



A M B I E N T

Hochschule der Medien
Fakultät Druck und Medien
Studiengang Computer Science & Media (M. Sc.)

Masterarbeit

Untersuchung eines Ansatzes für Testgenerierung anhand
natürlicher Sprache unter Verwendung von BDD

Autor Niklas Brocker
Email: niklas.brocke@web.de
Matrikelnummer: 39499

Erstbetreuer Prof. Walter Kriha
Hochschule der Medien

Zweitbetreuer Felix Schul
Ambient

Datum 16. November 2021

Ehrenwörtliche Erklärung

Hiermit versichere ich, Niklas Brocker, ehrenwörtlich, dass ich die vorliegende Masterarbeit mit dem Titel: "Untersuchung eines Ansatzes für Testgenerierung anhand natürlicher Sprache unter Verwendung von BDD" selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden. Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Stuttgart, den 16. November 2021



Niklas Brocker

Kurzfassung

Bei agiler Softwareentwicklung unter Anwendung des Behavior-Driven Developments (BDD) sind Auftraggebende am gesamten Prozess der Entwicklung beteiligt. Ihre Anforderungen werden in natürlicher Sprache festgehalten. Das garantiert gute, den Kundenwünschen entsprechende Produkte. BDD hat allerdings auch einen Schwachpunkt, der im Bereich des Testings liegt. Um zu überprüfen, ob ein System die in natürlicher Sprache festgehaltenen Anforderungen erfüllt, müssen Entwickelnde diese Anforderungen bei jedem einzelnen Schritt der Entwicklung manuell in Code übersetzen. Das ist zeitaufwendig. Außerdem entstehen viele Code-Schnipsel, die miteinander interagieren und die Wartbarkeit erschweren. Könnte man anhand der Feature-Dateien zugehörigen Test-Code automatisiert generieren, würde dies den Entwicklungsprozess effizienter und einfacher machen. Vor diesem Hintergrund stellt diese Arbeit einen Prototyp vor, der die in natürlicher Sprache geschriebenen Texte einer Feature-Datei zu Test-Suites für eine Django-API übersetzen kann. Im Gegensatz zu bisherigen Ansätzen soll die Software ohne eine existierende Implementierung vollständige Testfälle generieren können. Der Prototyp verwendet hierfür einen Compiler, der die Bedeutung und die Daten des Textes analysiert.

Der Prototyp wird mithilfe einer Simulation getestet, in der verschiedene Feature-Dateien umgewandelt werden. Es hat sich gezeigt, dass die entstandene Software in der Lage ist, ohne bestehende Implementierung Tests zu generieren. Der Prototyp ist dabei in aller Regel mindestens ebenso schnell wie in früheren Arbeiten beschriebene Testgenerierungsansätze. Außerdem bietet er interessante Möglichkeiten der Weiterentwicklung. Der Prototyp muss allerdings in einem Real-Life Projekt erprobt werden, bevor er abschließend bewertet werden kann.

Abstract

In agile software development using Behavior-Driven Development (BDD), clients are involved in the entire development process. Their requirements are recorded in natural language. This guarantees good products that meet customer needs. However, BDD also has a weak point which is in the area of testing. To verify that a system meets the requirements captured in natural language, developers must manually translate them into code at each step of development. This is time-consuming. It also creates many code snippets that interact with each other and make maintainability difficult. If the associated test code could be generated automatically based on the feature files, the development process would become easier and more efficient. With this in mind, this thesis presents a prototype that can transform the natural language text of a feature file into test suites for a Django API. In contrast to existing approaches, the software should be able to generate complete test cases without an existing implementation. To do this, the prototype uses a compiler that analyses the meaning and data of the text.

The prototype is tested using a simulation in which various feature files are converted. This work's analysis shows that the resulting software is capable of generating tests without an existing implementation. The speed is at a similar level to previous work. The prototype also offers interesting possibilities for further development. However, it must be used in a real-life project to allow a detailed evaluation.

Inhaltsverzeichnis

Abbildungsverzeichnis	VIII
Tabellenverzeichnis	X
Listings	XI
Produktionsregeln	XIV
Abkürzungsverzeichnis	XV
1 Einführung	1
1.1 Motivation	1
1.2 Related Work	3
1.3 Fragestellung	4
1.4 Ziel der Arbeit	4
1.5 Ambient	6
1.6 Struktur der Arbeit	6
2 Grundlagen	8
2.1 Agile Softwareentwicklung	8
2.1.1 Grundsätze	8
2.1.2 Scrum	10
2.2 Testing	12
2.2.1 Cases und Suites	12

2.2.2	Methoden	12
2.2.3	Ansätze	13
2.2.4	Ziel	13
2.2.5	Phasen	13
2.2.6	Level	14
2.2.7	Strategien in der Entwicklung	15
2.2.8	Test-Driven Development	16
2.2.9	Behavior-Driven Development	17
2.2.10	Gherkin	17
2.3	Generative Programmierung	19
2.3.1	Domänenpezifische Sprachen	20
2.3.2	Entwicklungsprozesse	21
2.3.3	Generative Domain Model	22
2.4	Grammatik	24
2.4.1	Grundlagen	24
2.4.2	Chomsky-Hierarchie	26
2.4.3	(Erweiterte) Backus-Naur-Form	28
2.5	Compiler	30
2.5.1	Lexer	31
2.5.2	Parser	31
2.5.3	Type-Checking	38
2.5.4	Code-Generator	38
2.6	Natural Language Processing	42
2.6.1	Tokenization	42
2.6.2	Stop Words	43
2.6.3	Lemmatization	43
2.6.4	POS Tagging	44
2.6.5	Dependencies	45
2.6.6	Similarity	46
2.7	Django	47
2.7.1	Apps	48
2.7.2	MVT	48
2.7.3	API	49

3 Anforderungsanalyse	50
3.1 Allgemein	50
3.2 Auslesen von Gherkin	51
3.3 Vorbereitung der Generierung	52
3.4 Generierter Code	53
4 Konzeptionierung	55
4.1 Test-Framework	56
4.2 Lexer	57
4.3 Parser	59
4.4 Code-Generierung	60
4.4.1 Generierungstechnik auswählen	61
4.4.2 Bestimmung der Bedeutung	62
4.4.3 Daten extrahieren	63
4.4.4 Erstellung des Output-ASTs	70
4.4.5 Umwandlung zu Code	72
5 Implementierung	76
5.1 Mixin	76
5.2 Gherkin-Parser	77
5.3 NLP	79
5.4 Tiler	81
5.5 Transformationsklassen	82
5.5.1 Umwandlung zu Code	82
5.5.2 Anpassungsfähigkeit	83
5.5.3 Events	84
5.6 Lookout	85
5.6.1 Möglicher Output	85
5.6.2 Keywords	86
5.6.3 Variationen	86
5.6.4 Similarity	88
5.6.5 Unsicherheit	89
5.7 ExtractorOutput	90
5.7.1 Token zu nativem Wert	91

5.7.2	Nativer Wert zu Output	92
5.8	Extractor	93
5.8.1	Verwendung von Output	93
5.8.2	ManyExtractorMixin	93
5.9	Converter	94
5.9.1	Kompatibilität	94
5.9.2	Datenbestimmung	96
5.9.3	Statements erstellen	96
6	Anwendungsanalyse	99
6.1	Aufbau	99
6.2	Ablauf	101
6.3	Durchführung	102
6.3.1	Schritt S-1	103
6.3.2	Schritt S-2	104
6.3.3	Schritt S-3	105
6.3.4	Schritt S-4	105
6.3.5	Schritt S-5	107
6.3.6	Schritt S-6	108
7	Diskussion	109
8	Fazit und Ausblick	112
Quellenverzeichnis		114
A	Anhang	124

Abbildungsverzeichnis

1.1	Ablauf der Generierung	5
2.1	Abschnitt 2.1 im Ablauf der Generierung	9
2.2	Abschnitt 2.2 im Ablauf der Generierung	12
2.3	Test-Last vs. Test-First	15
2.4	Abschnitt 2.3 im Ablauf der Generierung	20
2.5	Domain Engineering und Application Engineering Kreislauf	22
2.6	Generative Domain Model	23
2.7	Generative Domain Model aus Sicht der Transformation	23
2.8	Abschnitt 2.4 im Ablauf der Generierung	24
2.9	Abschnitt 2.5 im Ablauf der Generierung	30
2.10	Baum für Top-Down Parsing	32
2.11	Bäume für mehrdeutige Grammatik	34
2.12	Baum für Bottom-Up Parsing	37
2.13	Abschnitt 2.6 im Ablauf der Generierung	42
2.14	Dependencies eines Satzes	46
2.15	Farben in 3-dimensionalen Raum	47
2.16	Abschnitt 2.7 im Ablauf der Generierung	48
4.1	Schritte des Ghengo-Compilers	55
4.2	Schritte des Ghengo-Compilers nach Wahl des Test-Frameworks . .	57
4.3	Tokens mit Lexemes für Listing 4.3	59

4.4	Ablauf der Code-Generierung mit Ghengos Generator	61
4.5	Ablauf der Code-Generierung mit Ghengo nach Wahl der Generierungstechnik	62
4.6	Daten in einem Beispielsatz	64
4.7	Klassendiagramm für Wrapper	66
4.8	Abhängigkeiten bei Listen von Zahlen	69
4.9	Beispielhafter Ablauf der Template-Befüllung in Ghengo	73
4.10	Klassen für die Erstellung eines Models	74
5.1	Beispiel für Anwendung von Mixin	77
6.1	Aufbau der Datenbank am Start der Simulation	100
6.2	Aufbau der Datenbank nach Schritt S-4	102
A.1	Ausschnitt aus Klassendiagramm des Output-ASTs von Ghengo .	161
A.2	Ausschnitt aus Klassendiagramm für Zusammenspiel Code-Generator, Tiler und Converter	162
A.3	Vereinfachtes Klassendiagramm des ASTs von Gherkin	163
A.4	Klassendiagramm des Model- und RequestConverters	164

Tabellenverzeichnis

2.1	Beispiele für universelle POS Tags	44
2.2	Beispiele für POS Tags für Deutsch	44
2.3	Tagging an einem deutschen Satz	45
4.1	Tokens für Keywords aus Gherkin	58
4.2	Daten aus Parser-AST und deren Verwendungszweck für Generator	60
4.3	Zuordnung unterstützter Aktionen zu Step-Arten	63
6.1	API-Endpunkte am Start der Simulation	100
6.2	Dauer der verschiedenen Step-Arten in Simulationsschritt S-2 . . .	104
6.3	Dauer der verschiedenen Step-Arten in Simulationsschritt S-3 . . .	105
6.4	Dauer der verschiedenen Step-Arten in Simulationsschritt S-4 . . .	106
6.5	Dauer der verschiedenen Step-Arten in Simulationsschritt S-5 . . .	107
6.6	Dauer der verschiedenen Step-Arten in Simulationsschritt S-6 . . .	108

