von Firmen stellt die Erhebung von Anforderungen den ersten Schritt bei der Entwicklung von neuen Produkten dar. Diese Anforderungen manifestieren sich später in der zu entwickelnden Hard- und Software. Spätere Produktmerkmale müssen dabei weit vor der tatsächlichen Entwicklung berücksichtigt werden und fließen in das Anforderungsdokument ein. Die Erhebung und Identifikation dieser Anforderungen wird als *Requirements-Engineering*(RE) bezeichnet [BAL10].

Dieser Prozess wird dabei sowohl bei kompletten Neuentwicklungen, als auch bei der Adaption eines bereits vorhandenen Produktes auf neue Projekte durchlaufen. Zur Rolle von Requirements-Engineering in Projekten siehe etwa [MW02] und [HL01].

Dem Betrieb liegen anschließend die Spezifikationen des späteren Produktes vor, auf deren Basis die Entwicklung beginnen kann. Die Entwicklung unterliegt dabei bestimmten Kriterien und Faktoren, die den unternehmerischen Erfolg beeinflussen. Neben betriebswirtschaftlichen Einflüssen wie der Einordnung des Produktes in der Wertschöpfungskette sind es dabei besonders technische Anforderungen an das Produkt, die durch die Anforderungen definiert und während der Produktentwicklung eingehalten werden müssen.

Anforderungen im betrieblichen Entwicklungsprozess

Diese Anforderungen werden im sogenannten Lastenheft zusammengetragen, in Unternehmen häufig als CRS (*Customer Requirement Specification*) be-

zeichnet [DGE19]. Mithilfe dessen kann nun die Entwicklung eines Produktes begonnen werden. Ein später hergestelltes Produkt sollte möglichst allen beschriebenen Anforderungen entsprechen.

Hersteller fertiger Produkte werden auch als *OEM* (*Original Equipment Manufacturer*), zu Deutsch *Erstausrüster*, bezeichnet [IR09]. Diese stellen nur selten alle einzelnen Komponenten des späteren Produktes selbst her und sind daher auf Zulieferer angewiesen, die einzelne Komponenten entwickeln und produzieren. Aus Sicht der Zulieferer ist es dabei betriebswirtschaftlich sinnvoll, nicht bloß für einen Kunden ein komplettes Produkt zu entwickeln, sondern bereits vorhandene Entwicklungen auf möglichst viele Kundenprojekte zu adaptieren.

Der Zulieferer erhält dabei häufig ein Lastenheft vom OEM, dass alle Anforderungen für die Entwicklung enthält. Das bedeutet jedoch, dass auch die Produktentwicklung anhand des Lastenheftes an den Zulieferer ausgelagert wird. Dem Zulieferer werden damit Möglichkeiten geboten und Verantwortung übertragen, das Produkt im Rahmen der Anforderungen zu definieren. Einerseits können etwa interne Richtlinien und Erfahrungen auf dem Gebiet der Produktentwicklung aus früheren Projekten integriert werden, andererseits aber auch weitere Optimierungen sowie Merkmale, die noch nicht im Lastenheft des OEM dokumentiert sind. Somit agiert der Zulieferer nicht nur als produzierende Fabrik, sondern nimmt am Entwicklungsprozess teil.

Die Anforderungen an ein Produkt, etwa technische Rahmenbedingungen und Qualitätsanforderungen, setzen sich dabei aus einer Auflistung von funktionalen und nicht-funktionalen Anforderungen zusammen [BAL10]. Häufig enthält das Lastenheft auch weitere Informationen und abseits von Text Abbildungen, die einzelne Anforderungen ergänzen.

Umstände der Anforderungsdefinition

Die Anforderungen werden dabei von vielen verschiedenen Domänenexperten beim OEM formuliert und in das Lastenheft für die Entwicklung eingetragen. Verschiedenste Akteure aus einem Unternehmen sind dabei an der Festlegung der Anforderungen an ein Entwicklungsprojekt bzw. Produkt beteiligt. Dies sind etwa Produktdesigner, Ingenieure und Systemtechniker, die an verschiedenen Stellen im Lastenheft Anforderungen an eine Komponente festlegen und an entsprechenden Stellen vermerken. Diese Beteiligten sind in der Regel auf ihren Bereich spezialisiert und nicht interdisziplinär, da häufig an verschiedenen Orten und an spezifischen Aspekten eines Produktes in

mehreren Projektteams entwickelt wird.

Projektteams aus einem Bereich tauschen sich häufig nicht untereinander aus und legen unabhängig von anderen Beteiligten Anforderungen fest. Demzufolge sammeln sich im Lastenheft schon bei der Entstehung beim OEM verschiedenste Merkmale einer Komponente. Ferner gibt es sprachliche Eigenheiten der Autoren und sprachliche Fehler, die innerhalb des Teams nicht auffallen und später nicht mehr korrigiert werden. Auch gibt es unternehmensinterne Richtlinien für die Formulierung von Anforderungen beim OEM, die das Verständnis auf Seite des Zulieferers erschweren können. Christian Allmann, Lydia Winkler und Thorsten Kölzow haben in [AWK06] solche Herausforderungen für das Requirements-Engineering zwischen OEMs und Zulieferern identifiziert.

Durch die entstehende große fachliche Breite und Tiefe der Spezifikationen im Lastenheft, können mitunter Dokumente mit einem Umfang von mehreren tausend Seiten entstehen. Dies bedeutet einen hohen Aufwand bei der Transformation und Bündelung der Anforderungen in ein tatsächliches Produkt beim Zulieferer.

Prozessoptimierung durch Werkzeuge

Bei der Untersuchung von Projektlastenheften ist daher die Formalisierung der Anforderungen wesentlich, damit Korrektheit des späteren Produktes gewährleistet ist. Die Analyse der Anforderungen stellt dabei aus Gründen der Effizienz ein Problem dar, wenn jeder Projektbeteiligte manuell die für ihn relevanten Anforderungen aus dem Lastenheft prüfen muss. Auch müssen die Lastenhefte an die Formulierungen und Ausdrucksweisen für Requirements-Management im Unternehmen angepasst werden.

Diese Probleme beim Verständnis der Anforderungen bieten Potential für Unterstützung durch Methoden der Informatik, wenn entsprechende Werkzeuge entwickelt werden. In 2.3 werden einige Ansatzpunkte aus der Literatur vorgestellt.

Bislang gibt es jedoch kaum Werkzeugunterstützung, die effiziente Möglichkeiten zur automatisierten Überarbeitung und Anpassung einzelner Anforderungen aus dem Dokument bietet. Ansätze aus dem *Natural-Language-Proessing*, kurz NLP, stellen gleichzeitig vielversprechende Forschungsfelder in der Informatik dar, die eine solche automatisierte Verarbeitung auf Basis von Sprachanalyse ermöglichen. Diese Analysemethoden werden in Kap. 2.1 ausführlich erläutert. Syntax und Semantik der einzelnen Sätze und Zusam-

menhänge in Texten können auf Basis aktueller Trends wie Machine-Learning und dynamischer Programmierung zunehmend besser abgebildet werden.

1.2 Unternehmenskontext der Arbeit

Besonders betroffen von dem geteilten Entwicklungsprozess zwischen OEMs und Zulieferern ist die Automobilindustrie. Eine Vielzahl von Zulieferern beliefert die Automobilhersteller mit Komponenten für ihre Fahrzeuge. Einzelne Komponenten bestehen dabei häufig auch aus mehreren Einzelteilen, die wiederum von mehreren verschiedenen Zulieferern hergestellt werden. [AWK06]

1.2.1 Einsatz bei HELLA

Einer dieser Zulieferer ist die HELLA GmbH & Co. KGaA (im folgenden (die) HELLA genannt), die sich auf die Entwicklung und Produktion von Licht- und Elektronikkomponenten in Hard- und Software spezialisiert hat. HELLA ist ein international operierender deutscher Automobilzulieferer mit Hauptsitz in Lippstadt und zählt zu den Top 100 Automobilzulieferern weltweit [LV10]. Für weitere Information siehe [HE19] sowie die Firmenwebsite.

Im Rahmen dieser Arbeit waren die Autoren fünf Monate bei HELLA beschäftigt. Dies beinhaltete zwei Monate Arbeit als Praktikanten und die verbleibenden drei Monate als Bacheloranden. Die Hauptaufgabe dort war die Mitarbeit im Bereich Requirements-Engineering für die Abteilung der Elektronik-Vorentwicklung *E-AE* (*Electronics-Advances Engineering*).

Die Abteilung ist zuständig für die Planung und Umsetzung von Vorentwicklungsprojekten um die Ergebnisse an die Serienentwicklung oder verschiedene OEMs weiterzuleiten. Dies beinhaltet häufig, wie zuvor beschrieben, den Austausch von Informationen der Firma HELLA mit diversen OEMs bezüglich gemeinsamer Projekte. Die genaue Darstellung dieser Prozesse erfolgt in Kap. 3.1. Somit erhält diese Abteilung auch oft Lastenhefte der OEMs, welche dann an die Projektteams weitergeleitet werden.

1.2.2 Anforderungsmanagement im Unternehmen

Die Gewichtung einzelner Anforderungen in einem größeren Systemkontext fällt dort schwer, da nun Projektteams beim Zulieferer, die an der Entstehung des Lastenheftes nicht beteiligt waren, dieses verstehen sollen und auf ihre

Teilkomponente anwenden müssen. Zusätzlich zu den teilweise uneindeutig formulierten Anforderungen aus dem Lastenheft kommen weitere Vorgaben des Zulieferers, wie etwa die Produktkosten und unternehmensinternte Richtlinien, hinzu. Schlussendlich soll jedoch ein Produkt entwickelt werden, dass möglichst alle Anforderungen berücksichtigt. [MW02]

Zu diesem Zweck können bei Zulieferern spezielle Abteilungen existieren, die Projektteams bei der Identifikation und Gewichtung von Anforderungen unterstützen. Diese Abteilungen sind auf die Auswertung der Projektlastenhefte spezialisiert und erstellen Modelle zu Merkmalen des späteren Produktes. Die Experten stellen dabei die Eindeutigkeit und Verständlichkeit sicher und identifizieren relevante Anforderungen für die Mitglieder des Projektteams. Somit kann ein eigenes Anforderungsdokument für das Projektteam beim Zulieferer erstellt werden. Falls nicht vorhanden, muss das Team selbst über die Bewertung von Anforderungen entscheiden und Rücksprache mit dem OEM halten.

Bei HELLA existiert eine dementsprechende Abteilung nur für Requirements-Engineering. Im Rahmen unserer Beschäftigung konnte in enger Zusammenarbeit den RE-Experten genau ermittelt werden, wie Lastenhefte momentan verarbeitet werden, welche Verbesserungspotentiale dabei bestehen und wie diese genutzt werden können.

1.3 Hypothese

Hypothese dieser Arbeit ist, dass sich mithilfe von NLP Lastenhefte effizient automatisiert verarbeiten lassen. Damit kann die Arbeit von Requirements-Engineers, aber auch von Beteiligten an der Entwicklung beim Verständnis der Anforderungen, erleichtert werden. Die Verarbeitung von Lastenheften bedarf dabei der Modellierung von Syntax und Semantik. Insbesondere durch die hohe Komplexität der Semantik werden dazu Ontologien benötigt.

Die sprachliche Analyse kann einzelne Anforderungen erkennen und formalisieren. Insbesondere lassen sich die beschriebenen Probleme bei der Verarbeitung der Lastenhefte lösen oder zumindest vereinfachen. Dies kann mithilfe der in dieser Arbeit beschriebenen Programme erfolgen, welche konzeptuell dargestellt und prototypisch implementiert werden.

Weiterhin soll geprüft werden, wie stark ein Mehrwert bei der Nutzung solcher Programme für betriebliches RE ausfallen kann. Dies betrifft primär ersparte Arbeitszeit bzw. reduzierten manuellen Aufwand.

1.4 Methodik

Verschiedene Methoden aus dem Bereich Natural-Language-Processing werden zunächst anhand des Rückblicks auf die Projektarbeit theoretisch und anhand praktischer Beispiele erläutert. Der betriebliche Kontext der Firma HELLA wird durch genaue Darstellung der dortigen Verfahren beleuchtet.

Mit diesen Grundlagen wird dann die Unterstützung der dortigen Prozesse im Bereich Requirements-Engineering auf Basis von natürlichsprachlicher Textanalyse evaluiert. Die gewonnenen Erkenntnisse aus der Zusammenarbeit mit der zuständigen Abteilung für RE wurden in Anforderungen an die zu entwickelnden Werkzeuge zusammengefasst. Dadurch entstand eine genaue Vorstellung der zu entwickelnden Software, welche mit Hilfe der Programmkonzepte von Konstantin Zichler [ZH19] entworfen und implementiert werden konnte.

Das Ergebnis stellten zwei Software-Prototypen auf *java*-Basis dar, deren Struktur und Funktionsweisen anhand von Modellen und Codebeispielen dargestellt werden. Die Prototypen wurden mit exemplarischen Lastenheften von verschiedenen bei HELLA getestet und evaluiert.

1.5 Aufbau der Arbeit

In der folgenden Ausarbeitung wird zunächst in Kapitel 2 eine Überblick über den Stand der Technik gegeben. Dies beinhaltet einerseits einen Rückblick auf die zuvor verfasste Projektarbeit zu NLP und einer Erläuterung der Grundlagen von Ontologien und andererseits einen Überblick von Arbeiten, die sich der gleichen oder ähnlichen Problemstellungen gewidmet haben. Kapitel 3 beschäftigt sich zur Motivation der beiden Programme mit dem betrieblichen Umfeld und RE-Prozessen bei der HELLA. Hierzu wird zunächst vor dem Hintergrund der aktuellen Verfahren der Anwendungsfall der Programme genauer beschrieben. Somit lassen sich die Anforderungen an die Werkzeuge erheben, welche dann von den vorgestellten Konzepten abgedeckt werden sollen.

In Kapitel 4 wird das erste Werkzeug - der Requirements-to-Boilerplate-Converter(R2BC) - ausführlich beschrieben. Dies beinhaltet die Darstellung der Architektur, eine Vorstellung der Implementierung, eine Auflistung möglicher Erweiterungen und eine genaue Beschreibung durchgeführter Tests. Zu den Tests wird dabei präzise die Methodik vorgestellt und Ergebnisse einge-

ordnet.

Das zweite Werkzeug - der Delta-Analyser(DA) - wird in Kapitel 5 behandelt. Dabei werden ähnlich zu Kapitel 4 die Architektur, die Implementierung und mögliche Erweiterungen beschrieben.

Die Arbeit wird von Kapitel 6 abgeschlossen. Dieses beinhaltet eine Evaluation des Projekts, die Zusammenfassung der geleisteten Arbeit und einen Ausblick.

In der Evaluation wird dargestellt, ob und inwiefern das Nutzen der Werkzeuge einen Mehrwert für die Firma erwirtschaften kann und somit im betrieblichen Umfeld eingesetzt werden sollte.

Der Inhalt der Arbeit wird zusammengefasst und abschließend ein Ausblick für die Zukunft der Werkzeuge und die Lösung des Problems erläutert.

1.6 Autorenverzeichnis

- **1.: Ritter, Schul**
  - 1.1: Ritter
  - 1.2: Ritter
  - 1.3: Ritter, Schul
  - 1.4: Schul
  - 1.5: Schul
- **2.: Ritter, Schul**
  - 2.1: Ritter, Schul
    - 2.1.1: Schul
    - 2.1.2: Ritter
    - 2.1.3: Ritter
    - 2.1.4: Ritter
  - 2.2: Schul
  - 2.3: Ritter
- **3.: Ritter, Schul**
  - 3.1: Schul
  - 3.2: Schul
  - 3.3: Ritter
  - 3.4: Schul
  - 3.5: Ritter
- **4.: Ritter, Schul**
  - 4.1: Ritter, Schul
    - 4.1.1: Schul
    - 4.1.1: Ritter
  - 4.2: Ritter, Schul
    - 4.2.1: Schul
    - 4.2.2: Ritter
    - 4.2.3: Ritter
    - 4.2.4: Ritter
  - 4.3: Schul
  - 4.4: Ritter
- **5.: Ritter, Schul**
  - 5.1: Schul
  - 5.2: Ritter, Schul
    - 5.2.1: Schul
    - 5.2.2: Ritter
  - 5.3: Schul
- **6.: Ritter, Schul**
  - 6.1: Schul
  - 6.2: Ritter
  - 6.3: Schul

Kapitel 2

Stand der Technik

In diesem Kapitel wird zunächst ein Rückblick auf die zuvor von den Autoren verfasste Projektarbeit und die darin enthaltenen Grundlagen von NLP gegeben. Aufgrund ihrer möglichen Relevanz für die Programme werden anschließend zusätzlich Ontologien als Teil von NLP genauer beleuchtet. Außerdem wird mit Hilfe von verwandten Arbeiten ein Überblick über die Problematik und verschiedene Lösungsansätze geboten.

2.1 Rückblick auf die Projektarbeit

In [RS18] sind die Grundlagen von NLP im Rahmen einer Projektarbeit zusammengefasst. In diesem Abschnitt werden die für die Arbeit relevanten Aspekte kurz wiederholt.

2.1.1 Grundlagen zu Natural Language Processing

Definition und Ziel

Natural Language Processing ist zu verstehen als die automatisierte oder halb-automatisierte Verarbeitung von natürlicher Sprache mit Hilfe eines Computers. In NLP sind verschiedenste wissenschaftliche Bereiche verknüpft.[1] Hauptsächlich handelt es sich dabei um Linguistik und Informatik, jedoch

[1] Es ist anzumerken, dass verschieden weit gefasste Definitionen von NLP existieren. Manche schließen etwa auch die Erfassung von Sprache mit ein, etwa durch Interaktion mit Sprachcomputern. Andere beschränken sich ausschließlich auf den Verarbeitungsprozess der Sprachdaten im Computer selbst, siehe 2.1.2.

gibt es auch viele Verbindungen zur Psychologie, Philosophie, Mathematik und Logik. [COP04]

Ziel von NLP ist üblicherweise die Extraktion von Informationen aus einem Text durch kleinschrittige Textanalyse. Diese Informationen werden dem Text direkt über sogenannte *Annotationen* angehängt oder anders abgespeichert, etwa in Ontologien, welche in einem späteren Abschnitt genauer erläutert werden.

Beispiele von Forschung und Entwicklung

Die wissenschaftliche Erforschung von NLP entstand Mitte des 20. Jahrhunderts, als der Linguist Noam Chomsky zeigte, dass sich Merkmale der englischen Sprache formalisieren und somit automatisiert verarbeiten lassen [CHO57].

Frühe Projekte wie der Sprachcomputer ELIZA aus den 1960er Jahren waren somit bereits in der Lage, eine natürlichsprachliche Frage als Texteingabe entgegenzunehmen und eine plausible Antwort zu liefern. Siehe dazu [WEI66].

Heutzutage gibt es ein Vielzahl von Spracherkennungssoftware auf Smartphones oder Smart-Devices wie ALEXA, welche in der Lage sind, gesprochene Kommandos zu erkennen und über das Internet an Server zu leiten. Diese können diese Kommandos schnell und effizient in der Cloud verarbeiten [HAO14].

Herausforderungen von NLP

Durch die Verbindung von Linguistik und Informatik stellt sich NLP generell der Herausforderung, sowohl inhärente Probleme der Sprache, als auch Probleme der Automatisierung lösen zu müssen.

Zu den sprachlichen Problemen gehören Aspekte wie Mehrdeutigkeit, Sarkasmus und Umgangssprache bzw. Redewendungen. Diese lassen sich oft nicht einzig und allein durch den gegebenen Text erklären, sondern beruhen auf Kontext und Situation.

Vor allem bei der Erfassung von Semantik, also dem Inhalt der Texte, können komplexe Probleme auftreten. So gibt es beispielsweise Mehrdeutigkeiten, welche selbst für den Menschen nicht einfach zuzuordnen sind. „*Die Betrachtung des Studenten*“kann etwa bedeuten, dass der Student etwas betrachtet. Genauso kann es jedoch sein, dass jemand anderes den Studenten

betrachtet.

Eine andere häufige Fehlerquelle ist das korrekte Zuordnen von sprachlichen Bezügen. Betrachtet man die Frage „*Verkaufen Sie Handys und Computer von Samsung?*“, so ist diese eindeutig grammatikalisch korrekt. Dennoch ist es uneindeutig, ob sich die Frage auch nach der Marke der Handys richtet, oder ob dies lediglich auf die Computer bezogen ist.

Noch komplexer wird es, wenn sich diese Bezüge über verschieden Sätze bzw. Teilsätze erstrecken. So ist der Satz „*Tom schenkt Tim ein Fahrrad, weil er nett ist*“ ebenfalls auf zwei verschieden Arten zu deuten. Einerseits kann es sein, das Tom das Fahrrad verschenkt, weil Tom ein netter Mensch ist. Andererseits kann es sein, das Tom das Fahrrad verschenkt, weil Tim nett zu ihm ist.

Weiterhin können auch bei der korrekten Erfassung des Sprachinhalts weitere Probleme die Kommunikation stark beeinflussen. So kann etwa die Intention der Sprache dem Inhalt widersprechen, wie es bei Sarkasmus üblicherweise der Fall ist.

Bei der Implementierung ergeben sich dann Probleme, trotz der sprachlichen Besonderheiten konsistente Regelmäßigkeiten als Grundlage zur Automatisierung zu finden.

Linguistische Analyse

Um einen Text möglichst Fehlerfrei verarbeiten zu können bedarf es also einem kleinschrittigen und präzisen Verfahren. Dieses bezeichnet man als linguistische Analyse.

Sie ist aufgeteilt in vier wichtige Schritte, welche aufeinander aufbauen und an Komplexität zunehmen. Im folgenden sind diese Schritte aufgeführt und kurz erläutert [RS18]:

1. Morphologische Analyse - Die Zerlegung von Wörtern bezüglich ihrer Struktur. Die Komposition aus Präfix, Wortstamm und Suffix von Wörtern wird erkannt, um Fall, Tempus, Numerus etc. des Wortes zu bestimmen. Im Englischen zeigt so die Endung -ed bei Verben die Vergangenheitsform an. Dabei ist zu beachten, dass Mehrdeutigkeiten entstehen können.

2. Syntax - Aufbau von Sätzen durch einzelne Wörter. Jede Sprache besitzt syntaktische Regelungen bezüglich der auftauchenden Wörter; im Deutschen folgt etwa auf einen Artikel irgendwann ein Substantiv oder

bestimmte Signalwörter geben Satztyp und Tempus an. Auf Basis dieses Wissens können formale und strukturelle Analysen der Textbestandteile erfolgen.

3. Semantiken - Die Identifikation der Bedeutung von einzelnen Sätzen und Wörtern. Die Semantik eines Textabschnittes wird häufig als „Logik“ bezeichnet und fragt nach dessen Bedeutung bzw. Thema. Semantische Deutungen können anhand von Kompositionen einzelner Wörter und Sätze auf Basis der semantischen Analyse erkannt werden.

4. Kontext - Darstellung von satzübergreifenden Zusammenhängen syntaktischer und semantischer Natur. Mehrere Sätze können über Konstrukte wie etwa Pronomen verbunden sein, wenn sich das Pronomen des einen Satzes auf ein Subjekt des anderen bezieht.

Diese Aufteilung lässt sich jedoch in noch weitere Teilschritte für die Spracherkennung und -generierung als NLP-Architektur unterteilen. Diese wird im folgenden Abschnitt genauer erläutert.

2.1.2 Umsetzung der Verarbeitungsschritte natürlicher Sprache

Dieser Abschnitt beschäftigt sich primär mit den verschiedenen Schritten der sprachlichen Analyse. Diese Differenzierung kann dabei, wie zuvor beschrieben, anhand der linguistischen Analyse unterschiedlicher Bestandteile von Sprache vorgenommen werden. Diese sukzessive Verarbeitung wird im folgenden anhand verschiedener Algorithmen und Verfahren erläutert, die im wesentlichen in [RS18] zu finden sind.

Das Ziel jeder Analyse ist es, Wissen über einen Teilbereich von Sprache zu generieren. Das Wissen über den Zusammenhang von Wörtern eines Textes kann eindeutig sein, jedoch können besonders bei der morphologischen Analyse sprachliche Mehrdeutigkeiten auftreten. Dies wird in 2.1.3-Morphologie näher erläutert.

Die Präzision dieser Aussagen fällt dementsprechend abhängig von der Treffergenauigkeit und dem Abdeckungsgrad der linguistischen Analyse durch die verwendeten Verfahren ab. Es ist dabei zu beachten, dass die Analyse typischerweise als Architektur aufgebaut ist, da die verschiedenen Schritte

aufeinander aufbauen [COP04]. Somit können, basierend auf den Ergebnissen, inhaltliche Aussagen und Zusammenhänge über den zugrundeliegenden Text getroffen werden.

[COP04] identifiziert dabei eine allgemeine NLP-Architektur, die basierend auf der linguistischen Analyse diese Teilbereiche voneinander abgrenzt. Für ein besseres Verständnis der Verfahren werden die für diese Arbeit relevanten Schritte in [RS18] wie folgt kurz erläutert:

1. Input Processing - Erkennung der Dokumentensprache und Normalisierung des Textes. Im ersten Schritt geht es um das korrekte Format des Eingabedokumentes für die Verarbeitung. Dieser Vorgang stellt jedoch noch keine Analyse dar, sondern dient lediglich der Vorbereitung.

2. Morphologische Analyse - Die meisten Sprachen sind im Bezug auf ihre Grammatik und syntaktische Struktur der Wörter in Systemen abgeschlossen, etwa durch Modellierung mittels Automaten zur Erzeugung der Worte. Auch können Vokabeln in Wörterbüchern/Lexika gesammelt und Regeln auf ihnen formuliert und gespeichert werden.

3. Part-of-Speech-Tagging - Die einzelnen Wörter eines Satzes werden im Hinblick auf den Sach- oder Satzzusammenhang, wie auch auf die Stellung im Satz hin analysiert. Basierend auf Kontext und/oder Erfahrungswerten bzw. Heuristiken können die Wörter korrekt erfasst und deren Fall getaggt werden. Subjekte, Prädikate usw. werden identifiziert. Dazu kann eine Wissensbasis mit Trainingsdaten und die Einordnung des Kontextes darin verwendet werden.

4. Parsing - Die Ergebnisse der vorherigen Schritte werden weiter verarbeitet und in ein standardisiertes Format gebracht. Syntaktische Zusammenhänge, wie etwa das Zusammenfassen von Wörtern zu einer gemeinsamen Bedeutungsphrase und das Zuordnen von Verben zu einem Nomen können nach dem Parsing dargestellt werden.

5. Disambiguation - Die Entfernung von Mehrdeutigkeiten auf Basis der Ergebnisse des Parsings stellt einen entscheidenden Schritt für die semantischen Analysen dar. Nur wenn die Aussage und der Kontext eines Satzes klar erkennbar sind, kann dessen Semantik gezielt abgeleitet werden.

6. Context Module - Textbausteine, deren Interpretation semantisch auf dem Kontext anderer Sätze oder Wörtern beruhen, können erfasst werden. Bei Anaphern ist etwa das Verständnis des aktuellen Kontexts abhängig von einer vorherigen Deutung.

7. Text Planning - Die Sachzusammenhänge und Semantiken, die aus dem zugrundeliegenden Text extrahiert wurden, werden für die Darstellung im fertigen Text definiert. Es wird festgelegt, welche der Bedeutungen qualitativ übertragen und vermittelt werden sollen. Die Reihenfolge und Umfang der behandelten Themen wird geschätzt und definiert.

8. Tactical Generation - Die Bedeutungen werden in Form konkreter Zeichenketten generiert, die die gewünschte Bedeutung enthalten. Diese Textbausteine basieren häufig direkt auf den Ergebnissen des vorherigen Parsings, da dort schon Bedeutungen zusammengefasst werden können. Der quantitative Teil des Textes wird erzeugt.

9. Morpholigical Generation - Die erzeugten Sätze werden morphologisch an die Rahmenbedingungen der verwendeten Sprache angepasst. Grammatiken und Regeln der Satzkonstruktion werden auf die erzeugten Wörter angewendet, sodass ein lesbarer und nachvollziehbarer Text entsteht.

10. Output Processing - Der Text wird nun, nachdem dieser inhaltlich komplett erzeugt vorliegt, an das Design und Format des jeweiligen Einsatzzwecks angepasst. Sollte Text, anders als in dieser Arbeit der Einfachheit halber angenommen, nicht als geschriebenes Wort ohne besondere Formatierung vorliegen, kann dieser einem *Formatter* zur Designanpassung übergeben werden. Ferner kann etwa mihilfe von Text-to-Speech eine Audioausgabe erfolgen.

Wie später gezeigt wird, kann die in dieser Arbeit beschriebene Entwicklung der Software-Prototypen als eine Umsetzung eben dieser NLP-Architektur verstanden werden.

2.1.3 Syntaktische Analyse von Wörtern und Sätzen

In diesem Abschnitt wird die Zerlegung der Syntax anhand logischer Modelle von Sprache beschrieben. Begonnen wird mit der einfachen Struktur und

Zusammensetzung einzelner Wörter aus der Grammatik. Die Kombination aus Wortstämmen mit Prä- und Suffixen kann etwa in einem Lexikon gesucht werden, dass alle Wörter einer Sprache enthält. Basierend auf dieser Morphologie eines Wortes kann dann in der Analyse mehrerer Wörter auf die Rolle von Wörtern im Satzzusammenhang geschlossen werden.

Morphologie

Dieser Abschnitt befasst sich genauer mit der Analyse der Struktur und Zusammensetzung von Wörtern. Dies wird als morphologische Analyse bezeichnet. Wie im folgenden beschrieben, können auf dieser Basis logische Modelle wie etwa Automaten erstellt werden, die zulässige Wörter aus einer Sprache auf Basis grammatikalischer Regeln erzeugen können. Für ein besseres Verständnis später vorgestellten Ergebnisse beschränkt sich diese Arbeit auf die vergleichsweise einfach strukturierte englische Sprache. Aufgrund der großen Fortschritte und Erfahrungen in der Sprachforschung, werden die folgenden Beispiele auf Englisch behandelt.

In der englischen Sprache besteht jedes Wort aus einem Wortstamm, sowie den sogenannten Affixen. In einem Satz könnte etwa das Wort „*believable*“ auftauchen. Das Wort stammt von der Grundform „*(to) believe*“ (zu Deutsch: „glauben “) ab. Das *-able* stellt ein Suffix dar, also eine Wortendung. Zusammen mit den Präfixen, die dem Wortstamm vorangestellt sind, gehört *-able* somit zur Kategorie der Affixe.

Das Ziel der morphologischen Analyse ist nun, die Zusammensetzung des Wortes nachzuvollziehen und die Information für die weitere Verarbeitung bereitzustellen. Die Generierung des Wortes *believable* kann anhand von Buchstabierregeln erfolgen, die alle zulässigen Wörter auf englisch anhand der Wortstämme und Affixe erzeugen können. Es fällt auf, dass in unserem Beispiel *believable* vor dem Hinzufügen des Suffixes *-able* zunächst das *-e* von der Grundform *(to) believe* entfernt werden muss. In einer Grammatik, die zur Erzeugung dieser Wörter genutzt werden kann, muss dementsprechend eine solche Regel vorhanden sein. Gemäß [RS18] bedeutet dies formal ausgedrückt:

$$\texttt{e} \rightarrow \varepsilon\widehat{}\texttt{\_able}$$

In der Logik kann man diese Regeln als nicht-deterministische *Zustandswandler* visualisieren.[2] In Abb. 2.1 ist der Zustandswandler gezeigt, die das

[2]In [RS18] wurde darauf hingewiesen, dass nicht-deterministische Zusatandswandler

Suffix -*able* an entsprechende Wörter anhängen kann.

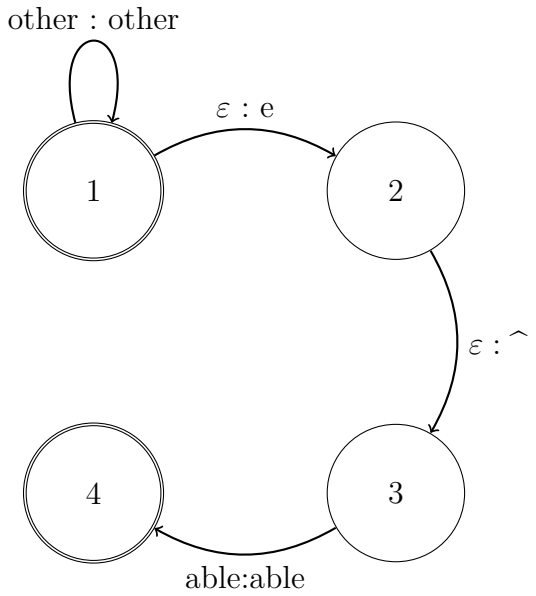

Abbildung 2.1: Nicht-deterministischer Zustandswandler der able-Regel [RS18]

Um alle zulässigen Wörter sammeln und identifizieren zu können, muss die morphologische Analyse über eine Menge an Informationen über die Sprache verfügen. Die Bestandteile von Wörtern können dabei in verschiedenen Lexika enthalten sein, welche die entsprechenden Informationen bereitstellen. Gemäß [COP04] bedarf es dabei drei verschiedener Lexika:

1. Liste von *Affixen* (zusammen mit den Informationen, die damit verbunden sind, z.B. Negation „`un-`")
2. Liste von *unregelmäßigen Wörtern* (zusammen mit den Informationen, die damit verbunden sind, z.B. „`went":simple past„(to) go`")

trotzdem mithilfe der *Potenzmengenkonstruktion* in einen äquivalenten deterministischen Zustandswandler überführt werden kann. Sie kommen somit zu einem eindeutigen Ergebnis *erzeugbar* oder *nicht erzeugbar* in der Sprache.

3. Liste von *Wortstämmen* (zusammen mit syntaktischer Kategorie, z.B. `„believe":verb`)

Es ist zu beachten, dass die in der morphologischen Analyse verwendeten Buchstabierregeln zwar zu jeweils eindeutigen Ergebnissen kommen können, die Analyse selbst jedoch nicht eindeutig sein muss. Dies kann, wie bereits zuvor beschreiben, durch sprachliche Mehrdeutigkeiten entstehen.

Das Wort „*even*“ könnte isoliert etwa einmal als *gleichmäßig*, also als Adjektiv kategorisiert, ins Deutsche übersetzt werden. Jedoch kann auch sinngemäß *sogar*, also das Adverb mit einem oder mehreren Bezugswörtern, gemeint sein.[3] Die mehreren möglichen Bedeutungen des einzelnen Wortes können jedoch in der Regel durch die Position und grammatikalische Funktion im Satz aufgelöst werden. Diese Eigenschaften werden durch das *Part-of-Speech-Tagging* identifiziert.