Listings

1.1	Eine einfache Feature-Datei	2
1.2	Implementierung eines Steps in Behave	2
2.1	Feature-Datei mit Hintergrund und Szenarien	18
2.2	Implementierung eines given-Steps	19
2.3	Beispiel-Input für einen Lexer	31
2.4	Beispiel eines Templates für eine If-Bedingung in Python	39
2.5	Beispiel eines Templates für eine If-Bedingung in beliebiger Sprache	39
2.6	Beispiel einer Code-Generierung per Transformation in Python . .	40
2.7	Beispielhafte Implementierung der Generierung einer Klasse in Py- thon	41
2.8	Generierter Code aus Listing 2.6	41
4.1	Pytest Testfall mit Decorator	56
4.2	Tag in Feature-Datei	56
4.3	Beispiel einer deutschen Feature-Datei	58
4.4	Generierter Code aus dem Text „Gegeben sei ein Auftrag 1, der abgeschlossen ist“	70
4.5	Generierter Code aus dem Text „Wenn Auftrag 1 gelöscht wird“ .	70
4.6	Nicht verwendete Variablen und lange Zeilen	72
4.7	Verbesserung von Listing 4.6	73
4.8	Code mit Parameter und Import	75

5.1	Erben von Mixins in Python	77
5.2	Validierung der Token-Sequenz der TerminalSymbol-Klasse	78
5.3	Verwendung der RuleOperator-Klassen	78
5.4	Anwenden von NLP mit Spacy	79
5.5	Anwenden von NLP mit Spacy und eigenem Cache	80
5.6	Tiler sucht den besten Converter	81
5.7	GivenTiler	82
5.8	Die Transformationsklasse für ein Attribut	83
5.9	Die Transformationsklasse PyTestModelFactoryExpression	84
5.10	Schritt 1 der Suche der Lookout-Klasse	85
5.11	Schritt 2 der Suche der Lookout-Klasse	86
5.12	Schritt 3 in der Suche der Lookout-Klasse	87
5.13	Schritt 4 in der Suche der Lookout-Klasse	88
5.14	Schritt 5 in der Suche der Lookout-Klasse	89
5.15	Beispiel für Fallback der Lookout-Klasse	89
5.16	Token zu nativem Wert in BooleanOutput-Klasse	92
5.17	Zum Generieren von Output erforderliche Schritte der BooleanOutput-Klasse	93
5.18	Phasen der Erstellung von Statements in Converter-Klasse	97
5.19	Beispiel für generierten Code vom AssertPreviousModelConverter	98
6.1	Start-Befehl für Ghengo	104
A.1	OnAddToTestCaseListenerMixin für das Hören auf Events	124
A.2	Komplette Suche der Lookout-Klasse	125
A.3	TemplateMixin von Ghengo	126
A.4	Replaceable-Klasse von Ghengo	127
A.5	Schritte der ExtractorOutput-Klasse, um nativen Wert zu erhalten	128
A.6	Token zu nativem Wert in ExtractorOutput-Klasse	129
A.7	Schritte der ExtractorOutput-Klasse, um den Output zu erhalten	130
A.8	Ausschnitt aus Extractor-Klasse	131
A.9	Wichtigste Methode der ManyExtractorMixin-Klasse	132
A.10	Dokument-Kompatibilität der AssertPreviousModelConverter-Klasse	133
A.11	Dokument-Kompatibilität der ObjectQuerysetConverter-Klasse . .	134

A.12	Finden von referenzierten Objekten und den zugehörigen Tokens im Converter	135
A.13	Algorithmus zur Bestimmung von Tokens, die Namen von Referenzen repräsentieren	136
A.14	Beispiel für die Verwendung von Transformationsklassen im Assert-PreviousModelConverter	137
A.15	Validierung der Token-Sequenz der Chain-Klasse	138
A.16	Validierung der Token-Sequenz der Optional-Klasse	139
A.17	Validierung der Token-Sequenz der OneOf-Klasse	139
A.18	Validierung der Token-Sequenz der Repeatable-Klasse	140
A.19	Validierung der Token-Sequenz der NonTerminalSymbol-Klasse . .	141
A.20	Teil 1 der Feature-Datei für Schritt S-2 der Simulation	142
A.21	Teil 2 der Feature-Datei für Schritt S-2 der Simulation	143
A.22	Teil 1 des generierten Codes aus Schritt S-2 der Simulation	144
A.23	Teil 2 des generierten Codes aus Schritt S-2 der Simulation	145
A.24	Teil 3 des generierten Codes aus Schritt S-2 der Simulation	146
A.25	Feature-Datei für Schritt S-3 der Simulation	147
A.26	Generierter Code der Simulation vor Implementierung (Schritt S-3)	148
A.27	Generierter Code der Simulation nach Implementierung (Schritt S-3)	149
A.28	Feature-Datei für Schritt S-4 der Simulation	150
A.29	Teil 1 des generierten Codes der Simulation vor Implementierung (Schritt S-4)	151
A.30	Teil 2 des generierten Codes der Simulation vor Implementierung (Schritt S-4)	152
A.31	Teil 1 des generierten Codes der Simulation nach Implementierung (Schritt S-4)	153
A.32	Teil 2 des generierten Codes der Simulation nach Implementierung (Schritt S-4)	154
A.33	Feature-Datei für Schritt S-5 der Simulation	155
A.34	Teil 1 des generierten Codes der Simulation (Schritt S-5)	156
A.35	Teil 2 des generierten Codes der Simulation (Schritt S-5)	157
A.36	Feature-Datei für Schritt S-6 der Simulation	158
A.37	Generierter Code der Simulation (Schritt S-6)	159

Produktionsregeln

2.1	Einfaches Beispiel	25
2.2	Ungültig für kontextsenstive Grammatik	26
2.3	Kontextsensitive Grammatik	27
2.4	Reguläre Grammatik	27
2.5	Definition der Variablen digit und letter (EBNF)	28
2.6	Verwenden von Variablen und Folgen (EBNF)	28
2.7	Optional (EBNF)	29
2.8	Zusammenfassung aller Regeln (EBNF)	29
2.9	Funktionsweise der Parser	32
2.10	Linksrekursive Grammatik	33
2.11	Mehrdeutige Grammatik	33
2.12	Grammatik für rekursive Fehlerbehandlung	36
A.1	Regeln für Gherkin	160

Abkürzungsverzeichnis

API Application Programming Interface

AST Abstract Syntax Tree

ATDD Acceptance-Test-Driven Development

BDD Behavior-Driven Development

BNF Backus-Naur-Form

CD Continous Delivery

CI Continous Integration

CRUD Create, Read, Update, Delete

CST Concrete Syntax Tree

DSL Domain-Specific Language

E2E End-to-End

EBNF Erweiterte Backus-Naur-Form

EOF End of File

EOL End of Line

GDM Generative Domain Model

GP Generative Programming

GPL General-Purpose Language

GUI Graphical User Interface

IDE Integrated development environment

IPA International Phonetic Alphabet

JSP Java Server Pages

KI Künstliche Intelligenz

NLP Natural Language Processing

PO Product Owner

POS Part-of-speech

RGB Red-Green-Blue

SQL Structured Query Language

TDD Test-Driven Development

UML Unified Modeling Language

URL Uniform Resource Locator

1. Einführung

1.1 Motivation

Agile Methoden sind im Bereich der Software-Entwicklung sehr verbreitet. Dazu zählt auch das Behavior-Driven Development (BDD), eine Technik, die Entwickelnde in ungefähr 20 Prozent der Engineering-Prozesse verwenden (Stand: 2020) [1][54]. Dort werden Anforderungen in natürlicher Sprache festgehalten und Auftraggebende in jeden Schritt der Entwicklung einbezogen - ein Vorgehen, das für gute Produkte und zufriedene Kundschaft sorgt [7][36].

Behaviour-Driven Development hat allerdings auch einen Schwachpunkt, der im Bereich des Testings liegt. Um zu überprüfen, ob ein System die in natürlicher Sprache festgehaltenen Anforderungen erfüllt, sind Entwickelnde gezwungen, diese Anforderungen händisch in Code zu übersetzen. Dabei liegen die Anforderungen als *Feature*-Dateien vor, die häufig in der Sprache *Gherkin* verfasst sind (Beispiel in Listing 1.1) [7][36].

Hinzu kommt, dass Entwickelnde diese Übersetzungsarbeit im Rahmen des Behavior-Driven Developments sehr häufig wiederholen müssen. Der Grund: Die Feature-Dateien beschreiben verschiedene Szenarien in einzelnen Steps (Given, When, Then). Und Entwickelnde müssen für jeden dieser Steps den zugehörigen Test-Code schreiben. Bei dem hier üblicherweise gewählten Test-First Ansatz entstehen diese Test bereits vor der Implementierung des Produkt-Codes, so dass Entwickelnde sich bei jedem Schritt fragen müssen: Wie sähe der Code aus, mit dem sich das erreichen oder

```
1 Feature:  
2 Scenario:  
3   Given a user Alice  
4   When Alice logs in  
5   Then she should see the list of orders
```

Listing 1.1: Eine einfache Feature-Datei

prüfen ließe, was der jeweilige Step beschreibt (s. Listing 1.2)? Anders als bei traditionellen Testfällen entstehen so viele kleine Code-Schnipsel für die Steps. Jeder dieser Code-Schnipsel kann mit jedem anderen beliebig interagieren, was zu erschwerter Wartbarkeit führt. Der entscheidende Nachteil aber ist, dass Entwickelnde viel Zeit auf die Übersetzung von Feature-Dateien zu Tests verwenden müssen. Das verursacht Kosten und verzögert die Fertigstellung eines Produktes [7][72].

```
1 from behave import *  
2 from core.utils import get_user  
3 from core.auth import AuthClient  
4  
5 @when('Alice logs in')  
6 def step_impl(context):  
7     alice = get_user()  
8     AuthClient.login(alice)
```

Listing 1.2: Implementierung eines Steps in Behave [72]

Könnte man anhand der Feature-Dateien automatisiert den zugehörigen Test-Code generieren, würde dies den Entwicklungsprozess effizienter und einfacher machen. Um eine bessere Wartbarkeit zu erreichen, wäre es wünschenswert, vollständige Testfälle und keine Code-Schnipsel zu erstellen.

1.2 Related Work

Es gibt bereits einige Arbeiten, die sich mit dem Generieren von Tests auseinander gesetzt haben.