Part-Of-Speech-Tagging

Die Ergebnisse der morphologischen Analyse von mehreren Wörtern, etwa in einem Satz, können zur Analyse von Zusammenhängen zwischen diesen Wörtern genutzt werden. Somit kann die grammatikalische Struktur des Satzes auf Basis der einzelnen Wörter dargestellt werden. Der grammatikalische Typ jedes Wortes wird durch das Part-Of-Speech-Tagging festgelegt und jedes Wort wird *annotiert*. Wie später in dieser Arbeit beschrieben wird, dienen solche Annotationen in NLP-Werkzeugen der Sammlung der bekannten Informationen zu einem Wort oder Satz.

Zur Festlegung dieser grammatikalischen Funktionen werden sog. *POS-Tagger* eingesetzt, die verschiedene Worttypen als Kategorien beinhalten. Diese werden dann jedem Wort eines Satzes als Annotation hinzugefügt. Die Aufgabe ist es nun, die korrekte Funktion eines Wortes im Satz auf Basis der morphologischen Analyse zu identifizieren.

Als Beispiel soll der Satz

```
I love trains.
```

[3] Aus Gründen der Vereinfachung wurde sich hier auf zwei erkennbar verschiedene Bedeutungen beschränkt. Tatsächlich stellt sich neben Bestimmung des grammatikalischen Typs auch die Frage, ob sich die verschiedenen Übersetzungen derselben Kategorie, z.B. *even* als *gerade* und *gleichmäßig*, in ihrer Bedeutung im Sachzusammenhang unterscheiden.

dienen. Das Part-of-Speech tagging zum Wort „*trains*“ soll hier veranschaulicht werden. Das Wort „*trains*“ allein ist ein Nomen im Plural, jedoch könnte es sowohl das Subjekt, als auch das Objekt im Satz darstellen. Im Englischen sollte ein POS-Tagger aber erkennen, dass *Subjekt-Prädikat-Objekt* eine gültige Satzkonstruktion darstellt, die zu dem Beispielsatz passt. Dementsprechend entfällt die Möglichkeit von *trains* als Subjekt, da es hinter einem Verb als Akkusativobjekt des Satzes auftaucht.

Es existieren verschiedene Techniken für diese Beurteilungen der POS-Tagger. Verschiedene gültige Satzkonstruktionen sind häufig in sog. *Corpora* gespeichert. Diese stellen repräsentative Trainingsdaten dar, die beispielsweise aus der Zeitung *Wall-Street-Journal* stammen. In diesem Corpus könnten etwa die annotierten Sätze

```
People(subj) love(vb) trains(obj) .(punc) und
I(subj) go(vb) home(obj) .(punc)
```

enthalten sein.[4]

Mithilfe dieser Menge von Daten kann POS-Tagging auch heuristisch, d.h. abhängig von Wahrscheinlichkeiten bzw. Metriken, erfolgen, indem das Auftauchen verschiedener Satzbauten im Trainingscorpus gezählt wird. Auf dieser Basis können dann Wahrscheinlichkeiten für verschiedene Möglichkeiten berechnet werden, wenn es mehrere mögliche Zuordnungen gibt.

Nutzen syntaktischer Analyse für NLP

Ein wichtiges Einsatzgebiet für POS-Tagger ist auch die Wortvorhersage bzw. *prediction* auf Basis solcher Heuristiken. Diese kann im Kontext der zuvor vorgenommen Annotationen von Wörtern erfolgen. Beispielsweise werden sog. *n-Gramme* verwendet, um unter Berücksichtigung der Annotation von n gelesenen Wörtern die Möglichkeiten für das folgende Wort zu berechnen. In [RS18] findet sich ein vollständiges Beispiel anhand eines *Bigramms* mit Corpus.

Die heutigen Verfahren zur syntaktischen Analyse nutzen zur Steigerung

[4] Die tatsächliche Annotation kann durch POS-Tagger i.d.R. feiner erfolgen, sodass etwa Pronomen von Subjekten unterschieden werden. Auch können, wie später in dieser Arbeit gezeigt wird, eigene Annotationen bei Bedarf nach neuen Kategorien hinzugefügt werden. Der Corpus hier stellt ein stark vereinfachtes Beispiel dar. Ein ausführliches Beispiel unter der Verwendung der CLAWS-7-POS-Taggers findet sich in [RS18].

der Genauigkeit u.a. Verfahren aus dem maschinellen und automatisierten Lernen und können somit hohe Trefferquoten erzielen.[5] Um ein tieferes Verständnis von Texten zu erzielen, reicht die in diesem Abschnitt beschriebene syntaktische Analyse anhand der Grammatik nicht aus. Die tatsächliche Abbildung von Wissen über semantische Zusammenhänge stellt eine verlässlichere Möglichkeit dar, sprachliche Zusammenhänge zu identifizieren.

2.1.4 Umgang mit Semantik

Wie im letzten Abschnitt aufgezeigt wurde, können nicht alle Informationen aus einem Text eindeutig anhand der Erfassung der Grammatik eines Satzes extrahiert werden. Die Syntax eines Satzes ist häufig mehrdeutig, da der Zusammenhang einzelner Wörter verschieden interpretiert werden kann. Auf der semantischen Ebene, d.h. der Bedeutung, existieren jedoch bereits bei einzelnen Wörtern verschiedene Deutungen.

Schon bei der Mensch-Mensch-Kommunikation kommt es somit häufig zu Missverständnissen, wenn die Bedeutung hinter einer Aussage nicht eindeutig ist. Auf Basis des Kontextes einer Aussage bzw. der Domäne des Inhalts kann jedoch meist zu einer mehrdeutigen Aussage eine eindeutige Deutung identifiziert werden.

NLP beschäftigt sich daher auch intensiv mit der Abbildung und Modellierung der Semantik von Sprache, die in der Logik häufig als *„Logik einer Aussage“* bezeichnet wird.[6] Die Formalisierung mehrerer Bedeutungen kann dabei mithilfe von Symbolen der Aussagenlogik erfolgen. Die Symbole dienen als Platzhalter für Wörter bzw. Satzteile. Etwa das Beispiel aus [RS18]: `Every man loves a woman.`

Zu deutsch *„Jeder Mann liebt...“* ist semantisch mehrdeutig zwischen

1. *„...die eine Frau“*: $\exists y[woman'(y) \bigwedge \forall x[man'(x) \Rightarrow love'(x, y)]]$ und
2. *„...irgendeine Frau“*: $\forall x[man'(x) \Rightarrow \exists y[woman'(y) \bigwedge love'(x, y)]]$

[5] In [SHE07] wird ein POS-Tagger auf der Basis von bidirektionaler Analyse und Machine-Learning zum Erstellen der Trainingsdaten beschrieben. Obwohl nur auf Basis der Textsyntax erzielt, sind die Ergebnisse zu 97,33% korrekt.

[6] Bereits in den 1980er Jahren wurden Konzepte auf Basis logischer Modelle, mit denen Semantik abgebildet werden kann, mithilfe von Theorembeweisern und Model-Checking beschrieben. [SB88] [KN85]

Bereits durch den Menschen fällt die Deutung anhand von Symbolen der Aussagenlogik nicht leicht. Daher wurden verschiedene Arten von Zusammenhängen, u.a. durch Abbildung auf Symbole der Mengenlehre definiert, anhand welcher die Deutung vereinfacht und klarer erkennbar wird.

Meaning-Postulates

Linguisten und Philosophen haben Regeln für die Abbildung von Zusammenhängen auf eindeutige Kategorien formalisiert. Dabei wurden verschiedene Bedeutungen von Wortzussammenhängen identifiziert, die als (Bedeutungs-)Mengenbeziehung bzw. Relation zwischen Symbolen visualisiert werden können [CAR52]:

1. *Meaning-Postulates*: A beinhaltet B und C

   Der Begriff A setzt sich logisch aus der Relation zwischen B und C zusammen, etwa:

   $\forall x[bachelor(x) \Rightarrow man(x) \bigwedge unmarried(x)]$

   sinngemäß:„*Ein Junggeselle ist ein Mann und nicht verheiratet.*“

2. *Hyponomy*: A ist ein B

   ```
   Ein Hund ist ein Tier.
   ```

   Also hat ein Hund auch alle Eigenschaften eines Tieres. Auf Basis solchen Wissens kann eine Kategorisierung erfolgen bzw. lassen sich verscheidene Ausprägungen der gleichen Sache zusammengefassen [RS18].

3. *Meronymy*: A $\subseteq$ B, A ist Teil von B

4. *Synonymy*: A $\equiv$ B, A ist semantisch äquivalent zu B

5. *Antonymy*: A $\equiv \neg$B, A bedeutet das Gegenteil von B

Feinheiten der Deutung, wie etwa der Unterschied von „*A semantisch äquivalent zu B*“ zu „*A ist ein B*“ müssen dabei während der Erstellung semantischer Modelle unterschieden werden, da die Kategorisierung sonst zu grob ausfällt. Es können sonst falsche Zusammenhänge und Beziehungen entstehen, die Fehlinterpretationen der Aussagen erzeugen.

In der Praxis stellt sich daher die Frage, in welchem Umfang Semantiken für NLP tatsächlich für eine Verbesserung der Textanalyse sorgen und wie detailliert eine Modellierung dazu erfolgen muss. Mit diesem Ansatz beschäftigen sich die *Ontologien* in Kapitel 2.2.

2.2 Ontologien

Dieser Abschnitt befasst sich genauer mit dem Begriff der Ontologien im Kontext der Informatik. Ein weit verbreitetes Werkzeug für die Arbeit mit Ontologien ist *Protegé*. Dieses wird im Folgenden für die Modellierung verwendet.

2.2.1 Definition

Ontologien lassen sich allgemein definieren als „formale, schematische Abbildungen eines Wissensbereichs, bestehend aus einem Vokabular und Regeln zu seiner Zusammensetzung“ [WE13].

Zunächst bedeutet dies, dass sich eine Ontologie stets nur auf Wissen aus einer bestimmten Domäne bezieht und nicht versucht, das gesamte über die Welt bzw. sehr allgemeines Wissen abzubilden.

Des Weiteren besteht eine Unterteilung der Ontologie in ein Vokabular und in eine Menge von Regeln.

Einer der Aspekte von Ontologien, welche sie für die Informatik so geeignet machen, ist, dass sie in maschienenlesbaren Ontologiesprachen verfasst werden. Somit können Computersysteme diese automatisch auslesen, ggf. interpretieren und daraus schlussfolgern.

Da Terminologie und bestimmte Formalitäten zwischen Ontologiesprachen variieren können, wird im Folgenden der Bezug auf die weit verbreitete Ontologiesprache „Web Ontology Language“ (OWL) angenommen.

2.2.2 Aufbau

Der Aufbau von Ontologien lässt sich grob in zwei Aspekte unterteilen:

1. *Das Vokabular*: Hier sind alle Dinge innerhalb des Wissensbereiches aufgeführt, welche von der Ontologie abgedeckt werden sollen. Dazu gehören sowohl die Oberbegriffe dieser Dinge, als auch die Instanzen selbst.

2. *Die Regeln*: Diese beschreiben die Dinge genauer und bilden deren Beziehung untereinander ab. Sie können eigenständig existieren oder im Bezug zu den vorhandenen Dingen stehen.

Klassenhierarchie

Die Hierarchie einer Ontologie, also der strukturelle Aufbau, lässt sich in die folgenden drei Komponenten unterteilen:

1. *Begriffe*: Die Grundlage für die Struktur einer Ontologie stellt der hierarchische Aufbau von *Begriffen* (*concepts*) dar. Diese repräsentieren eine Art Klassenstruktur und werden daher oft auch als Klassen bezeichnet. Sie können in aus der Informatik bekannten Ansätzen wie Ober- und Unterklassen angeordnet werden. Begriffe sind üblicherweise sehr generell gehalten und könnten etwa die Klasse *Auto* abbilden.

2. *Typen*: In vielen Fällen lässt sich ein Begriff in verschiedene *Typen* aufteilen. Ein Auto kann beispielsweise nach verschiedenen Automarken typisiert werden. Diese Typen werden dann ebenfalls durch Klassen verkörpert und bilden in dem hierarchischen Aufbau eine Unterklasse des jeweiligen Begriffs. Da sowohl Begriffe, als auch Typen in Klassen verkörpert werden, wird in der Praxis oftmals nicht zwischen den beiden unterschieden. Man kann jedoch üblicherweise über einer Unterklasse als Typ der Oberklasse sprechen.

3. *Instanzen*: Die sogenannten *Instanzen* (*instances*) verkörpern konkrete Objekte dieser Klassen und werden nicht weiter aufgespalten bzw. typisiert. Ein Instanz wäre dann etwa ein bestimmtes Modell einer bestimmten Marke von Autos.

Diese Hierarchie ist ggf. beliebig erweiterbar. So könnte man für eine bessere Klassifizierung z.B. die Marke eines Autos zunächst in eine Produktreihe oder nach einer Produkteigenschaft wie *Kombi* typisieren und erst davon abhängig die verschiedenen Instanzen zuordnen.

In Abbildung 2.2 ist eine stark vereinfachte Klassenhierarchie aus Oberklassen, Unterklassen und Instanzen beispielhaft dargestellt.

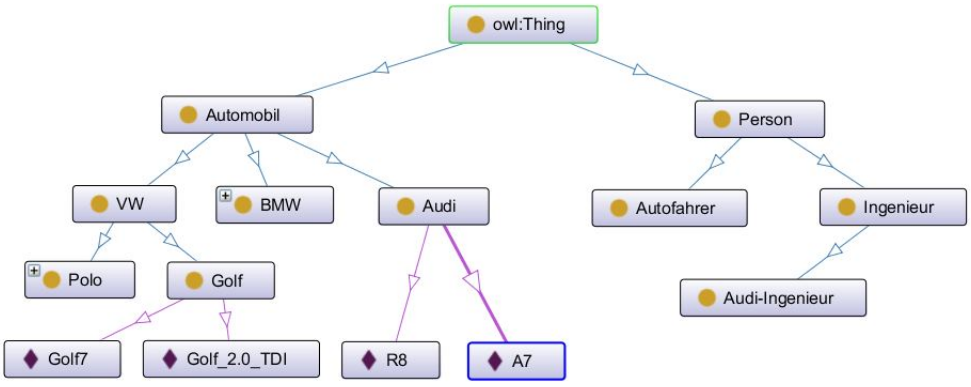

Abbildung 2.2: Beispielhafte Ontologie als Klassenhierarchie in Protegé [eigene Darstellung]

Auf der obersten Ebene der Klassen-Hierarchie befindet sich die Klasse *owl: Thing*. Von dieser erben alle Begriffe durch die *hasSubclass*-Relation. Dies ist vergleichbar mit der objektorientierten Programmiersprache *Java* bei welcher alle Klassen direkt oder indirekt von der Oberklasse *Object* erben.

Auf der nächsten Ebene befinden sich die beiden Begriffe *Automobil* und *Person*. Diese können entweder direkt in Instanzen übergehen, oder zunächst durch Typisierung unterteilt werden.

Somit finden sich in der nächsten Ebene die drei nach Marke getrennten Automobil-Typen *VW*, *BMW* und *Audi*. Außerdem werden Autofahrer und Ingenieure als Personen unterschieden. Es ist anzumerken, dass es nicht ausgeschlossen ist, das ein Ingenieur auch Autofahrer ist. Aufgrund der möglichen Mehrfachvererbung lassen sich jedoch Typen oder Instanzen erzeugen, welche sowohl von Autofahrer, als auch von Ingenieur erben. Somit stellt eine Unterteilung in Typen ohne gegenseitigen Ausschluss in der Praxis kein Problem dar und kann sogar nützlich sein, wie später anhand von Relationen gezeigt wird.

In der vierten Ebene sind einzig zu Demonstrationszwecken einige der Typen weiter unterteilt und andere nicht. Dies zeigt, dass die Hierarchie also keineswegs gleichmäßig sein muss, um eine sinnvolle Unterteilung zu gewährleisten.

In der letzten Ebene befinden sich einige Instanzen von Autos, welche durch konkrete Modelle dargestellt werden. Theoretisch ließen sich auch diese

weiter unterteilen, etwa nach Innenausstattung. Diesbezüglich ist je nach Verwendungszweck der Ontologie genau festzulegen, wie die Hierarchie zu strukturieren und wie weit sie auszuführen ist. Beispielsweise macht es für einen großen Autohersteller keinen Sinn, wenn in einer konzernübergreifenden Ontologie jeder Ingenieur instantiiert ist.

Regeln

Die Regeln in einer Ontologie lassen sich in die folgenden zwei Komponenten unterteilen:

1. *Relationen*: Einen wichtigen Bestandteil von Ontologien bilden die sogenannten *Relationen* (*properties*). Relationen beschreiben Beziehungen zwischen Begriffen und werden oft auch als Eigenschaften verwendet und bezeichnet. Ähnlich zu vielen Programmiersprachen können diese Relationen und Eigenschaften eines Begriffs an die Instanzen des Begriffs vererbt werden. Grundsätzlich ist dabei auch eine Mehrfachvererbung möglich.

2. *Axiome*: Des Weiteren können Ontologien durch sogenannte *Axiome* (*restictions*) genauer modelliert bzw. eingeschränkt werden. Axiome verkörpern dabei Regeln in der Ontologie, welche stets zutreffen müssen. Üblicherweise werden diese dazu verwendet, Einschränkungen vorzunehmen, die nicht effizient durch den abgebildeten Wissensbereich darzustellen sind. Dies könnte Beispielsweise in einer Ontologie über verschiedene Automodelle verschiedenster Marken eine generelle Definition von Automobilen sein. So ließe sich die Voraussetzung, dass ein Automobil motorisiert sein muss, durch eine Klasse *Bauteil* mit der Unterklasse *Motor* darstellen, welche dann über eine *muss verbaut sein in*-Relation mit der Klasse *Automobil* verbunden ist. Ist es jedoch nicht Zweck der Ontologie, Bauteile genauer abzudecken, wäre dies unnötiger Aufwand, der außerdem die Übersichtlichkeit der Ontologie einschränken könnte.

Abbildung 2.3 zeigt eine beispielhafte, stark vereinfachte Ontologie zu Automobilen und bestimmten im Kontext relevanten Personen.

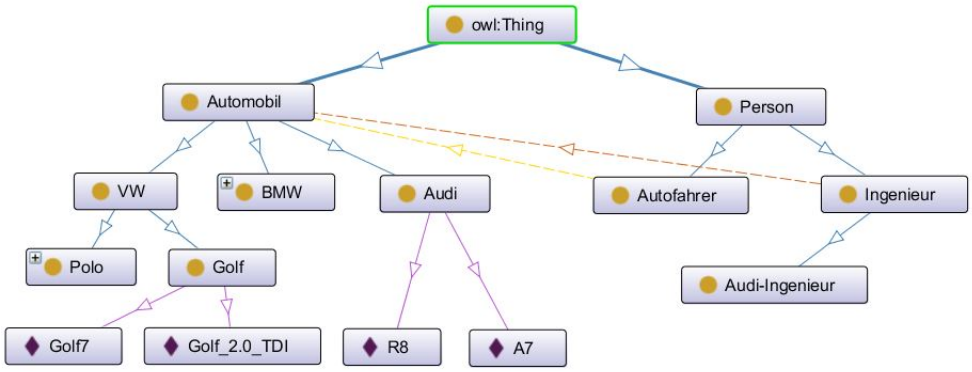

Abbildung 2.3: Klassenstruktur mit zusätzlichen Relationen [eigene Darstellung (in Protegé)]

Die Relationen zwischen den Knoten der Ontologie sind durch Linien mit einem mittigen Pfeil dargestellt. Die durchgezogenen Linien zeigen den Grundaufbau der Hierarchie. Die blauen Linien verkörpern die *has subclass*-Beziehung, welche auf Vererbung hinweist. Die violetten Linien stehen für die *has individual*-Beziehung und zeigen auf eine Instanz der Klasse. Außerdem wurden zwei Objekt-Attribute (*object properties*) manuell hinzugefügt. Diese können verwendet werden, um Relationen zwischen Klassen aufzuzeigen, welche nicht voneinander erben, sondern sich in unterschiedlichen „Teilbäumen“ der Ontologie befinden. So verweist, dargestellt durch den gelben Pfeil, eine *benutzt*-Relation von dem Autofahrer auf das Automobil. Dies bedeutet also, dass Autofahrer Automobile verwenden.

Ähnlich dazu zeigt von der Klasse *Ingenieur* eine *entwickelt*-Beziehung (brauner Pfeil) auf *Automobil*.

Verwendung von Ontologien

Wie eingehend bereits erwähnt, sind Ontologien vor allem in der Informatik durch ihre Maschinenlesbarkeit verbreitet. Sie sind außerdem auch geeignet, komplexere Sachzusammenhänge abzubilden. Dies ist mit primitiven Datentypen, wie man sie in einer relationalen Datenbank abspeichern kann, nicht möglich. Simple Verbindungen, wie sie etwa in den in 2.1.4 beschriebenen Bedeutungspostulaten beschrieben sind, lassen sind in Ontologien schnell und

effizient darstellen.

Betrachtet man zum Beispiel das Satzschema „B ist ein A“, so würde dies durch eine einfache *hasSubclass*-Beziehung von der Klasse *A* zu der Klasse *B* dargestellt. Erstellt man nun die beiden Instanzen *b1* und *b2* der Klasse B, so liegen implizit bereits die Informationen vor, dass sie beide auch ein *A* sind. Außerdem ist ihre Homogenität durch die Abstammung von der gleichen Klasse gewährleistet. Daher brauchen die Instanzen hierfür weder eine Beziehung untereinander, noch zu der Oberklasse *A* ihrer Elternklasse *B*.

Im Zusammenhang dieser Arbeit könnten Ontologien später zur Modellierung der Anforderungen aus Lastenheften dienen. Die durch die Anforderungen beschriebenen Dinge sind sind dabei stark domänenbezogen und somit optimal geeignet, um in Ontologien abgespeichert zu werden.

Eine üblicherweise vorausgesetzte Eigenschaft für Anforderungen ist etwa Atomarität [HP12], was bedeutet, dass jeder Satz nur genau eine Aussage beinhalten soll. Dies hat zur Folge, dass viele betriebliche Anforderungen tatsächlich den vorgestellten, knappen Satzschemata entsprechen. Maßnahmen zur Herstellung konsistenter Anforderungen sind in Abs. 3.3.1 beschrieben.

2.3 Verwandte Arbeiten

Dieses Kapitel stellt einen Überblick über bereits veröffentlichte Arbeiten dar, die sich mit dem Themen Natural Language Processing für Anforderungsmanagement, sowie die Unterstützung durch Werkzeuge beim Requirements-Engineering befassen.

Grundlage dieser Arbeit

K. Zichler und S. Helke stellen in [ZH17] und [ZH19] die wesentlichen Aspekte für den Entwurf einer Softwareunterstützung für Requirements-Engineering in Firmen vor.

In [ZH17] wird zunächst der Bedarf nach solcher Unterstützung besonders für die Automobilindustrie beschrieben, da die Verbesserung herkömmlicher Methoden im Requirements-Engineering dort wesentliches Potential bietet. Der Vorschlag beinhaltet die Unterstützung von Projektteams durch ein Expertenteam aus Requirements-Experten, die Anforderungen aufbereiten und deren konsistente Umsetzung verbessern.

Auf dieser Basis stellen sie in [ZH19] konkret einen Entwurf eines Werkzeugs für die automatisierte Analyse von Projektlastenheften vor. Das Werkzeug gliedert sich dabei in eine Komponente zur Überführung natürlichsprachlicher Anforderungen in vordefinierte Sprachschablonen, genannt *Requirements-to-Boilerplate-Converter* (R2BC). Auf dieser Basis arbeitet dann der *Delta-Analyser* (DA), der die Unterschiede zwischen mehreren Lastenheften, die in Schablonen überführt sind, darstellen kann. Dabei kommen auch Ontologien und NLP-Methoden zum Einsatz. Die Prototypen basieren dabei auf der Open-Source-Software *GATE* zur sprachlichen Auswertung und sollen als 3-Schichten-Architektur umgesetzt werden.

Diese Arbeiten stellen die wesentliche Grundlage für die in dieser Arbeit entworfenen Prototypen dar. Die Implementierung erfolgte in Zusammenarbeit mit dem Autoren Konstantin Zichler bei der Firma HELLA, die den konkreten Anwendungsfall für seine Entwürfe und unsere Umsetzung dargestellt hat.

Es ist anzumerken, dass eine Vielzahl betrachteter Arbeiten das Verbessern von Requirements mithilfe von Schablonen zum Thema haben, sich diese jedoch auf eine Unterstützung bei der Formulierung von Anforderungen beschränken. Eine Umsetzung einer automatisierten Verbesserung und Anpassung bestehender Anforderungen wird häufig zwar umrissen, jedoch selten implementiert bzw. real getestet.

Im Kontext der Beziehung zwischen OEM und Zulieferer kann jedoch aus Sicht des Zulieferers kein Einfluss auf die Qualität der beim OEM formulierten Anforderungen genimmen werden. Für diesen Anwendungsfall stellt eine Unterstützung der Formulierung also generell keinen Lösungsansatz dar. Die folgenden Arbeiten beinhalten jedoch vielversprechende Techniken für die Modellierung von Anforderungen.

Literatur

In [KR93] untersuchte Kevin Ryan bereits vor modernen Entwicklungen im Bereich NLP die Möglichkeiten einer Unterstützung von Requirements-Engeneering durch natürlichsprachliche Analyse (NLP). Dabei weist er auf die Vorteile einer formalen Einschränkung bereits bei der Spezifikation von Anforderungen hin, womit die Möglichkeit einer Fehlinterpretation vermindert wird. Ryan stuft jedoch auch die Verifikation von Anforderungen durch NLP als geeignet ein, um Konsistenz und Eindeutigkeit der Anforderungen

zu gewährleisten. Dazu generieren sie Schemas aus der Analyse der Anforderungen, die dann inhaltlich mit neu dazukommenden Anforderungen im späteren Entwicklungsprozess abgeglichen werden können. Auch schlägt er die Einteilung besonders von Text im Anforderungsdokument in Kategorien vor, damit die Sichtbarkeit von Anforderungen verbessert wird.

In [FF11] beschäftigen sich Farfelder et. al mit der Unterstützung von Requirements-Experten bei der Anforderungsspezifikation durch vorgefertigte Sprachschablonen und Ontologien, womit Fehler bei der Konsistenz von Anforderungen vermieden werden. Sie stellen das Tool *DODT* vor, dass als Alternative zur natürlichsprachlichen Formulierung das Erstellen von Anforderungen anhand von Schablonen anbietet, aus denen der Nutzer im Programm auswählen kann. Dabei greift das Werkzeug auf eine Ontologie zu, welche alle möglichen Begriffe zum Einsetzen in die Schablone enthält. Die Eintragungen des Nutzers stellen dann eine Anforderung dar, die in der Ontologie gespeichert wird.

In [FH14] wird das Vorgehen für die Dokumentation und logische Einordnung von funktionalen Anforderungen mithilfe von Sprachschablonen beschrieben. Dazu werden aus den Anforderungen für jedes Gesamtsystem zunächst die einzelnen Komponenten benannt und anhand ihrer Eigenschaften gespeichert. Die verschiedenen Komponenten des Modells können dann in Beziehung zueinander gesetzt werden. Auf Basis der einzelnen Komponenten und der Beziehungen kann dann ein Netz aus allen Komponenten des Gesamtsystems erstellt werden. Zur Erstellung dieser formalen Modelle mittels verschiedener Sprachschablonen haben sie das Werkzeug *ReqPat* entwickelt, das als Erweiterung in Anforderungsmanagament-Programmen wie *IBM Rational-DOORS* integriert werden kann. Ferner kann zur grafischen Darstellung des erstellten Modells ein Export aus ReqPat in UML-Diagramme erfolgen. Dies deckt jedoch nur funktionale Anforderungen ab und kann daher nicht für vollständige Lastenhefte verwendet werden.

Sampaio et. al. stellen in [AS05] den *EA-Miner* vor, der bei der Identifizierung von Systemaspekten hilft, die dann in Anforderungen überführt werden. Dazu modellieren sie inhaltliche Bezüge von Anforderungen zueinander auf Basis von NLP und Sprachschablonen, die der Nutzer im Programm anlegen kann. Dazu nutzen sie sog. *aspect-orientated-Requirements-Engineering* (AORE), das Anforderungen anhand verschiedener Use-Cases bzw. Blick-

punkten auf das Gesamtsystem identifiziert und einordnet. Dieser Ansatz hilft insbesondere dann, wenn mehrere Autoren aus verschiedenen Fachbereichen unabhängig Anforderungen definieren.

[CA15] beinhaltet einen Vorschlag von Arora et. al. für das automatisierte Überführen von natürlichsprachlichen Anforderungen in Sprachschablonen mittels NLP. Sie stellen eine Methode vor, die sprachliche Muster in den Anforderungen erkennt und daraus eine Grammatik in Backus-Naur-Form (BNF) erzeugt. Diese Grammatik kann somit zulässige Sätze in einem Anforderungsdokument anhand der Satzbausteine erzeugen bzw. finden. Durch die Übersetzung in eine JAPE-Regel kann auf dieser Basis die Erkennung der gefundenen sprachlichen Regeln durch das NLP-Werkzeug GATE erfolgen, das die Konformität der Anforderungen auf Basis der Regeln prüft.

Heßeler et. al untersuchen in [HE13] formale Prozesse für das Anforderungsmanagement in Unternehmen. Sie zeigen die möglichen Verbesserungen des projektbezogenen Requirements-Engineerings durch die Definition verschiedener Rollen für die beteiligten Entwickler auf. Dabei beschreiben sie explizit Kriterien für die Auswahl von Tools für eine Werkzeugunterstützung der Requirements-Engineers in Firmen. Konkret beschreiben sie Werkzeuge für die Unterstützung der Formulierung von Anforderungen bzw. für die Verifikation von Anforderungen mithilfe grafischer Modelle. Das Tool *DOORS Telematic* dient dabei zur Sammlung von Anforderungen und der Verknüpfung durch Relationen in einer Datenbank, die alle Anforderungen hierarchisch enthält. Tau Generation2 wird für die Visualisierung von Systemeigenschaften in UML2.0-Diagrammen und für den Export als DOORS-Objekte in die Anforderungs-Datenbank genutzt.

Kapitel 3

Betriebliches Umfeld - Use-Case HELLA

In diesem Kapitel wird der betriebliche Kontext dieser Arbeit genauer vorgestellt. Der im folgenden skizzierte Anwendungsfall des Automobilzulieferers HELLA stellt sie Basis für die später vorgestellten Werkzeuge dar. Die Evaluation und das Konzept für diese Werkzeuge erfolgte vor Ort mit den Experten für Anforderungsmanagement bei HELLA. Daher wird im Folgenden insbesondere der Umgang mit Anforderungen in der Firma, als Basis für die softwaretechnische Entwicklung, beschrieben.

3.1 Allgemeines zu HELLA

In diesem Abschnitt wird zur besseren Einordnung die Firma HELLA kurz beschrieben. Dies beinhaltet die Geschichte und die momentane Aufstellung des Unternehmens. [HE18] [HE19]

Geschichte

Die HELLA GmbH & Co. KGaA wurde 1899 von Sally Windmüller unter dem damaligen Namen „Westfälische Metall-Industrie Aktien-Gesellschaft(WMI)“ in Lippstadt gegründet, wo sich bis heute auch ihr Hauptsitz befindet.

Die hergestellten Produkte beschränkten sich zu diesem Zeitpunkt auf Ballhupen und Lampen für Kutschen. Im Jahr 1908 tauchte mit dem ersten entwickelten elektrischen Scheinwerfer auch der Name *HELLA* zum ersten

mal als Warenzeichen auf. Dieser wurde jedoch erst 1986 offiziell in die Firmenbeschreibung aufgenommen. Es wird davon ausgegangen, dass der Name eine Mischung aus der Anlehnung an den Namen von Windmüllers Frau Helene und dem Wortspiel mit „heller“ ist.

Die erste Tochtergesellschaft gründete HELLA 1951 in Todtnau und dem Namen „Metallwerke Todtnau“. 1961 begann das Unternehmen sich auch international aufzustellen. Dies geschah durch die Gründung der ersten internationalen Produktionsstätte in Mentone in Australien. Das Zentrallager, welches auch heute noch unter dem Namen *HELLA Distribution GmbH* besteht, wurde 1973 in dem Lippstadt nahegelegenen Erwitte errichtet.

Aktueller Stand des Unternehmens

Die HELLA ist in die vier Geschäftsbereiche *Elektronik*, *Scheinwerfer*, *Aftermarket* und *Special Applications* unterteilt und verfügt über 125 Standorte in mehr als 35 Ländern.

Nach dem Geschäftsjahr 2017/2018 werden insgesamt über 40.000 Mitarbeiter beschäftigt. Davon arbeiten mehr als 7000 Mitarbeiter in der Forschung und Entwicklung.

So konnte HELLA in diesem Geschäftsjahr fast 7,1 Milliarden Euro Umsatz verbuchen, wovon ca. ein drittel außerhalb des europäischen Marktes erzielt wurde. Damit gehört HELLA international zu den Top 40 der weltweiten Automobilzulieferer und befindet sich unter den Top 100 der größten deutschen Industrieunternehmen. [HE19]

3.2 Requirements-Engineering innerhalb der Firma

In diesem Abschnitt wird das Requirements-Engineering (RE) der Firma HELLA genauer behandelt. Dies beinhaltet eine Vorstellung der zuständigen Abteilungen, sowie einen Überblick über HELLA-internen RE-Prozesse. Des Weiteren werden die Herausforderungen innerhalb dieser Prozesse anhand von Use-Cases (Anwendungsfällen) aufgeführt. In den darauffolgenden Abschnitten werden daraus Anforderungen an eine Lösung erhoben, welche mit den Konzepten der Werkzeuge umzusetzen versucht wird.

Die getroffenen Aussagen stützen sich dabei auf die angegebenen Lektüren, Unterlagen der Firma und/oder Gespräche mit firmeninternen Experten.

3.2.1 Abteilung für RE

Wie bereits erwähnt, existiert bei HELLA eine auf Requirements-Engineering spezialisierte Fachabteilung. Die Abteilung ist unter anderem für folgende Aufgaben verantwortlich:

1. Standardisierung von Prozessen und reibungsfreies Verknüpfen von Arbeitsabläufen
2. Einführung von neuen Standards und Hilfe bei der Umsetzung dieser
3. Entwicklung neuer und effizienterer Arbeitsmethoden zur Einsparung von Zeit und Kosten
4. Entwicklung, Einführung und Verwaltung neuer (Software-) Werkzeuge, welche in der Lage sind, einen Mehrwert für die Firma zu erzeugen
5. Empfangen Firmen-externer (Anforderungs-) Dokumente, welche vor dem Weiterleiten an Fachabteilungen gegebenenfalls zunächst auf die firmeninternen Standards angepasst werden müssen
6. Unterstützung von Mitarbeitern und Projektteams bei dem Umgang und der Verarbeitung von Lastenheften

Oftmals steht diese Abteilung in Verbindung mit den Abteilungen der Vorentwicklung. Dabei spielt vor allem die System-Sparte der elektronischen Vorentwicklung eine wichtige Rolle. Hier versuchen Mitarbeiter mit Hilfe von OEM-Lastenheften, einen kompletten Überblick eines Systems zu erlangen, um darauf bezogene Projekte besser einschätzen und durchführen zu können.