Pynguin, Evosuite und Randoop generieren Code anhand bestehender Implementierung. Sie arbeiten also mit einem Test-Last Ansatz und erstellen Tests, mit denen man kontrollieren kann, ob ein System nach Änderungen immer noch wunschgemäß funktioniert [32][57][63]. Es ist also nicht möglich, die drei Ansätze zur Generierung von Tests im Behavior-Driven Development einzusetzen. Zum einen kann hier nicht davon ausgegangen werden, dass die Implementierung schon existiert. Zum anderen sollen Tests im Behavior-Driven Development die Anforderungen an das System überprüfen.

Es gibt auch einige Arbeiten, die sich mit Codegenerierung im Behavior-Driven Development, also im Umfeld agiler Entwicklung, auseinander gesetzt haben.

- **behave _ nicely** ist eine Software, die mithilfe von Feature-Dateien die Code-Schnipsel der einzelnen Steps generieren kann (s. Listing 1.2). Somit müssen sich Entwickelnde nicht mehr um ihre Wartung kümmern [84].
- Soeken, Wille und Drechsler haben sich mit der Frage beschäftigt, wie aus natürlicher Sprache Informationen entnommen werden können. Dabei ist eine Software entstanden, die semi-automatisch Tests generiert. Eine nutzende Person gibt dabei die Steps ein und erhält Code-Schnipsel, die diese repräsentieren. Entwickelnde können diese Schnipsel dann korrigieren und weiterverwenden. Hier ist also das Zusammenspiel von Mensch und Maschine notwendig [77].
- **Kirby** nutzt Feature-Dateien, um vollständige Testfälle zu generieren. Die Generierung läuft hierbei zyklisch und in mehreren Schritten ab. Im ersten Schritt entsteht das Outline der Tests. Im nächsten Schritt müssen Entwickelnde Teile der Implementierung erstellen. Kirby kann im Anschluss weitere Teile der Tests befüllen. Dies wiederholt sich so lange, bis die Implementierung fertig gestellt ist und Kirby die Tests vollständig generiert hat. Dieser Ablauf orientiert sich am Red-Green-Refactor Zyklus, der im Test-Driven Development verbreitet ist [48].

Diese Thesis verwendet und kombiniert einige Ideen dieser Arbeiten.

1.3 Fragestellung

Keiner der in Abschnitt 1.2 vorgestellten Arbeiten ist in der Lage, ohne vorherige Implementierung traditionelle **Testfälle** zu generieren. Dies könnte damit zusammenhängen, dass die vorgestellten Software-Ideen auf jede Art von Programm angewendet werden können und nicht spezialisiert sind.

Daraus leiten sich folgende globale Forschungsfragen ab:

1. **Kann man mithilfe von natürlicher Sprache und ohne bestehende Implementierung Anforderungen zu Tests übersetzen, wenn die assistierende Software auf diesen Bereich spezialisiert ist?**
2. **Hat diese Strategie einen positiven Effekt auf die Entwicklung in einem agilen Umfeld?**

Diese Arbeit fokussiert den ersten Punkt und beschäftigt sich speziell mit der Generierung von Tests für eine Django-API. Die spezifizierte Forschungsfrage lautet also:

Kann man mithilfe natürlicher Sprache und ohne bestehende Implementierungen Tests für Django-APIS generieren, so dass Entwickelnde Anforderungen nicht mehr händisch zu Tests übersetzen müssen?

1.4 Ziel der Arbeit

Diese Arbeit wird einen Prototyp vorstellen, der Feature-Dateien in Tests für Python umwandelt und es ermöglicht, die definierten Anforderungen zu testen. Die entstehenden Tests sollen für eine verbreitete Test-Library in Python ausführbar sein.

Der Prototyp soll in einem agilen Test-First Ansatz verwendbar sein und somit ohne vorherige Implementierung arbeiten können. Um dies zu ermöglichen, spezialisiert er sich auf einen Anwendungsfall: auf die Generierung von Tests für eine Django-API. Bei der Implementierung dieses Prototyps fließen Ideen aus den in Abschnitt 1.2 genannten Arbeiten ein.

Abbildung 1.1 zeigt den groben Ablauf. Eine auftraggebende Person hat Anforderungen an eine Django-API. Diese werden in einer *Feature*-Datei festgehalten. Der Prototyp wertet diese Datei aus und generiert Python-Tests. Diese Tests überprüfen, ob die API die Anforderungen erfüllt.

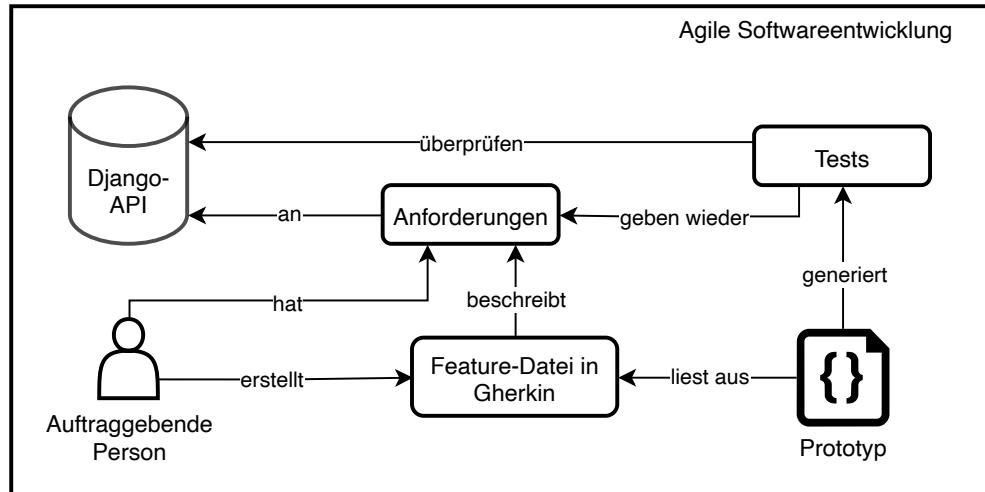


Abbildung 1.1: Ablauf der Generierung (eigene Darstellung)

Normalerweise würden Auftraggebende in der agilen Softwareentwicklung die Anforderungen aus der Sicht einer nutzenden Person beschreiben [70]. In einer solchen Applikation gäbe es auch noch ein Frontend. Dies ist nicht Teil dieser Arbeit, bietet aber Raum für künftige Weiterentwicklung.

Um die formulierten Ziele zu erreichen, ist zu klären, wie der Prototyp den Input verarbeiten kann. Es muss eine Übersetzung des Textes in eine interne Struktur möglich sein, die den Text beschreibt. Der Prototyp soll den Text verwenden, um unter anderem folgende Fragen zu beantworten: Welche Abschnitte der Texte enthalten Daten, die entnommen werden müssen? Wie kann man diese Daten extrahieren? Was bedeutet ein Satz? Wie kann man Sätze so kategorisieren, dass die Übersetzung der Bedeutung eines Satzes zu Code möglich ist? Nachdem diese Fragen geklärt sind, geht es um einen passenden Ansatz für die Transformation von Text zu Tests. Dabei ist es wichtig, dass die Software erweiterbar für andere Programmiersprachen und Test-Libraries ist und dabei performant bleibt.

Die Anforderungen an den Prototyp sollen initial festgehalten werden. Sie dienen dazu, ihn abschließend zu bewerten und über Entscheidungen in der Implementierung zu reflektieren. Es findet allerdings keine Bewertung der Software anhand eines echten Projekts statt. Diese Arbeit wird lediglich die Grundlage für eine solche Bewertung schaffen.

1.5 Ambient

Diese Masterarbeit wurde in Kooperation mit Ambient erstellt. Die Firma ist eine Agentur für IT, Web- und Appentwicklung. Sie wurde 2010 als Spin-Off des Lehrstuhls Wirtschaftsinformatik und Informationsmanagement der Universität zu Köln gegründet [93]. Ambient hat aktuell einen Sitz in Köln und beschäftigt zur Zeit ungefähr 75 Mitarbeiter [4].

Die Firma charakterisiert sich mit vier Werten: „Gemeinsam auf Augenhöhe“, „Herzlich professionell“, „Mutig zukunftsorientiert“ und „Verantwortungsvoll freiheitslebend“ [97]. Ambient führt sämtliche Projekte mithilfe von Scrum durch. Das Unternehmen verspricht sich davon Flexibilität und optimale Testbarkeit der Projekte [74]. Ambient verwendet hierbei verschiedene Test-Arten. Dazu zählen E2E-Tests, Unit-Tests und Akzeptanztests [20][23]. Ambients Entwickelnde haben also in ihrer täglichen Arbeit mit Tests im agilen Umfeld zu tun. Ambient nutzt in vielen Projekten Django - hauptsächlich als Backend mit einer API [22].

Mit dieser Arbeit erhält Ambient einen Einblick in die Generierung von Tests anhand von Akzeptanzkriterien. Aufgrund der häufigen Verwendung von Django liegt hier die Spezialisierung des Prototyps.

1.6 Struktur der Arbeit

- **Kapitel 1**

Kapitel 1 erläutert die Zielsetzung dieser Arbeit.

- **Kapitel 2**

Kapitel 2 vermittelt theoretische Grundlagen für die Bearbeitung der Forschungsfrage.

- **Kapitel 3**

Kapitel 3 beschreibt die Anforderungen, die sich auf die in 1.4 beschriebenen Ziele beziehen.

- **Kapitel 4**

Kapitel 4 beinhaltet die Konzeption der Teilbereiche, die in der Arbeit behandelt wurden.

- **Kapitel 5**

Kapitel 5 beschreibt die Implementierung der einzelnen Bereiche, wobei auch kurz auf Probleme eingegangen wird.

- **Kapitel 6**

Kapitel 6 analysiert den Prototyp mithilfe einer Simulation.

- **Kapitel 7**

Kapitel 7 diskutiert die Ergebnisse der Simulation und beschreibt Stärken und Schwächen des Prototyps.

- **Kapitel 8**

Kapitel 8 fasst die in Kapitel 7 genannten Punkte zu einem Fazit zusammen und gibt einen Ausblick.

2. Grundlagen

Die folgende Abschnitte betrachten den Ablauf der Generierung in Abbildung 1.1 aus verschiedenen Blickwinkeln und erläutert zentrale Begriffe, die in dieser Arbeit eine Rolle spielen. Zu diesen Begriffen zählen agile Entwicklung, Testing, generative Programmierung, Grammatik, Compiler, Natural Language Processing und Django. Jeder Abschnitt zeigt dabei auf, für welche Schritte der Generierung die angesprochenen Sachverhalte wichtig sind und welche Details näher ausgeführt werden sollen.

2.1 Agile Softwareentwicklung

Der aus dieser Arbeit resultierende Prototyp soll in der agilen Softwareentwicklung eingesetzt werden. Dieser Abschnitt befasst sich mit agilen Methoden und den Personen, die dahinter stehen (s. Abbildung 2.1). Dies geschieht am Beispiel von Scrum. Zuvor werden einige Grundsätze des agilen Arbeitens erläutert.

2.1.1 Grundsätze

Alle agilen Methoden basieren auf den gleichen Werten und verfolgen Prinzipien die im sogenannten *Agile Manifesto* festgehalten sind. Die vier Werte agiler Entwicklung sind [8][24]:

- **Individuals and interactions over processes and tools**

Agile Methoden stellen den Menschen über den Prozess. Der Grund: Wenn strikte Prozesse die Entwicklung leiten, ist es schwer, auf spontane Wünsche

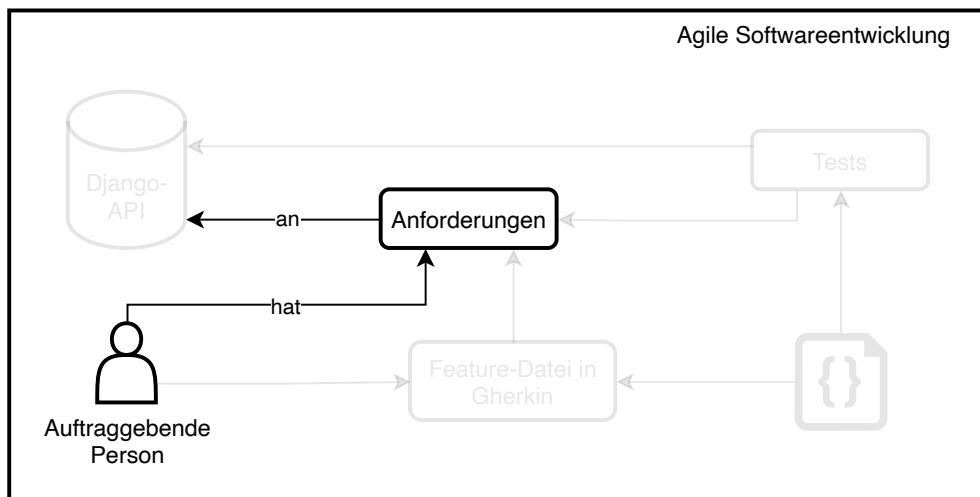


Abbildung 2.1: Abschnitt 2.1 im Ablauf der Generierung (eigene Darstellung)

des Auftraggebenden oder der nutzenden Person einzugehen. Ein wichtiger Teil des agilen Arbeitens ist die zwischenmenschliche Kommunikation.

- **Working software over comprehensive documentation**

In der agilen Entwicklung wird vergleichsweise wenig Zeit auf das Erstellen von Dokumentation für das finale Produkt verwendet. So bleibt mehr Zeit für die eigentliche Entwicklungsarbeit.

- **Customer collaboration over contract negotiation**

Bei traditionellen Methoden halten Projektleitende und Auftraggebende vor der Entwicklung alle Anforderungen detailliert fest. Ein Beispiel dafür ist das Wasserfallmodell, das Abschnitt 2.2.7 genauer erläutert. Beim agilen Arbeiten dagegen entstehen und konkretisieren sich die Anforderungen im laufenden Projekt. Auftraggebende sind während des gesamten Entwicklungsprozesses eingebunden und bringen an dieser Stelle ihre Wünsche ein.

- **Responding to change over following a plan**

Auch dieser Punkt spricht gegen die detaillierte Definition der Anforderungen zu Beginn des Projekts: Es ist wichtiger, spontan auf Änderungen zu reagieren, als sich an einen definierten Plan zu halten. Durch die direkte Einbindung von Auftraggebenden können diese mehr oder weniger spontan entscheiden und reevaluieren. Die wichtigsten Anforderungen lassen sich jederzeit nachschärfen.

Ergänzt werden diese vier Werte durch zwölf Prinzipien [9]. Sie beschreiben eine Arbeitskultur, in der Änderungen möglich sind und in der Auftraggebende im Zentrum stehen. Im Folgenden sind beispielhaft einige dieser Ziele genannt [24].

- **Customer satisfaction through early and continuous software delivery**

Auftraggebende sollen früh und schnell Ergebnisse sehen und Prototypen ausprobieren können.

- **Collaboration between the business stakeholders and developers throughout the project**

Während der gesamten Entwicklungsphase stehen Anspruchsgruppe und Entwickelnde in engem Kontakt zueinander.

- **Enable face-to-face interactions**

Sämtliche Mitglieder eines Teams sollten sich regelmäßig treffen, damit sie sich kennen und auf Augenhöhe begegnen können. Dies ist besonders wichtig, wenn einzelne Mitglieder nicht vor Ort arbeiten.

Die Anforderungen sind für diese Arbeit besonders wichtig. Sie sollen später in Tests umgewandelt werden.

2.1.2 Scrum

Scrum zählt zu den agilen Methoden, die am weitesten verbreitet sind [54, S. 6]. Auch Ambient verwendet Scrum. Folglich geht diese Arbeit davon aus, dass die Anforderungen an ein System (s. Abbildung 2.1) im Umfeld von Scrum definiert werden.

Dieser Abschnitt beschreibt zunächst wie ein Scrum-Team aufgebaut ist und wie auftraggebende Personen Anforderungen mithilfe von User Stories definieren.

Scrum-Team

Innerhalb eines Scrum-Teams nehmen die teilnehmenden Personen eine von drei Hauptrollen ein [37, S. 197][65, S. 304ff]:

- **ScrumMaster**

ScrumMaster leiten das Team, tragen aber keine Verantwortung. Sie sollen da-

für sorgen, dass sich das Team für das Produkt verantwortlich fühlt. Außerdem lenken sie den Scrum-Prozess.

- **Product Owner**

Die Product Owner (kurz PO) haben die Vision und das Ziel des Projekts im Blick. Sie erstellen und priorisieren Anforderungen und Tickets. Zusätzlich kontrollieren sie, ob die Umsetzung alle definierten Kriterien erfüllt und die gesteckten Ziele erreicht werden.

- **Development Team**

Das Development Team als Rolle (im Gegensatz zum gesamten Scrum-Team) ist für die Lieferung und das Erstellen des Produkts zuständig. Das Development Team ist selbstorganisiert, so dass jede Person Zusagen einhalten und pünktlich liefern kann.

Neben den Hauptrollen gibt es einige Nebenrollen. Diese sind im Rahmen dieser Arbeit allerdings nicht weiter von Bedeutung.

User Stories

User Stories, kurz Stories, stellen Ziele (z.B. für neue Funktionen) aus Sicht der nutzenden Personen dar. Sie enthalten die Anforderungen in einfacher Sprache. Diese Art der Definition hebt hervor, dass der Mensch im Mittelpunkt steht. User Stories folgen üblicherweise der Form:

Als [Rolle] möchte ich [Funktionalität], damit [Grund]

User Stories werden im ganzen Team besprochen, um Fragen und Einzelheiten zu klären [70].

Akzeptanzkriterien

Akzeptanzkriterien definieren alle Bedingungen, die erfüllt werden müssen, bevor nutzende Personen oder Product Owner Teilschritte der Entwicklung und schließlich das fertige Produkt abnehmen. Akzeptanzkriterien enthalten also die Erwartungen an das System und beschreiben, wie es sich im Rahmen einer User Story verhalten sollte [67].

2.2 Testing

Da der Prototyp Tests generieren soll, ist Testing ein zentrales Thema dieser Arbeit. Dieser Abschnitt geht auf wichtige Begriffe aus diesem Bereich ein und beschreibt Methoden, Test-Arten und Ziele. Zuletzt betrachtet er Testing im Umfeld der agilen Softwareentwicklung und stellt mit *Gherkin* eine Sprache zum Beschreiben von Tests vor (s. Abbildung 2.2).

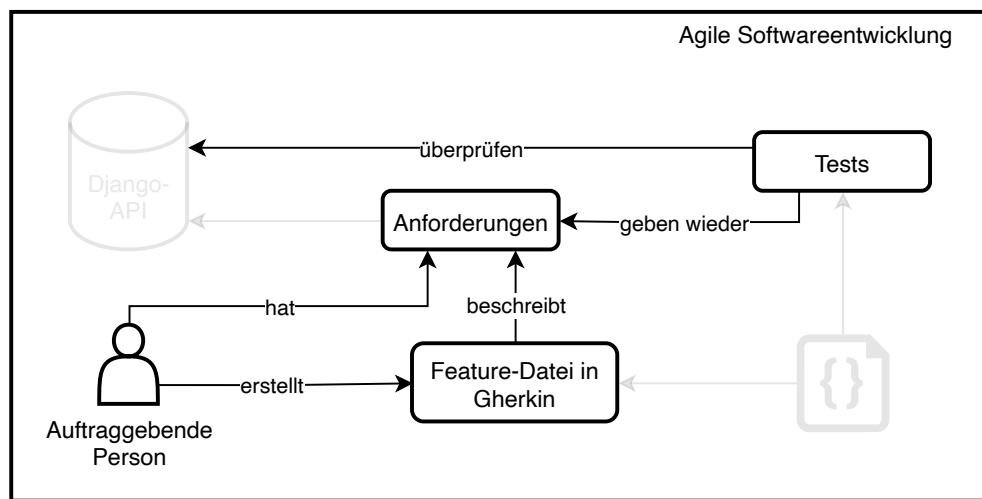


Abbildung 2.2: Abschnitt 2.2 im Ablauf der Generierung (eigene Darstellung)

2.2.1 Cases und Suites

Test Cases (deutsch Testfälle) überprüfen, ob Teile der zugehörigen Software so funktionieren wie sie sollten. Sie enthalten verschiedene Instruktionen, die einzelne Sachverhalte kontrollieren. **Test Suites** sind eine Ansammlung von Test Cases. Sie geben die Möglichkeit der Kategorisierung [53].

2.2.2 Methoden

Verschiedene Methoden erlauben es, eine Applikation und ihren Code zu testen. Dabei kann man hauptsächlich **manuelles** und **automatisiertes** Testen unterscheiden. Manuelles Testen wird von Menschen durchgeführt, die die benutzenden Personen repräsentieren. Sie überprüfen das Verhalten des Systems händisch. Beim automatisierten Testen übernimmt ein Programm diese Aufgabe [58]. Automatisiertes Testen

steht oft in Verbindung mit Continuous Integration (kurz CI) und Continuous Delivery (kurz CD) [71].