Diese Aufgabenstellung bringt besondere Herausforderungen mit sich, da Anforderungsmanagement ein komplexes Aufgabenfeld ist. Hierzu gehören unter anderem eine hohes Maß an Fachkenntnissen für alle relevanten Bereiche des Produkts, ein gründliches und somit aufwendiges Erfassen aller Aspekte des Produkts und die Fähigkeit, mit Hilfe dieses Wissens mögliche Probleme oder Fehlerquellen des Produkts zu erfassen. Mitarbeiter haben zum Ausdruck gebracht, dass diese Aufgaben besondere Konzentration und Auffassungsgabe erfordern.

Diese Aufgabe kann jedoch von Requirements-Engineers (wie in 6. erwähnt) unterstützt werden. Da es erforderlich ist, dass ein einzelner Mitarbeiter der Vorentwicklung einen Überblick erhält, bietet eine klassische

Arbeitsteilung, bei welcher der möglicherweise enorme Umfang dieser Lastenhefte auf verschiedene Mitarbeiter aufgeteilt wird, keine Lösung.

Dennoch können klar formulierte, gut sortierte und strukturierte Anforderungen die Arbeit der Systemingenieure stark vereinfachen. Dies reduziert ein wenig den benötigten Arbeitsaufwand des einzelnen Mitarbeiters, verringert aber vor allem die Herausforderungen bei der Analyse. Des Weiteren kann die Fachabteilung für RE bei Rückfragen als Sprachrohr zwischen dem OEM und den internen Mitarbeitern fungieren, um einen effizienteren Ablauf zu gewährleisten.

3.2.2 RE-Prozess

In diesem Abschnitt wird ein Überblick über den RE-Prozess von HELLA geboten. Hierzu wird dieser zunächst in den Kontext eines Entwicklungsprojektes für ein neues Produkt eingeordnet. Daraufhin werden die einzelnen Schritte des Requirements-Engineering genauer beleuchtet.

Einordnung in den Entwicklungsprozess

Üblicherweise steht der RE-Prozess direkt zu Beginn jeder Entwicklung. Entweder gibt es ein neues Produktkonzept, zu welchen die Anforderungen abhängig von den geplanten Aufgaben und Eigenschaften des Produktes erhoben werden, oder es wird ein Auftrag eingereicht, zu welchem bereits ein komplettes Anforderungsdokument existiert, welches es dann umzusetzten gilt.

In der Ausgangssituation liegt also ein Dokument vor, in welchem alle Anforderungen an das Produkt zusammengefasst sind. Dies bedeutet auch, dass die Entwicklung erfolgreich war, sobald alle dieser Anforderungen durch das Produkt erfüllt werden. Dies muss jedoch auch für jene Anforderungen gelten, welche sich innerhalb der Entwicklung ändern bzw. hinzugefügt werden.

Dies verdeutlicht, dass sich der Prozess des Requirements-Engineering nicht auf den Beginn des Gesamtprozesses beschränken lässt. Dennoch befindet sich dort der wichtigste Bereich des RE, da zumindest in hohem Maße feststehen muss, was es zu entwickeln gilt, bevor mit der Entwicklung begonnen werden kann.

Abbildung 3.1 zeigt die planmäßige Einordnung von RE in den Gesamtprozess der Produktentwicklung bei HELLA.

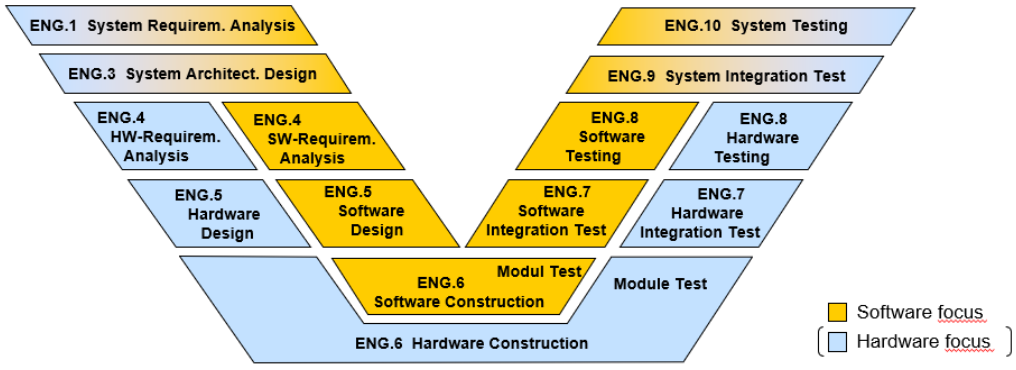

Abbildung 3.1: Hella-Prozesse mit ISO 26262. RE in den Stufen ENG. 1 und ENG. 4 [EA19]

Hierbei ist anzumerken, dass in dem Modell zwischen den Systemanforderungen (*system requirements*) und den funktionalen Anforderungen (*functional requirements*) unterschieden wird.

Die Systemanforderungen[1] beschreiben Anforderungen, die das System in der Rolle als Umfeld für die Funktionalitäten zu erfüllen hat. Dazu gehören üblicherweise Eckdaten des Systems, welche eher generelle Informationen bieten. Eine beispielhafte Anforderung könnte etwa lauten: „Das System muss einer Spannung von bis zu 48V standhalten.“ oder „Das System soll den Standards XY der Norm Z entsprechen.“

Funktionale Anforderungen sind hingegen etwas spezifischer und beschreiben das genaue Handeln einzelner Komponenten und die Ausführung von Funktionalitäten. Hierzu könnten beispielhafte Anforderungen wie folgt aussehen: „Wenn eine Spannung von 24V erreicht wird, soll Komponente XY die Funktion Z ausführen.“ oder „Die Funktion Z lässt die Leuchten A und B für 10 Sekunden leuchten.“

Die Unterscheidung zwischen den verschiedenen Anforderungstypen kann jedoch sehr problematisch sein, da unterschiedliche Definitionen existieren können. Da es also keine einheitliche Abgrenzung gibt und Anforderungen in natürlicher Sprache verfasst werden, ist es praktisch unmöglich, alle Anforderungen eines Dokuments eindeutig den Kategorien zuzuweisen. Dies ist jedoch auch nicht zwangsläufig nötig, wenn man sich dazu entscheidet, alle

[1] Häufig als nicht-funktionale Anforderungen bezeichnet bzw. von den funktionalen Anfordeurngen abgegrenzt

Anforderungen zunächst gleich und dann abhängig von Inhalt zu verarbeiten. Daher verzichten auch die später vorgestellten Konzepte auf eine erzwungene Unterscheidung verschiedener Anforderungstypen.

Werkzeuge im RE-Prozess

Ein klassischer Requirements-Engineering-Prozess der RE-Abteilung beginnt damit, dass von einem OEM ein Lastenheft zu einem bestimmten Auftrag bei der HELLA eingeht. Dieses Lastenheft erhält die PMT-Abteilung üblicherweise in Form einer *DOORS*-Datei (Dynamic Object Oriented Requirements System).

DOORS ist ein Datenbankmanagementsystem für Anforderungen, welches im Requirements-Engineering verwendet wird. Mithilfe dieses Werkzeugs werden Anforderungsdokumente gespeichert. Es ist vergleichbar mit einer herkömmlichen Datenbank. Dort wird in einer Tabellenstruktur der gesamte Inhalt eines Anforderungsdokumentes hinterlegt. Die einzelnen Anforderungen sind dabei zeilenweise angeordnet. Zusätzlich befinden sich in verschiedenen Spalten ergänzende Informationen zu der jeweiligen Anforderungen, wie z.B. die eindeutige Identifikationsnummer oder die Unterscheidung zwischen Anforderung und genereller Information.

DOORS bietet zusätzlich eine Vielzahl von Erweiterungen zur Verarbeitung sowie die Möglichkeit, das DOORS-Objekt als PDF-Datei zu exportieren. Da man Informationsverlust verhindern möchte, werden auch alle Zusatzinformationen aus den Spalten der Tabelle in der extrahierten Datei mit aufgenommen, was jedoch zu stark fragmentierten Dokumenten bzw. Texten führt. Dies ist unter anderem ein Problem, welches im Kapitel Implementierung noch einmal aufgegriffen wird.

Eine der oben erwähnten Erweiterungen ist das von HELLA entwickelte *ReqPat*. Dieses wird dazu verwendet Teile von Anforderungsdokumenten auf Konsistenz zu überprüfen. Dazu werden sogenannte *Boilerplates* als Satzschablonen verwendet um die natürliche Sprache der Anforderungen in eine formalisierte Form zu bringen. Daraus kann dann ein prüfbarer Graph erstellt werden. Grundsätzlich bestehen hier Parallelen zu dem später vorgestellten Konzept, jedoch ist ReqPat dabei auf bestimmte (funktionale) Anforderungen beschränkt.

Die Anforderungen aus dem DOORS-Objekt werden nun von den RE-Experten überprüft, ggf. an firmeninterne Standards angepasst oder in eine formalisierte Form überführt und dann für die verantwortliche Abteilung

freigegeben. Dieser Ablauf ist in der Abbildung 3.2 präziser dargestellt.

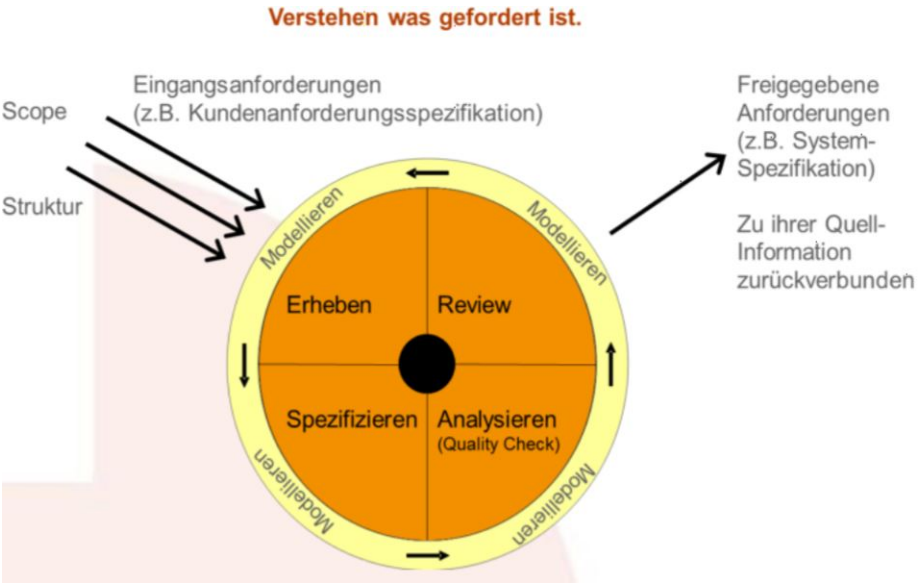

Abbildung 3.2: Prozess der Aufbereitung von OEM-Anforderungen für weitere Verwendung [HP12]

Dies kann etwa mit dem beschrieben ReqPat geschehen. Da dieses Werkzeug jedoch nur für funktionale Anforderungen entwickelt wurde, werden nicht-funktionale Anteile im Lastenheft damit nicht abgedeckt.

3.3 Anforderungen im RE-Management

Dieses Kapitel beschäftigt sich mit der Motivation und den Rahmenbedingungen für die später in dieser Arbeit skizzierte Werkzeugunterstützung. Wesentliche Erkenntnisse entstanden dabei in Rücksprache mit dem Requirements-Experten Colin Hood, der als externer Berater für die RE-Abteilung von HELLA arbeitet.

Auf dieser Basis konnten Anforderungen an die Werkzeuge für den Einsatz zur Transformation von Lastenheften während des RE erhoben werden.

Mithilfe der Konzepte kann, wie später gezeigt wird, ein Beitrag zur Verbesserung des Anforderungsmanagements im Unternehmen erbracht werden.

Das Ziel ist es, die im vorigen Abschnitt benannten Komplikationen beim Umgang mit Anforderungen zu bewältigen.

3.3.1 Kriterien der Qualität von Anforderungen

Das Ziel von Abteilungen, die sich mit Requirements-Engineering in Unternehmen beschäftigen, ist die Verbesserung der Qualität von Anforderungsdokumenten für die spätere Verwendung. Die Requirements-Experten sind meist zwar nicht direkt am Entwicklungsprozess beteiligt, unterstützen die Experten in der Entwicklung jedoch beim Verständnis und der Umsetzung der Lastenhefte.

Wie im vorigen Abschnitt beschrieben, sind in den Anforderungen oftmals nicht alle Informationen für ein vollständiges Verständnis beim Zuieferer enthalten, wenn dort mit der Entwicklung begonnen werden soll. Diese Schwierigkeiten entstehen durch Herausforderungen bei der Kommunikation, z.B. aufgrund von Geheimhaltung oder fehlender Informationen während des Entwicklungsprozesses. Es kann vorkommen, dass Anforderungen nicht vollständig sind und Lastenhefte Inkonsistenzen enthalten. Requirements-Experten beim Zulieferer zielen also auf eine Formalisierung der Anforderungen aus einem Lastenheftes für die nachgelagerte Entwicklung ab, um solche Missverständnisse zu vermeiden.

Qualitätsmerkmale

In diesem Zusammenhang existieren Qualitätsmerkmale, die gemäß [ZG02] wesentliche Kriterien für die Bewertung von Anforderungen darstellen. Requirements-Experten orientiereren sich dabei an der Unterscheidung der drei wesentlichen Qualitätsmerkmale *Consistency-Correctness-Completeness*, die sich wie folgt auffassen lassen:

1. *Consistency - Einheitlichkeit*: Die Anforderungen dürfen sich an verschiedenen Stellen im Lastenheft nicht widersprechen, sondern sollen sich vielmehr an geeigneten Stellen ergänzen. Beispielsweise darf später keine Anforderung im Lastenheft enthalten sein, die im direkten Widerspruch zu anderen Anforderungen steht, die bereits zuvor anders spezifiziert wurden. Entwickler, die nur die für sie relevanten Stellen des Lastenheftes erhalten, würden somit fundamental unterschiedliche

Produkte entwerfen. Inkonsistenzen und Abhängigkeiten zwischen Anforderungen bedürfen also Rücksprache mit dem OEM bei der Bewertung.

2. *Completeness - Vollständigkeit*: Es sollten zu allen relevanten Merkmalen des Endproduktes entsprechende Anforderungen existieren. Das Lastenheft sollte also alle Produktmerkmale inhaltlich abdecken, die für die Entwicklung beim Zulieferer relevant sind. Wenn der OEM etwa konkrete Vorstellungen zum Design des Produktes hat, diese jedoch im Lastenheft fehlen, erhält der Zulieferer diese Informationen nicht (etwa implizite Informationen oder Erläuterungen). Es gehen mitunter entscheidende Informationen und Kriterien für den Entwicklungsprozess verloren.

3. *Correctness - Richtigkeit*: [ZG02] beschreibt die Richtigkeit von Anforderungen als ein prinzipiell vages Konzept aus zwei verschiedenen Standpunkten:

   (a) Formal betrachtet ist ein Anforderungsdokument korrekt, wenn es einheitlich und vollständig ist, d.h. frei von Widersprüchen und inhaltlich alles beinhaltet, was als „wahr“ über das Produkt bezeichnet werden kann.

   (b) Praktisch erfolgt die Bewertung der Richtigkeit eines Dokumentes anhand von Kriterien, die Stakeholder auswählen. Das Lastenheft wird demnach als richtig empfunden, wenn es Bedürfnisse zufriedenstellend abbildet. Aus Sicht der Kostenkalkulation wird etwa beurteilt, ob die Kosten durch die zu verwendenden Materialien, die im Lastenheft vermerkt sind, gering gehalten werden.

Es lassen sich gemäß [ZG02] weitere Abhängigkeiten zwischen den drei Merkmalen beobachten, die die Umsetzung in konkreten Dokumenten erschweren können. 1. und 2. stehen demnach häufig in Wechselwirkung zueinander.

Zur Herstellung von Konsistenz im Lastenheft existiert der Ansatz, einzelne Anforderungen entweder schwerer zu gewichten als andere, oder unpassende Anforderungen komplett zu entfernen. Dies führt jedoch zu verminderter Vollständigkeit.

Informationsmangel kann zwar ebenfalls recht einfach durch das Hinzufügen neuer Anforderungen ausgeglichen werden, jedoch muss dabei darauf

geachtet werden, dass die Spezifikation dann nicht zu umfänglich ausfällt. Die Verbesserung der Vollständigkeit kann somit auch die Richtigkeit des beschriebenen Produktes beeinflussen, sollten entsprechende Informationen ergänzt werden.

Beispiel problematischer Anforderungen

Zur Verdeutlichung des resultierenden Aufwands bei der Herstellung hochqualitativer Anforderungen soll das folgende fiktive Beispiel dienen. Die Analyse der Vollständigkeit soll hier vernachlässigt werden, da es sich nur um einen minimalen Ausschnitt handelt. Folgende funktionale Anforderungen könnten in einem Lastenheft enthalten sein:

```
1: Design: Das Gehäuse soll rot gestrichen sein.
...
2: Materialien: Das Gehäuse soll nach außen aus 1mm starkem,
rostfreiem Stahl bestehen.
```

Bei der Prüfung auf Vollständigkeit fallen zunächst keine Konflikte auf, jedoch ist `1` recht vage formuliert. Genaue Angaben zur verwendeten Farbe und der genaue Farbton fehlen in der Anforderung. Dieses Problem wird bei der Analyse der Abhängigkeiten offensichtlich, da die Anforderung „rostfreier Stahl“ die Verwendung entsprechender Farbe notwendig macht.

Eine solche Anforderung könnte etwa wie folgt lauten:

```
...
3. Materialien: Die Lackierung erfolgt mit Metallschutzlack.
```

Isoliert betrachtet wäre diese Anforderung auch für alle Lackierungen denkbar, nicht nur für das Gehäuse. Daher muss das Ausmaß dieser Anforderung auf das Gehäuse beschränkt werden, etwa durch `1.1` als zugehörig zu `1`.

Aus Perspektive der Richtigkeit könnten je nach Stakeholder, der die Bewertung vornimmt, weiter spezifiziert werden. Ein Designer könnte etwa einen genauen Farbcode zur Farbanforderung „rot“ hinzufügen. Aus Perspektive der Kosten könnte `2` auch komplett gegenüber einer günstiger umzusetzenden Anforderung entfallen, die etwa „Kunststoff“ anstelle von „rostfreiem Stahl“ vorschreibt.

Problematisch ist bei der Betrachtung des Beispiels auch, dass in der Re-

gel interdisziplinäre Teams jeweils nur einen Abschnitt des Lastenheft erhalten und entwerfen. Sie verfügen somit nur über einen Teilbereich des Wissens bzw. nur einen Teil der Anforderungen. Es könnte also der Fall eintreten, dass das Design-Team nur `1` und das Ingenieur-Team nur `2` erhält.[2]

Die Relevanz dieser drei Kriterien für das betriebliche Anforderungsmanagement wurde auch in Rücksprache mit internationalen Requirements-Experten bei HELLA betont. Auf dieser Basis realisieren die dortigen Teams Analysen der Projektlastenhefte.

3.3.2 Kriterien zur Herstellung qualitativer Anforderungen

Harant und Poethen stellen in [HP12] weitere Maßnahmen für das Requirements-Management auf Basis des *HOOD Capability Models* (HCM) vor. Dabei ordnen sie die verschiedenen Schritte anhand eines mehrstufigen Prozesses ein, der in jedem Entwicklungspojekt von Requirements-Experten durchgeführt werden soll (siehe Abb. 3.3).

Diese Expertengruppe dient dabei abstrakt zur Unterstützung der tatsächlichen Entwicklungsarbeit. Weitere Merkmale von qualitativen Anforderungen werden durch das HCM beschrieben und verschieden gewichtet.

Dabei unterscheidet das Modell zwischen 3 verschiedenen Qualitätsstufen von Anforderungen. Diese werden jeweils durch verschiedene Kriterien dargestellt. Das Ziel der Arbeit der Experten bei HELLA stellt demnach das Herstellen einer möglichst hohen Qualitätsstufe der Lastenhefte dar. Im besten Fall machen die Experten die Anforderungen somit *vollständig & verfolgbar*.

3.3.3 Umgang mit natürlicher Sprache in Lastenheften

Problematische Anforderungen lassen sich meist auf falsche Annahmen der Stakeholder beim OEM zurückführen, die implizite Annahmen bei der Formulierung von Anforderungen treffen und diese Annahmen nicht explizit erneut aufführen. Ergänzenden Informationen zu Anforderungen können je-

[2] Hier ist kurz ein naiver Ansatz zur Verdeutlichung der Komplikationen beschrieben. [ZG02] beschreibt detailliert die Prüfung der Qualitätsmerkmale anhand verschiedener Verfahren und wie diese verbessert werden können. Mithilfe von Werkzeugen können diese auf Basis automatisierter Textanalyse der Anforderungen umgesetzt werden.

Abbildung 3.3: Überblick der 3 Qualitätsstufen von Anforderungen im *Requirements-Management & Engineering* (RM & E) aus [HP12]

doch, wie zuvor gezeigt, durchaus andersartig kategorisiert im Lastenheft vermerkt sein.

Dies resultiert laut [PR15] daher, dass die verschiedenen Autoren des Lastenheftes beim OEM als selbstverständlich erachten und diese nicht nochmal für externe Leser hinzufügen. Es stellt sich jedoch die Frage, ob unterschiedliche Menschen jemals ein verlässlich gleiches Verständnis der gleichen Sache haben.

Colin Hood formuliert Anforderungen grundsätzlich als „individuelle Bedürfnisse“, die immer abhängig von dem Autoren sind.[3] In [KR93] wird die Verwendung von Sprachschablonen für Lastenhefte im Vergleich zu natürlicher Sprache evaluiert. Die grundsätzliche Eignung natürlicher Sprache für die Formulierung von Anforderungen wird dort bezweifelt.

Hood stellte die Eingrenzung der Sprache, die für Projektlastenhefte genutzt werden darf, als eine vielversprechende Möglichkeit vor, sprachliche Eigenheiten der Autoren aus verschiedenen Bereichen effektiv zu vermeiden.

[3]Im Rahmen unserer Arbeit konnten wir mehrmals Interviews zur Ausrichtung bzw. dem Ansatz unserer Prototypen führen.

Diese Transformation natürlichsprachlicher Anforderungen in formale Sprache als Schablonen (Boilerplates) stellt auch das Ziel des ersten entworfenen Werkzeugs dar.

3.4 Anwendungsfall bei HELLA

Der Anwendungsfall für die zu entwickelnden Werkzeuge ist ähnlich zu dem oben beschriebenen. Es wird davon ausgegangen, dass die HELLA eine Anfrage für ein Produkt von einem OEM erhält. Es ist jedoch bekannt, dass ein ähnliches Produkt, etwa eine frühere Version, bereits von HELLA entwickelt wurde. Somit liegt ein Lastenheft eines OEM in natürlicher Sprache vor. Außerdem existiert zu dem alten Produkt der HELLA bereits ein Pflichtenheft in formalisierter Form.

Ziel ist es, die beiden Anforderungsdokumente inhaltlich zu vergleichen, um alle Unterschiede der beiden Produkte zu ermitteln. Diese könnten dann etwa genutzt werden, um eine sehr frühe und recht präzise Kostenabschätzung der neuen Produktentwicklung an den OEM geben zu können.

Hierzu muss jedoch zunächst das Lastenheft des OEM in eine vergleichbare und semi-formalisierte Form gebracht werden. Erst sobald sich also beide Anforderungsdokumente in einer gleichartigen Form befinden, kann ein direkter Vergleich der von den Dokumenten beschriebenen Produkte stattfinden. Diese Heterogenität von Anforderungsdokumenten ist in der Automobilindustrie vor allem bei der Beziehung zwischen OEM und Zulieferer ein bekanntes und wiederkehrendes Problem.

Daher wird zunächst ein Werkzeug benötigt, welches mit kleinstmöglichem Nutzeraufwand und größtmöglicher Sicherheit ein natürlichsprachliches Anforderungsdokument formalisieren kann.

Verarbeitet man mit diesem Werkzeug das Lastenheft das OEM, so erhält man zwei homogene Dokumente. Hierzu soll außerdem ein Werkzeug entwickelt werden, welches in der Lage ist, zwei solcher Dokument zu vergleichen und Unterschiede darzustellen.

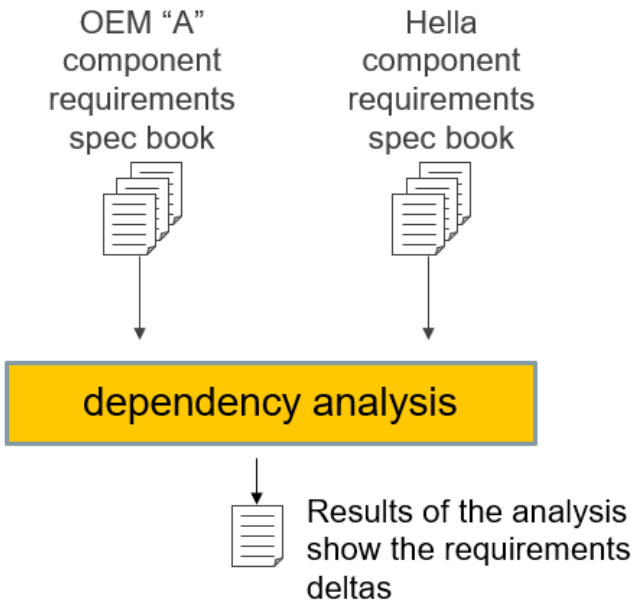

Abbildung 3.4: Anwendungsfall des R2BC und DA [eigene Darstellung]

Die Ansätze und Konzepte solcher Werkzeuge sind in dem folgenden Abschnitt genau beschrieben.

3.5 Ansatz und Konzept der Werkzeuge

Wie bereits unter beschrieben, existiert eine Vielzahl an Arbeiten, die sich mit der verlässlichen Überführung der Anforderungen in natürlicher Sprache in Sprachschablonen beschäftigen. Dazu werden Methoden aus dem NLP und Ontologien genutzt, um den Text der Anforderungen zunächst genau zu annotieren. Dieser Ansatz stellt die Basis für die Vorstellung unserer Werkzeuge in diesem Kapitel dar.

In [ZH17] wurde das Potential einer Verbesserung von RE-Prozessen in Firmen durch Werkzeuge aufgezeigt. Die in den vorigen Kapiteln beschriebenen Anforderungen aus dem Requirements-Management schaffen somit die Basis für den konkreten Entwurf von Werkzeugen, die entsprechende Funktionalitäten bereitstellen.

3.5.1 Anforderungen an die Entwicklung

Für den Entwurf der Prototypen stellt sich einerseits die Frage, an welcher Stelle im abstrakten Modell ein Potential für Werkzeugunterstützung entsteht. Auf der anderen Seite müssen aber auch Zielgruppen bzw. Stakeholdern im tatsächlichen Betrieb bei HELLA definiert werden, die nachher mit der Software interagieren und diese täglich nutzen.

Für die Entwicklung unseres Werkzeugs wurden in Zusammenarbeit mit RE-Experten einige Ziele identifiziert, die eine Unterstützung für die Arbeit ausmachen würden:

1. Zeitersparnis bei Arbeit mit dem Werkzeug, da der Aufwand einer manuellen Prüfung reduziert wird. Dieser Aufwand wird an das Werkzeug abgegeben.

2. Hoher Automatisierungsgrad bei der Verarbeitung, um Konflikte in den Anforderungen im Voraus aufzulösen.

3. Hoher Abdeckungsgrad verschiedener Problemfälle, da eine einfache Wiederverwendbarkeit den Aufwand zur Anpassung an andere Domänen reduziert.

4. Anforderungen an NLP-Werkzeuge umsetzen, da eine präzise sprachliche Analyse eine wesentliche Grundlage für die Verlässlichkeit der Verarbeitung darstellt.

3.5.2 Zuordnung einzelner Werkzeuge

Verschiedene Verfahren, bereits bei der Spezifikation von Anforderungen im Zusammenhang eines Lastenheftes durch Werkzeuge zu unterstützen, sind unter Abs. 2.3 beschrieben. Es existieren jedoch durch die Besonderheiten im geteilten Entwicklungsprozess zwischen OEM und Zulieferer besondere Herausforderungen für das RE, welche Potential für eine Werkzeugunterstützung bieten.

Wie bereits in Kap. 3.3 beschrieben, resultieren die Probleme auf Seiten des Zulieferers beim Verständnis der Anforderungen aus dem Umstand, dass dort ein Lastenheft in natürlicher Sprache vorliegt. Dieses verursacht einerseits einen Arbeitsaufwand durch die Probleme beim Umgang mit natürlicher Sprache wie z.B. sprachlichen Mehrdeutigkeiten (Abs. 2.1.1). Andererseits

können enthaltene Anforderungen teilweise nicht korrekt bzw. von niedriger Qualität sein, was zu inhaltlichen Problemen für die Produktentwicklung führt.

Auch kann es mehrere Versionen der Spezifikation eines Produktes im Laufe der Zeit geben, deren Unterschiede erkannt werden müssen. Auswirkungen der Überarbeitung des Produktes auf Basis einer neuen Version des Lastenheftes müssen kategorisiert werden.

Es gilt also, Werkzeuge zu entwerfen, die insbesondere bei der Lösung von sprachlichen Problemen und bei der Identifikation von Unterschieden zwischen Anforderungen in Lastenheften unterstützen.

Daher werden im Folgenden die Ansätze aus [ZH17] beschrieben, die eine Transformation von bereits spezifizierten Anforderungen in bestehenden Lastenheften in eine vordefinierte und eindeutige formalere Sprache vorschlagen. Dieser Schritt soll für jedes neu eintreffende Lastenheft durchgeführt werden, damit dieses dann standardisiert vorliegt. Die zweite Software-Komponente dient zum automatisierten Vergleich mehrerer verschiedener Lastenhefte.

Diese Ansätze haben zur konkreten Entwicklung von zwei Werkzeugen geführt, dem *Requirements-to-Boilerplate-Converter* (R2BC) und dem *Delta-Analyser* (DA). Diese identifizierten Software-Artefakte werden für die Verwendung in einer prototypischen Werkzeugkette hintereinander angeordnet. Somit erfolgt auf Basis der Konvertierung in Schritt 1 die Delta-Analyse in Schritt 2.

R2BC

Zu Beginn des Verarbeitungsprozesses liegen die Lastenhefte in natürlicher Sprache vor, die der Zulieferer vom Kunden erhalten hat. Im ersten Schritt werden diese natürlichsprachlich formulierten Anforderungen durch den R2BC in Sprachschablonen überführt.

Nach der Verarbeitung liegt das Lastenheft des OEM somit in semiformaler Sprache vor. Diese Version des Lastenheftes kann dann für den Entwicklungsprozess genutzt werden. Das Ziel des R2BC ist es also, die Lesbarkeit bzw. das Potential der Anforderungen durch Formalisierung zu verbessern. Auf dieser Basis kann dann der Inhalt des Lastenheftes besser durch die Entwickler beim Zulieferer verstanden werden. Der Prozess ist in Abb. 3.5 dargestellt.

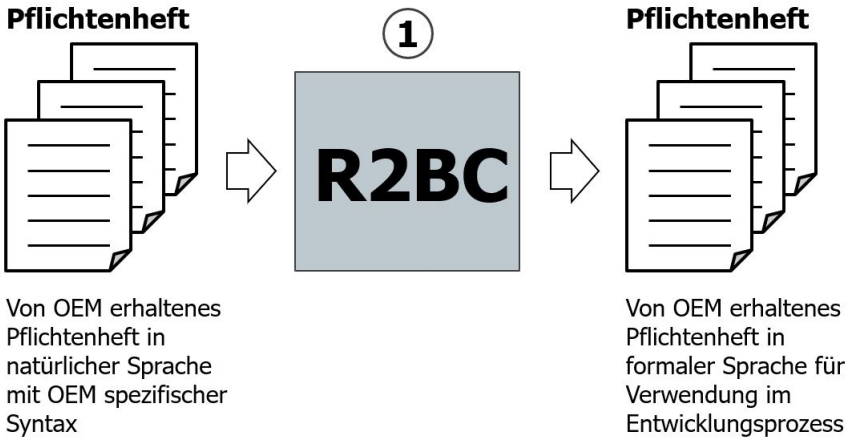

Abbildung 3.5: Szenario für Unterstützung durch R2BC-Komponente nach [ZH19]

DA

Diese angepasste Syntax der Anforderungen ermöglicht auch einen Vergleich mehrerer Lastenhefte anhand der Struktur der einzelnen Anforderungen. Diese Unterschiede können durch den DA identifiziert und in Form eines Delta-Reports ausgegeben werden. Unterschiede können anschließend etwa für einen RE-Experten auf dem Bildschirm bzw. in eine Datei ausgegeben werden. Der Ablauf ist in Abb. 3.6 dargestellt.

Alternativ können Abweichungen auch automatisiert durch eine Integration der Anpassungen in das interne Lastenheft übernommen werden. Dazu müssen im Programm Verfahren zur Bewertung der Deltas bzw. Metriken zur Entscheidung zwischen Anforderungen hinterlegt sein. Auf Basis der Annotationen kann eine solche Kategorisierung erfolgen.

Angesichts der Aufgaben für die Werkzeuge stellt sich die Frage nach dem erreichbaren Grad der Automatisierung bei hoher Verlässlichkeit im Gegensatz zu möglichen erforderlichen Interaktionspunkten mit dem Nutzer.

Der prinzipielle Aufbau beider Werkzeuge wird in den folgenden Abschnitten beschrieben.

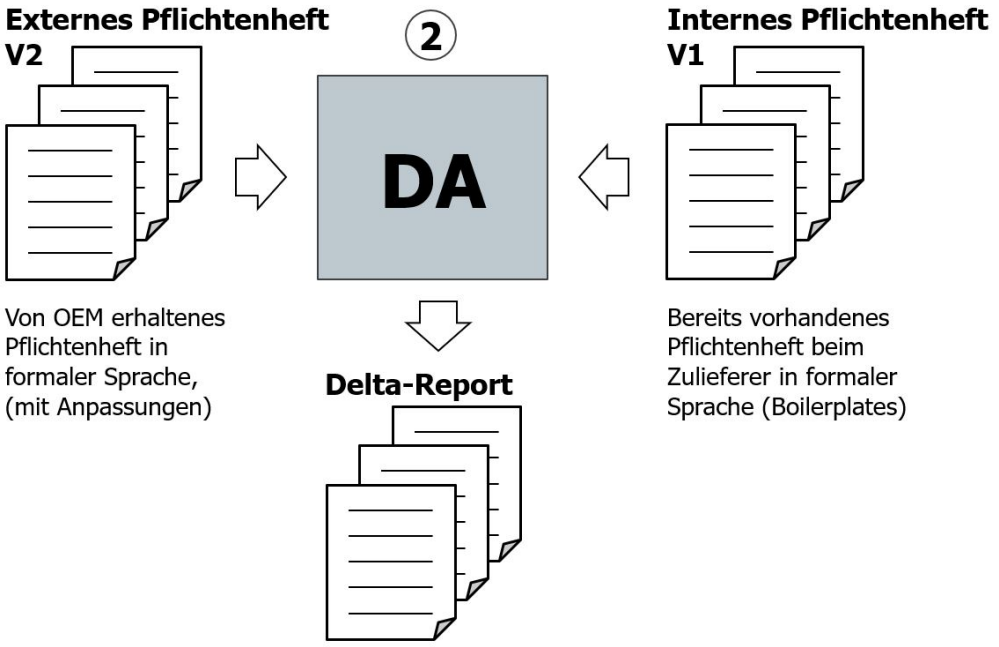

Abbildung 3.6: Szenario für Unterstützung durch DA-Komponente nach [ZH19]

3.5.3 R2BC-Funktionalität

Wie bereits zuvor beschrieben wurde, stellt der *Requirements-To-Boilerplate-Converter* die erste wesentliche Stufe bei der automatisierten Verarbeitung von Lastenheften dar. Die Textverarbeitung erfolgt dabei sukzessive bzw. satzweise aus dem Quelldokument in ein Zieldokument, dass am Ende alle Anforderungen in angepasster Form enthält.