2.2.3 Ansätze

Der Ansatz beschreibt, welchen Zugang man aus Sicht der Testenden zum Projekt bzw. zur Applikation hat. Beim **Blackbox Testing** geschieht dies aus der Sicht einer Person, die den Code und die Architektur der Applikation nicht kennt. Beim **Whitebox Testing** hingegen ist dieses Wissen gegeben. Man hat hier auch Zugriff auf den Source-Code. **Greybox Testing** ist eine Mischform der beiden Ansätze. Es sind nicht alle, aber einige Informationen gegeben, damit die Tests nicht komplett „blind“ durchgeführt werden müssen [58].

2.2.4 Ziel

Tests können unterschiedliche Ziele haben. Mithilfe **nichtfunktionaler** Tests etwa lassen sich Probleme entdecken, die nicht direkte Funktionen einer Applikation sind. Dazu gehören Probleme in den Bereichen Performance, Kompatibilitäten oder Sicherheit. **Funktionale** Tests gewährleisten, dass die Funktionen einer Applikation wunschgemäß laufen [58].

2.2.5 Phasen

Tests können in unterschiedlichen Phasen des Buildings stattfinden. „Building“ umfasst hierbei den gesamten Prozess, der beim Erstellen einer Applikation durchlaufen wird. Ein Beispiel für eine Building-Phase ist das Installieren von Abhängigkeiten, z.B. Libraries. **Smoke Testing** beschreibt Fälle, in denen nur kurz überprüft wird, ob das System läuft. Gibt es generelle Probleme mit der Applikation? Kann sie gebaut werden? **Regression Tests** gewährleisten, dass bestehende Funktionen nach Änderungen immer noch intakt sind. **Sanity Tests** stellen sicher, dass Ergänzungen im Code so funktionieren wie sie sollten. Sobald die Ergänzungen nicht mehr als „neu“ zu werten sind, werden Sanity Tests Teil der Regression Tests [39][58].

2.2.6 Level

Der Test für den Code einer Applikation kann auf verschiedenen Levels (oder auch Ebenen) durchgeführt werden [58]. Die folgende Auflistung gibt einen Überblick und weist dabei an einigen Stellen auf unterschiedliche Interpretationsansätze hin. Dabei geht es ausschließlich um funktionale Tests [78].

- **Unit-Testing**

Unit Tests überprüfen einzelne Module oder Bereiche einer Applikation bzw. eines Systems isoliert und ohne Abhängigkeiten. Sie kontrollieren also, ob diese Module in sich stimmig sind und den Anforderungen entsprechend funktionieren [88].

- **Integration Testing**

Integration Testing kontrolliert, ob verschiedene Module einer Applikation immer noch laufen, wenn sie miteinander kombiniert werden [88].

- **E2E Testing**

End-to-End (kurz E2E) Testing (auch Chain Testing genannt) befasst sich mit der kompletten Applikation und allen Workflows. Man versucht dabei das gegebene System so nah wie möglich an die Produktivumgebung zu bringen. Hier sind Netzwerkkommunikation und die Anbindung an eine Datenbank üblich. Ein Beispiel für einen Workflow wäre der Ablauf durch den Prozess einer Bestellung [58][86]. Bibliotheken wie Cypress ermöglichen die Erstellung von E2E-Tests [46].

- **System Testing**

Der Unterschied zwischen System Testing und E2E Testing ist nicht ganz eindeutig und lässt Spielraum für Interpretation. Einige beschreiben **System Testing** als Gesamtheit aller funktionalen und nicht funktionalen Tests [86]. Andere definieren den Begriffe eher im Sinne einer Überprüfung des Systems. Dabei bilden E2E Tests eine Teilmenge der System Tests. E2E Tests konzentrieren sich vor allem auf einzelne Workflows in Applikationen. System Tests hingegen liegen eine Ebene darüber und berücksichtigen alle Funktionen [21][38][58].

- **Acceptance Testing**

Mit dieser Ebene wird überprüft, ob die Anforderungen an das System so erfüllt

sind, wie sie die auftraggebende Person vorgegeben hat. Hierbei handelt es sich normalerweise um den letzten Testschritt vor der Abnahme [58][92].

Die Test-Levels werden üblicherweise in der hier genannten Reihenfolge durchlaufen. Die Menge an Tests nimmt dabei sukzessive ab. Das heißt: Es gibt z.B. mehr Unit-Tests als E2E Tests [88].

2.2.7 Strategien in der Entwicklung

Unterschiedliche Strategien der Entwicklung legen fest, in welchem Kontext Entwickelnde Tests schreiben. Die für diese Arbeit relevanten Strategien sind **Test-First** und **Test-Last**. Abbildung 2.3 gibt einen Überblick über die Unterschiede der beiden Strategien.

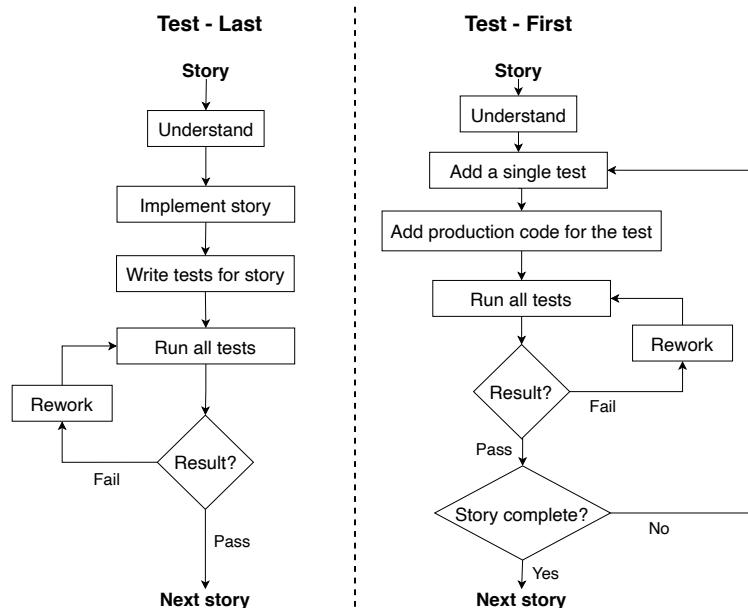


Abbildung 2.3: Test-Last vs. Test-First [13, S. 2]

Bei der Test-Last Strategie wird zunächst eine Story bzw. eine Anforderung implementiert. Im Anschluss schreiben Entwickelnde Tests für diese Anforderung. Dann lassen sie die Tests laufen. Schlagen diese fehl, müssen Tests oder Implementierung angepasst werden. Die Entwickelnden wiederholen diese Schritte so lange, bis die Tests problemlos durchlaufen. Erst dann stellt man sich der nächsten Anforderung.

Einen vergleichbaren Prozess beschreibt das Wasserfallmodell. Es unterscheidet verschiedene Phasen, die die vorherigen bestätigen. Die üblichen Phasen sind: Analyse der Anforderungen, Design, Implementierung, Testing und Betrieb. Dieser Prozess erinnert an den Test-Last Ansatz und ist womöglich ein Grund dafür, warum Test-Last Ansätze eher mit traditionellen Entwicklungsstrategien in Verbindung gebracht werden [13, S. 1f][89, S. 2].

Bei der Test-First Strategie laufen die zentralen Arbeitsschritte in umgekehrter Reihenfolge ab. Hier schreiben Entwickelnde zunächst einen Test und anschließend die zugehörige Implementierung. Wenn das Ausführen der Tests fehlschlägt, müssen Test und / oder Implementierung angepasst werden. Die Entwickelnden wiederholen diesen Prozess so lange, bis alle Anforderungen aus der Story erfüllt sind. Test-First Ansätze stehen oft mit agilen Prozessen in Verbindung. Ein typisches Beispiel für einen solchen Prozess ist Test-Driven Development [13, S. 2].

2.2.8 Test-Driven Development

Test-Driven Development (kurz TDD) orientiert sich an der Test-First Entwicklungsstrategie. Typisch für TDD sind kurze, sich wiederholende Entwicklungsphasen. Für jede Funktionalität gibt es drei Phasen, die man zyklisch durchläuft [40][47][75]:

1. Test
2. Code
3. Refactor

Dieses Vorgehen nennt sich auch Red-Green-Refactor Cycle. Dabei repräsentiert *rot* einen fehlgeschlagenen Test. *Grün* steht für den nach Fertigstellung der Implementierung funktionierenden Test, *Refactor* für das Aufräumen des Codes bzw. für Änderungen am Code. Das frühe Erstellen von Tests garantiert, dass durch Änderungen verursachte Fehler sofort auffallen [49][69]. TDD ist eine Methode, deren Anwendung sich auf das Team der Entwickelnden beschränkt [75].

TDD verspricht unter anderem eine exzellente Code-Qualität, Sicherheit bei der Überarbeitung von Code, frühe Entdeckung von Bugs und eine gute Code-Coverage (Anteil des getesteten Codes im Projekt) [13, S. 4f][40]. Es bietet sich also an, dieses Vorgehen in agile Entwicklung bzw. Scrum zu integrieren (s. Abschnitt 2.1.2).

2.2.9 Behavior-Driven Development

Behavior-Driven Development (kurz BDD) ist eine Erweiterung von TDD. Bei BDD konzentriert sich alles auf das gewünschte Verhalten eines Systems. Anforderungen werden in einfacher Sprache festgehalten (s. Abschnitt 2.2.10) und sind damit sowohl für Entwickelnde als auch für Auftraggebende verständlich. Die entwickelten Anforderungen werden im Anschluss in Tests „übersetzt“.

BDD bezieht das gesamte Scrum-Team und nicht nur die Entwickelnden ein. Allen involvierten Parteien ist es dadurch möglich, die ablaufenden Prozesse besser zu verstehen. Anforderungen und Tests verschmelzen miteinander [75][87].

Einen dem BDD ähnelnden Ansatz verfolgt auch das Acceptance-Test-Driven Development (kurz ATDD) [75]. ATDD ist für diese Arbeit nicht relevant und wird deswegen nicht näher erläutert.

2.2.10 Gherkin

Verschiedene Tools ermöglichen die Entwicklung mit BDD, unter anderem das Open-Source Tool **Cucumber** [75]. Gherkin ist die Sprache, mit der man Tests schreiben kann, die Cucumber anschließend auswertet. Gherkin lässt sich in vielen verschiedenen natürlichen Sprachen (z.B. Englisch oder Deutsch) schreiben, wobei jeweils unterschiedliche Keywords verwendet werden. Dieser Abschnitt klärt die wichtigsten Begriffe und veranschaulicht den Ablauf eines Tests anhand eines Beispiels. Die deutschen Übersetzungen der Keywords sind in der folgenden Auflistung in Klammern angegeben [36][100].

- **Feature (Funktion)**

Jede *Feature*-Datei beschreibt eine Funktion, die getestet wird. Die Beschreibung der Funktion wird nach einem Doppelpunkt hinzugefügt.

- **Rule (Regel)**

Regeln repräsentieren eine Business-Regel im System. Eine Regel gruppiert mehrere Szenarien. Auch hier kann eine Beschreibung ergänzt werden.

- **Background (Hintergrund)**

Ein Hintergrund enthält Schritte, die in allen Szenarien auf der gleichen Ebene relevant sind. Dies ist nützlich, wenn in mehreren Szenarien die gleichen

Bedingungen durch *Given*-Schritte erfüllt werden müssen.

- **Scenario/ Example (Szenario/ Beispiel)**

Ein Szenario beschreibt einen Fall für eine Regel oder eine Funktion. Es definiert einen konkreten Testfall und enthält die einzelnen Schritte.

- **Scenario Outline (Szenariogrundriss)**

Ein Szenariogrundriss beschreibt ein Szenario, aber eröffnet auch die Möglichkeit, einzelne Werte in den Schritten als Variablen zu übergeben.

- **Given, When, Then, And, But (Gegeben sei, Wenn, Dann, Und, Aber)**

Sogenannte **Steps** beschreiben die einzelnen Schritte, die in Tests durchgeführt werden sollen. **Given** versetzt dabei das System immer in den Wunschzustand.

When repräsentiert eine Aktion. **Then** definiert anschließend den Wunschzustand im System. **And** und **But** sind Füllworte, die man verwenden kann, um die Sätze verständlicher zu machen.

Listing 2.1 zeigt eine Feature-Datei, die zwei Testfälle definiert (Zeile 5-7 und 9-11). Da für beide Testfälle ein *User* notwendig ist, liefert Zeile 2 zusätzlich den nötigen *Background*. Er sorgt dafür, dass in beiden Testfällen ein User namens Alice gegeben ist.

```
1  Feature: Check Login for user
2    Background:
3      Given a user Alice
4
5    Scenario: Check that user sees orders after successful login
6        When Alice logs in with valid credentials
7        Then she should see the list of orders
8
9    Scenario: Check that user sees error with incorrect credentials
10       When Alice logs in with invalid credentials
11       Then she should see an error "Invalid credentials"
```

Listing 2.1: Feature-Datei mit Hintergrund und Szenarien

Bei der Entwicklung mit Gherkin müssten Entwickelnde für alle einzelnen Schritte eine Implementierung schreiben und damit gewissermaßen eine Übersetzung durchführen: Was heißt *Given a user Alice*? Was muss im Code passieren, damit Cucumber diesen Schritt verwenden kann? Eine beispielhafte Übersetzung in Python zeigt Listing 2.2. Das hier Beschriebene ist für alle einzelnen Schritte durchzuführen. Anschließend führt Cucumber die Tests automatisch aus und überprüft, ob die einzelnen Schritte erfüllt sind [100].

```

1  @given('a user Alice')
2  def step_impl(context):
3      # add user to context for later use in other steps
4      context.user = get_user(name="Alice")

```

Listing 2.2: Implementierung eines given-Steps

Ein Szenario ist ein funktionaler Test, der sich an gegebenen Akzeptanzkritieren orientiert. Man könnte die Kontrolle der Szenarien also auch als Acceptance Testing sehen. Die Definition der Tests in natürlicher Sprache sorgt für ein gutes Verständnis beim Auftraggebenden, hat aber auch Nachteile: Die Implementierung der Schritte fügt eine Komplexitätsebene hinzu. Bei jeder Funktion müssen bisherige „Übersetzungen“ überprüft werden: Kann man Schritte wiederverwenden? Muss etwas an bisherigen Schritten verändert werden? Gibt es sehr viele Schritte, können die verschiedenen Code-Schnipsel unübersichtlich sein. Sie enthalten gleichzeitig generischen und spezifischen Code [10].

2.3 Generative Programmierung

In der generativen Programmierung (engl. Generative Programming oder kurz GP) versucht man, ein System anhand gegebener Spezifikationen mit textuellen oder grafischen domänenpezifischen Sprachen, z.B. Gherkin, automatisch zu generieren [16, S. 5][17, S. 326f]. Der Output nennt sich Generat. Das den Output erzeugende Programm wird Generator genannt [81, S. 44].

Nach dieser Definition ist der im Rahmen dieser Arbeit zu entwickelnde Prototyp eingebunden in einen Prozess, der sich in Teilen ebenfalls unter den Begriff des gene-

rativen Programmierens fassen lässt (s. dazu Abbildung 2.4). Der Prototyp ist dabei als Generator zu verstehen. Mithilfe einer in Gherkin geschriebenen Feature-Datei liefert er Tests als Generat.

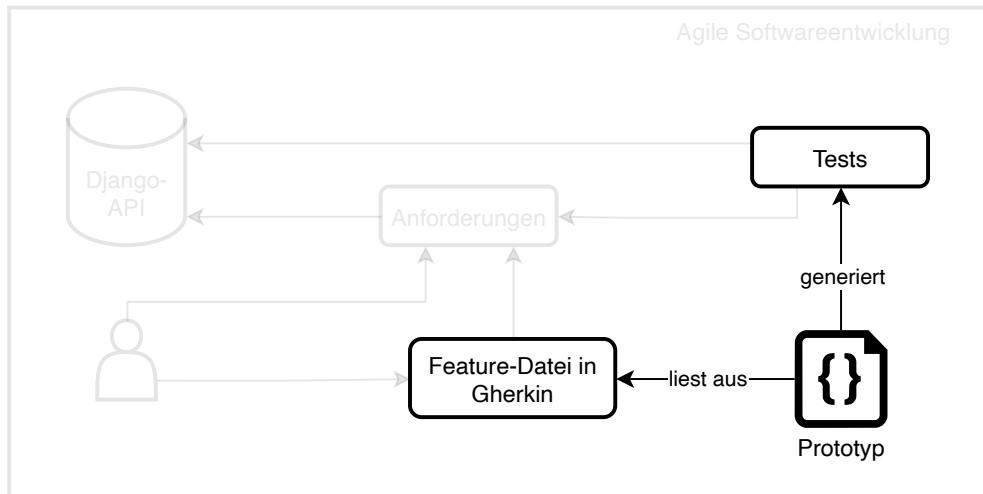


Abbildung 2.4: Abschnitt 2.3 im Ablauf der Generierung (eigene Darstellung)

Vor diesem Hintergrund sollen einige für diese Arbeit relevanten Aspekte des generativen Programmierens näher erläutert werden. Abschnitt 2.3.1 wird sich mit Domänenspezifischen Sprachen befassen, die in der generativen Programmierung eine wichtige Rolle spielen. Sie halten Konzepte und Funktionen in einer Notation fest und vereinfachen dadurch die Anwendung in der betreffenden Domäne. Abschnitt 2.3.2 gibt Einblick in Entwicklungsprozesse der generativen Programmierung. Und Abschnitt 2.3.3 liefert ein abstraktes Modell, zu der Frage, wie man eine domänen-spezifische Sprache in ein Generat umwandeln kann.

2.3.1 Domänenspezifische Sprachen

Domänenspezifische Sprachen (engl. Domain-Specific languages, kurz DSL) sind spezialisierte und problemorientierte Sprachen für eine gewählte **Domäne**. Im Gegensatz dazu stehen General-Purpose Languages (kurz GPL), wie Java oder Python, bei denen keine Spezialisierung existiert [15, S. 29f].

Die gewählte Syntax einer DSL sollte [16, S. 137ff][82, S. 97f]:

- die Domäne gut repräsentieren.
- eine einfache Weiterentwicklung ermöglichen.
- für eine Person aus der gewählten Domäne verständlich sein.

Der Begriff „domänenspezifische Sprache“ lässt zunächst auf eine Sprache der textuellen Art schließen. Ein Beispiel hierfür ist das in Abschnitt 2.2.10 erwähnte Gherkin. Es ist eine Sprache für die Domäne *Testing* und zeichnet sich durch einfache und für alle Beteiligten lesbaren Texte aus [62]. Ein weiteres Beispiel ist Structured Query Language (kurz SQL), eine auf Datenbanken spezialisierte Sprache [56]. Allerdings kann man eine DSL auch über einen GUI Builder definieren [68]. Benutzende Personen würden beispielsweise die grafische Oberfläche einer Applikation festhalten, womit der zugehörige Code generiert wird [16, S. 137].

DSLs können zudem in interne und externe Varianten eingeteilt werden. Eine interne DSL ist in eine andere Sprache eingebettet. Beliebte Programmiersprachen für interne DSL-Entwicklung sind Ruby und LISP. In diesen Sprachen ist es dem Programmierenden möglich, tief in die interne Logik der Programmiersprache einzugreifen. Externe DSLs sind unabhängig von anderen Sprachen entwickelt. Gherkin und SQL passen in diese Kategorie. Ein weiteres Beispiel für eine solche Sprache ist TeX[16, S. 138][82, S. 98].

Damit Computer externe DSLs verstehen bzw. auswerten können, übersetzt man die Sprache üblicherweise in eine GPL. Der entstehende Code kann anschließend weiterverarbeitet werden [15, S. 29f].

2.3.2 Entwicklungsprozesse

Die Entwicklung von Systemen in der generativen Programmierung ist in zwei Hauptbereiche unterteilt: Domain Engineering und Application Engineering. Das Domain Engineering bzw. „development for reuse“ befasst sich mit der Entwicklung von wiederverwendbaren Assets, z.B. Generatoren. Man überlegt, was Applikationen einer Domäne brauchen könnten, und bildet dies mithilfe von Assets ab. Das Application Engineering bzw. „development with reuse“ verwendet diese Assets, um die Entwicklung von Applikation zu erleichtern [17, S. 328f].

Die beiden Bereiche ergänzen einander und die darin stattfindenden Prozesse werden üblicherweise zyklisch durchlaufen. Zunächst entstehen Assets, die eine Applikation verwendet. Beim Application Engineering stellt man unter Umständen fest, dass Anforderungen einer Applikation nicht passend abgebildet werden oder dass es zu Fehlern kommt. Diese Informationen greift das Domain Engineering auf und erstellt damit verbesserte Assets. Es entsteht somit ein Kreislauf, der in Abbildung 2.5 erkennbar ist [17, S. 329f].