Sprachschablonen

Dabei greift der R2BC auf eine semiformale Sprache zurück, die in verschiedenen Schablonen repräsentiert und im Programm gespeichert ist. Diese eingeschränkte Sprache ist zuliefererintern fest dokumentiert und wurde zuvor durch RE-Experten eindeutig definiert, was zu einer verbesserten Verständlichkeit führt. Sprachschablonen stellen demnach, im Kontrast zu natürlicher Sprache, eine eindeutige Deutung der Anforderungen sicher. Im besten Fall

wurden somit insbesondere sprachliche Mehrdeutigkeiten im Sinne der Atomarität von Anforderungen eliminiert bzw. aufgeschlüsselt.

Im Rahmen dieser Arbeit wurden zwei repräsentative Anforderungstypen aus dem betrieblichen Umfeld als Sprachschablonen (Boilerplate) implementiert. Dies beinhaltet eine Boilerplate, die durch ein System abgedeckte Funktionen beschreibt. Die zweite fügt dieser Funktionsbeschreibung zusätzlich noch eine Bedingung hinzu, unter der das System die Funktion ausführen soll.

Die orientieren sich dabei namentlich an den intern bei HELLA vorkommenden Boilerplates und wurden auch für die spätere Implementierung so genannt. Dort existiert eine Vielzahl verschiedener Boilerplate-Typen, die alle einen spezifischen Typ Anforderungen abdecken. Diese Schablonen beinhalten dabei immer feste Satzteile, wie etwa Satzzeichen, und Variablen für die Spezifikation der Anforderung. Dies ermöglicht die semi-formale Formulierung von Anforderungen.

Für den in dieser Arbeit entworfenen Prototypen wurden die Boilerplates *65c* und *85* implementiert.

1. *Boilerplate65c* Die Sprachschablone dient zur Erfassung von Anforderungen, die zu einem System eine Funktion spezifizieren. Dazu werden in einem Satz zwei Variablen benötigt. Die Schablone sieht wie folgt aus:

   ```
   The complete system <<Variable System>> shall perform
   <<Variable Funktion>>.
   ```

2. *Boilerplate85* Typ-85-Boilerplates stellen im Wesentlichen eine Erweiterung der Typ65c-Boilerplates dar. In dieser Schablone wird eine Bedingung als dritte Variable hinzugefügt:

   ```
   Under the condition: <<Variable Bedingung>> the actor:
   <<Variable Akteur>> shall <<Variable Funktion>>.
   ```

In Kap. 4 wird genauer erläutert, wie sich diese Schablonen in Programmen erzeugen lassen. Dazu wird das NLP-Werkzeug GATE genutzt, um zu einer Anforderung eine passende Boilerplate zu finden. Diese wird dann durch das Programm erzeugt, wobei die variablen Bestandteile identifiziert und passend in die Schablone eingefügt werden.

Der R2BC verarbeitet die Lastenhefte unter Zugriff auf diese formalen Sprachschablonen (*Boilerplates*) automatisiert und ein Eingreifen des Nutzers ist nur im Ausnahmefall erforderlich.

Programmablauf und Artefakte

Auf Basis dieses Ansatzes wurde ein Algorithmus zur satzweisen Verarbeitung der Sätze unter Verwendung des NLP-Werkzeugs GATE entwickelt, der aus natürlichsprachlichen Eingaben (Input) eine Ausgabe (Output) in eingeschränkter Sprache erzeugt. Dabei werden auch Metainformationen über den Text generiert und zur weiteren Verarbeitung ausgegeben.

Der prinzipielle Programmablauf gliedert sich wie folgt:

1. Zunächst werden die Sätze für eine sukzessive Verarbeitung getrennt. Dies erfolgt mithilfe des NLP-Werkzeugs GATE.

2. Für jeden Satz wird geprüft, ob und in welchem Maße Anpassungen an die formale Sprache der Boilerplates erforderlich sind. Dazu werden Syntax bzw. Satzbestandteile mittels GATE analysiert.

3. Passende Boilerplate-Typen für die Anforderung werden identifiziert. Diese leitet sich aus den unter 2. gefundenen Merkmalen ab. Es können etwa Kategorien von Anforderungen unterschieden sowie Bedingungen für die Anforderung modelliert werden, unter welchen sie gilt.

4. Sind Anpassungen erforderlich, so wird eine Konvertierung in die jeweils passende Boilerplate durchgeführt und diese werden im Programm gesammelt.

5. Die erzeugte Boilerplate, die am besten zum Inhalt der Anforderung passt, wird identifiziert. Dies kann automatisiert oder unter Einbezug des Nutzers geschehen. Der Nutzer kann dazu beispielsweise aus mehreren Vorschlägen auswählen.

6. Sind alle Anforderungen aus dem Quelldokument verarbeitet, wird auf Basis der erzeugten Anforderungen in Boilerplates ein Zieldokument für die weitere Verwendung erzeugt.

Auf Basis dieses Konzeptes wurde in [ZH17] ein Vorschlag zur Aufteilung der Ressourcen in verschiedene Bereiche gemacht. Dabei gliedern sich die

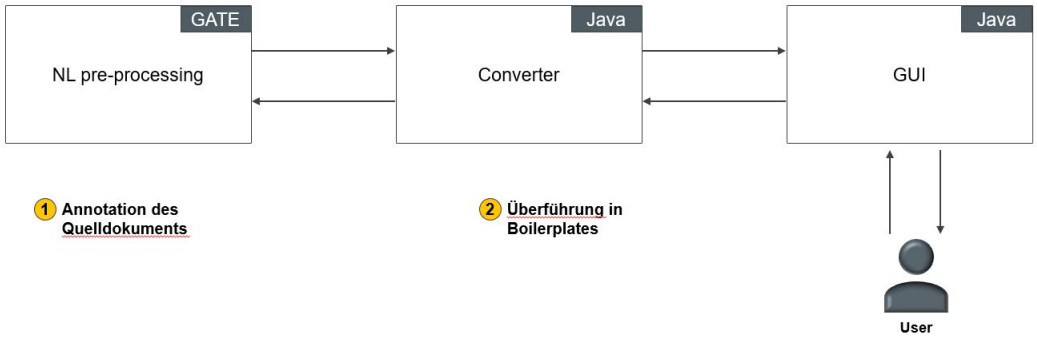

Abbildung 3.7: Verteilung der R2BC-Ressourcen [eigene Darstellung]

Funktionalitäten in drei Software-Komponenten, die in Abb. 3.7 dargestellt sind.

Der Programmablauf ist durch die Schritte 1 und 2 grob dargestellt und verteilt sich hauptsächlich auf die zwei entsprechenden Artefakte. Die Komponenten lassen sich dabei wie folgt definieren:

- Die *NL pre-processing* Komponente stellt das backend für die Sprachverarbeitung des Quelldokumentes dar. Dabei werden verschiedene Textbereiche mithilfe des NLP-Werkzeugs GATE annotiert und somit relevante Informationen über den Text generiert. Das Ziel ist es, konkrete Anforderungen im Lastenheft eindeutig zu identifizieren. GATE stellt ein in hohem Maße konfigurierbares Werkzeug auf *java*-Basis dar, dass in Abs. 4.2.1 näher beschrieben wird.

- Die zweite Komponente beinhaltet die Konvertierung der Sätze aus dem Quelldokument in Sprachschablonen. Der *Converter* nutzt dabei die zuvor erstellten Annotationen, um für jeden Satz im Anforderungsdokument eine oder mehrere passende Sprachschablonen zu finden. Diese Schablonen werden dann durch das Programm generiert bzw. passend „befüllt“. Der *Converter* wird ebenfalls mit *java* implementiert.

- Die Interaktion mit dem Nutzer wird durch ein herkömmliches *GUI* realisiert, dass dem Nutzer die Navigation im Programm ermöglicht. Auch kann dort etwa die Anzeige und Auswahl von einer aus mehreren möglichen Konvertierungen erfolgen.

In Kap. 4 erfolgt auf dieser Basis die konkrete Implementierung eines Prototypen für den R2BC.

3.5.4 DA-Funktionalität

Auf Basis des zuvor beschriebenen R2BC stellt der *Delta-Analyser* die Softwarekomponente für einen kriteriengeleiteten Vergleich zweier Anforderungsdokumente dar. Das Ziel ist es, auf Basis der Informationen zu jeweils einem der Lastenhefte den Grad der inhaltlichen Überschneidung zum anderen Lastenheft zu messen.

Im Kontext des Use-Cases dieser Arbeit geht es dabei insbesondere um den Vergleich von zwei verschiedenen Anforderungsdokumenten zur Version V1 und V2 eines gleichen Produktes. Zu Anforderungen aus dem Lastenheft V2 müssen also inhaltlich passende Anforderungen aus Lastenheft V1 gefunden werden, um anschließend Deltas zwischen diesen Anforderungen zu identifizieren bzw. für den Nutzer hervorzuheben. Diese Verarbeitung erfolgt dabei satzweise für das komplette Lastenheft.

Für ein automatisiertes Finden passender Requirements im DA müssen daher die Anforderungen in einheitlicher Form dargestellt sein. Somit können verlässlich nach einzelnen Satz- bzw. Textbestandteilen in den Anforderungen gesucht und Anforderungen verglichen werden. Dabei stellt, wie zuvor beschrieben, die Überführung der Anforderungen aus den Dokumenten aus der natürlichen in eine fest definierte, formalere Sprache, eine wesentliche Grundlage dar. Der zuvor skizzierte R2BC bietet eben diese Funktionalität.

Programmablauf und Artefakte

Die beschriebenen Funktionen ermöglichen den Entwurf eines Verfahrens für einen automatisierten Vergleich von zwei Lastenheften im DA. Dazu wird jede Anforderungen aus dem Lastenheft V2 mit denen aus V1 abgeglichen, um jeweils bestmöglich zueinander passende zu finden. Folgendes Vorgehen wurde dazu entwickelt:

1. Die einzelnen Anforderungen in Boilerplates werden für Lastenheft V1 und V2 geladen.

2. Für jede Boilerplate aus Lastenheft V2 wird geprüft, ob es identische, vergleichbare oder keine zuzuordnende Anforderungen in Lastenheft V1

gibt. Diese drei Kategorien werden auf Basis der GATE-Annotationen aus dem R2BC identifiziert.

3. Entsprechend des gefundenen Übereinstimmungsgrades werden die Deltas zwischen den Anforderungen identifiziert:

   (a) *Identische Anforderung*: Es wurde eine identische Anforderung in beiden Lastenheften gefunden. Die Anforderung kann ohne gesonderte Betrachtung übernommen werden.

   (b) *Vergleichbare Anforderung*: Es wurden nur eine teilweise übereinstimmende Anforderung gefunden. Dies ist etwa der Fall, wenn sich Anforderungen geändert haben bzw. überarbeitet wurden. Zwischen diesen Anforderungen existiert also ein Delta, das eine Änderung in den Anforderungen signalisiert und somit eine geänderte Produktspezifikationen bedeuten würde.

   Eine feingranulare Aufschlüsselung der erwartbaren Deltas findet sich in der Implementierung in Kap. 5.2.2 .

   (c) *Keine vergleichbare Anforderung*: Es konnte keine Anforderung aus V1 gefunden werden. In Lastenheft V2 könnten etwa neue Anforderungen enthalten sein, die in V1 noch in keiner Weise spezifiziert wurden. Dies kann etwa Eigenschaften betreffen, die erst später im Entwicklungsprozess definiert wurden oder zu der vorher noch keine Informationen bekannt waren. Im Kontext anderer Anforderungen muss dabei auf Konsistenz geprüft werden.

4. Falls es noch mehr mindestens vergleichbare Anforderungen in V1 gibt, wird die am besten passende Anforderung gefunden.

5. Der Delta-Report für den Nutzer wird unter Zugriff auf die Ergebnisse aus 4. erzeugt, wenn V2 vollständig durchlaufen wurde.

Insofern Deltas gefunden wurden, muss immer auch die Umsetzbarkeit dieser neuen Anforderungen gesondert geprüft werden. Die weitere Korrektheit des Lastenheftes V2 zu V1 kann bei vielen Deltas nicht gewährleistet werden, was eine Umsetzung der Anforderungen erschwert. Es können massive Unterschiede in den Spezifikationen auftreten, was zu einer teilweisen oder kompletten Neuentwicklung des Produktes führen kann.

Die beschriebene Funktionalität hat in der Entwicklung des Werkzeugs zur konkreten Herausbildung von drei Kernbestandteilen geführt. Das Werkzeug beinhaltet dabei Komponenten zur Abbildung von Fachlogik und GUI. Die Verteilung der Ressourcen ist in Abb. 3.8 dargestellt.

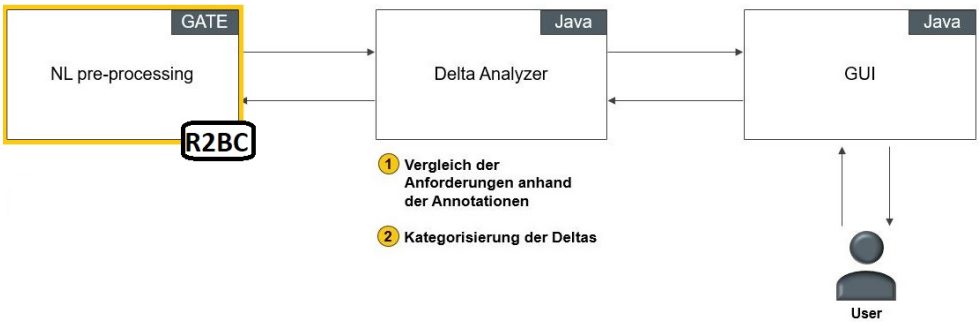

Abbildung 3.8: Verteilung der DA-Ressourcen [eigene Darstellung]

Die Komponenten realisieren dabei die zwei Hauptfunktionalitäten 1 und 2 in Form von GATE und java-Ressourcen. Auf dieser Basis können den einzelnen Komponenten Funktionalitäten wie folgt zugeordnet werden:

- *NL-pre-processing* Da der DA die zweite Stufe der Verarbeitung darstellt, nutzen dieser die Ergebnisse aus dem R2BC, d.h. die dort generierten Annotationen. Diese werden unter Zugriff auf das Programm in den DA eingelesen. Somit können die Anforderungen, die nach dem R2BC in Boilerpaltes vorliegen, anhand ihrer Annotaionen verglichen werden.

- Der *Delta-Analyser* prüft die verschiedenen Anforderungen aus Lastenheft V2 auf ihren Übereinstimmungsgrad mit den Anforderungen aus V1. Dabei greift er auf die geladenen Annotationen beider Lastenhefte zu und vergleicht daran verschiedene Anforderungen. Schlussendlich wird ein Delta-Report erstellt, der alle Deltas zwischen den Anforderungen übersichtlich darstellt.

- Die *GUI* ermöglicht dem Nutzer die Eingabe der zwei zu vergleichenden Lastenhefte. Nach der Verarbeitung kann der Nutzer dort außerdem den Delta-Report exportieren, der die jeweils verglichenen Anforderungen enthält.

Die Implementierung des DA auf Basis des zuvor implementierten R2BC ist in Kap. 5 beschrieben. Dort werden auch konkrete Erweiterungen und Verbesserungspotentiale des Programms beschrieben, die nach der Entwicklung identifiziert wurden.

Kapitel 4

Requirements-To-Boilerplate-Converter (R2BC)

Dieses Kapitel behandelt den *Requirements-to-Boilerplate-Converter* (*R2BC*) ausführlich. Hierzu wird zunächst der Aufbau des Tools aus den verschiedenen Klassen und der verwendeten Software beschrieben. Daraufhin wird die Implementierung wichtiger Funktionalitäten und Datenstrukturen mit Auszügen des Quellcodes gezeigt und erläutert. Anschließend wird das Testverfahren für das Werkzeug beschrieben und die dadurch erzielten Ergebnisse aufgeführt. Die Evaluation der Ergebnisse findet in einem späteren Kapitel statt. Abschließend wird ein Katalog mit möglichen Erweiterungen und Verbesserungen für das Werkzeug geliefert.

4.1 Architektur

In diesem Abschnitt soll die Architektur des R2BC vorgestellt und erläutert werden. Dazu gehört die zur Entwicklung verwendete Software sowie ein Überblick über die Klassenstruktur des Werkzeugs.

4.1.1 Verwendete Software

Als verwendete Software werden sämtliche Programme aufgeführt, welche eine Hilfe bei der Implementierung oder Teile der Funktionalität des Pro-

grammes dargestellt haben.

Implementierungsunterstützung

Das Werkzeug wurde auf Basis der Programmiersprache *java* geschrieben. Hierzu wurde das JDK (*java development kit*) 8.1 verwendet.

Die Wahl der Programmiersprache fiel aufgrund von zwei Hauptfaktoren auf java:

1. java ist eine weit verbreitete und entwicklerfreundliche Programmiersprache. Vorteile der Sicherheit von java werden außerdem in [RS18] aufgeführt.

2. Das Programm, welches für das NL-preprocessing verwendet wird ist java-basiert. Somit fällt die Integration des Programms in das Werkzeug einfacher. Dies wird jedoch noch genauer in einem späteren Abschnitt behandelt.

Als Programmierumgebung[1] wurde die Netbeans 8.2 IDE verwendet. Diese ist gemeinsam mit *Eclipse* eine der am weitesten verbreitete Programmierumgebungen für java-Programme.

Außerdem wurde die Software *Apache Maven* als Build-Management-Tool (*BMT*) verwendet. Dieses wird vor allem für java-Programme verwendet und war daher sehr geeignet für die Verwendung in unserem Werkzeug. Außerdem wurde dieses BMT auch in dem später beschriebenen Tool zur Bereitstellung bestimmter Funktionalitäten verwendet.

Build-Management-Tools helfen grundsätzlich dabei, bestimmte, zuvor nicht vorhandene Bibliotheken und deren Funktionalitäten nachzuladen und in ein Programm zu integrieren. Außerdem helfen sie bei dem späteren Ausliefern des Programms, also der Erstellung eines unabhängig ausführbaren Programms außerhalb der Programmierumgebung. Im Fall des R2BC wurde also eine *Executable JAR* (ausführbares JAVA-Archiv) erstellt.

Funktionalitätsunterstützung

Das zuvor bereits erwähnte Programm zur Bereitstellung von Funktionen ist bekannt unter dem Namen GATE (*General Architecture for Text Enginee-*

[1] Entwicklungswerkzeuge lassen sich als IDE, also *integrated development enviroment*, abkürzen

ring). Dies ist ein Werkzeug zur Erstellung von NLP-Programmen und wird ausführlich in [RS18] behandelt.

Für diesen Prototypen des R2BC wird *GATE Embedded* verwendet. Dies ist eine Ansammlung von java-Bibliotheken, welche alle Funktionen von GATE bieten. Man erhält also das gesamte Programm mit Ausnahme der Benutzeroberfläche. Im Folgenden wird daher im Zusammenhang auf die entwickelten Werkzeug nicht zwischen GATE und GATE Embedded unterschieden.

GATE ist, wie in Abbildung 3.7 (Konzept) zu sehen ist, für das NLP, also die Sprachverarbeitung in unserem Werkzeug zuständig und bildet somit das *Backend* des Programms.

Die genaue Umsetzung von NLP im Programm und der damit verbundene Einsatz von GATE sind jedoch Teil der Implementierung und werden somit in Abschnitt 4.2.1 behandelt.

4.1.2 Klassenstruktur

Im Rahmen der Entwicklung des Proof-of-concepts wurde die Funktionalität aus Abs. 3.5.3 auf verschiedene Komponenten aufgeteilt. Hier soll lediglich ein grober Überblick der Ressourcenverteilung im Programm gegeben werden, der Programmablauf findet sich in Kap. 4.2.3.

Verschiedene Programmbausteine sind dabei modular aufgebaut und verteilen sich auf die Pakete *R2BC*, der die Fachlogik enthält, *R2BCGUI*, das eine simple GUI-Klasse enthält, und *Boilerplates*, das die Objekte zur Erzeugung von Sprachschablonen enthält (siehe Abb. 4.1).

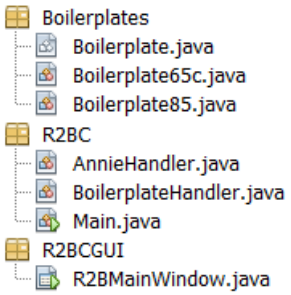

Abbildung 4.1: Zuordnung der R2BC-Klassen zu Paketen nach Abb. 3.7 [eigene Darstellung]

Die Funktionalität wurde dabei hauptsächlich mit verschiedenen java-Klassen umgesetzt, die die Umsetzung der Fachlogik darstellen. Die Anbindung an das NLP-Werkzeug GATE Embedded erfolgte dabei unter Verwendung verschiedener java-Bibliotheken, die extern im Programmordner gespeichert sind. Die GATE-Ressourcen wurden dabei zur Annotation von Text genutzt. Die selbst entwickelten java-Klassen einerseits für die GATE-Konfiguration, andererseits für die Auswertung der dort erstellten Annotationen.

Auf Basis des in Kap. 3.5.3 beschriebenen Algorithmus können die Aufgaben der verschiedenen Klassen in Abb. 4.2 wie folgt erläutert werden:[2]

- *Main* - Die Hauptklasse ermöglicht den Zugriff auf alle Funktionen des Programms. Es erfolgt die Initialisierung aller Programmkomponenten. Es werden dazu verschiedene Objekte für die Textverarbeitung (*AnnieHandler*, *BoilerplateHandler*) sowie für die GUI (*R2BMainWindow*) erzeugt. Hauptsächlich dient die Hauptklasse auch der Initialisierung der GUI, um die dortigen Schaltflächen mit Funktionen zu belegen. In der GUI ist beispielsweise ein Button zum Starten der Konvertierung enthalten, der eine Funktion des BoilerplateHandler aufruft.

- *AnnieHandler* - Der Annie-Handler realisiert den Zugriff auf die Bestandteile der Sprachverarbeitung aus dem Werkzeug GATE. Hauptsächlich betrifft dies die Verarbeitungslogik ANNIE, die konkrete Komponenten zur Textverarbeitung enthält (siehe Abs. 4.2.1). Im AnnieHandler werden dazu neben den Verarbeitungswerkzeugen auch die Textressourcen, die im Programm verarbeitet werden sollen, initialisiert. Diese *Language-Ressourcen* aus GATE beinhalten neben dem Text auch alle generierten Annotationen. Der AnnieHandler beinhaltet nach der Verarbeitung somit alle Informationen zu den Anforderungen in einer Liste. Auf Basis dieser Informationen kann dann die Transformation von Anforderungen in Boilerplates im *BoilerplateHandler* erfolgen.

- *BoilerplateHandler* - Die Klasse dient der Erzeugung von jeweils passenden Boilerplates zu einer Anforderung, indem aus dem Text alle relevanten Informationen zur Erzeugung passender Boilerplates extrahiert

[2]Die Darstellung der GUI-Klasse in Abb. 4.2 ist für eine bessere Übersichtlichkeit vereinfacht worden. Enthaltene Objekte und Funktionalitäten beziehen sich dabei jedoch nur auf die GUI-Elemente.

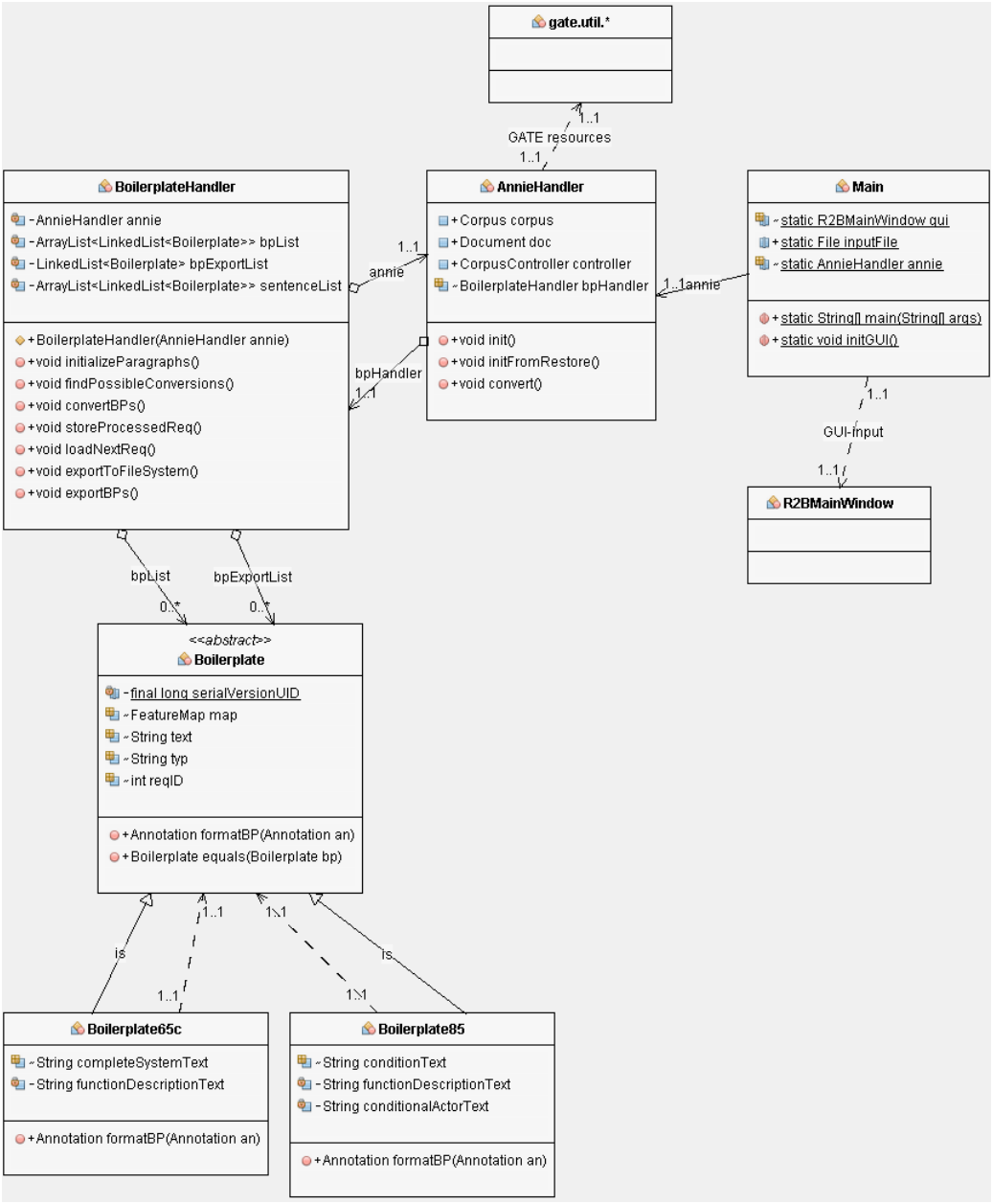

Abbildung 4.2: Klassendiagramm R2BC [eigene Darstellung]

werden. Der BoilerplateHandler beinhaltet dazu einerseits Funktionen zur Identifikation passender Boilerplates auf Basis der Annotationen aus der GATE-Verarbeitung. Andererseits ist auch die Erzeugung von konkreten Boilerplate-Objekten aus der abstrakten Klasse Boilerplate möglich, wenn die passenden Annotationen dazu gefunden wurden.

- *Boilerplate* - Die Klasse bündelt alle Kernmerkmale der Boilerplates, die durch das Programm erzeugt wurden. Dies betrifft in erster Linie den Gesamtsatz, die einzelnen Satzbestandteile und Metainformationen über die Anforderung, wie etwa eine ID.

- *Boilerplate65c* und *Boilerplate85* - Wie bereits unter 3.5.3 beschrieben, wurden im Zuge der Entwicklung dieses Werkzeugs zwei prototypische Boilerplates implementiert. Diese konkreten Boilerpalte-Objekte beinhalten eine Repräsentation der GATE-Annotation einzelner Satzbestandteile als Zeichenketten, um daraus ganze Sätze zu erzeugen. Der Boilerplate-Handler „befüllt“ somit konkrete Boilerplates mit allen erforderlichen Begriffen anhand der dort definierten Bestandteile.

    - Die Klasse *Boilerplate65c* stellt eine Schablone für Systemfunktionen dar. Demnach enthält sie Platzhalter für ein System und eine Funktion dieses Systems, die dann bei der Erzeugung konkreter Boilerplate-Objekte in das Grundgerüst des Satzes eingefügt werden. Typ 65c Boilerplates sehen demnach etwa so aus:

      `The complete system <<System>> shall perform <<FunctionDescription>>.`[3]

    - Die Klasse *Boilerplate85* stellt vereinfacht eine Erweiterung der Boilerplate65c um eine Bedingung dar. Dementsprechend existiert auch hier ein System und eine Funktion, aber auch eine Bedingung, unter der die Funktion ausgeführt wird. Diese drei Bestandteile werden für die Erzeugung von Typ 85 Boilerplates benötigt. Das Grundgerüst der Sätze mit bedingten Funktionen sieht dabei wie folgt aus:

      `Under the condition: <<conditionText>> the actor: <<conditionalActorText>> shall`

[3] «System» und «FunctionDescription» werden durch die Verarbeitung des Satzes in GATE eindeutig anhand passender Annotationen identifiziert und zugeordnet. Der Rest des Satzes ist fest hinterlegt.

`<<functionDescriptionText>>`.[4]

Die Funktionalität des Gesamtprogramms, das sich aus dem Zusammenspiel der Komponenten ergibt, ist in den folgenden Abschnitten dargestellt.

4.2 Werkzeug-Implementierung

In diesem Abschnitt werden wichtige Aspekte der praktischen Umsetzung des Programms genau erläutert. Dies bezieht sich sowohl auf die präzisere Vertiefung der Konzepte, als auch auf Quellcode-Beispiele der Implementierung zur Erläuterung des Programmvorgehens und die daraus resultierenden Interaktionsmöglichkeiten der Benutzer.

4.2.1 Sprachverarbeitung mit GATE

Dieser Abschnitt beschäftigt sich mit der automatischen Sprachverarbeitung der Anforderungsdokumente mit Hilfe des angekündigten NLP-Werkzeugs *GATE*. Dies beinhaltet das Einlesen der Lastenhefte, die Erzeugung einer Verarbeitungslogik, die Erstellung und Auswertung von Verarbeitungsregeln und die Einbindung von GATE in das spätere Programm. Der Fokus liegt dabei auf der praxisnahen Verwendung in dem Projekt. Ein genereller Überblick und Einordnung des Werkzeugs finden sich in der vorherigen Arbeit [RS18].

Einlesen von Lastenheften

In GATE benötigt man zum Einlesen und Verwenden von Dokumenten zunächst einen *Corpus*, also einen Textkörper. Dieser kann später mehrere Dokumente enthalten und diese verwalten. Außerdem wird eine Verarbeitungslogik immer auf einem Corpus (nicht auf einem Dokument) ausgeführt. Im Zusammenhang dieser Arbeit war es nicht nötig, einen Corpus mit verschiedenen Dokumenten zu befüllen. Daher ist zukünftig die Verarbeitung eines Corpus gleichbedeutend mit der Verarbeitung eines Dokuments.

[4] Eine Unterscheidung zu Typ 65c scheint zunächst nicht trivial, da conditionText auch einfach leer sein kann und somit Typ 65c fälschlicherweise als Typ 85 erzeugt werden können. Durch die unterschiedliche Annotation der beiden Typen schon bei der GATE-Verarbeitung kann dennoch eine verlässliche Unterscheidung erfolgen.

Ein Corpus kann in der Benutzeroberfläche von GATE schnell als neue *language resource* (sprachliche Ressource) erzeugt werden. In GATE Embedded ist dies ohne Benutzeroberfläche natürlich nicht möglich, lässt sich jedoch ähnlich einfach durch den Aufruf von *new Corpus()* durchführen.

Sobald der Corpus erstellt ist, lassen sich dort Dokumente einfügen.[5] Auch dies geschieht in GATE direkt über die Benutzeroberfläche. In GATE Embedded verwendet man hierzu den *add()*-Aufruf eines Corpus-Objekts mit einem übergebenen Dokument.

Sobald man einen Corpus mit mindestens einem Dokument hat, kann man auf diesem Corpus eine Verarbeitungslogik aufrufen (oft aufgrund der Aneinanderreihung mehrerer Verarbeitungskomponenten auch als Verarbeitungs-Pipeline bezeichnet). Diese werden im folgenden Abschnitt genauer beschrieben.

GATE-Verarbeitungslogik

In GATE wird die sprachliche Analyse durch eine Verarbeitungslogik durchgeführt, welche aus verschiedene Verarbeitungsressourcen (*processing resource*) aufgebaut ist. Eine Standard-Verarbeitungslogik wird durch das GATE-Plugin *ANNIE* zur Verfügung gestellt. Die einzelnen ANNIE-Verarbeitungsressourcen sind in [RS18] beschrieben.

Führt man eine Verarbeitungslogik auf einem Corpus bzw. auf dessen Dokumenten aus, so werden durch die verschiedenen Verarbeitungsressourcen an bestimmten Stellen im Text bestimmte Informationen hinzugefügt. Dies bezeichnet man als *annotieren*. Dabei wird der Text an sich nicht geändert, die Annotationen können jedoch etwa durch farbige Unterlegung des betroffenen Texts in der GATE-Benutzeroberfläche gezeigt und als XML-Datei exportiert werden.

[5] Alternativ kann auch erst ein Dokument eingelesen und anschließend ein Corpus um dieses Dokument angelegt werden.

Abbildung 4.3: Anforderung mit farbig markierten Token-Annotationen in GATE [eigene Darstellung]

Einer Annotation können sog. *Features* also Eigenschaften zugewiesen werden, welche dann noch weitere Informationen enthalten können. Dies sind z.B. die Länge der Annotation und die betroffene Zeichenkette.