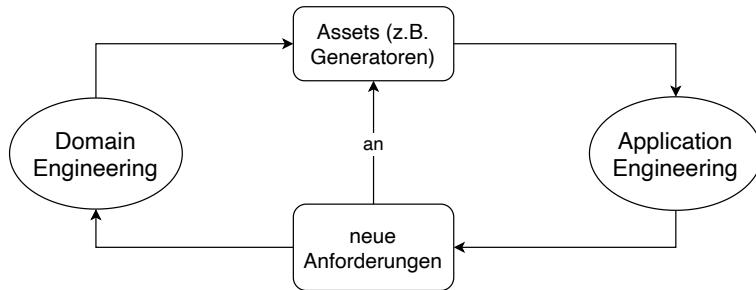


Abbildung 2.5: Domain Engineering und Application Engineering Kreislauf (entnommen aus [17, S. 329])

Dieser Prozess kann auch auf die in dieser Arbeit gestellten Ziele aus Abschnitt 1.4 übertragen werden. Diese Arbeit erstellt einen Prototyp, der Code generiert. Die Entwicklung zählt somit zum Domain Engineering und der Prototyp ist ein daraus entstehendes Asset. Bei einer künftigen Bewertung des Prototyps würde man ihn auf Applikationen bzw. Projekten anwenden. Danach ließen sich die resultierenden Informationen verwenden, um den Prototyp zu verbessern und weiterzuentwickeln.

2.3.3 Generative Domain Model

Ein zentrales Konzept in der generativen Programmierung ist das Mapping, also das Verknüpfen von *Problem Space* und *Solution Space*. Das Konzept nennt sich auch *Generative Domain Model* (kurz GDM). Es ist in Abbildung 2.6 dargestellt und beschreibt, wie man mithilfe eines Ansatzes von einem Problem zur Lösung kommt. Man kann das Modell grundsätzlich aus zwei Sichtwinkeln betrachten: aus der Sicht der Konfiguration und aus der Sicht der Transformation. Problem Space und Solution Space haben dann jeweils unterschiedliche Inhalte [17, S. 330]. Für diese Arbeit ist nur die Sicht der Transformation wichtig.

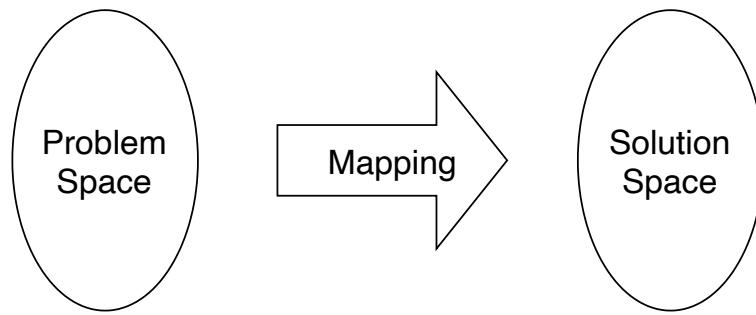


Abbildung 2.6: Generative Domain Model (entnommen aus [17, S. 330])

Aus der Sicht der Transformation ist der Problem Space eine DSL. Der Solution Space ist die übersetzte Implementierung in einer GPL. Das Mapping zwischen ihnen ist eine Transformation, die die DSL nimmt und als GPL ausgibt [17, S. 331f]. Ein Beispiel für diese Form des Mappings sind **Compiler**. Die Sicht der Transformation auf das GDM ist in Abbildung 2.7 dargestellt. Es wird deutlich, dass der dort zu sehenden Ablauf dem in Abbildung 2.4 sehr ähnlich ist. Damit liegt der Gedanke nahe, dass ein Compiler ein interessanter Ansatz sein könnte für die Entwicklung eines Prototyps, der Feature-Dateien zu Tests umwandeln kann.

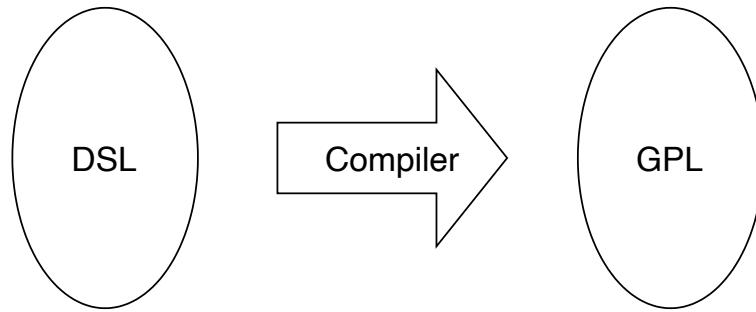


Abbildung 2.7: Generative Domain Model aus Sicht der Transformation (entnommen aus [17, S. 330])

2.4 Grammatik

Unter Grammatik versteht man in der Informatik Modelle, die zur Beschreibung formaler Sprachen dienen. Grammatiken werden unter anderem im Compilerbau angewendet und legen eindeutig fest, ob ein Wort Element einer Sprache ist [73, S. 76].

Da der hier zu entwickelnde testgenerierende Prototyp Gherkin umwandeln soll, muss dieser in der Lage sein, die Syntax der Sprache bzw. deren Grammatik analysieren zu können. Abschnitt 2.4.1 beschreibt zunächst die grundlegenden Eigenschaften einer Grammatik und formaler Sprachen. Es folgen Ausführungen zur Chomsky-Hierarchie, die verschiedenen Arten von Grammatiken enthält. Abschließend geht es um die Notierung von Grammatiken. Ziel ist es an dieser Stelle, einen Überblick über Regeln zu bekommen, die der Prototyp verwenden soll.

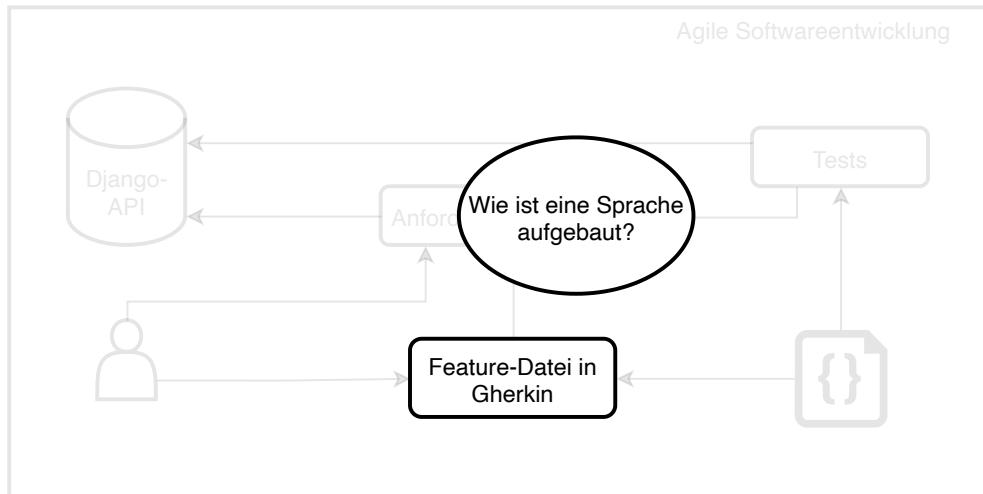


Abbildung 2.8: Abschnitt 2.4 im Ablauf der Generierung (eigene Darstellung)

2.4.1 Grundlagen

Eine Grammatik enthält verschiedene Regeln, die eine Sprache L beschreiben bzw. erzeugen. Eine Grammatik besteht aus vier Komponenten [73, S. 75ff]:

- einer endlichen Menge V , deren Elemente Variablen (oder **Nichtterminal-symbole**) heißen. Sie sind dadurch charakterisiert, dass man sie ersetzen kann (z.B. A). Nichtterminalsymbole werden normalerweise groß geschrieben.

- einer endlichen Menge Σ , deren Elemente **Terminalsymbole** (oder kurz **Terminale**) genannt werden. Die Elemente können nicht weiter aufgebrochen oder ersetzt werden (z.B. a). Die Menge heißt auch **Alphabet**. Terminalsymbole sind meistens durch Kleinschreibung kenntlich gemacht. Eine Kombination mehrerer Terminalsymbole im Alphabet heißt **Wort**.
- einer endlichen Menge P , die Produktionsregeln enthält. Diese Regeln folgen der Form: $A \rightarrow aa$. Das Nichtterminalsymbol A ließe sich also durch die Terminalsymbole aa ersetzen.
- einem Nichtterminalsymbol S , das den Start einer Sequenz definiert.

Eine formale Sprache besteht aus allen Wörtern, die mithilfe der Produktionsregeln gebildet werden können. Diese Menge aus Wörtern kann leer, endlich oder unendlich sein [73, S. 75ff]. Schauen wir uns als Beispiel einige Produktionsregeln für eine Grammatik an:

$$\begin{aligned} S &\rightarrow AB \quad (1) \\ A &\rightarrow aB \quad (2) \\ B &\rightarrow ba \quad (3) \end{aligned}$$

Produktionsregeln 2.1: Einfaches Beispiel

S , A und B sind die Nichtterminalsymbole. a und b stellen die Terminalsymbole dar. Und S definiert den Start der Sequenz. Generell können sowohl auf der linken als auch auf der rechten Seite einer Produktionsregel Terminalsymbole und Variablen stehen. Jede Regel hält fest, wie Terminals bzw. Variablen ersetzt werden dürfen. Mit den Regeln 1, 2 und 3 lässt sich zum Beispiel die folgende Sequenz aus Terminalen bilden:

$$S \xrightarrow{(nach \ 1)} AB \xrightarrow{(nach \ 2)} aBB \xrightarrow{(nach \ 3)} abaB \xrightarrow{(nach \ 3)} ababa \quad (\text{F-2.1})$$

Die einzelnen Schritte werden auch Ableitungen genannt. Produktionsregeln definieren hierbei, welche Ableitungen möglich sind. Die Klammern über den Pfeilen gehören nicht zur Notation. Sie dienen lediglich dem Verständnis.

2.4.2 Chomsky-Hierarchie

Nach der Chomsky-Hierarchie gibt es vier verschiedene Arten von Grammatiken, die sich durch die jeweiligen Produktionsregeln unterscheiden und aufeinander aufbauen [73, S.79ff].

- **Grammatik ohne Einschränkungen (Typ-0)**

Diese Grammatiken haben keine Einschränkungen.

- **Kontextsensitive Grammatik (Typ-1)**

Für alle Produktionsregeln einer Typ-1 Grammatik muss gelten:

$$w_1 \rightarrow w_2, \text{ dann } |w_1| \leq |w_2| \quad (\text{F-2.2})$$

Wenn also ein Nichtterminalsymbol ersetzt wird, dann muss der neue Wert mindestens so lang sein wie der vorherige. Folgende Produktionsregel wäre demnach nicht gültig:

$$AAA \rightarrow ab$$

Produktionsregeln 2.2: Ungültig für kontextsensitive Grammatik

- **Kontextfreie Grammatik (Typ-2)**

Bei allen Produktionsregeln einer kontextfreien Grammatik darf auf der linken Seite nur ein Symbol stehen. Ist eine Sequenz aus Symbolen gegeben, betrachten Typ-2 Grammatiken immer nur ein Symbol. Darin liegt der zentrale Unterschied zu Typ-1 Grammatiken. Um dies klar zu machen, sei als Gegenbeispiel eine **kontextsensitive Grammatik** gegeben:

$$\begin{aligned} S &\rightarrow AAAB \\ AA &\rightarrow aB \\ AB &\rightarrow aaB \\ B &\rightarrow a \end{aligned}$$

Produktionsregeln 2.3: Kontextsensitive Grammatik

Auf der linken Seite der Produktionsregeln $AA \rightarrow aB$ und $AB \rightarrow aaB$ stehen mehrere Symbole. Beim Ableiten muss man also 2 Symbole betrachten und klären, ob die erste oder zweite Produktionsregel zu verwenden ist. Befindet sich auf der linken Seite nur ein Symbol, ist dies nicht mehr erforderlich.