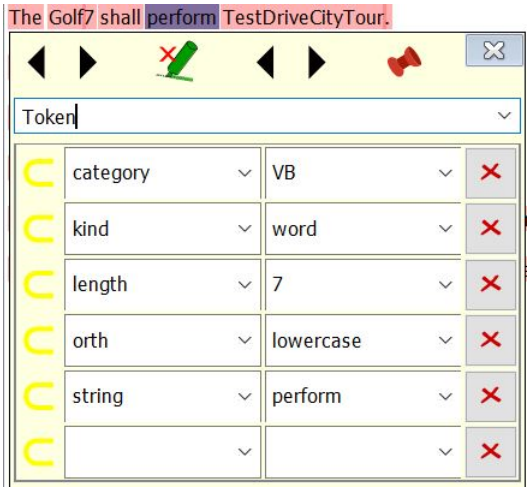

Abbildung 4.4: Ausgewählte Token-Annotation und dessen Features in GATE [eigene Darstellung]

Die für diese Arbeit verwendeten und besonders relevanten ANNIE- Verarbeitungsressourcen und die dadurch entstehenden Annotationen sind:

1. *ANNIE Tokenizer*: Der Tokenizer unterteilt den Text oberflächlich in Zeichen oder Zeichenketten, welche als Zahlen, Wörter (Groß- und Kleinschreibung wird unterschieden) und Interpunktion kategorisiert werden. Dies fügt an den betroffenen Stellen die *Token*-Annotation

hinzu, welche jeweils Typ (Zahl, Wort usw.) und Länge in Zeichen beinhalten. Die Verarbeitung durch den Tokenizer soll so effizient wie möglich sein und nur als Grundlage, etwa für spätere Grammatikregeln, dienen.

2. *ANNIE Gazetteer*: Ein Gazetteer beinhaltet relevante Eigennamen aus bestimmten Bereichen. Es handelt sich dabei um eine geschachtelte Liste, welche für jeden Bereich (beispielsweise Städte, Flughäfen, Fußball Clubs) über eine Liste der jeweils bekannten Eigennamen verfügt. Gazetteers werden verwendet, um domänenspezifisch Entitäten zu erkennen, da diese in einem normalen Wörterbuch nicht zu finden sind. Die vorgefertigten Listen sind also je nach Dokument individuell zu erweitern bzw. zu erstellen. Wird ein Listenelement im Text wiedererkannt, so erhält es standardmäßig die *Lookup*-Annotation. Das lässt sich jedoch auch schon im Gazetteer ändern und jeweils Listen-abhängig bestimmen.

3. *ANNIE Sentence Splitter*: Der Sentence-Splitter unterteilt den gesamten Text in einzelne Sätze. Dadurch wird jeder komplette Satz von einer *Sentence*-Annotation umschlossen. Dies wird später für den Part-of-Speech Tagger benötigt. Der Splitter verwendet dazu einen Gazetteer mit Abkürzungen, um Interpunktion wie etwa Fragezeichen, Ausrufezeichen und Punkte von anderen Sonderzeichen wie Semikola und Doppelpunkten zu unterscheiden.

4. *ANNIE POS-Tagger*: Der Part-of-Speech-Tagger annotiert alle Wörter und Symbole gemäß ihrer grammatikalischen Funktion. Hierfür wird keine neue Annotation angelegt, es werden jedoch das wichtige *category*-Feature an die Token-Annotationen angefügt. Dies ist eine wichtige Grundlage für grammatikalische Analyse von Sätzen. Dazu nutzt dieser Standard-Lexika und Regelwerke, welche auf dem Corpus-Training des *Wall Street Journal* beruhen. Siehe dazu auch Abs. 2.1.1.

Abbildung 4.5: Verwendete ANNIE-Verarbeitungsressourcen in GATE [eigene Darstellung]

JAPE Transducer

Die für die Verarbeitung wichtigsten Komponenten sind in der normalen ANNIE-Verarbeitungslogik jedoch noch nicht enthalten. Die sogenannten *JAPE Transducer* werden individuell angelegt und verwenden die Annotationen der vorherigen Verarbeitungsressourcen, um dem Text wichtige Informationen hinzuzufügen. Diese können im Gegensatz zu den vorherigen Annotationen jedoch auch von großer inhaltlicher bzw. semantischer Relevanz und spezifischer sein. JAPE Transducer lassen sich wie die anderen Verarbeitungsressourcen hintereinander reihen und somit in eine Verarbeitungslogik einfügen.

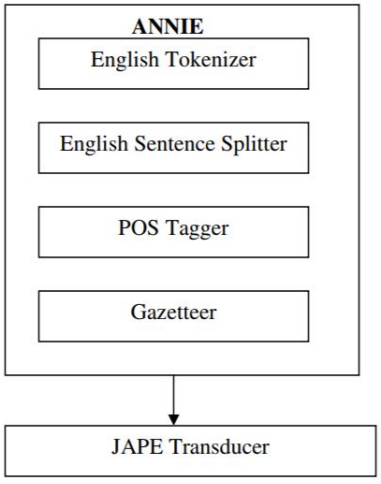

Abbildung 4.6: Verarbeitungslogik aus ANNIE und JAPE Transducern [TOL09]

In JAPE-Transducern[6] wird die im Namen vorhandene *JAPE* (Java Annotation Patterns Engine) verwendet. JAPE fungiert dabei als endlicher Zustandswandler[7], der reguläre Ausdrücke prüft und abhängig davon Annotationen erzeugt oder java-Code ausführt.[TOL09]

Um ein Verständnis von JAPE Transducern aufzubauen ist jedoch zunächst eine Erläuterung von JAPE-Dateien bzw. JAPE-Regeln nötig, welche im Folgenden gegeben wird.

Einführung zu JAPE-Regeln

JAPE Transducer werden immer mit einer *JAPE-Datei* initialisiert. JAPE-Dateien sind erkennbar an der *.jape*-Endung und können entweder eine JAPE-Regel oder eine JAPE-Hauptklasse (*main.jape*) darstellen.

Eine Hauptklasse wird dazu verwendet, mehrere JAPE-Regeln zu vereinen. Dadurch lassen sich auch Transducer mit mehreren Regeln initialisieren und unnötig lange Verarbeitungs-Pipelines verhindern.

Die JAPE-Regeln bilden die eigentliche Logik der jeweiligen Verarbeitungsressource und ähneln kleinen Programmen mit der Erkennung einer Eingabe und der Erzeugung einer bestimmten Ausgabe. Daher wird im Folgenden auch von JAPE-Code gesprochen.

Der Code in einer JAPE-Regel ist immer in einen Regel-Kopf, eine linke Seite (*LHS* für left hand side) und eine rechte Seite (*RHS* für right hand side) unterteilt. Bei der Betrachtung einer JAPE-Regel kann diese Benennung jedoch irreführend sein, da die Regeln von oben (Kopf, LHS) nach unten (RHS) aufgebaut sind, so wie es für Quellcode üblich ist. Abbildung 4.7 zeigt beispielhaft eine stark vereinfachte JAPE-Regel.

[6] sowie in einigen der anderen Verarbeitungsressourcen

[7] Für Erläuterung siehe [RS18]

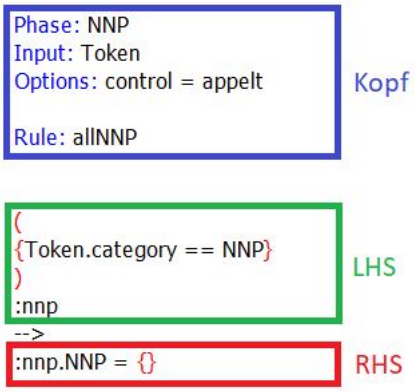

Abbildung 4.7: Beispielhafte JAPE-Regel [eigene Darstellung]

Im Kopf der Regel befinden sich die Parameter *Phase*, *Input*, *Options* und *Rule*.

1. Phase: Hier lässt sich eine Phase festlegen, in welcher die Regel ausgeführt werden soll. Dies kann wichtig sein, wenn z.B. eine Regel eine Annotation verwenden möchte, welche zuvor von einer anderen Regel angelegt werden muss. In diesem Projekt wurden für eine bessere Übersicht verschiedene Transducer angelegt, welche dann hintereinander durchlaufen werden. Daher wird die Phase im weiteren Verlauf der Arbeit nicht weiter betrachtet.

2. Input: Das Definieren des Inputs ist vor allem später für die LHS von entscheidender Bedeutung. Hier wird angegeben, welche Annotationen in den regulären Ausdrücken abgefragt werden können.

   Wenn ein Annotationstyp im Input nicht angegeben ist, jedoch in der LHS verwendet wird, führt dies zu einem Fehler beim Erstellen des Transducers. Wenn im Input Annotationen angegeben sind, welche in der LHS nicht verwendet werden, schadet dies unnötig der Laufzeit des Transducers.

3. Options: Hier können verschiedene Optionen zum verhalten der Regel in bestimmten Fällen angegeben werden. Dies beinhaltet hauptsächlich die Anpassung der *control*-Option. Diese entscheidet über das Resultat der Regel, wenn sich etwa mehrere gleiche Annotationen überschneiden. So kann etwa jede einzelne Annotation oder auch nur die längste

Annotation (welche die kleineren abdeckt) ausgegeben werden. Andere Optionen wie *negationGrouping* werden benötigt, um bestimmte reguläre Ausdrücke in der LHS zu ermöglichen.

4. Rule: Hier wird der Name der Regel definiert. Dieser weist üblicherweise auf die Funktion der Regel, bzw. die erzeugte Annotation, hin. Außerdem wird der Name oftmals als Feature der erzeugten Annotationen hinzugefügt, um die Nachvollziehbarkeit einer Verarbeitungslogik zu erhöhen.

In der JAPE-Regel befindet sich unterhalb des Kopfes die LHS. Diese umfasst eine Menge von regulären Ausdrücken, basierend auf den Annotationen, welche als Input deklariert wurden. Diese werden in dem Text gesucht und auf die angegebenen Bedingungen geprüft. Sollten bestimmte Textstellen diese Bedingungen erfüllen, werden diese mit einem Label versehen.

Ein Pfeil trennt die LHS von der RHS. Unterhalb der Pfeils befindet sich die RHS, in welcher abhängig von den vergebenen Labels nun neue Annotationen und deren Features mit Werten definiert werden können.

Alternativ kann sich auf der RHS in einer JAPE-Regel auch java-Code befinden, um erweiterte Funktionalitäten nutzen zu können. Dies war für das Projekt jedoch nicht nötig, da die Ergebnisse der JAPE-Regeln ohnehin später in eine java-Programm verwendet wurden. Daher wird diese Option im Verlauf der Arbeit nicht weiter berücksichtigt.

Der Aufbau von JAPE-Regeln soll anhand des Beispiels in Abbildung 4.7 genauer erläutert werden:

Zunächst wird die Phase *NNP* benannt und die *Token*-Annotation als Input deklariert. Die *control*-Option wird auf *appelt* gesetzt. Diese wird im Normalfall verwendet und bewirkt, dass eine betroffene Stelle im Text nur von einer Regel genutzt werden kann.[8] Die Regel selbst wird mit *allNNP* benannt, dies wird jedoch im Rest der Regel nicht wieder verwendet.

In der LHS wird mit der Abfrage

$$\{Token.category == NNP\}$$

[8]Setzen zwei Regeln am gleichen Startpunkt an, so erhält die Regel mit der längeren Übereinstimmung Vorrang, sind beide gleich lang, so erhält die Regel mit der höheren Priorität Vorrang, haben beide die gleiche Priorität, so erhält die Regel Vorrang, welche als erste in der Grammatik definiert wurde.[TOL09]

das *category*-Feature der *Token*-Annotation auf den Wert *NNP* geprüft.[9] Die Abfrage entspricht also der Form

$\{Annotation.Feature == Wert\}$

Alle Tokens, welche diese Bedingung erfüllen, werden daraufhin mit dem Label *nnp* versehen, welches auf die RHS der Regel übernommen wird.

Unterhalb des Pfeils wird nun durch die Zeile

$: nnp.NNP = \{\}$

jede mit *nnp* markierte Stelle mit der *NNP*-Annotation versehen. In den geschweiften Klammern könnten dieser Annotation außerdem Features übergeben werden. Möchte man wie zuvor erwähnt den Namen der Regel als Feature hinzufügen, so lässt sich dies mit der folgenden Ergänzung tun:

$: nnp.NNP = \{rule = \text{“}allNNP\text{“}\}$

Damit ist eine neue Annotation namens *NNP* mit dem Feature *rule* erzeugt, welche von späteren Regeln mit der einfachen Abfrage $\{NNP\}$ genutzt werden kann, ohne den Wert eines Features abfragen zu müssen.

JAPE-Regeln in der Praxis

In diesem Abschnitt werden einige der in der Sprachverarbeitung verwendeten JAPE-Regeln und deren Funktionsweise genauer erläutert.

Ursprünglich war der Ansatz für die Erkennung von Systemnamen, eine genaue grammatikalische Analyse zu nutzen, um solche Namen komplett erfassen zu können. Dieser wurde jedoch nach verschiedenen Tests für zu unpräzise befunden. Grund dafür war, dass in den Systemnamen keine ausreichende Regelmäßigkeit erkannt werden konnte, um eine verlässliche Erfassung zu gewährleisten.

Daher wurde als Nächstes ein Ansatz mit einer simpleren, jedoch auch besser umsetzbaren Methode verfolgt. Durch die Nutzung eines Gazetteers, kann die korrekte Erkennung von Eigennamen stark verbessert werden. Der einzige Nachteil ist, dass diese Eigennamen zuvor bekannt sein müssen. Dies

[9] Erinnerung: Die *Token*-Annotation wird von dem Tokenizer erstellt, das *category*-Feature und dessen Wert jedoch von dem POS-Tagger

ist jedoch im betrieblichen Kontext gut umsetzbar. Um dies zu demonstrieren, wurde ein ANNIE-Gazetteer mit einigen Systemnamen angelegt und sowohl für Beispiele, als auch für komplette Lastenhefte getestet. Daher wird im Folgenden anstelle einer JAPE-Regel der verwendete Gazetteer vorgestellt.

Solch ein Gazetteer besteht aus einer Menge von *.lst*-Dateien, welche als Listen fungieren und mindestens einer *.def*-Datei, welche definiert, wie mit den einzelnen Listen umzugehen ist. In Abbildung 4.8 ist ein Ausschnitt einer solchen Listendatei zu sehen. Abbildung 4.9 zeigt die direkte Definition einer bestimmten Annotation innerhalb der Definitionsdatei. Dies ermöglicht das Einsparen einer JAPE-Regel, welche die standardisierte *Lookup*-Annotation abfragen und einordnen muss.

```
TIM
RKE
LED driver
RF transmitter and Receiver
BCM
Remote
receiver
transponder
```

Abbildung 4.8: Ausschnitt der *SystemName*-Liste des ANNIE-Gazetteers [eigene Darstellung]

```
complete_system.lst::::CompleteSystem
module.lst::::Module
```

Abbildung 4.9: Definitionsdatei des ANNIE-Gazetteers [eigene Darstellung]

Die Regel *recognize_conditional_actor* wird dazu verwendet, im Falle einer Anforderung mit einer Bedingung den Akteur zu ermitteln, dessen Funktion durch die Bedingung eingeschränkt wird. Diese Anforderungen können dann später in die *Boilerplate85* überführt werden, welche in Abschnitt 3.5.3 dargestellt ist.

In der Phase und der Benennung der Regel sind soweit keine Besonderheiten vorhanden. Als Input sind hier jedoch *Token*, *Split* und *Condition* enthalten. Token ist die bereits bekannte Annotation des Tokenizers und

```
Phase: ConditionalRequirement
Input: Token Split Condition
Options: control = appelt negationGrouping=false

Rule: ConditionalRequirement


({Condition})?
({Token.string==","})?
({Token.category=="DT"})?


(

(
( {Token, !Split, !Token.string=="shall", !Token.string=="should"} )*
):conditionalactor

({Token.string=="shall"} | {Token.string=="should"})

)

-->
:conditionalactor.ConditionalActor = {}
```

Abbildung 4.10: JAPE-Regel zur Erkennung von eingeschränkten Akteuren [eigene Darstellung]

Split ist eine Annotation des Sentence Splitters, welche das Ende eines Satzes (z.B. durch Punkt oder Leerzeile) markiert. Außerdem wird die eigens erstellte Condition-Annotation verwendet, welche auf Grundlage von Schlagwörtern wie *if* oder *when* die Bedingung in einem Satz erkennen und markieren kann.[10] Außerdem wird die Option *negationGrouping* auf *false* gesetzt, um es zu ermöglichen, mehrere negierte Abfragen auf der gleichen Token-Annotation auszuführen.

Generell muss die Regel die folgenden zwei Satzstellungen berücksichtigen können:

1. Unter der Bedingung A, soll das System B die Aktion C durchführen.
2. Das System B soll die Aktion C durchführen, unter der Bedingung A.

Die LHS der Regel beginnt daher mit einer optionalen Abfrage auf eine Condition, ein Komma und einen *Determiner* (the, this, a usw.). Dies sorgt dafür,

[10] Auch zu dieser Annotation gibt es natürlich eine Regel, diese wird jedoch aus Platzgründen nicht genauer erläutert und befindet sich mit den restlichen Regeln in den Anlagen der Arbeit

dass diese Elemente nicht mit als *ContitionalActor* markiert werden, falls sie wie in *1.* vor dem Akteur im Satz auftauchen.

Daraufhin werden die folgenden Tokens als *conditionalactor* getaggt, jedoch nur solange:

1. keine Split-Annotation enthalten ist,
2. kein *shall*
3. und kein *should* im Text auftauchen.

Nachlaufend ist es jedoch nötig, dass eines dieser Wörter auftaucht (außerhalb des conditionaleactor-Taggings), da der Satz sonst nicht für die gewollte Boilerplate geeignet sein kann.

Damit ist LHS bzw. die Erkennung abgeschlossen und die als conditionalactor getaggten Bereiche werden mit der *ConditionalActor*-Annotation versehen. Features werden dabei nicht benötigt.

Sobald alle benötigten Satzbestandteile erfasst sind, lässt sich auf deren Grundlage nun eine geeignete Boilerplate definieren. Diese wird mit der JAPE-Regel *BP85_bsp* in Abbildung 4.11 erkannt und markiert.

Im Regelkopf ist hierbei zu beachten, dass alle für die Boilerplate benötigten Satzbestandteile als Input definiert werden. In diesem Fall werden also die *Condition*-, *ConditionalActor*- und *FunctionDescription*-Annotationen verwendet.

Daraufhin werden in den regulären Ausdrücken der LHS die zwei Satzschemata abgefragt, welche schon bei der Erkennung des *ConditionalActor* eine wichtige Rolle gespielt haben:

1. Unter der Bedingung A, soll das System B die Aktion C durchführen.
2. Das System B soll die Aktion C durchführen, unter der Bedingung A.

Zwischen den Satzbestandteilen vorkommende Tokens werden dabei ignoriert, solange diese keine *Split*-Annotation enthalten, also mit Punkten oder Zeilenumbrüchen den Satz beenden. Alle Sätze, die mit den Satzschemata entsprechen und die gestellten Bedingungen erfüllen werden als *ann* gelabelt und auf der RHS der Regel verarbeitet.

Im Gegensatz zu den vorherigen Regeln ist auf dieser RHS die Vergabe von Features von entscheidender Bedeutung. Da man in dieser Regel verschiedene Annotationen zusammenfasst, ist es wichtig, Informationsverlust

```
Phase: BoilerPlateConversion
Input: Token Split Condition ConditionalActor FunctionDescription
Options: control = appelt  debug = true

Rule: BP85_bsp
(

({Token, !Split})*
({Condition})
({Token, !Split})*
({ConditionalActor})
({Token, !Split})*
({FunctionDescription})
({Token, !Split})*

|

({Token, !Split})*
({ConditionalActor})
({Token, !Split})*
({FunctionDescription})
({Token, !Split})*
({Condition})
({Token, !Split})*


)
:ann

-->
:ann.Boilerplate85 = {rule=BP85_bsp,
text=:ann@cleanString,
conditionText=:ann.Condition@cleanString,
conditionalActorText=:ann.ConditionalActor@cleanString,
functionDescriptionText=:ann.FunctionDescription@cleanString
}
```

Abbildung 4.11: JAPE-Regel zur Markierung von Sätzen für Boilerplate85 [eigene Darstellung]

zu verhindern. Vor allem deshalb, weil die Satzbestandteile später noch vorliegen müssen, um die Boilerplates zu befüllen. Daher erhält jede erstellte Boilerplate-Annotation auch ihren kompletten Text sowie jeweils den Text jedes wichtigen Satzbestandteils als Feature. Dies macht es außerdem später im Programm einfacher, diese abzufragen. Alle Features der Annotation sind in Abbildung 4.12 zu sehen.

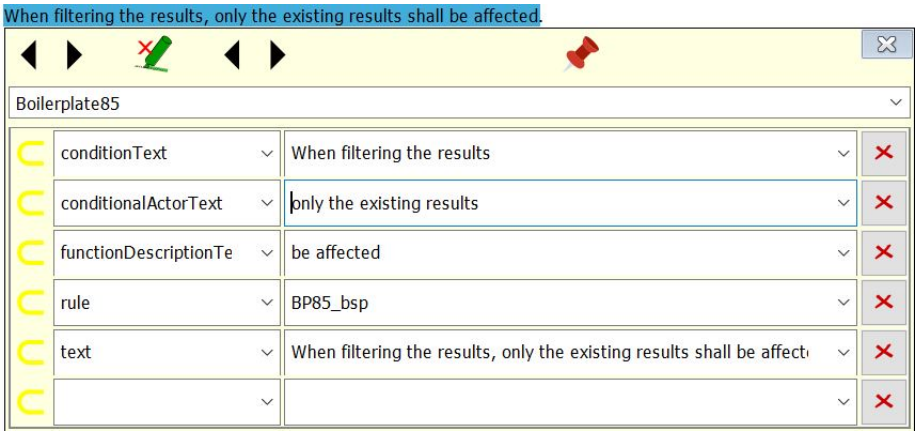

Abbildung 4.12: Alle Features eines *Boilerplate85*-Satzes in GATE [eigene Darstellung]

JAPE Transducer in GATE

Sobald alle benötigten JAPE-Regeln definiert sind, lassen sich diese unter JAPE-Hauptdateien zusammenfassen, um die benötigte Anzahl von JAPE Transducern zu verringern. Es sollte jedoch nur soweit zusammengefasst werden, dass einzelne Transducer noch austauschbar sind, wenn man die Sprachverarbeitung ändern möchte.

Die Transducer lassen sich (mit dem ANNIE-Plugin) über die Benutzeroberfläche von GATE leicht als Verarbeitungsressource anlegen und erhalten beim Initialisieren direkt die jeweilige JAPE-Datei. Syntaxfehler innerhalb der Dateien werden sofort erkannt und im Hauptfenster ausgegeben. Sollten keine Fehler vorhanden sein, erhält man die fertige Verarbeitungsressource, welche man an beliebiger Stelle in die Verarbeitungs-Pipeline einbauen kann.

Obwohl in dem Programm GATE Embedded verwendet wurde und die Benutzeroberfläche somit nicht direkt verwendet werden konnte, bietet GATE die wichtige Funktion, Corpora und Verarbeitungslogiken zu speichern und zu laden. Somit kann man eine Verarbeitungslogik in GATE erstellen diese als sogenannte *.gapp*-Datei speichern und anschließend in der Embedded-Version von GATE innerhalb des Programms laden. Außerdem lässt sich dabei der verwendete Corpus leicht ändern bzw. neu bestimmen. Das Laden einer solchen Datei in das Programm wird im folgenden Abschnitt genauer erläutert.

4.2.2 Converter-Funktionalität

In diesem Abschnitt wird die Funktionalität der Converter-Komponente des Programms erläutert. Diese dient, wie bereits beschrieben, zur automatisierten Transformation der gelesenen Lastenhefte in sog. Boilerplates. Im Programm greift die Verarbeitungslogik dabei auf die Funktionen der unter 4.1.2 beschriebenen Klassen bzw. Methoden, sowie auf die im vorigen Abschnitt behandelten JAPE-Regeln zur Textannotation zu.

Diese Ressourcen werden im Programm dann nacheinander genutzt, um die Boilerplate-Konvertierung durchzuführen. In der Regel sind verschiedene Programmfunktionen dabei zur Nutzerinteraktion Buttons in der GUI zugeordnet (siehe Abs. 4.2.3). Für ein besseres Verständnis der Zusammenhänge ist das R2BC-Klassendiagramm, sowie der kommentierte Quellcode des Prototypen im Anhang hilfreich.

Listen-Architektur

Der Convert-Vorgang teilt sich auf die zwei Klassen AnnieHandler und BoilerplateHandler auf. AnnieHandler dient dabei zur Erstellung von Annotationen über den Text durch die Anbindung an GATE. BoilerplateHandler erstellt auf Basis der dort gefundenen Annotationen anschließend entsprechende Boilerplates.

Es ist dabei zu beachten, dass mehrere Deutungen bzw. erzeugbare Boilerplates aus einem Satz existieren können. Ferner muss vor dem Export des konvertierten Textes in ein Ausgabedokument die eine richtige Konvertierung jedes Satzes gespeichert werden.

Das Datenmodell des Programms muss daher sowohl den ursprünglichen Text mitsamt Annotationen beinhalten, als auch alle möglichen Konvertierungen zu einem Satz in Boilerplates auflisten können.

Dieser Ansatz hat zur konkreten Erstellung von drei Listen im Programm geführt, *sentenceList*, *bpList* und *bpExportList*. Die Listen sind in Abb. 4.13 prinzipiell in ihrer Verknüpfung dargestellt.

- *sentenceList* repräsentiert dabei alle Paragraphen des Quelldokuments, da Anforderungen immer durch einen Zeilenumbruch getrennt sind.[11] soll dabei gefundene GATE-Annotationen zu jedem Satz beinhalten,

[11] In unserem Programm wird konkret die Sentence-Annotation aus dem ANNIE-SentenceSplitter von GATE verwendet, um Sätze zu trennen.

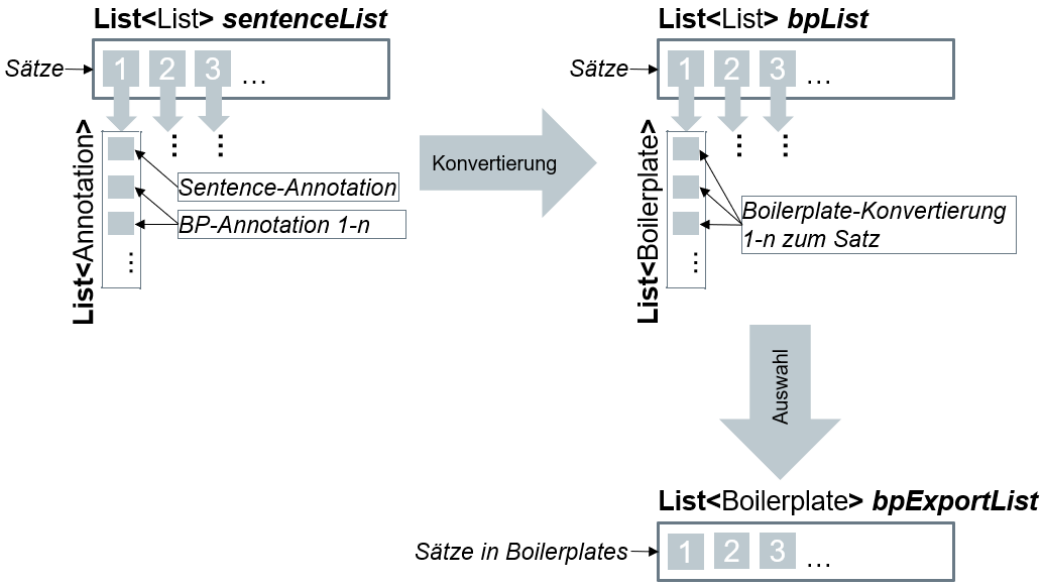

Abbildung 4.13: R2BC-Datenmodell aus Listen zur Speicherung von Annotationen(1), möglichen Konvertierungen(2) und Export(3) [eigene Darstellung]

die mithilfe der Klasse AnnieHandler angebundenen Verarbeitung generiert werden.

- Auf dieser Basis erfolgt dann die Befüllung der *bpList* mit möglichen Boilerplate-Konvertierungen zu jedem Satz in der Klasse BoilerplateHandler.[12] Die Annotationen dienen dabei zur Erzeugung der Boilerplates für die bpList.

- In der *bpExportList* sind alle Sätze enthalten, die nach dem Programmdurchlauf exportiert werden sollen. Nach der Wahl der zu speichernden Konvertierung durch den Nutzer wird der entsprechende Satz hier eingetragen.

sentenceList und *bpList* sind dementsprechend zweidimensional und beinhalten jeweils Annotationen (sentenceList, horizontal) und Boilerplate- Konvertierungen (bpList, horizontal) zu jedem Satz des Lastenheftes (vertikal).

[12]Wie bereits gezeigt, wurden dazu verschiedene verschachtelte JAPE-Regeln entwickelt, die eine mögliche BP-Konvertierung signalisieren bzw. annotieren.

Durch den Aufruf verschiedener Methoden der Klasse BoilerplateHandler werden beide Listen automatisch initialisiert und befüllt.

Anzumerken ist, dass *bpExportList* prinzipiell auch automatisch befüllt werden kann, wenn etwa keine Konvertierung gefunden wurde und somit der ursprüngliche Satz beibehalten werden soll. Auch könnte bei hinreichender Genauigkeit gleich die bestmögliche Konvertierung des Satzes gespeichert werden. Im Regelfall wird die Konvertierung jedoch durch den Nutzer im Programm ausgewählt bzw. bestätigt.

AnnieHandler

Die Klasse *AnnieHandler* stellt den ersten Schritt bei der Verarbeitung der Lastenhefte dar. Die Sätze aus dem Lastenheft müssen vor der tatsächlichen Konvertierung in Boilerplates mithilfe des NLP-Werkzeugs GATE annotiert werden. Daher beinhaltet der AnnieHandler verschiedene Objekte zur Verwaltung der GATE-Ressourcen.

Dazu dienen ein *Document* für das eingelesene Dokument, sowie ein *Corpus* und ein *CorpusController*, der auf die ANNIE-Module in GATE Embedded zur Verarbeitung zugreift.

Zunächst wird in der Methode *initFromRestore()* (Abb. 4.14) die ANNIE-Pipeline initialisiert, indem eine GATE-Sitzung aus einer *.gapp*-Datei wiederhergestellt wird. Diese Datei wurde vorher angelegt, um einen direkten Zugriff auf die dort hinterlegten Module aus dem R2BC zu ermöglichen.

```
// initialise the GATE library
Gate.init();

//restore from File
controller = (CorpusController)
        PersistenceManager.loadObjectFromFile(new File("ANNIE.gapp"));
```

Abbildung 4.14: Initialisierung der GATE-Ressourcen in *initFromRestore()*-Methode der Klasse AnnieHandler [eigene Darstellung]

Nachdem nun ein Dokument eingelesen wurde, kann dies dem *Document*-Objekt zugeordnet und auf dieser Basis der *Corpus* zur Speicherung der Annotationen über das Lastenheft initialisiert werden. In diesen Corpus werden während der Verarbeitung Annotationen über den Text gesammelt.

Für den weiteren Ablauf betrifft dies insbesondere gefundene Boilerplate-Annotationen, sowie die dafür nötigen Satzbestandteile, die gesammelt werden sollen.

Dazu kann nun mit der Verarbeitung durch die ANNIE-Pipeline begonnen werden. Die Methode *execute()* führt dazu den CorpusController, der die konkrete ANNIE-Pipeline für den R2BC enthält, auf dem Corpus aus (Abb. 4.15). Der AnnieHandler beinhaltet dazu die Methode *convert()*, die in Abb. 4.15 abgebildet ist.

```
//create Corpus for document from file system
corpus = Factory.newCorpus("MyCorpus");
controller.setCorpus(corpus);
File source = Main.inputFile;
doc = Factory.newDocument(source.toURI().toURL());
corpus.add(doc);

controller.execute();

bpHandler = new BoilerplateHandler(this);

//Load paragraphs and initialize Lists
bpHandler.initializeParagraphs();

//Traverse BP-Annotations and add to sentenceList
bpHandler.findPossibleConversions();

//convert annotated sentences to matching BPs, store Boilerplates
bpHandler.convertBPs();
```

Abbildung 4.15: Ausführung der ANNIE-Pipeline, Verarbeitung der Annotationen in BoilerplateHandler in *convert()*-Methode [eigene Darstellung]

Nach der Ausführung der GATE-Verarbeitung liegen die gefundenen Annotaitionen innerhalb des *Corpus*-Objektes vor, wodurch mit der weiteren Verarbeitung im BoilerplateHandler begonnen werden kann. Dazu wird mithilfe von new ein neuer BoilerplateHandler erzeugt. In der Methode *convert()* findet daher auch direkt der Aufuf verschiedener Methoden der Klasse BoilerplateHandler statt, der die Konvertierung der gefundenen Sätze in Boilerplates und die Initialisierung der Listen aus dem vorigen Abschnitt durchführt.

BoilerplateHandler

Im ersten Schritt wird dazu der Text satzweise getrennt. Dies erfolgt unter Nutzung der verschiedener Annotationen aus GATE, die nach der Textverarbeitung vorliegen und in einem *AnnotationSet* gesammelt werden können.

Da sich GATE-Annotationen, wie zuvor beschrieben, aus einem Startknoten, einem Endknoten und einer ID zusammensetzen, können die Sätze nun durchnummeriert und deren Länge bestimmt werden. Da nun bekannt ist, wie viele Sätze es gibt, kann die erste Dimension der Listen initialisiert werden. Die einzelnen Sätze werden nun nach ihrer *Sentence*-Annotation in die *sentenceList* eingetragen, die alle extrahierten Anforderungen aus dem Eingabedokument beinhaltet. In die bpList werden diese auch eingetragen, um Zugriff auf die ursprüngliche Anforderung zu haben.

Dazu wird die Methode *initializeParagraphs()* (Abb. 4.16) verwendet.

```
// Lists with size of sentence count
sentenceList = new ArrayList<>(sentenceCount);
bpList = new ArrayList<>(sentenceCount);

AnnotationSet annotations = annie.doc.getAnnotations().get("Sentence");

annotations.forEach((anno) -> {
    //Initialize sentenceList
    LinkedList<Annotation> liste = new LinkedList<>();
    liste.add(anno);
    sentenceList.add(liste);

    //Initialize bpList with source sentence as BP as first element
    LinkedList<Boilerplate> output = new LinkedList<>();
    output.add(new Boilerplate() {
        @Override
        public Boilerplate formatBP(Annotation an) {
            setMap(an.getFeatures());
            this.setText(Utils.stringFor(annie.doc, anno));
            return this;
        }
    });
    output.getFirst().formatBP(anno);
    bpList.add(output);
});
```

Abbildung 4.16: Initialisierung des Datenmodells aus Listen der Sätze in *initializeParagraphs()*-Methode [eigene Darstellung]

Nachdem nun die Annotationen getrennt vorliegen, kann mit dem ersten Schritt der Konvertierung der Sätze begonnen werden. Zunächst muss für jeden Satz geprüft werden, ob zwischen dem Start- und Endknoten Boilerpalte-Annotatioenn enthalten sind, die durch GATE erstellt wurden.

Daher wird die Methode *findPossibleConversions()* genutzt, um innerhalb der annotierten Sätze nach *Boilerplates*-Annotationen zu filtern. Falls Boilerplates vorhanden sind, bedeutet dies, dass alle Bestandteile für eine Boilerplate im Satz vorhanden sind und somit eine Konvertierung möglich ist. Dies vermerkt die Methode durch das Hinzufügen der Annotation an den entsprechenden Satz in der *sentenceList* (Abb. 4.17).

```
//Traverse each paragraph in sentenceList
AnnotationSet bpSet = annie.doc.getAnnotations("Boilerplates").get();

sentenceList.forEach((paragraph) -> {
    //find matching BPs within span of paragraph
    bpSet.stream().filter((bpAnno)
            -> (bpAnno.withinSpanOf(paragraph.get(0)))).
            forEachOrdered((e) -> paragraph.add(e));
});
```

Abbildung 4.17: Suche nach Boilerplate-Annotationen in *findPossibleConversions()*-Methode [eigene Darstellung]

Die tatsächliche Konvertierung findet in der Methode *convertBPs()* statt, die in Abb. 4.18 abgebildet ist. Diese beinhaltet einen Algorithmus zur sukzessiven Konvertierung der Sätze, sollten Boilerplates enthalten sein. Dazu wird die sentenceList iteriert, die nun die Sätze mit ihren *Boilerplates*-Annotationen enthält, die auf eine mögliche Konvertierung hinweisen.

Sollten aus einer Anforderung Boilerplates erstellt werden können, liegt eine entsprechende Annotation *Boilerplate65c* oder *Boilerplate85* für den Satz in der sentenceList vor. Dementsprechend können nun konkrete Objekte der jeweiligen Klasse, die die Boilerplate repräsentieren, erzeugt werden.[13]

Boilerplate-Objekte

Die Grundbausteine für die Erstellung von Boilerplates aus Anforderungen sind in der abstrakten Klasse *Boilerplate* zusammengefasst.

[13]Im Programm sind die Boilerplates hierarchisch angeordnet, wie im nächsten Abschnitt erläutert wird.

```
//Hierarchy of BPs: if 85 is found, 65c is always wrong and is not processed
for (LinkedList<Annotation> paragraph : sentenceList) {

    //find if BP85 in contained
    boolean found85 = false, found65c = false;
    //run for lowest hierarchy level
    for (Annotation an : paragraph) {
        if ("Boilerplate85".equals(an.getType())) {
            found85 = true;
            Boilerplate85 bp85 = new Boilerplate85();
            bpList.get(i).add(bp85.formatBP(an));
        }}

    if (!found85) {
        for (Annotation an : paragraph) {
            if ("Boilerplate65c".equals(an.getType())) {
                found65c = true;
                Boilerplate65c bp65 = new Boilerplate65c();
                bpList.get(i).add(bp65.formatBP(an));
            }}}
```

Abbildung 4.18: Überführung der Anforderungen in passende Boilerplates in *convertBPs()*-Methode [eigene Darstellung]

Boilerplate-Objekte bestehen dabei im wesentlichen aus ihrem Typen, einer ID, die die Anforderung eindeutig identifiziert, sowie den enthaltenen Features. Diese bilden, durch das Einfügen an der entsprechenden Stelle in einer festen Satzschablone, den Text für die Anforderungen in der Boilerplate. Durch die Methode *formatBP()* wird der Text der Boilerplate entsprechend formatiert, indem die entsprechenden Features aus der *FeatureMap* des Satzes extrahiert und als Text eingefügt werden (Abb. 4.19).

Verschiedene Features für eine Boilerplate definieren dabei eine jeweilige konkrete Klasse. Im Rahmen dieses Prototypen konnten zwei verschiedene Klassen für Boilerplate-Objekte erzeugt werden, *Boilerplate65c* und *Boilerplate85*.

Das Ergebnis stellt dann die Anforderung als formatierte Boilerplate im Objekt dar, die in die *bpList* als mögliche Konvertierung eingfügt wird.

Im Laufe der Entwicklung hat bereits frühzeitig die hierarchische Anordnung der Boilerplates signifikante Verbesserungen erzeugt. Es wurde gezeigt, dass die Boilerplate für bedingte Funktionen (BP85) der Boilerplate für Funktionen ohne Bedingung (BP65c) in jedem Fall überlegen ist, sollten

```
map = an.getFeatures();
completeSystemText = (String) map.get("completeSystemNameText");
functionDescriptionText = (String) map.get("functionDescriptionText");
text = "The complete system <<" + completeSystemText +
        ">> shall <<" + functionDescriptionText + ">>.";
```

Abbildung 4.19: Formatierung des Textes für Typ-65c-Boilerplates in *format-BP()*-Methode [eigene Darstellung]

beide gefunden werden. In der BP65c wird die Bedingung in manchen Fällen einfach abgeschnitten und nicht aus der Quellanforderung übernommen. Die BP85 enthält diese Bedingung explizit als *condition*-Annotation.

Da die BP85 also effektiv eine Spezialisierung der BP65c darstellt, konnte dieser Fehler durch entsprechende Regeln beseitigt werden. Dies kann durch eine verbesserte Erkennung möglicher Konvertierungen im NL pre-rpcessing verhindert werden. Für verbesserte Testedingungen wurde diese Optimierung im Quellcode durchgeführt.

In Abs. 4.3 werden weitere mögliche Ansätze einer verbesserten Erkennung der Boilerplates behandelt. Dort wird auch die Kompensation von sprachlichen Fehlern in den Anforderungen thematisiert, die in der Praxis auftreten und die Erkennung erschweren.

4.2.3 GUI - Nutzerinteraktion und Workflow

Im Kontext dieser Arbeit stand besonders die die Verdeutlichung des späteren Workflows bei der Verwendung des Werkzeugs im Vordergrund. Für konkrete Tests der Funktionalität des WErkzeugs wurde eine prototypische GUI implementiert. Diese diente auch zur Verdeutlichung des Ansatzes und zum Einholen von Feedback der RE-Experten bei HELLA.

In diesem Abschnitt wird daher anhand eines prototypischen Lastenheftes der Umgang der späteren Nutzer mit dem Werkzeug evaluiert. Es soll demonstriert werden, inwiefern solche Werkzeuge Potential für eine Unterstützung der RE-Prozesse bieten.

Die prototypische Interaktion verschiedener Nutzergruppen mit NLP-Werkzeugen wurde bereits in [RS18] skizziert. Das dort beschriebene Aktivitätsdiagramm dient dabei zur Visualisierung der Interaktionspunkte mit dem Werkzeug von Anfang bis Ende der Nutzung. Die enthaltenen Aktivitäten haben sich in einer spezifischen GUI für das Werkzeug manifestiert.

Werden die dort enthaltenen Aktivitäten auf das in dieser Arbeit vorgestellte Werkzeug angewandt, ergibt sich die Konkretisierung des Arbeitsprozesses, den ein Nutzer (hier RE-Experte) mit dem Tool durchläuft (Abb. 4.20).

Programmoberfläche

Um zu prüfen, wie Nutzer mit dem Programm interagieren, wurde neben der Fachlogik auch eine prototypische GUI implementiert. Diese orientiert sich grob an dem Vorschlag aus [ZH17] und beinhaltet einige weitere Elemente, die in Rücksprache mit den Experten evaluiert wurden.

Sie beinhaltet demnach Buttons zum laden und speichern von Dokumenten, sowie zur Bewertung der angezeigten Sätze bzw. Konvertierungen als richtig, falsch und unklar, wenn die Anforderung uneindeutig bzw. unverständlich formuliert ist. Die Anforderungen aus dem Quelldokument werden mit den vorgeschlagenen Konvertierungen in den Textfeldern angezeigt. Die *Progress-Bar* und die Anzeige unterhalb stellen den Anteil der bereits bewerteten Anforderungen prozentual bzw. absolut dar.

In Abb. 4.21 ist die GUI aus dem *java*-Programm dargestellt.

Je nach Aktivität, in der sich der Nutzer mit dem Programm befindet, ist nur eine Auswahl der Buttons der GUI aktiv. Der Arbeitsprozess gliedert sich dabei wie folgt:

1. *Initialisierung und Konvertierung* Zunächst kann über einen Klick von *Load document...* über ein Dialogfenster ein Lastenheft ausgewählt werden, das konvertiert werden soll. Über den Convert-Button wird anschließend der komplette Verarbeitungsprozess, der im vorigen Kapitel beschrieben ist, durchlaufen.

2. *Bewertung der Konvertierungen* Nach der Analyse wird automatisch die erste Anforderung geladen. Oben erscheint das Original, in den unteren Textfeldern drei mögliche Boilerplates, die der R2BC dazu ermittelt hat.

   Der Nutzer kann nun durch die Radio-Buttons rechts eine davon auswählen und über den Button *Confirm* bestätigen. Alternativ kann er, sollte keine Konvertierung bzw. passende Boilerplate gefunden werden, über *Skip* zur nächsten Anforderung springen. Die Methode *storeProcessedReq()* speichert die Auswahl für den Export. Der Button *Review*

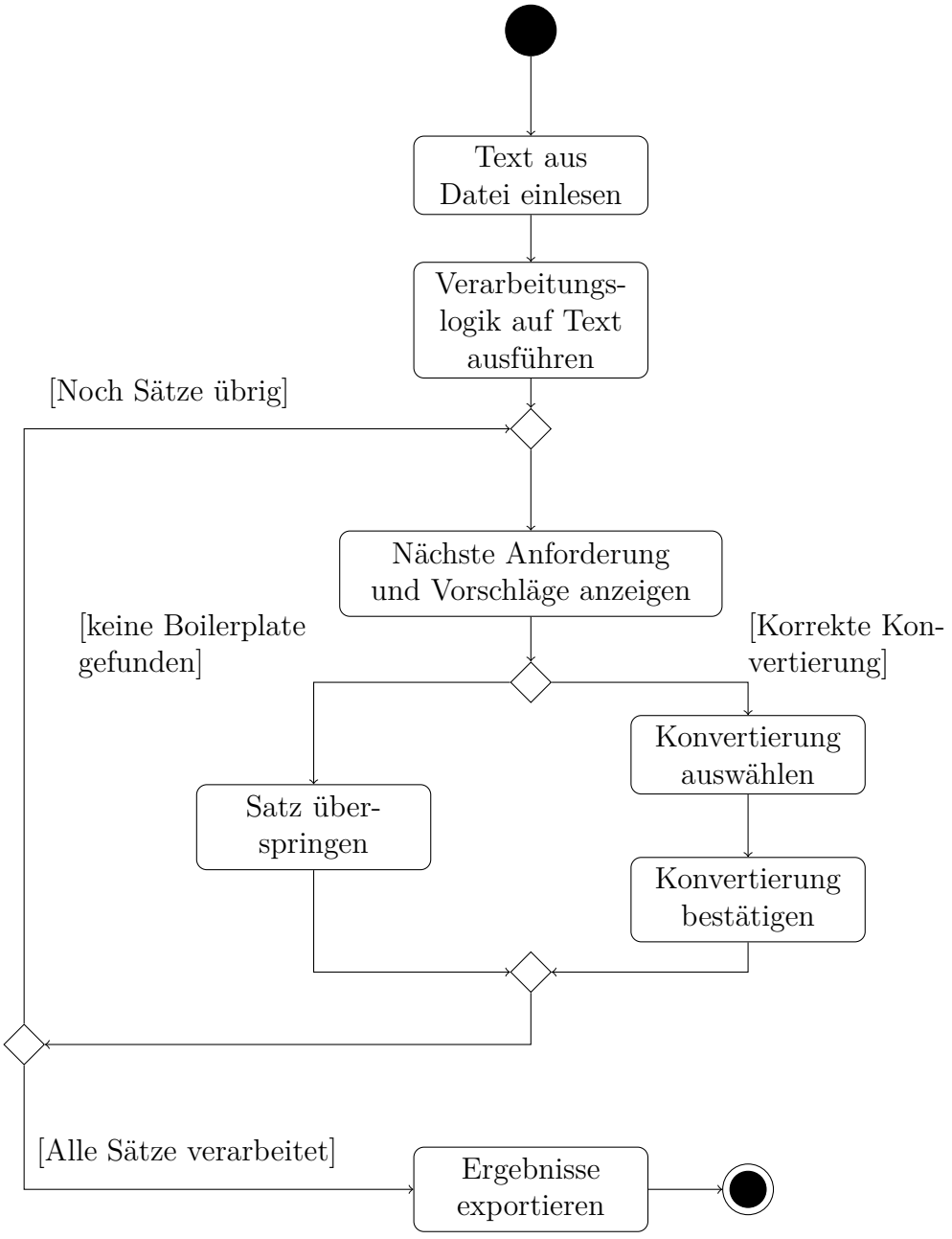

Abbildung 4.20: UML-Aktivitätsdiagramm User-Workflow mit dem Werkzeug [eigene Darstellung]

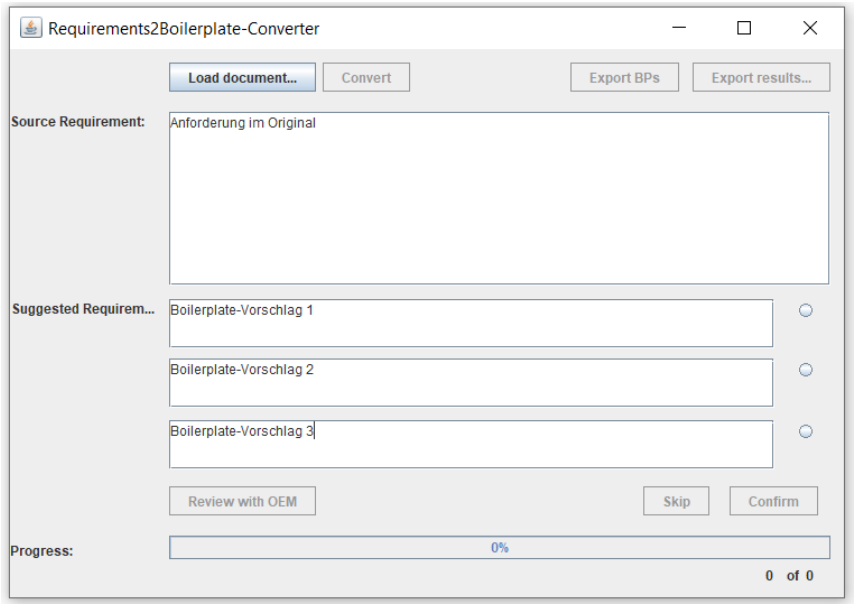

Abbildung 4.21: R2BC-GUI beim Programmstart [eigene Darstellung]

with OEM dient zur Markierung der Anforderung als unklar, wenn Klärungsbedarf mit dem Autoren beim OEM besteht.

Das nächste Tupel aus Original und Boilerplate-Liste einer Anforderungen wird anschließend über die Methode *loadNextReq()* geladen.

3. *Export der Ergebnisse* Wenn alle Anforderungen verarbeitet sind, können die Ergebnisse in Textform über *Export results...* in eine Datei geschrieben werden. Zur weiteren Verarbeitung der Ergebnisse, z.B. durch den DA, können auch die Boilerplate-Objekte über den Button *Export BPs* serialisiert und exportiert werden.

Beispielhafte Arbeit mit dem Werkzeug

In diesem Abschnitt wird der Arbeitsprozess mit dem R2BC näher beschrieben.

Für ein besseres Verständnis wird sich hier auf ein kleines Beispiel beschränkt, ausführliche Tests mit Lastenheften aus dem betrieblichen Alltag werden unter 4.4 behandelt. Die Implementierung der Nutzerinteraktion lässt

sich anhand der Konvertierung der folgenden Sätze demonstrieren, die in einem eingelesenen Lastenheft enhalten sein könnten:

1. The Golf7 shall perform TestDriveCityTour.

2. The EBus2X is part of Golf7.

3. The durability of EBus2X shall be at least 12000 h.

4. If there is a connection to the internet, the
Bluetooth connected phone shall transmit the data to a server.

Dieses prototypische Lastenheft wird im Laufe dieses Abschnittes von natürlicher Sprache in formalere Boilerplates konvertiert. Die vier Beispielanforderungen sind dabei an betriebliche Produktspezifikationen angelehnt, z.B. an ein Auto *Golf7*. Jede Anforderung repräsentiert konkrete Szenarien, die im Programm bei der Analyse und Konvertierung auftreten können. Dies beinhaltet sowohl die Boilerplates, die für jeden Satz infrage kommen, als auch typische Fehler bei der Erkennung, die bei der Verarbeitung auftreten.

Zunächst wird der R2BC mit dem Lastenheft initialisiert und die Konvertierung gestartet, indem der Convert-Button geklickt wird. Anschließend liegen die verarbeiteten Anforderungen im Programm vor und der Nutzer beginnt mit dem *Review* der Konvertierungen.

Zunächst wird Anforderung 1 angezeigt, zu der eine passende Boilerplate gefunden wurde (Abb.4.22):

Aus dieser Anforderung wurde im R2BC die Boilerplate65c erzeugt und nun als einziger Vorschlag angezeigt. Die Konvertierung ist korrekt, da sowohl die Boilerplate richtig erkannt wurde, als auch das CompleteSystem *Golf7* und die FunctionDescription aus der Anforderung extrahiert und in die Schablone der Boilerplate65c eingebettet sind. Der Satz ist also stimmig und die Konvertierung wird über *Confirm* bestätigt.

Zu Anforderung 2 wurde im Programm keine passende Boilerplate gefunden, da die *is-part-of*-Beziehung nicht durch eine entsprechende Boilerpalte abgedeckt ist. Das Programm kann also keine Konvertierung durchführen, da der Satz nicht alle Elemente für die Befüllung der Schablonen enthält (Abb. 4.23).

Dementsprechend wird der Satz mit *Skip* übersprungen und die Anforderung damit ohne Konvertierung in die *exportList* geschrieben.

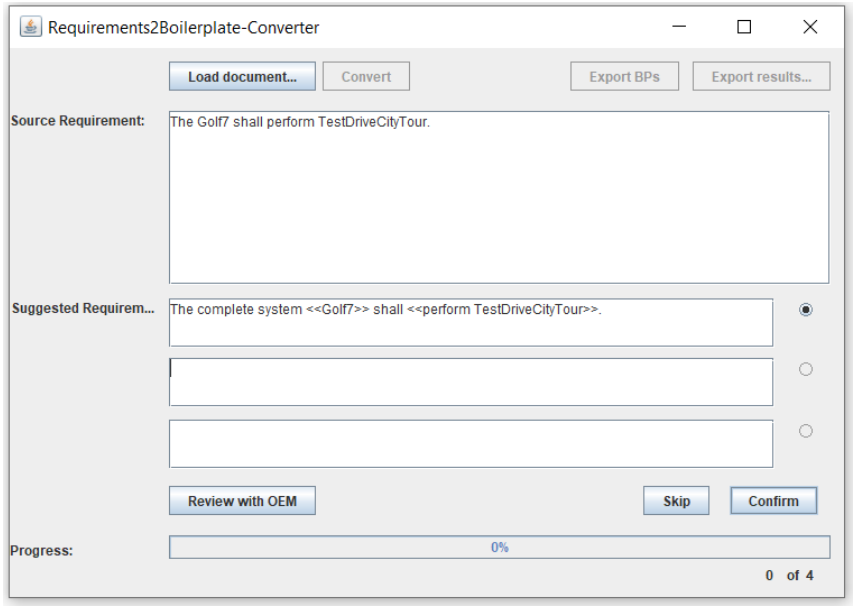

Abbildung 4.22: Anforderung 1, Boilerplate65c korrekt übersetzt [eigene Darstellung]

Anforderung 3 stellt beispielhaft den Fall einer fehlerhaften Erkennung bzw. Übersetzung dar. Die erkannte Boilerplate ist prinzipielle nicht geeignet, da die Anforderung eine Eigenschaft des Systems definiert (*durability of ... at least 12000 h.*) und im Gegensatz zur Boilerplate65c keine Funktionen eines Systems. Ferner wurde bei der Übersetzung der Boilerplate der Teil *durability of* vor dem System abgeschnitten.

Der Nutzer hat nun zwei Möglichkeiten: entweder überspringt er die fehlerhafte Konvertierung mit *Skip*, oder er fügt *durability of* vor *EBus2X* hinzu, um die sinngemäße Aussage zu erhalten und eine Boilerplate zu erhalten.

Zu Anforderung 4 erscheint die korrekte Konvertierung in die Boilerplate85. Die Bedingung *if ...* vor der Beschreibung der Funktion hat dazu geführt, dass der R2BC diese identifiziert hat. Somit wurde keine Boilerplate65c erzeugt, für die ja prinzipiell der Teilsatz nach dem Komma ausreichen würde. Die korrekte Konvertierung wird erneut durch *Confirm* bestätigt.

In Abs. 4.4 wird auf Basis der prototypischen Implementierung die Funktionalität des gesamten Programms mit realen Lastenheften erprobt. In Abschnitt 4.3 werden mögliche konzeptuelle Ergänzungen der GUI vorgestellt.

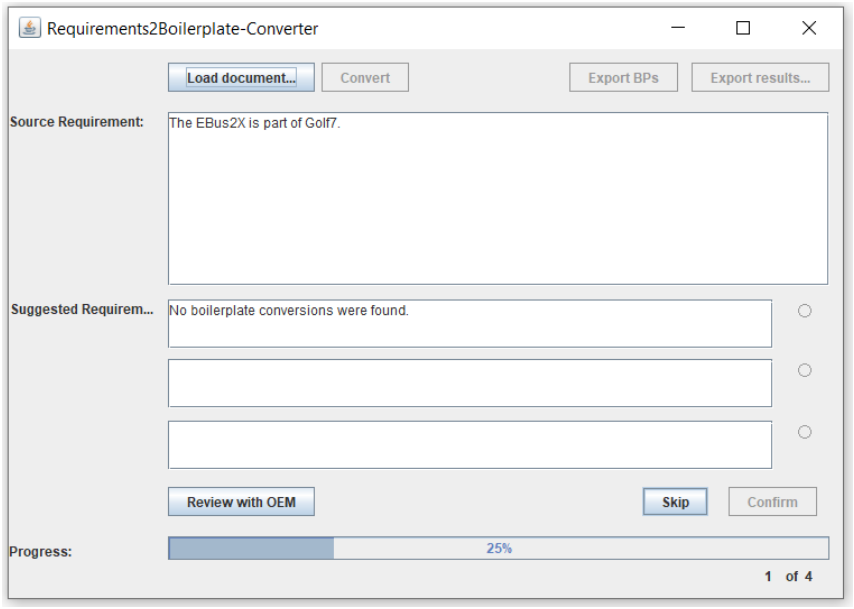

Abbildung 4.23: Anforderung 2, keine passende Boilerplate [eigene Darstellung]

4.2.4 Umsetzung der NLP-Architektur

In diesem Abschnitt wird die Einordnung des Prototypen als NLP-Werkzeug in die von [COP04] definierte Architektur behandelt. Die verschiedenen Schritte aus Abs. 2.1.2. zur Analyse und Verarbeitung der Lastenhefte wurden in den oben beschriebenen Programmbestandteilen umgesetzt. Anhand der prinzipiellen Aufgaben von NLP-Werkzeugen kann die Verarbeitungskette unseres Werkzeugs dort wie folgt eingeordnet werden:

1. *Input Processing* - Einlesen von Quelldaten: Zunächst kann der Nutzer Dateien für die Verarbeitung im Programm eingeben. Dazu wählt der diese Dateien in der GUI des Programms aus. Unser Werkzeug extrahiert daraus dann den enthaltenen Text.

2. *Morphologische Analyse* - Annotation der Worttypen: Mithilfe der Analyse des Textes durch die GATE-Verarbeitungslogik ANNIE werden einzelne Wörter aus dem Text auf ihre sprachliche Funktion hin überprüft. Dies ermöglicht die weitere Untersuchung bestimmter Wortty-

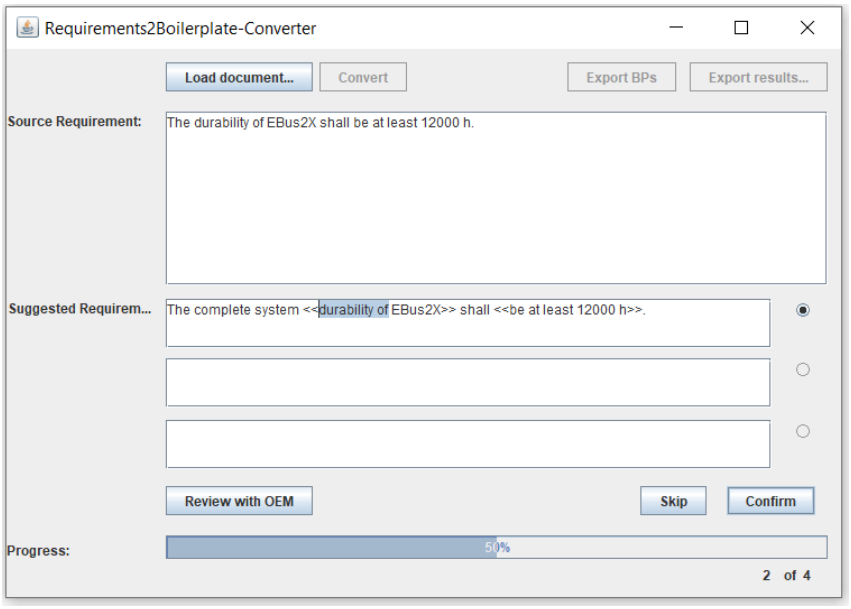

Abbildung 4.24: Anforderung 3, fehlerhafte Konvertierung [eigene Darstellung]

pen, wie z.B. Nomen, nachdem diese gesammelt wurden. Nomen können etwa in JAPE-Regeln als *NNP* abgefragt werden (siehe 4.2.1)

3. *Part-of-Speech-Tagging* - Erkennung von Satzbausteinen: Für ein genaues Verständnis, worum es in einer Anforderung geht, müssen die einzelnen Bestandteile des Textes zugeordnet werden. Im Programm werden gefundene Wörter etwa dahingehend geprüft, ob diese ein System darstellen können, wenn sie zuvor als Nomen kategorisiert wurden.

   Auf dieser Basis können dann passende Boilerplates identifiziert werden. Für die spätere Befüllung der Boilerplates mit den korrekten Begriffen ist die korrekte Erkennung von Worten im Satzzusammenhang daher wesentlich.

4. *Parsing* - Überführung in Boilerplates: Mithilfe des POS-Tagging wurden zuvor passende Boilerplates auf Basis gefundenen Annotationen innerhalb der Anforderung gefunden. Dies stellt nun die Grundlage für die Zusammensetzung der Boilerplates des entsprechenden Typs dar, indem Boilerplate-Objekte im Programm erzeugt werden.

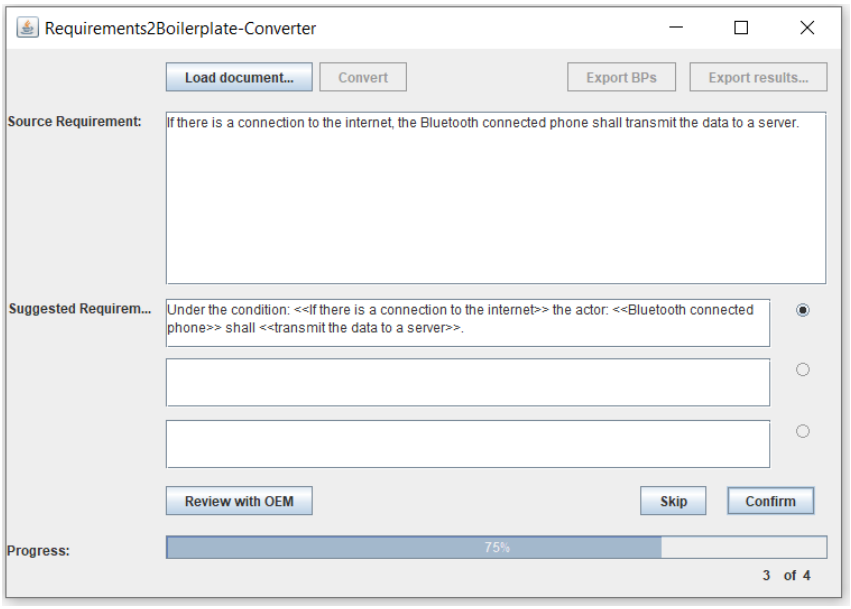

Abbildung 4.25: Anforderung 4, Boilerplate85 korrekt übersetzt [eigene Darstellung]

Die Platzhalter werden dann von den Begriffen aus der Anforderung ersetzt, womit schlussendlich eine Anforderung in der semi-formalen Sprache der Boilerplates vorliegt.

5. *Disambiguation* - Zuordnung der richtigen Konvertierung: Im Parsing werden zunächst alle möglichen Konvertierungen auf Basis möglicher Boilerplate-Typen durchgeführt. Nach der Erzeugung der Boilerplates liegen somit möglicherweise verschiedene Möglichkeiten der Deutung vor, von denen einzelne unpräzise oder fehlerhaft sein können.

   Unpräzise Formulierungen können etwa durch eine hierarchische Anordnung der Boilerplates reduziert werden, wenn eine spezifische Boilerplate offensichtlich besser den Inhalt der Anforderung repräsentiert als eine eher allgemein formulierte. Die allgemeineren Boilerplates werden somit gleich ausgeschlossen, wenn eine bessere Konvertierung möglich ist. Siehe dazu auch Abs. 4.2.2.

   Diese Unterschiede werden, sofern vorhanden, als mehrere Boilerplates im Programm dargestellt und an den Nutzer ausgegeben. Der Nutzer

kann in der GUI dann die am besten zum ursprünglichen Anforderung passende Konvertierung auswählen.

6. *Context Module* - Bewertung im Zusammenhang: Bei der Identifikation der best passendsten Boilerplate wurden in Abs. 4.3 Vorschläge zur präzisen Erkennung von Entitäten. Diese Ansätze ermöglichen dann die Automatisierung der Auflösung bestimmter Mehrdeutigkeiten, wenn diese das Verständnis des Kontextes aus dem Zusammenhang mehrerer Sätze erfordern.

   Im Kontext mehrerer Anforderungen kann der Nutzer im Rahmen des Review-Prozesses weitere Informationen über den Dokumentenkontext mit einfließen lassen. Dies kann etwa Informationen über vorige Anforderungen beinhalten, die bereits verarbeitet wurden, oder Meta-Informationen über das Anforderungsdokument wie etwa ergänzende, mitgeltende Unterlagen.

7. *Text Planning* - Speichern der korrekten Boilerplate: Nach der Auswahl von einer richtigen Konvertierung innerhalb der GUI hat sich der Nutzer für eine Boilerplate entschieden, die in das Export-Dokument übertragen werden soll. Durch diese Festlegung wird im Programm die entsprechende Boilerplate in die OutputList der Konvertierung eingetragen.

8. *Tactical Generation* - Erzeugung von Text aus BP-Objekten: Die Boilerplates liegen nach der Konvertierung zunächst als Objekte im Programm vor. Vor der Erzeugung des Ausgabe-Dokuments, das alle Anforderungen als Boilerplates enthält, muss aus diesen Objekten der Text extrahiert werden.

   Dieser Text liegt in den einzelnen Objekten als Zeichenkette vor und wird mithilfe einer Methode formatiert. Im Programm wird bei der Dokumentenerzeugung der Text aller übernommenen Boilerplates abgefragt.

9. *Morphological Generation* - Sprachliche Anpassung: Im ersten Entwurf der Prototypen wurde die Analyse auf englischen Text optimiert, da die Lastenhefte häufig auf englisch verfasst sind (siehe Kap. 3.3). Die Boilerplates im Programm sind dementsprechend in englischer Sprache verfasst, wie auch das Ausgabe-Dokument.

Da der Text bereits innerhalb der Boilerplate-Objekte formatiert wurde bzw. vorliegt, entfällt hier eine aufwändige Anpassung an Sprache und ihre Grammatik.

10. *Output-Processing* - Erzeugung des Anforderungsdokuments aus Boilerplates: Nachdem die einzelnen Zeichenketten aus den Boilerplates extrahiert wurden, liegen nun die einzelnen Anforderungen in formaler Sprache als Text vor. Die Anforderungen müssen nun vor dem Export in der richtigen Reihenfolge angeordnet und an das Layout des Dokumentes angepasst werden.

    Der Nutzer wählt dazu eine Datei über die GUI aus, in welche die Anforderungen exportiert werden. Der Dateityp ist dabei prinzipiell unbeschränkt, zur Vereinfachung wurde sich jedoch auf reine Textdateien (.txt) beim Export beschränkt.

    Das Programm schreibt nun alle Anforderungen, getrennt durch Leerzeilen, in die Datei, sodass schlussendlich alle Anforderungen als Boilerplates exportiert sind.

4.3 Mögliche Erweiterungen

In diesem Abschnitt werden mögliche Verbesserungen bzw. Erweiterungen des Programms aufgelistet und erläutert. Die Erweiterungen sind zur Übersichtlichkeit in drei wichtige Bereiche unterteilt:

1. Benutzeroberfläche: Hier werden die Erweiterungen dargestellt, welche im direkten Zusammenhang mit der Nutzerinteraktion des Programms stehen und neue Ansichten auf der Benutzeroberfläche ermöglichen.

2. Nutzbarkeit: Hierzu gehören alle Erweiterungen, welche die Verwendbarkeit des Programms verbessern, jedoch nicht durch den Nutzer gesteuert werden. Dazu gehören die Aspekte Performanz, Abdeckungsgrad und präzise Sprachverarbeitung.

3. Konzept: Hier sind alle Erweiterungen aufgelistet, welche das Programm konzeptuell verändern, wichtige Vorarbeiten leisten oder alternative Technologien verwenden.

4.3.1 Benutzeroberfläche

Im Folgenden sind die Erweiterungen zur Verbesserung der Nutzeroberfläche und dessen Funktionalität aufgeführt.

Kontext-Anzeige

Eine mögliche Verbesserung der Benutzeroberfläche könnte durch das Anzeigen von relevantem Kontext zu der aktuellen Anforderung erzielt werden. Dies bedeutet, dass der Nutzer die aktuelle Anforderung besser einordnen kann, wenn die beispielsweise mit der vorherigen oder der darauffolgenden zu tun hat.

Wie weit der relevante Kontext ausfällt ist hierbei an konkreten Lastenheften zu testen. Mögliche Ausführungen wären etwa das Anzeigen der umgebenden Anforderungen oder der letzten Information vor der Anforderung.

Außerdem könnte dies von entscheidender Bedeutung bei der Auflösung von Referenzen (wie zum Beispiel durch Pronomen) sein. Wörter wie „it“, „that“ usw. sollten zwar in Lastenheften wegen der Entstehung von Uneindeutigkeiten nicht verwendet werden, jedoch kann man dies in der Praxis nicht annehmen. Daher wären die Markierung von betroffenen Wörtern oder gar deren automatische Auflösung sehr nützlich.

Markieren von Unterschieden

Eine wichtige Hilfestellung bei der Übersichtlichkeit und somit bei der Verarbeitungsgeschwindigkeit durch den Programmnutzer könnte durch die farbliche Markierung von relevanten Textstellen erzeugt werden.

Wie in Abbildung 4.21 gezeigt, werden der Ursprungssatz und mögliche Übersetzungen in Boilerplates auf dem selben Nutzer-Bildschirm abgebildet. Jedoch kann es durch die vorgegebene Satzstruktur der Boilerplates zur Umstellung der Eingabesatzes kommen. Dies ist für den Nutzer nur schwer auf den ersten Blick zu erkennen.

Eine farbige Markierung der wichtigen Satzbestandteile im Eingabesatz und der entsprechenden Gegenstücke in der übersetzten Boilerplate könnten dort für einen sofortigen Überblick sorgen. Die Markierung könnte abhängig von den durch GATE vergebenen Annotationen stattfinden und würde es dem Nutzer ermöglichen, einige der angezeigten Möglichkeiten direkt auszuschließen bzw. zu bestätigen.

Vorher/Nachher Abgleich

Falls der Nutzer beabsichtigt, die Ergebnisse der Dokumentkonvertierung noch einmal zu überprüfen, könnte eine eingebaute Ansicht sehr nützlich sein. Diese zeigt das Eingabe- und das Ausgabedokument nebeneinander an. Gegebenenfalls könnten die Eingabe-Anforderungen passend zu den entsprechenden Übersetzungen markiert oder gefärbt werden.

Dadurch würden Fehler durch den Nutzer noch weiter eingeschränkt und somit die Konsistenz des Ausgabedokuments stark verbessert. Vor allem für die spätere Weiterverarbeitung durch den Delta-Analyser ist dies wichtig.

Eingabe des Dokumenttyps

Eine weitere Möglichkeit, für den Nutzer das Programm zu verbessern, könnte darin bestehen, die Art des Eingabedokuments zu spezifizieren. Da in der Automobilindustrie viele verschiedene OEM mit verschiedenen Standards Lastenhefte für die Zulieferer verfassen, sind die eingehenden Anforderungsdokumente sehr heterogen und können auch bei der Formatierung voneinander abweichen.

Wenn der Nutzer jedoch die Möglichkeit hat, z.B. den OEM schon vor der Verarbeitung einzugeben und das Programm dementsprechend dynamisch angepasst wird, könnten auch ohne großen Aufwand für den Nutzer die Lastenhefte verschiedener OEMs verarbeitet werden.

4.3.2 Nutzbarkeit

Hier werden die Erweiterungen behandelt, welche sich positiv auf die Nutzbarkeit des Programms auswirken könnten. Dazu gehören vor allem Performanz- und Automatisierungsaspekte.

Implementierung weiterer Boilerplates

Eine naheliegende Verbesserung des Programms wäre das Umsetzen weiterer Boilerplates. Dies ist jedoch nicht nur wichtig, um die Abdeckung des Testes zu verbessern, sondern auch, um die Auswahl an Boilerplates zu verbessern.

Zunächst ergibt sich durch das Implementieren weiterer Boilerplates die Möglichkeit, mehr Anforderungen des Quelldokuments in Boilerplates zu überführen. Damit geht jedoch auch ein Problem einher. Bei mehr Boilerplates steigt auch die Anzahl der vorgeschlagenen Boilerplates zu den ein-

zelnen Sätzen. Im Optimalfall wird jedoch nur genau eine Boilerplate als Übersetzung vorgeschlagen, und zwar die richtige.

Es gilt also, einen Anstieg an unpassend vorgeschlagenen Boilerplates so gering wie möglich zu halten und somit die Nutzbarkeit des Programms zu gewährleisten. Dies lässt sich erreichen, indem man die hinzugefügten Boilerplates abhängig von den verwendeten Satzbestandteilen in einer Hierarchie anordnet.

Angenommen eine Boilerplate *A* enthält die Satzbestandteile *1*, *2* und *3* und die Boilerplate *B* enthält *1*, *2*, *3* und *4*. Ein Satz, der die Bestandteile *1-4* beinhaltet, würde in beide Boilerplates konvertierbar sein. Da die Boilerplate *A* jedoch Bestandteil *4* nicht berücksichtigen kann, führt dies zu Informationsverlust und ist somit falsch. Es lässt sich also schließen, dass die Boilerplate A niemals richtig sein kann, sobald Boilerplate *B* eine mögliche Übersetzung darstellt. Somit muss auch Boilerplate *A* nicht als möglicher Vorschlag angegeben werden.

JAPE-Optimierung

Die in der Sprachverarbeitung implementierten JAPE-Regeln wurden lediglich im Hinblick auf Funktionalität entworfen. Aufgrund der generell niedrigen Laufzeit[14] wurde die Performanz der Regeln nicht extra optimiert. Es ist also möglich, dass dort noch Verbesserungspotential ausgeschöpft werden kann.

Die Implementierung weiterer Boilerplates sollte die Laufzeit des Programms nicht sonderlich beeinflussen, da der mit Abstand größte Annotations-Aufwand bei dem Tokenizer und dem Part-Of-Speech-Tagger liegt. Diese werden jedoch auf jeden Fall genau ein mal ausgeführt.

Außerdem ist es möglich die JAPE-Transducer durch sogenannte *JAPE+* -Transducer zu ersetzen. Diese sind vor allem bei der Verarbeitung langer Dokumente sehr viel effizienter als herkömmliche JAPE-Transducer. Daher sind sie in dem Anwendungsfall mit Anforderungsdokumenten besonders geeignet. Abbildung 4.26 zeigt Testergebnisse der GATE-Entwickler zu den verschiedenen Transducern [GM19].

[14] Die Testergebnisse zur Laufzeit des Programms werden in Abschnitt 4.4.1 genauer erläutert

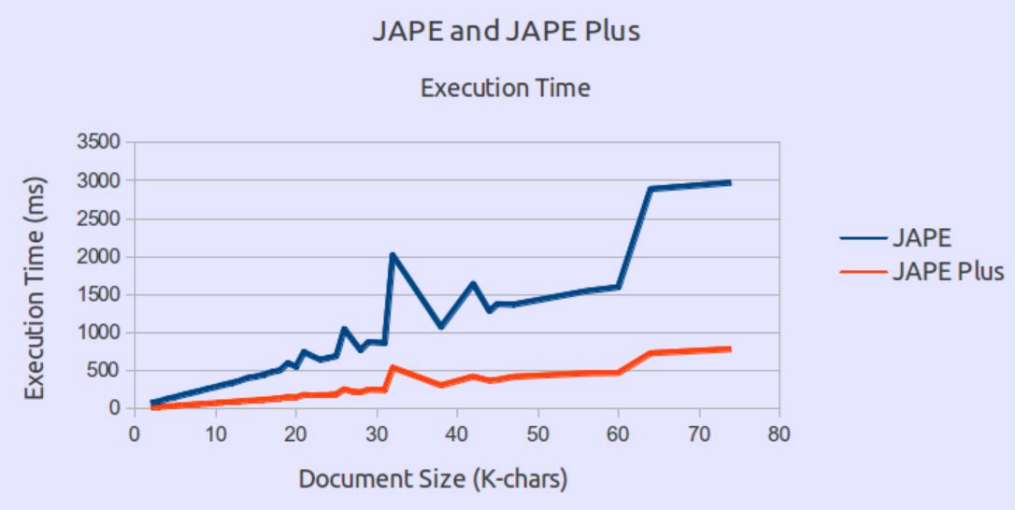

Abbildung 4.26: JAPE und JAPE+ Zeitaufwand abhängig von der Dokumentlänge [GM19]

Größtmögliche Automatisierung

Um dem Nutzer so viel Arbeit wie möglich zu ersparen, ist es wichtig, so viele Arbeitsschritte wie möglich zu automatisieren. Dabei darf die Fehlerquote jedoch nicht steigen. Unter der Annahme, dass ein Nutzer 99% der Anforderungen richtig verarbeitet, könnte man die Verarbeitung von Anforderungen bei einer Trefferquote des Programms von über 99% automatisieren.

Dies könnte beispielsweise geschehen, indem bei bestimmten Boilerplates mit sehr hoher Trefferwahrscheinlichkeit der Nutzer die Boilerplate nicht mehr überprüfen und bestätigen muss. Voraussetzung für eine Automatisierung könnte etwa eine Trefferquote von 99,5% sein, in Verbindung mit nur einer möglichen vorgeschlagenen Boilerplate.

Um dies zu realisieren sind jedoch die Implementierung einer ausführlichen Heuristik und weitläufige Tests nötig. Sollte dies gelingen, könnte der Nutzen enorm sein, da die Nutzerinteraktion mit großem Abstand die meiste Zeit der Verarbeitung benötigt.

Filtern von Anforderungen

Eine weitere Möglichkeit, die Zeit der Nutzerinteraktion zu verringern könnte durch das Filtern von Anforderungen aus dem Dokument realisiert werden. Dies wird im Programm bereits teilweise getan, so wurden Zeilen des DOORS-Exportes wie
`H_ObjectContent: Requirement` und `ID: 382`
bereits nicht mehr dem Nutzer angezeigt. Aus diesen könnten jedoch auch einige Informationen genutzt werden, wie die Erfassung und Speicherung der ID-Nummer und das Erkennen des *ObejctContent*. Damit könnte außerdem abgewogen werden, ob es möglich ist, auf die Verarbeitung der *Information*-Objekte zu verzichten und sich auf die *Requirement*-Objekte zu beschränken. Dies kann jedoch in der Praxis schwierig sein, da oftmals *Information* und *Requirement* nicht zuverlässig deklariert werden oder ein Großteil der Sätze mit *TBD*(to be determined/ noch zu bestimmen) gekennzeichnet sind.

4.3.3 Konzept

In diesem Abschnitt werden Modifizierungen des Programms auf der Konzeptebene erläutert. Dies beinhaltet etwa das Nutzen anderer Technologien und Anbindungs-Konzepte.

Preprocessing

Ein großes Problem bei der Verarbeitung der aus DOORS exportierten PDF-Dateien war die Zerstückelung der Sätze, auf denen die Sprachverarbeitung aufbaut. Oftmals waren beispielsweise Anforderungen aufgeteilt in einen Hauptteil und danach eine Auflistung von Bedingungen als eigenen Anforderungen.

Dies ist problematisch, weil die Sprachverarbeitung auf ganzen Sätzen beruht und aus Bruchstücken nur schlecht wichtige Informationen filtern kann. Daher wäre eine Preprocessing des Dokuments hilfreich, welches etwa den Hauptteil noch einmal vor jede der Bedingungen knüpft. Später wären die Bedingungen außerdem besser miteinander verbunden, da immer der gleiche Hauptteil erkannt wurde.

Des Weiteren könnte schon im Preprocessing versucht werden, einige grammatikalische Schwierigkeiten wie fehlende Kommasetzung und Nutzung von Referenzen oder Passiv-Konstruktionen zu verhindern.[15]

[15]Passiv-Konstruktionen sind nicht direkt falsch, sind jedoch in vielen Standards falls

Direkte Anbindung an DOORS und verschiedene Dateiformate

Im betrieblichen Kontext ist ein besonders wichtiger Aspekt des Programms die An- bzw. Einbindung in bestimmte Arbeitsabläufe. Da ein Großteil des Requirements Engineering mit Hilfe von DOORS durchgeführt wird, ist eine gute Anbindung an das Werkzeug sehr zu empfehlen.

Andere Programme wie z.B. *ReqPat*, welche in der Abteilung für RE verwendet werden, befinden sich bereits in DOORS oder haben zumindest eine Anbindung. Außerdem stellt DOORS auch einige der zuvor in diesem Kapitel vorgeschlagenen Erweiterungen wie Benutzeransichten zur Verfügung.

Zukünftig könnte es ansonsten auch nützlich sein, das Programm an verschiedene Dateiformate wie .pdf und .docx anzupassen. Diese können in GATE bereits eingelesen und verwendet werden [RS18], jedoch werden dabei Bilder nicht berücksichtigt. Außerdem liegen in GATE nur noch den Klartext und die entsprechenden Annotationen vor, also auch keine Formatierung oder ähnliches.

Wenn als Ergebnis der R2BC-Konvertierung also wieder eine .pdf- oder .docx-Datei exportiert werden soll, ist diese selbst zu generieren. Eine Hilfestellung könnte jedoch die von GATE bereitgestellte *Doc.alter()*-Methode bieten. Diese ermöglicht es, Text an der entsprechenden Stelle des GATE-internen Dokuments direkt zu ändern und dabei die Reihenfolge des Textabschnittes nicht zu verfälschen.

Machine Learning

Während des Entwicklungsprozesses wurde oftmals nach der Nutzung von *Machine Learning* bzw. künstlicher Intelligenz (*KI*) gefragt.

Anhaltspunkt für die Nutzung von KI ist etwa die Verbesserung der Präzision (bis hin zur möglichen Automatisierung) durch das Erfassen von Nutzerentscheidungen. Dazu wird davon ausgegangen, dass der Nutzer aus einer Menge von Vorschlägen immer den richtigen (bzw. keinen falls nicht vorhanden) akzeptiert.

Der Grundstein für diese Erfassung ist durch die Modularität des Programms bereits gegeben. Die in Abs. 4.2.2 beschriebenen internen Listen beinhalten immer alle ausgegebenen Vorschläge und die jeweilige Entscheidung des Nutzers, was den Abgleich und die Datensammlung stark vereinfachen sollte.

möglich zu vermeiden

Außerdem könnten Ansätze des Machine Learning verwendet werden, um die Erkennung bestimmter Satzbestandteile zu verbessern und somit die Präzision des Werkzeugs zu erhöhen.

4.4 Test

In diesem Abschnitt wird die Funktionalität des R2BC anhand von realen Tests geprüft. Wie bereits erwähnt, bestand durch die Zusammenarbeit mit RE-Experten bei HELLA die Möglichkeit, die Konvertierung unter Verwendung realer Lastenhefte aus dem betrieblichen Alltag zu testen. Dazu konnten diverse Indikatoren für die Trefferquote des Programms definiert werden, die während der Konvertierung von drei zur Verfügung gestellten Lastenheften gemessen wurden.

Die Tests, die für dieses Kapitel durchgeführt wurden, sind dabei bereits in der Veröffentlichung [ZH19] beschrieben, die parallel zu dieser Arbeit veröffentlicht wurde.

4.4.1 Methodik

In diesem Abschnitt wird das genaue Testszenario hinsichtlich der zu messenden Werte definiert. Das Programm entspricht dem unter 4.2.3 skizzierten Ablauf bei der Nutzung, d.h. die möglichen Konvertierungen wurden durch den R2BC vollautomatisch ermittelt und erzeugt. Zu jeder Anforderung wurden dann diese Konvertierungen angezeigt und bewertet. Auf dieser Basis konnten korrekte und fehlerhafte Verhaltensweisen des Programms gezählt und nachvollzogen werden.

Konkret existieren zwei primäre Werte, anhand derer die Erkennungsrate durch die JAPE-Regeln zu den Boilerplates und Übersetzungen im Converter evaluiert werden können. Bei der Konvertierung der Lastenheften sind dabei folgende Fehler, die aufgetreten sind, zu unterscheiden:

- *Recall - Erkennungsgrad der Boilerplates*

  Als recall wurde gemessen, ob alle passenden Boilerplates typmäßig zu einer Anforderung gefunden wurden. Es wird gemessen, ob zu einem Satz des Quelldokuments, zu dem eine passende Konvertirung existiert, diese auch im Programm gefunden wird. Sollte eine Konvertierung fehlen, obwohl sie zu der Anforderung gepasst hätte, liegt ein Fehler in der Erkennung vor.

- *Precision - Übersetzung in erzeugte Boilerplate*

  Der precision-Score stellt dar, wieviele von den richtig erkannten Boilerplates (*recall*) anschließend korrekt in eine erzeugte Boilerplate-Anforderung übersetzt wurden. Hier liegt ein Fehler immer dann vor, wenn z.B. Teile eines *CompleteSystem* oder einer *Condition* bei der Übersetzung abgeschnitten wurden und somit in der Boilerplate fehlen. Dies kann etwa dann passieren, wenn nur ein Teil des Systemnamens im Gazetteer enthalten ist.

Testdokumente

Im Rahmen der Tests wurden insgesamt drei Lastenhefte getestet, die aktuelle Produktspezifikationen von zu entwickelnden Fahrzeugkomponenten enthalten. Diese wurden dabei als repräsentative Dokumente von den RE-Experten zur Verfügung gestellt. Dazu wurden sie aus dem Programm *DOORS* als .pdf exportiert.[16]

Informationen

	Seiten	Absätze	Ladezeit	Konvertierung	Dateigröße
LH1	88	2693	0,07 sek.	10,97 sek.	344 KB
LH2	567	17766	0,29 sek.	98,63 sek.	3492 KB
LH3	60	1669	0,15 sek.	7,71 sek.	1156 KB

Tabelle 4.1: Metainformationen zu Testlastenheften [eigene Darstellung]

Tabelle 4.1 enthält die Metainformationen zu den getesteten Anforderungsdokumenten, die vor bzw. während des Programmdurchlaufs ermittelt wurden.[17] Auf dieser Basis konnte dann zu jeder Anforderungen die Programmausgabe auf die Genauigkeit hin geprüft werden.

Lastenheft 1 (LH1) enthält demnach 2693 Anforderungen und wurde innerhalb von 10,97 Sekunden konvertiert. LH1 stellt inhaltlich eine Teilspe-

[16] Beim DOORS-Export sind neben den Anforderungen als Text auch Bilder und Meta-Informationen wie etwa die Objekt-ID der Anforderung als DOORS-Objekt exportiert worden, die für die Tests nicht mit konvertiert wurden.

[17] Absätze repräsentieren stellvertretend einzelne Anforderungen, da im Programm die *sentence*-Annotation von GATE zur Trennung von Anforderungen verwendet wird. Dabei kann es vorkommen, dass mehrere Anforderungen als ein Satz zusammengefasst analysiert wurden und dass etwa Aufzählungen als verschiedene Sätze annotiert wurden. Dies hat keinen direkten Einfluss auf die Konvertierung.

zifikation des mit 567 Seiten und 17766 Anforderungen wesentlich größeren Lastenheft 2 dar, dass innerhalb von ca. 99 Sekunden automatisch konvertiert wurde. LH3 enthält einige bruchstückhafte bzw. teilvollständige Anforderungen; der R2BC hat 1669 Sätze unterschieden.

4.4.2 Durchführung

LH1 wurde zunächst manuell durchlaufen, um auf dieser Basis Regelmäßigkeiten in der Satzstruktur zu finden und Gazetter-Einträge hinzuzufügen. Dabei wurden aus jedem Dokument Systemnamen und -eigenschaften hinzugefügt. Auf dieser Basis wurden dann die JAPE-Regeln für die Analyse definiert. In Abs. 4.3 werden Ansätze zum automatischen Sammeln der Entitäten beschrieben. Bis auf den Gazetteer, der spezifische Eigennamen für z.B. Komponenten enthält, wurde für alle drei Lastenhefte mit dem identischen Programm getestet.

Testresultate

	Boilerplate 65c		Boilerplate 85	
	Recall	Precision	Recall	Precision
LH1	100%	100%	89,3%	66,7%
LH2		93,3%		67,7%
LH3		100%		84,7%

Tabelle 4.2: Testergebnisse nach R2BC-Durchlauf [eigene Darstellung]

Wie in Tab. 4.2 sichtbar ist, wurden bei LH1 für die Boilerplate 65c 100% recall und 100% precision erreicht. Der *recall*-Wert zeigt, dass alle passenden Anforderungen für diese Boilerplate erkannt wurden. 100% Präzision der Übersetzung lassen sich auf den Umstand zurückführen, dass alle zu erkennenden Systemnamen für die BP65c im Gazetteer vorhanden waren und dementsprechend gefunden werden konnten. Nichtsdestotrotz wurde bei 37,5% der Boilerplates das *completeSystemName* unvollständig eingetragen, wenn mehrere Einträge im Gazetteer vorhanden waren. Wenn im Gazetteer etwa sowohl *controller*, als auch *headlamps controller* enthalten sind, wurde stets nur *controller* in die Boilerplate eingetragen, obwohl *headlamp controller* als System in der Anforderung stand. Dieser prinzipielle Mangel lässt sich jedoch durch Optimierung oder hierarchische Anordnung mehrerer Gazetters beheben und wurde daher nicht als *precision*-Fehler gewertet.

Für Boilerplate 85 wurde 66,7% Erkennungsrate und 89,3% korrekte Übersetzung ermittelt. Wurde also eine bedingte Anforderung erkannt, hat das Programm diese in den meisten Fällen auch korrekt übersetzt. Eigenheiten der Formulierung der Autoren und Rechtschreibfehler in den Anforderungen haben den *recall*-Score verringert. Manche Autoren nutzen etwa „*In the event of*“ anstelle von „*if*“ (zu Deutsch: „Im Fall von“ anstatt „Wenn“) oder haben nach dem *if* fälschlicherweise kein Komma gesetzt. Auch wurde beobachtet, dass manche Anforderungen mehrere Bedingungen auf einmal oder verschachtelte Bedingungen enthielten, die zu einer Uneindeutigkeit der Anforderung führen können.

Lastenheft 2 wurde, aufgrund des großen Umfangs des Dokumentes, nur auf die Präzision der Übersetzung hin geprüft. Bereits durch den Test von LH1 konnte manches Verbesserungspotential in der Erkennung der Boilerplates identifiziert werden, weshalb nur die Korrektheit der übersetzten Boilerplates getestet wurde. LH2 wurde außerdem nicht gänzlich, sondern nur auf Basis der ersten 400 Seiten getestet. Die Konvertierungszeit belief sich dabei auf lediglich 89 Sekunden und hat zu positiven Testresultaten geführt.

Boilerplates 65c wurde mit 93,3% Genauigkeit übersetzt, was sich wieder auf den Einsatz des Gazetteers zurückführen lässt. Teilweise wurden erneut Satzbestandteile nur teilweise erkannt bzw. in die Boilerplate übersetzt, wenn mehrere Gazetteereinträge einen Teilwort aus dem Satzteil enthielten.

Boilerplate 85 wurde zu 67,6% korrekt erzeugt. Dies lässt sich erneut damit begründen, dass die BP85 nur für eine Bedingung pro Anforderung geeignet ist, manche Autoren aber fälschlicherweise mehrere Bedingungen für eine Systemfunktion in eine Anforderung verpacken. Aus diesen Anforderungen werden dann die zusätzlichen Bedingungen bei der Konvertierung abgeschnitten.

Lastenheft 3 zeigte beim Test ähnliche Werte für *precision*, wie durch den Test von LH2. Auffällig war, dass Teilsätze durch den R2BC teilweise zu mehreren neuen Sätzen geführt haben, die inhaltlich aber nicht zur Füllung von Boilerplates genügt haben. Dieser Umstand lässt sich auf die Textformatierung zurückführen bzw. auf die Verwendung der *sentence*-Annotation, die Sätze nicht korrekt abgrenzt. Daher wurde auch hier auf die Messung des *recall*-Wertes verzichtet.

4.4.3 Bewertung der Ergebnisse

Insgesamt lässt sich jedoch sagen, dass insbesondere Precision-Fehler durch die Bewertung der Anforderung in der GUI leicht auffallen und korrigiert werden können. Diese Fehler sind meist weniger drastisch und erfordern nur kleine Korrekturen, etwa wenn ein Systemname nicht ganz korrekt übernommen wurde. Ein Großteil der Arbeit im Programm entsteht bei der Konvertierung, die keinen Eingriff und Aufmerksamkeit des Nutzers erfordert.

Momentan arbeitet der R2BC semi-automatisch, da der Nutzer jede Konvertierung manuell bestätigen muss. Bei hinreichend hohen Trefferquoten und Implementierung einer Heuristik, die die best-passende Konvertierung aus mehreren Möglichkeiten identifiziert, wäre jedoch auch ein vollautomatischer Einsatz des Werkzeugs denkbar. Parallel zum Test der Funktionalität wurde daher auch die Bearbeitungsdauer der Lastenheften unter verschiedenen Programmeinstellungen gemessen.

Bei der manuellen Prüfung jeder Anforderung im Programm ließ sich LH2 innerhalb von ca. 42 Min. verarbeiten, da das Programm weiterhin einzelne Anforderungen anzeigt und der Nutzer diese nur überspringen muss, sollte keine Boilerplate existieren.

Wenn der Nutzer nur gefundene Boilerplates bewerten muss (nur die Übersetzung) und automatisch alle nicht-konvertierbaren Sätze übernommen werden, belief sich die Verarbeitungszeit auf lediglich 8,37 Min. . Dabei fallen aber mögliche Boilerplates zu Anforderungen, die fälschlicherweise nicht erkannt wurden, nicht auf (*recall* wird ignoriert).

Insgesamt lässt sich daher sagen, dass der Ansatz des R2BC wesentliches Potential für Zeitersparnisse gegenüber der manuellen Prüfung der Lastenhefte im RE bietet.

Kapitel 5

Delta-Analyser (DA)

In diesem Kapitel wird das zweite entwickelte Tool, der *Delta Analyser* (DA), vorgestellt. Dieser wird dafür verwendet, zwei Lastenhefte, welche zuvor durch den R2BC verarbeitet wurden, abzugleichen.

Dabei sollen für den Nutzer wichtige Informationen über Unterschiede und Gemeinsamkeiten erkannt, gefiltert und in Form eines *Delta-Reports* ausgegeben werden. Auf Basis des Delta-Reports soll dann die Arbeit mit dem OEM-Lastenheft stark vereinfacht und eine Kostenabschätzung für dessen Umsetzung ermöglicht werden.

Ähnlich zur Vorstellung des R2BC wird dabei zunächst die Architektur des Programms vorgestellt. Daraufhin werden wichtige Aspekte der Implementierung erläutert und mögliche Erweiterungen des Programms aufgeführt.

5.1 Architektur

In diesem Abschnitt wird die Klassenstruktur des Programms vorgestellt und erläutert, um einen groben Überblick über die Funktion des Programms zu geben.

Die verwendete Software ist die selbe wie für den R2BC (siehe Abschnitt 4.1.1), weswegen der entsprechende Abschnitt für den Delta-Analyser entfällt. Die einzige Ausnahme ist die Verwendung von GATE.

Da die Sprachverarbeitung durch GATE bereits im R2BC integriert ist, können dessen Ergebnisse im DA weiterverwendet werden[1]. Daher ist es nicht

[1]Dies wird im Implementierungs-Abschnitt genauer erläutert.

nötig, GATE explizit in den DA zu integrieren oder dort eine sprachliche Analyse durchzuführen.

5.1.1 Klassenarchitektur

Der Delta Analyser ist aufgeteilt auf drei Pakete mit insgesamt sieben Klassen[2] (und einer *Enumeration*). Das Paket *DAGUI* enthält dabei gesondert die Benutzeroberfläche in der *DAMainWindow*-Klasse. Die restlichen Ressourcen befinden sich zusammen in dem *DA*-Paket.

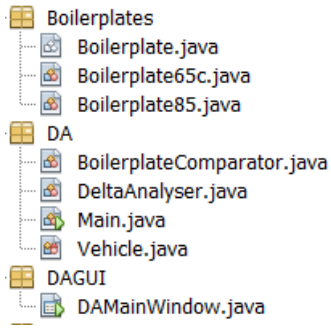

Abbildung 5.1: Zuordnung der DA-Klassen zu Paketen nach Abb. 3.8 [eigene Darstellung]

Abbildung 5.2 zeigt ein UML-Klassendiagramm des Programms mit den Klassen, sowie deren relevanten Attributen und Methoden. Im Folgenden sind die einzelnen Ressourcen kurz erläutert.

1. *DAMainWindow.java*: Diese Klasse wurde als *Swing-JFrame* angelegt und für den Entwurf der Benutzeroberfläche verwendet. Neben der Deklaration der verschiedenen GUI-Elemente, enthält diese Klasse auch alle nötigen *Getter*- und *Setter*-Methoden um die entsprechenden Elemente von den anderen Klassen aus verwenden zu können.

2. *Main.java*: Die Hauptklasse des Programms wird verwendet, um ein Objekt der *Delta Analyser*-Klasse (beschrieben im nächsten Punkt) zu instanziieren und danach die GUI zu initialisieren.

[2]Das Boilerplate-Paket mit dessen drei Klassen wurde im R2BC-Kapitel bereits ausreichend erläutert und wird daher hier nicht wiederholt.

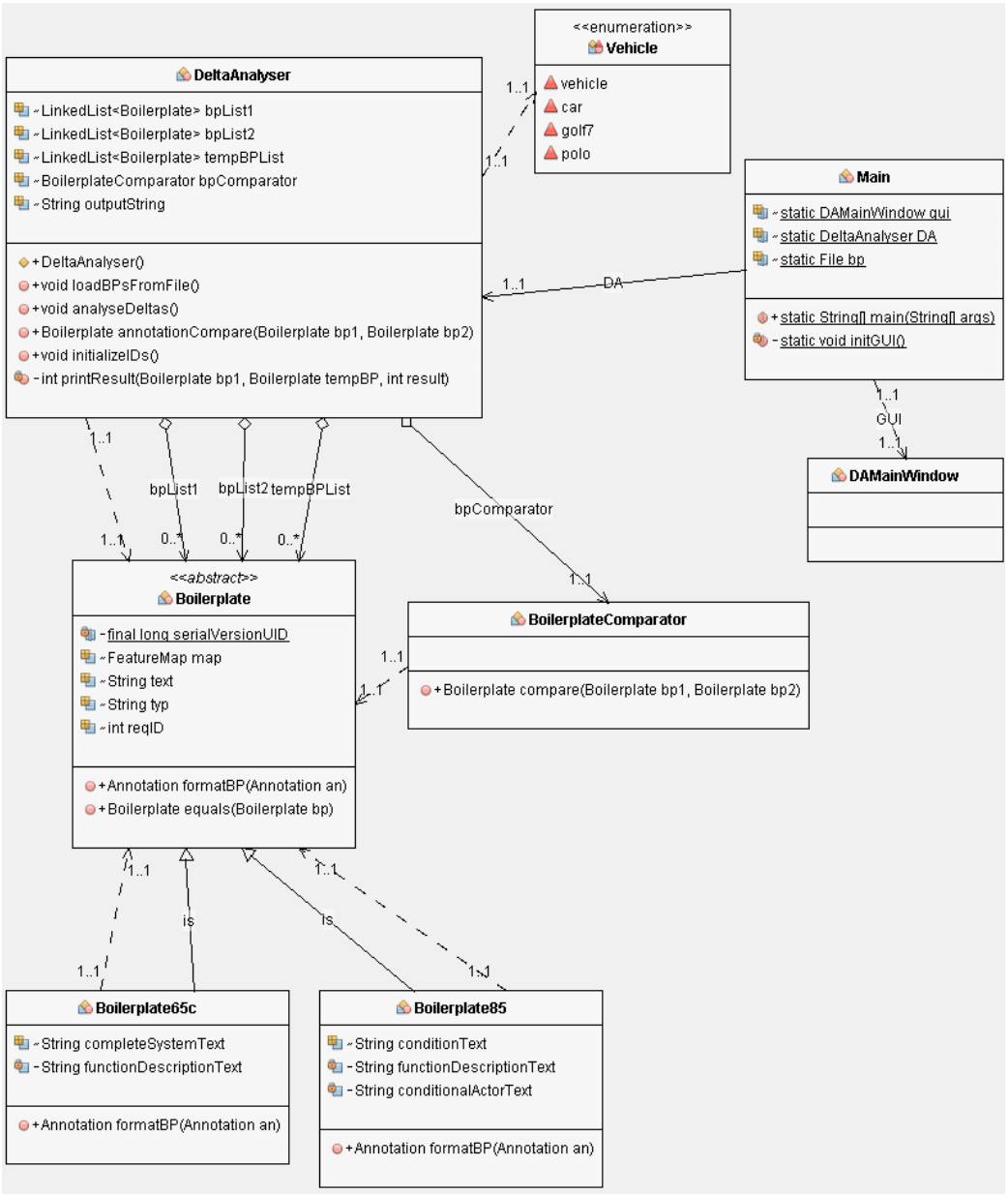

Abbildung 5.2: UML-Klassendiagramm DA [eigene Darstellung]

Beim Initialisieren der GUI werden den in der *DAMainWindow*-Klasse erstellten Oberflächen-Objekten über sogenannte *ActionListener* die

Funktionalitäten verliehen. Diese werden dann beim Betätigen der Oberfläche ausgeführt.

3. *DeltaAnalyser.java*: In dieser Klasse befindet sich der Hauptteil der Verarbeitungslogik. Zunächst werden hier die Boilerplate-Listen deklariert, in welche die Ergebnisse des R2BC geladen werden können. Außerdem wird hier ein Objekt der *BoilerplateComperator*-Klasse instanziiert (beschrieben im nächsten Punkt). Die Kernfunktionalitäten des Programms werden in den Methoden

   - *loadBPsFromFile*,
   - *analyseDeltas*,
   - *annotationCompare*
   - und *printResult*.

   implementiert. Diese werden jeweils genau im Implementierungsabschnitt erläutert.

4. *BoilerplateComparator.java*: Diese Klasse dient als Vergleichsklasse von zwei Boilerplate-Objekten. Dazu ist hier die *compare*-Methode implementiert, welche die Boilerplates auf Vergleichbarkeit prüft und eine Voreinordnung vornimmt. Dies ist unabhängig von der Art der zu vergleichenden Boilerplates möglich.

5. *Vehicle.java*: Diese Aufzählung von möglichen Fahrzeugbezeichnungen wurde als möglicher Platzhalter für andere Entitätserkennungen in das Programm integriert. Alternativ könnten auch Gazetteers oder Ontologien (siehe mögliche Erweiterungen) verwendet werden.

   Die Aufzählung wird dazu verwendet, voneinander unterschiedliche Elemente eventuell einer gemeinsamen Gruppe zuordnen und somit zwischen komplett oder nur leicht unterschiedlichen Elementen unterscheiden zu können. Dazu ist in der Aufzählung die *contains*-Methode implementiert, welche ein Element auf das vorkommen in der Aufzählung prüfen kann.

5.2 Werkzeug-Implementierung

In diesem Abschnitt werden wichtige Aspekte der Implementierung des Delta Analysers erläutert. Wichtig sind dabei vor allem die interne Haltung der zu vergleichenden Boilerplates und die Methoden zum Vergleichen und Bewerten dieser. Außerdem wird die Nutzerinteraktion mit dem Programm dargestellt.

5.2.1 Analyse-Funktionalität

Haltung der Boilerplates

Um in der Lage zu sein, zwei komplette Lastenhefte zu vergleichen, sind zwei Datenstrukturen nötig, welche jeweils eines der Lastenhefte komplett beinhalten. Hierzu kann man praktischerweise die im R2BC erstellten Boilerplate-Listen verwenden. Entsprechend der *bpExportList* des R2BC ist es ausreichend, wenn diese Listen eindimensional sind und Boilerplate-Objekte aufnehmen können.

Diese Listen sind Teil der *DeltaAnalyser*-Klasse und können durch die *loadBPsFromFile*-Methode befüllt werden. Sie wird von den zwei *Import*-Buttons aufgerufen (genauer bei Nutzerinteraktion erläutert), deserialisiert eine gespeicherte Boilerplate-Liste und speichert diese in die entsprechende interne Liste ab.

Als Platzhalter für das automatische Auslesen der Requirements-IDs (siehe R2BC/mögliche Erweiterungen 4.3) wurde die *initializeIDs*-Methode implementiert. In dieser werden die internen Listen iteriert und die enthaltenen Boilerplate-Objekte fortlaufend nummeriert, um den Vergleich der Boilerplates später nachvollziehbarer zu machen.

Vergleich der Boilerplates

Der Vergleich der Lastenhefte findet vorrangig in der *analyseDeltas*-Methode statt und wird dort teilweise an andere Methoden ausgelagert.

In *analyseDeltas()* werden die beiden internen Lastenhefte in einer geschachtelten *for*-Schleife durchlaufen. In der äußeren Schleife wird das OEM-Lastenheft und in der inneren das internen Lastenhefts iteriert. Es wird also für jede Boilerplate im OEM-Dokument jede Boilerplate des internen Dokuments durchlaufen.

Dabei wird für eine grobe Voreinordnung die *compare*-Methode der *BoilerplateComparator*-Klasse aufgerufen. Diese prüft den Type der Boilerplates

auf Gleichheit. Ist dieser gleich, wird die komplette Boilerplate abgeglichen. Bei einem Treffer gibt die Methode 0 zurück, bei dem gleichen Typen eine 1 und bei keinerlei Übereinstimmung eine 2.

Abhängig von diesem Ergebnis wird in der *analyseDeltas*-Methode weiter vorgegangen.

- Bei einer 0 ist eine perfekte Übereinstimmung gefunden und die Suche muss für diese Boilerplate nicht weitergeführt werden.

  Die interne Boilerplate, welche den Treffer ergab, wird zunächst in ein Boilerplate-Objekt namens *tempsBP* abgespeichert. In diesem Objekt befindet sich immer die bisher passendste interne Boilerplate zu der momentanen OEM-Boilerplate.

- Wurde eine 1 zurückgegeben, wird auf den beiden Boilerplates die *annotationCompare*-Methode aufgerufen. Hier werden auch einzelne Satzbestandteile der beiden Boilerplates verglichen. Die vorherige Überprüfung der Boilerplate-Typen garantiert, dass hier vergleichbare Satzbestandteile vorliegen.

```
public int annotationCompare(Boilerplate bp1, Boilerplate bp2) {
    switch (bp2.getTyp()) {
        case "65c": //Type Boilerplate65c
            Boilerplate65c b651 = (Boilerplate65c) bp1;
            Boilerplate65c b652 = (Boilerplate65c) bp2;
            if (b651.getCompleteSystemText().equals(b652.getCompleteSystemText())) {
                //Same CompleteSystem
                return 1;
            } else if ((Vehicle.contains(b651.getCompleteSystemText().toLowerCase())
                    && Vehicle.contains(b652.getCompleteSystemText().toLowerCase()))) {
                //Generalization of CompleteSystem is used
                return 2;
            } else {
                //Systems are different, but same BP.
                return 3;
            }
```

Abbildung 5.3: Ausschnitt der *annotationCompare*-Methode [eigene Darstellung]

An dieser Stelle kommt die *Vehicle*-Aufzählung zum Einsatz und ermöglicht es, Satzbestandteile wie *CompleteSystemText* nicht nur auf komplette Gleichheit, sondern auch auf Typgleichheit zu prüfen. Hierzu wird zu beiden Systemtexten mit Hilfe der *contains*-Methode überprüft, ob sie in der gleichen Aufzählung auftauchen.

Abhängig des Übereinstimmungsgrads der Satzbestandteile kann dann eine genauere Bewertung der Unterschiede erfolgen. Ist diese Bewertung besser als die momentane Bewertung der *tempBP*-Boilerplate, so wird diese durch die betroffene neue Boilerplate ersetzt und die Suche wird fortgesetzt.

- Gibt die *compare*-Methode eine 2 zurück, so sind die verglichenen Boilerplates nicht kompatibel und es wird im HELLA-Dokument weiter nach einer passenderen Boilerplate gesucht.

Abbildung 5.4 zeigt den hier beschriebenen Ausschnitt der *analyseDeltas*-Methode.

```
for (Boilerplate bp1 : bpList1) {
    result = 6;
    for (Boilerplate bp2 : bpList2) {
        switch (bpComparator.compare(bp1, bp2)) {
            case 0: //Identical
                result = 0;
                tempBP = bp2;
                break;
            case 1: //Comparable with Deltas
                //If we can reach a smaller result
                if (result > 1 && annotationCompare(bp1, bp2) < result) {
                    //Just do it
                    result = annotationCompare(bp1, bp2);
                    tempBP = bp2;
                }
                break;
            case 2: //Completely different
                if (result > 5) {
                    result = 5;
                    tempBP = bp2;
                }
                break;
        }
        System.out.println(result);
        //No further search necessary
        if (result == 0) {
            break;
        }
    }
    System.out.println(printResult(bp1, tempBP, result));
    //write into Delta-Report
    bos.write(printResult(bp1, tempBP, result));
    bos.newLine();
}
```

Abbildung 5.4: Ausschnitt der *analyseDeltas*-Methode [eigene Darstellung]

Die Funktionsweise der gesamten *analyseDeltas()*-Methode wir außerdem in Abbildung 5.5 in Form eines UML-Sequenzdiagramms dargestellt.

Effektiv löst diese Art des Iterierens das Minimierungsproblem der Abweichung zwischen den Boilerplates. Es wird als immer die interne Boilerplate

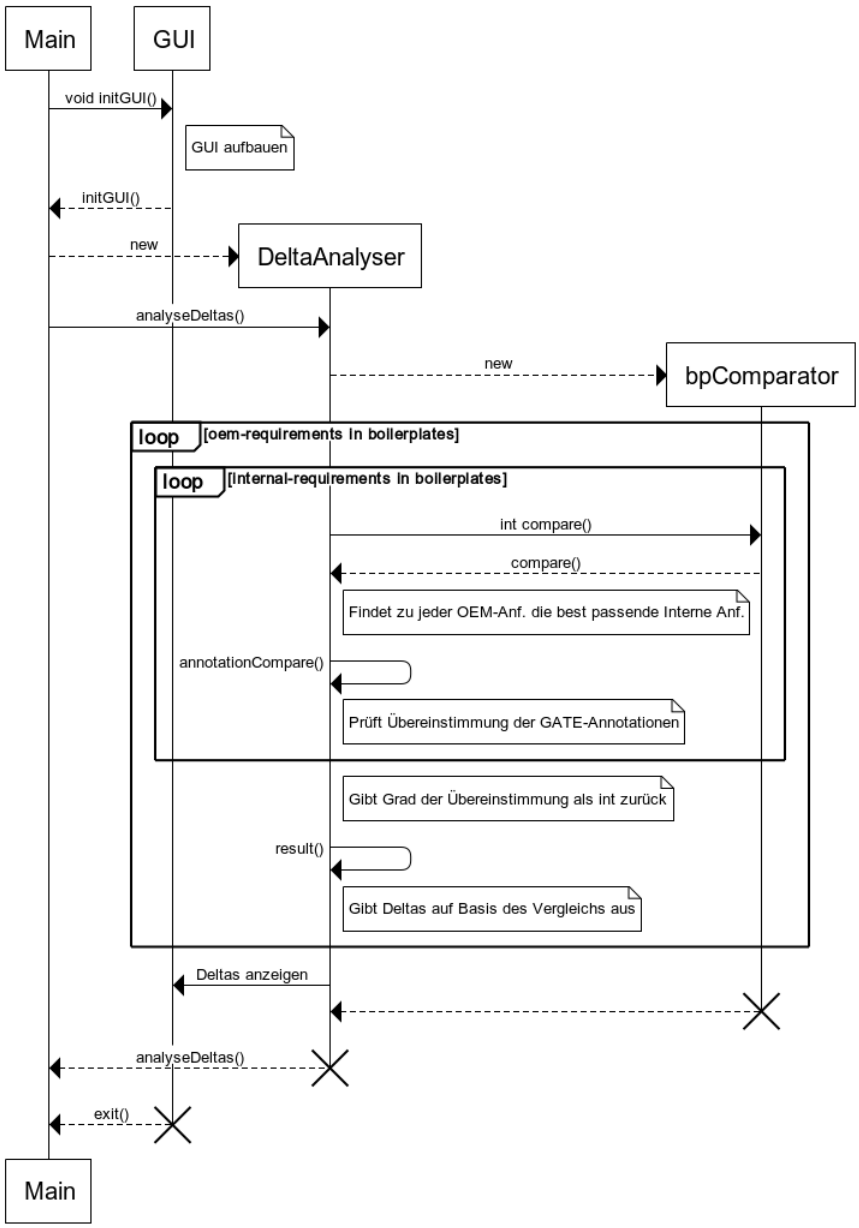

Abbildung 5.5: Sequenzdiagramm der *analyseDeltas()*-Methode [eigene Darstellung]

mit der geringsten Abweichung (gespeichert in der *result*-Variablen) bzw. der höchsten Übereinstimmung zu der momentanen OEM-Boilerplate gefunden.

Ausgabe des Delta-Reports

Sobald dies geschehen und spätestens wenn das HELLA-Dokument durchlaufen ist, wird mit den beiden Boilerplates und deren Übereinstimmungsfaktor die *printResult*-Methode aufgerufen.

Diese gibt je nach Boilerplate-Typ und Übereinstimmung der Boilerplates einen Satz aus, welcher das Ergebnis zusammenfasst und erläutert. Bei Unterschieden in bestimmten Satzbestandteilen werden beispielsweise beide betroffenen Komponenten zum Vergleich ausgegeben.

Dieser Satz wird für jede OEM-Anforderung über einen *FileOutputStream* in eine Datei geschrieben, welche später als Delta-Report verwendet wird. Sobald die letzte OEM-Anforderung verarbeitet ist, werden zusätzlich die Anzahl der kompletten Übereinstimmungen und die Anzahl der nicht verbundenen Boilerplates in die Datei ausgegeben. Ein fertiger Delta-Report wird am Ende des nächsten Kapitels gezeigt und erläutert.

5.2.2 GUI - Nutzerinteraktion und Workflow

Die Arbeit mit dem Delta-Analyser stellt das zweite Werkzeug der Verarbeitungskette dar, wenn Lastenhefte verglichen werden sollen. Ein Nutzer nutzt daher zunächst den R2BC, um Lastenhefte in formalere Sprache zu bringen, die sich dann im DA vergleichen lassen. In diesem Abschnitt wird daher anhand des Vergleichs zweier Lastenhefte im DA hauptsächlich auf den Delta-Report eingegangen, der als Ergebnis aus dem DA exportiert wird.

Programmoberfläche

Für den Test dieser Werkzeugkette und zur Verdeutlichung der Funktionalität vor den Experten, wurde daher auch für den DA eine minimale GUI entwickelt, anhand derer sich der Arbeitsablauf mit dem Programm zeigen lässt. Im Vergleich zum R2BC entfällt jedoch die manuelle Bewertung der Anforderungen durch den Nutzer, weshalb die GUI stark vereinfacht werden konnte.

Da die Analyse der Deltas prinzipiell automatisch anhand von Kriterien bzw. dem Überinstimmungsgrad erfolgt, beinhaltet die GUI im wesentlichen

Buttons zum laden der zwei Lastenhefte, die verglichen werden sollen, und zum Start der Analyse, in der auch direkt der Delta-Report erzeugt und als Datei exportiert wird.

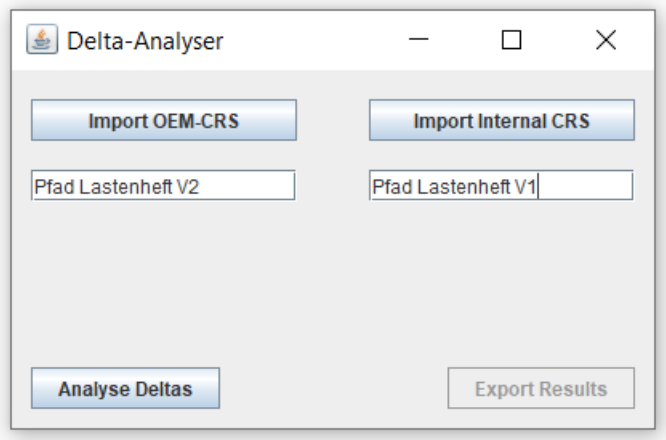

Abbildung 5.6: DA-GUI beim Programmstart [eigene Darstellung]

Die GUI orientiert sich dabei an dem in Abs. 3.4 gezeigten Use-Case, wonach der Delta-Analyser ein Lastenheft V2 auf die Deltas zu einem bereits vorhandenen Lastenheft V1 hin analysieren soll. V2 wird im DA als *OEM-Lastenheft* bezeichnet, da ein OEM neue Anforderungen als Kunde an HELLA gibt, die umgesetzt werden sollen. V1 wird als *internal* bzw. *HELLA*-Lastenheft bezeichnet, da es ein Lastenheft darstellt, dass bereits in der Firma bekannt ist.

Beim Programmstart wählt der Nutzer über *Import OEM CRS...* ein ein Lastenheft mit Kundenanforderungen aus, zu denen passende Anforderungen aus einem internen Lastenheft *Import internal CRS...* gefunden werden sollen.

Beispielhafte Arbeit mit dem Werkzeug

In diesem Abschnitt wird der bei der Analyse entstehende Delta-Report näher beschrieben. Die Implementierung der Vergleichsalgorithmen lässt sich anhand der zwei folgenden, prototypischen Lastenhefte nachvollziehen. Diese sind zuvor, wie bereits beschrieben, durch den R2BC konvertiert worden, womit der Vergleich im DA ermöglicht wurde. Als V1 könnte bereits ein Auszug

aus dem in Abs. 4.2.3 verarbeiteten Lastenheft in der Firma vorhanden sein:

1. The complete system «Golf7» shall «perform TestDriveCityTour».

2. The durability of EBus2X shall be at least 12000 h.

3. Under the condition: «If there is a connection to the internet» the actor: «Bluetooth connected phone» shall «transmit the data to a server».

Diese Anforderungen liegen somit firmenintern vor und neu hinzukommende Anforderungen können mit diesen abgeglichen werden. Die folgenden Anforderungen könnten als neue Anforderungen in V2 enthalten sein:

1. The complete system «Golf7» shall «perform TestDriveCityTour».

2. The complete system «vehicle» shall «perform Charge4FreeCharging».

3. Under the condition: «If there is a connection to the internet» the actor: «vehicle» shall «transmit the data to a server».

4. Under the condition: «If the car is turned off» the actor: «vehicle» shall «not transmit the data to a server».

Nach dem Klick auf *Analyse Deltas* ermittelt der R2BC zu jeder Anforderung aus V2 die am besten passende Anforderung aus V1. Der Grad der Übereinstimmung wird zusammen mit den Deltas zwischen den Anforderungen in den Delta-Report exportiert. Dieser ist für die beiden prototypischen Lastenhefte in Abb. 5.7 dargestellt:

Die Einträge zu jeder Anforderung aus V2 im Report sind wie folgt zu bewerten:

1. Zu ersten Anforderung aus V2 wurde die identische Anforderung 1 in V1 identifiziert, da diese nach der Konvertierung durch den R2BC die identische Boilerplate hervorgebracht haben. Der DA hat also die bestmögliche Übereinstimmung gefunden.

```
OEM-Anf 1 und Int-Anf 1 sind identisch:
 OEM: The complete system <<Golf7>> shall <<perform TestDriveCityTour>>.
--------------------------------------------------------------------------------
OEM-Anf 2 und Int-Anf 1
UNTERSCHIEDLICHE INFORMATIONEN zum GLEICHEN SYSTEMTYP:
 OEM: perform Charge4FreeCharging, vehicle
 Int: perform TestDriveCityTour, Golf7
--------------------------------------------------------------------------------
OEM-Anf 3 und Int-Anf 3:
UNTERSCHIEDLICHE SYSTEME zur SELBEN BEDINGUNG und FUNKTION:
 OEM: vehicle
 Int: Bluetooth connected phone
--------------------------------------------------------------------------------
OEM-Anf 4: Keine Übereiinstimmung gefunden, noch nicht in Int. spezifiziert:
 Under the condition: <<If the car is turned off>> the actor: <<vehicle>> shall
        <<not transmit the data to a server>>.
--------------------------------------------------------------------------------
Insgesamt 5 OEM-Anforderungen.

Davon sind identisch: 1.
Keine Übereinstimmung bei: 1.

Insgesamt 80% der OEM-Anf. sind bereits vergleichbar in Int. spezifiziert.
```

Abbildung 5.7: Delta-Report der Testlastenhefte [eigene Darstellung]

2. Bei Anforderung 2 liegt ein Delta in den vergleichbaren Anforderungen aus V1 vor. Der DA hat erneut die am besten passende Anforderung 1 identifiziert. Durch Nutzung des Enumerates wurde identifiziert, dass der Systemtyp von *vehicle* und *Golf7* übereinstimmen und somit zumindest vergleichbare Anforderungen vorliegen. Da sich die *function-Description* beider Anforderungen jedoch unterscheidet, ist der Übereinstimmungsgrad eher gering.

3. Bei Anforderung 3 liegt erneut der Fall vor, dass eine vergleichbare Anforderung in V1 gefunden wurde. Es fällt auf, dass sich lediglich der *ConditionalActor* zu den bekannten Anforderungen unterscheidet, weshalb der DA einen hohen Überinstimmungsgrad beider Anforderungen erreicht. Es muss lediglich geprüft werden, ob tatsächlich eine einfache Adaption von *Bluetooth connected phone* hin zu *vehicle* möglich ist.

4. Anforderung 4 scheint zunächst sehr ähnlich zu Anf. 3, unterscheitet sich jedoch in sämtlichen Satzbestandteilen von den Anforderungen aus

V1. Am ehesten vergleichbar wäre wieder Anf. 3, jedoch sind sowohl *ConditionalActor*, als auch *Condition* und *FunctionDescription* unterschiedlich. Der DA kann also, bis auf den gleichen Typ der Boilerplate, keine Übereinstimmung zu bekannten Anforderungen feststellen.

Es wurde gezeigt, dass eine recht feingranulare Differenzierung der Deltas im DA möglich ist, die dann an den Nutzer ausgegeben werden. Prinzipiell können auf Basis der Erkennung von Deltas auch weitere Informationen zur Umsetzbarkeit der abweichenden Anforderungen gegeben werden, wenn diese im Programm hinterlegt werden (etwa wenn Materialien direkt mit Kosten verknüpft sind). Dies kann die Bewertung der Anforderungen stark vereinfachen und den Aufwand der Machbarkeitsabschätzung reduzieren. Im nächsten Abschnitt werden mögliche Ergänzungen vorgestellt, die auf dem Konzept der DA-Implementierung basieren.

5.3 Mögliche Erweiterungen

In diesem Abschnitt werden mögliche Erweiterungen des Delta-Analysers erläutert. Diese sind aufgeteilt in Erweiterungen der Benutzeroberfläche (für eine bessere Interaktion mit dem Programm) und technische Erweiterungen, welche sich auf Konzept oder Funktionsweise des Programms beziehen.

5.3.1 Erweiterungen der Benutzeroberfläche

DOORS-Anbindung

Wie auch beim R2BC gilt für den Delta Analyser, dass eine Anbindung an DOORS von großem Nutzen sein kann. So könnte beispielsweise nach dem Delta Analyser ein direkter Vergleich der beiden Dokumente in DOORS erfolgen.

Dazu könnte eine farbige Unterlegung der Boilerplates auf Grundlage des besten gefundenen Übereinstimmungsfaktors genutzt werden. Die Übersichtlichkeit würde für den Nutzer dadurch stark verbessert und eine grobe Einschätzung der Dokumente ermöglicht.

Möglichkeiten Anzeigen

Momentan wird stets nur eine interne Boilerplate mit einer OEM-Boilerplate verbunden und später angezeigt. Es ist jedoch auch möglich, dass verschie-

dene Boilerplates den gleichen Übereinstimmungsfaktor haben und nur der Nutzer entscheiden kann welche am besten passt.

Dazu könnte man bei Uneindeutigkeit dem Nutzer (ähnlich zum R2BC) die besten Möglichkeiten anzeigen und ihn eine davon wählen lassen.

Delta Report

Der wichtigste Teil für die Verwendbarkeit des Programms ist ein übersichtlicher und praktischer Delta Report. Momentan wird dieser nur prototypisch erstellt und ist noch nicht konkret auf die Bedürfnisse bestimmter Nutzer angepasst.

Feldtests und Experteninterviews im Rahmen einer Anforderungsanalyse könnten die nötigen Informationen für einen optimierten Delta Report liefern.

Zusammenlegung

Da bei dem Delta Analyser keine komplexe Nutzerinteraktion nötig ist, sondern nur Dateien eingelesen und Ergebnisse ausgegeben werden müssen, ließe sich der Delta Analyser evtl. als Modul des R2BC in dessen Benutzeroberfläche integrieren.

Dies würde Modularität der Module etwas verringern, die Anwendbarkeit durch den Nutzer jedoch erleichtern.

5.3.2 Technische Erweiterungen

Analyse präzisieren

Im Delta Analyser werden bereits viele verschiedene Fälle berücksichtigt und unterschiedlich bewertet. Dies ist jedoch abhängig von der Komplexität der verglichenen Boilerplates noch erweiterbar, um Unterschiede noch präziser hervorzuheben.

Ontologien zur Entity-Einordnung

Die komplexeste Art der Deltas tritt bei Boilerplates auf, welche unterschiedlich voneinander sind, aber dennoch Gemeinsamkeiten aufweisen. Hierbei ist eine genau Einordnung der Gemeinsamkeiten wichtig, um die Deltas besser bewerten zu können.

Im Delta Analyser wird momentan eine Aufzählung (*Vehicle.java*) verwendet, um unterschiedliche Systeme einer gleichen Gruppe zuzuordnen. Hierfür könnte jedoch auch eine Ontologie verwendet werden, welche eine mächtigere Datenstruktur ist.

Die Baumstruktur von Ontologien würde es beispielsweise ermöglichen, anstelle der Abfrage nach der gleichen Klasse den „größten gemeinsamen Nenner“ der beiden Komponenten zu finden. Im Optimalfall wäre dies dann die gleiche Klasse, was jedoch andernfalls auch einfach abgestuft würde.

Kapitel 6

Fazit

6.1 Evaluation des Projekts

Im Zeitraum der Entwicklung wurde das zugrundeliegende Problem der aufwändigen, manuellen Verarbeitung von Lastenheften mit verschiedenen Beschäftigten von HELLA diskutiert.

Zu diesen Angestellten gehörten nicht nur RE-Experten und der externe Berater Colin Hood. Auch bei Mitarbeitern der Vorentwicklung bis hin zu den Vorgesetzten der Vorgesetzten gab es bereits eigene Erfahrung mit dem Problem. Sämtliche Befragten gaben dabei an, dass die manuelle Verarbeitung der Lastenheften mit manuellem Aufwand, hoher Fehleranfälligkeit und großen Übersichtsproblemen verbunden sei. Diese Problematik findet sich auch oftmals in der Literatur wieder[HE13][MW02].

Die Experten schätzten die vorgestellten Konzepte als praktisch wertvoll und bereits die im Rahmen eines Proof-of-Concept erstellten Prototypen als Arbeitserleichterung ein. Dies wurde beim Testen mit betrieblichen Lastenheften bestätigt und eine Konvertierungszeit von knapp 1,5 Minuten für das längste Lastenheft zeigte, dass auch die benötigte Rechenleistung keine kritische Herausforderung darstellt.

Zusätzlich fiel bei der Entwicklung auf, dass die Programmkonzepte für weitere Anwendungsfälle zum Einsatz kommen und bei verschiedenen Arten von Anforderungsdokumenten genutzt werden könnten. So ist etwa keine Unterscheidung zwischen funktionalen und nicht-funktionalen Anforderungen nötig.

Dadurch wurde die in Kap. 1.3 aufgestellte Hypothese größtenteils veri-

fiziert. Ontologien sind für die Programme entgegen der Erwartungen zwar nicht nötig, können jedoch als Erweiterung, wie etwa in Abschnitt 5.3 beschrieben, verwendet werden.

Des Weiteren wurde noch während der Entwicklung viel Potential für eine Weiterführung der Programme entdeckt. Dieses ließe sich beispielsweise mit den aufgeführten Erweiterungen noch ausschöpfen, um den Mehrwert der Werkzeuge weiter zu erhöhen.

6.2 Zusammenfassung

Die Entwicklung und Konzeption der zwei prototypischen Werkzeuge *Requirements-To-Boilerplate-Converter*(R2BC) und *Delta-Analyser*(DA) als Unterstützung von Require- ments-Engineering(RE)-Prozessen bei der Auswertung von Lastenheften stellten die Hauptthemen dieser Arbeit dar.

Im Laufe dieser Arbeit wurde eine Einführung in die Bereiche Natural Language Processing(NLP) und betriebliches RE in der Automobilindustrie gegeben. Dazu wurden die detaillierten Schritte der Analyse und Nutzung von Anforderungsdokumenten, wie auch die Kriterien und Ziele des RE vorgestellt. In Zusammenarbeit mit RE-Experten der Firma HELLA konnten Anforderungen und Funktionen für die softwaremäßige Umsetzung identifiziert und modelliert werden. Der Bezug zur Informatik wurde im Anschluss daran durch den konkreten Entwurf und die prototypische Implementierung der Werkzeuge hergestellt.

Aus den Grundlagen zu NLP und Ontologien wurde besonders die hohe Komplexität natürlicher Sprache und der damit begründete hohe Aufwand bei der Entwicklung entsprechender Werkzeuge verdeutlicht. Ein häufig anzutreffender Ansatz aktueller Forschungen im Bereich Anforderungsmanagement ist dabei, für ein verbessertes Verständnis zunächst natürlichsprachliche Dokumente in formalere Sprachen zu überführen.

Im Teil der betrieblichen Informationen wurde aus Sicht von RE-Experten auf die Herausforderungen beim Umgang mit Anforderungen im Entwicklungsprozess und Ziele für das RE hingewiesen. Machbarkeit und Qualität von Anforderungen werden häufig manuell geprüft, bieten jedoch großes Potential für Automatisierung und Use-Cases für die in dieser Arbeit skizzierten Werkzeuge. Diese nutzen die Konvertierung der Lastenhefte in semi-formale Sprachschablonen (Boilerplates), um Lastenhefte für die spätere Umsetzung

aufzubereiten.

Zuletzt erfolgte der softwaretechnische Entwurf und die Implementierung der Werkzeuge. Durch den betrieblichen Einsatz entstanden dabei der R2BC und DA auf java-Basis unter Nutzung der NLP-Software GATE.

Der R2BC stellt die erste Stufe der Verarbeitung dar, in welcher natürlichsprachliche Lastenhefte in Boilerplates überführt werden. Der DA realisiert auf dieser Basis die zweite Stufe der Verarbeitung, dessen Use-Case der Vergleich von zwei Lastenheften auf Basis der enthaltenen Anforderungen ist. Dazu nutzt er die im R2BC genereierten Textinformationen und kann Deltas präzise identifizieren. Die automatisierte Konvertierung und Vergleich von Lastenheften wurde anhand dieser Prototypen umgesetzt.

Die Evaluation und Tests der Werkzeuge fielen insgesamt positiv aus und deuten das weitere Potential solcher automatisierten Analysewerkzeuge an.

6.3 Ausblick

Die prototypischen Implementierungen der Werkzeuge haben gezeigt, dass durch den Einsatz von Software im Requirements-Engineering viel Verbesserungspotential ausgeschöpft werden kann.

Der Einsatzbereich für die vorgestellten Konzepte ließe sich also zukünftig auf weitere betriebliche Felder erweitern. Hinzu kommt, dass die Prototypen selbst noch viele Möglichkeiten zur Erweiterung bieten und somit weiterer Mehrwert für deren Nutzer geschaffen werden können.

Mit dem Abschluss dieser Arbeit werden die erstellten Programme daher nicht einfach archiviert, sondern mit der Zeit weiterentwickelt und neue Funktionen eingepflegt. Auf Anfrage der Requirements-Experten werden die Werkzeuge außerdem an HELLA-interne Entwickler übergeben, welche die Implementierung der Werkzeuge für den Betrieb weiterführen sollen.

Literaturverzeichnis

[AS05] Américo Sampaio, et al. „EA-Miner: a tool for automating aspect-oriented requirements identification.“ Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM, 2005.

[AWK06] Christian Allmann, Lydia Winkler, Thorsten Kölzow. „The requirements engineering gap in the OEM-supplier relationship.“, Journal of Universal Knowledge Management 1.2 (2006): 103-111.

[CA15] Chetan Arora, et al. „Automated checking of conformance to requirements templates using natural language processing.“ IEEE transactions on Software Engineering 41.10 (2015): 944-968.

[BAL10] Helmut Balzert. „Lehrbuch der softwaretechnik: Basiskonzepte und requirements engineering.“, Springer-Verlag, 2010.

[CAR52] Rudolf Carnap, „Meaning Postulates.“ , Philospohical Studies vol.3 S.65-73, 1952.

[CHO57] Noam Chomsky, „Syntactic Structures“ , Mouton & Co., Feb. 1957.

[COP04] Ann Copestake, University Of Cambridge, „Natural Language Processing“, 2004.

[DGE19] Deutsche Gesellschaft für EMV-Technologie e.V. „CRS Customer Requirements Specification“, aus https://www.demvt.de/publish/viewfull.cfm?ObjectID=ba9a0878_e081
_515d_74a3411df6771be8, abgerufen am 09.01.19.

[EA19] Dietmar Kinalzyk. „Von der Anforderung zur fertigen Safety-Software-Architektur“, aus https://www.elektroniknet.de/elektronik-automotive/bordnetz-vernetzung/von-der-anforderung-zur-fertigen-safety-software-architektur-132206.html, abgerufen 20.02.19

[FF11] Farfeleder, Stefan, et al. „DODT: Increasing requirements formalism using domain ontologies for improved embedded systems development. “ Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2011 IEEE 14th International Symposium on. IEEE, 2011.

[FH14] Fockel, Markus, and Jörg Holtmann. „A requirements engineering methodology combining models and controlled natural language.“ Model-Driven Requirements Engineering Workshop (MoDRE), 2014 IEEE 4th International. IEEE, 2014.

[GM19] https://gate.ac.uk/sale/tao/split.html GATE User Guide, aufgerufen am 12.02.2019

[HAO14] Hao Wu, et al. „ILLINOISCLOUDNLP: Text Analytics Services in the Cloud.“ LREC. 2014.

[HE13] Alexander Heßeler, et al. „Anforderungsmanagement: Formale Prozesse, Praxiserfahrungen, Einführungsstrategien und Toolauswahl.“ Springer-Verlag, 2013.

[HE18] HELLA Geschäftsbericht des Geschäftsjahres 2017/2018, von https://www.hella.com/hella-com/assets/media_global/2018.08.10_HELLA_Geschaeftsbericht_2018_geschuetzt.pdf, aufgerufen am 16.01.2019

[HE19] Firma HELLA. „Unternehmensinformationen in Kürze.“, aus https://www.hella.com/hella-com/de/HELLA-im-Ueberblick-723.html, abgerufen am 09.01.19.

[HL01] Hubert F. Hofmann, Franz Lehner. „Requirements engineering as a success factor in software projects.“IEEE software 4 (2001): 58-66.

[HP12] Andrea Harant, Miriam Poethen. „HOOD Capability Model für Requirements Definition und Requirements Management “ HOOD Ltd. 2012.

[IR09] BGH, „Urteil vom 6. Juli 2000, I ZR 244/97“Artikel beim Institut für Rechtsinformatik von der Universität des Saarlandes, 13. Oktober 2009

[KN85] Kapur, Deepak, and Paliath Narendran. „An equational approach to theorem proving in first-order predicate calculus.“ ACM SIGSOFT Software Engineering Notes 10.4 (1985): 63-66

[KR93] Ryan, Kevin. „The role of natural language in requirements engineering.“ Proceedings of the IEEE International Symposium on Requirements Engineering. IEEE, 1993.

[LV10] F. Langenscheidt, B. Venohr. „Lexikon der deutschen Weltmarktführer: Die Königsklasse deutscher Unternehmen in Wort und Bild.“, Deutsche Standards–Gabal Verlag Google Scholar (2010).

[MW02] Matthias Weber, Joachim Weisbrod. „Requirements engineering in automotive development-experiences and challenges.“, Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on. IEEE, 2002.

[PR15] Pohl, Klaus, Chris Rupp. „Basiswissen Requirements Engineering: Aus-und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering Foundation Level.“ dpunkt. verlag, 2015.

[RS18] Felix Ritter, Aaron Schul. „Analyse aktueller NLP-Methoden und -Werkzeuge am Beispiel von GATE.“ Projektarbeit Fachhochschule Dortmund 2018, 18.11.2018

[SB88] Muggleton, Stephen, Wray Buntine. „Machine invention of first-order predicates by inverting resolution.“ Machine Learning Proceedings 1988. 1988. 339-352

[SHE07] L. Shen, G. Satta, A. K. Joshi. In Meeting of the Association for Computational Linguistics (ACL), „Guided learning for bidirectional sequence classification“ , 2007.

[TOL09] Dhaval Thakker, Taha Osman, Phil Lakin. „Gate jape grammar tutorial.“ Nottingham Trent University, UK, Phil Lakin, UK, Version 1, 2009.

[WE13] Weller, Katrin. „Ontologien.“ 2013. S. 207-218.

[WEI66] Joseph Weizenbaum, „ELIZA—a computer program for the study of natural language communication between man and machine.“ Communications of the ACM 9.1, 36-45, 1966

[ZG02] Zowghi, Didar, and Vincenzo Gervasi. „The Three Cs of requirements: consistency, completeness, and correctness.“ International Workshop on Requirements Engineering: Foundations for Software Quality, Essen, Germany: Essener Informatik Beitiage. 2002.

[ZH17] Zichler, Konstantin, Steffen Helke. „Ontologiebasierte Abhängigkeitsanalyse im Projektlastenheft.“ Automotive-Safety & Security 2017-Sicherheit und Zuverlässigkeit für automobile Informationstechnik. 2017.

[ZH19] Zichler, Konstantin, Steffen Helke. „R2BC: Tool-Based Requirements Preparation for Delta Analyses by Conversion into Boilerplates “ Gesellschaft für Informatik. 2019.