- **Reguläre Grammatik (Typ-3)**

Eine reguläre Grammatik darf nur folgende Arten von Produktionsregeln enthalten, wobei ϵ für eine leere Sequenz steht:

$$\begin{aligned} A &\rightarrow \text{Terminalsymbol Nichtterminalsymbol } \mathbf{ODER} \\ A &\rightarrow \text{Terminalsymbol } \mathbf{ODER} \\ A &\rightarrow \epsilon \end{aligned}$$

Produktionsregeln 2.4: Reguläre Grammatik

Für jede Grammatik gelten auch die Regeln der Grammatiken eines niedrigeren Typs. Die nach den Regeln der jeweiligen Grammatik erzeugten Sprachen werden so zur Teilmenge einer anderen Sprache. Das heißt: Ist L_i gegeben, wobei i der Typ der Grammatik dieser Sprache ist (von 0 bis 3), gilt [79]:

$$L_3 \subset L_2 \subset L_1 \subset L_0 \quad (\text{F-2.3})$$

2.4.3 (Erweiterte) Backus-Naur-Form

J. Backus und P. Naur haben die Backus-Naur-Form (kurz BNF) 1960 entwickelt. Sie wurde in ihrer ursprünglichen Form eingeführt, um die Sprache Algol 60 zu beschreiben. Niklaus Wirth erweiterte diese Notation später, um sie lesbarer und einfacher zu machen. Die von ihm entwickelte Notation heißt erweiterte (oder auch extended) Backus-Naur-Form (kurz EBNF). Sie kann die Produktionsregeln kontextfreier Grammatiken beschreiben [101, S. 7f].

Diese Arbeit verwendet lediglich die Syntax von EBNF. Sie sieht folgendermaßen aus:

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
letter = "A" | ... | "Z"
```

Produktionsregeln 2.5: Definition der Variablen digit und letter (EBNF)

Die Produktionsregeln 2.5 definieren Zahlen und Buchstaben. Einzelne Terminal-symbole sind also mit "<char>" festgehalten. | steht für das logische Oder. Würde man einen String definieren, ergäben sich die Produktionsregeln 2.6.

```
character = letter | digit
string = "'", {character}, "'"
```

Produktionsregeln 2.6: Verwenden von Variablen und Folgen (EBNF)

Hier wird ein *character* so definiert, dass er entweder ein *digit* oder ein *letter* ist. Zusätzlich besteht ein *string* aus einem Anführungszeichen, einer beliebigen Anzahl an *characters* und einem weiteren Anführungszeichen. { } steht also für 0 bis N Folgen des Inhalts. Ein + nach der Klammer deutet hingegen auf 1 bis N hin. , steht für eine Verkettung von Symbolen, die aufeinander folgen müssen. Das Komma kann alternativ durch ein einfaches Leerzeichen ersetzt werden.

```
integer = [ "-" ], {digit}+
```

Produktionsregeln 2.7: Optional (EBNF)

Für diese Arbeit ist noch ein weiteres, in Produktionsregel 2.7 festgehaltenes Zeichen wichtig. Das Symbol – ist als optional definiert und durch die Klammern [] hervorgehoben. Das von der jeweiligen Programmiersprache abhängige Limit, also der maximale und minimale Wert eines Integers, wurde an dieser Stelle bewusst ignoriert, um das Beispiel einfach zu halten.

Schreibe man all diese Regeln übereinander, stünden Regeln mit Terminalsymbolen immer unten. Die Regel mit dem Startsymbol steht am Anfang. Die Produktionsregeln 2.8 zeigen, wie dies aussieht. Die darin eingeführte Syntax beschreibt eine Sprache, in der zuerst ein String kommen muss, auf den ein Integer folgt [101, S. 7ff].

```
syntax = string, integer
string = "", {character}, ""
character = letter | digit
integer = [ "-" ], {digit}+
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
letter = "A" | ... | "Z"
```

Produktionsregeln 2.8: Zusammenfassung aller Regeln (EBNF)

Neben der EBNF können auch Automaten Grammatiken beschreiben. Beispiele dafür sind endliche Automaten, Kellerautomaten oder die Turingmaschine [73, S. 86-101]. Eine weitere Option für eine Notation sind Syntaxdiagramme [73, S. 95ff]. Automaten und Syntaxdiagramme sind für diese Arbeit nicht relevant.

2.5 Compiler

Compiler spielen in der Informatik eine zentrale Rolle. Sie übersetzen einen Input (üblicherweise einen Quelltext) so, dass dieser weiterverwendet werden kann. Ein klassisches Beispiel dafür sind Programme. In diesem Fall wäre der zugehörige Code in einer Programmiersprache der Input. Der Rechner versteht diese Sprache allerdings nicht direkt. Er interpretiert lediglich Instruktionen bzw. Befehle. Das Programm, das die Programmiersprache in die Instruktionen für den Rechner übersetzt, ist der Compiler [101, S. 1]. Beispiele dafür sind die in Abschnitt 2.3.1 behandelten DSLs, die in eine GPL umgewandelt werden müssen, damit man sie ausführen kann [15, S. 31].

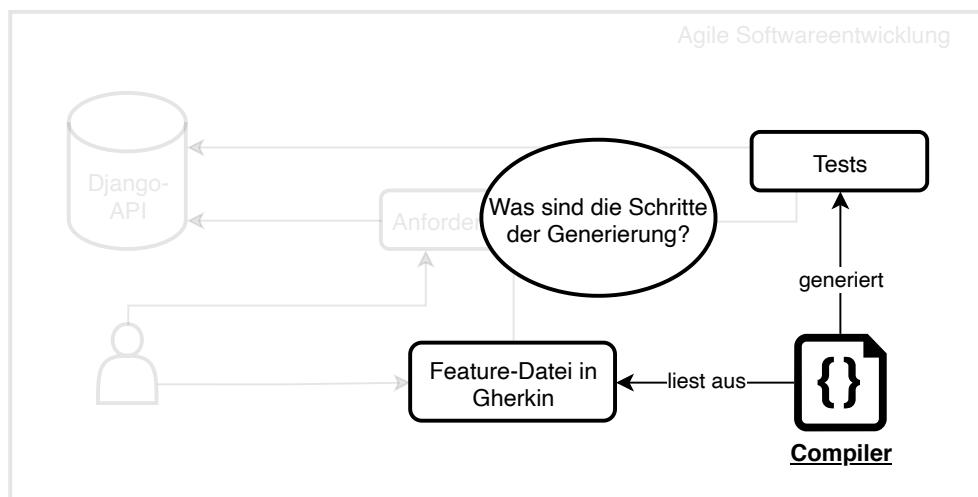


Abbildung 2.9: Abschnitt 2.5 im Ablauf der Generierung (eigene Darstellung)

Gherkin gilt als eine DSL (s. Abschnitt 2.3.1). Entwickelnde müssen Gherkin normalerweise händisch zu Test-Code in einer GPL übersetzen (s. Abschnitt 2.2.10). Ein Compiler kann eine solche Umwandlung ebenfalls durchführen und eignet sich damit für die Umsetzung des in Abbildung 2.9 hervorgehobenen Teils des in dieser Arbeit betrachtete Gesamtprozesses.

Dieser Abschnitt beschreibt die Funktionsweise eines Compilers und stellt dessen Komponenten vor: Lexer, Parser, Type-Checking und Code-Generator.

2.5.1 Lexer

Den ersten Schritt des Compilers führt der Lexer (auch Scanner genannt) aus. Der Name leitet sich aus seiner Funktion ab: lexical analysis - **lexikalische Analyse**. Der Lexer ist dafür zuständig, den Input in Tokens umzuwandeln. Diese Tokens repräsentieren das Vokabular einer Sprache. Tokens können z.B. Wörter, Zahlen, Zeichen oder eine Kombination aus ihnen sein [101, S. 1].

```
1 foo + 2
```

Listing 2.3: Beispiel-Input für einen Lexer

Wenn also ein Lexer den Input aus Listing 2.3 erhält, dann wäre *foo* vermutlich eine Variable, *+* ein Operator und *2* eine Zahl. Der Lexer würde hier also eine Folge aus drei Tokens erkennen: *Variable*, *Operator* und *Integer*. Der jeweilige Input, der zu diesem Token geführt hat, wird **Lexeme** genannt. Ein **Pattern** beschreibt eine Form, wie die Lexemes eines Tokens aussehen können [2, S. 111].

2.5.2 Parser

Nachdem der Lexer den eingelesenen Quelltext in lexikalische Einheiten, die Tokens, unterteilt hat, steht der nächste Schritt des Compilings an: Ein Parser überprüft, ob der eingelesene Quellcode zu einer gewählten Grammatik passt. Dies heißt auch **syntaktische Analyse**. Dabei wandelt der Parser Tokens in eine Struktur um, die der Compiler intern weiter verwenden kann. Diese Struktur nennt sich **Abstract Syntax Tree** (kurz AST). Der Baum, der die Struktur der Grammatik wiedergibt, wird **Concrete Syntax Tree** (kurz CST) genannt [11][73, S. 106f][80].

Der Parser erhält also eine Token-Sequenz und muss überprüfen, ob diese zu einer Grammatik passt bzw. ob diese Sequenz valide ist. Diese Überprüfung ist in zwei verschiedenen Richtungen möglich: Von *S* ausgehend zur Sequenz oder umgekehrt. Anhand der Ableitungen aus F-2.1 (s. Seite 25) könnte der Parser also überprüfen: Kommt man bei rückwärts ablaufender Ableitungen von *ababa* zu *S*? Oder alternativ: kann man mithilfe bestimmter Ableitungen von *S* zu *ababa* kommen [73, S. 106f]?

Diese zwei Ansätze werden durch zwei Parsertypen umgesetzt: den Top-Down Parser und den Bottom-Up Parser. Bei beiden muss eine kontextfreie Grammatik gegeben sein.

Top-Down Parser

Der Top-Down Parser (auch LL-Parser genannt) arbeitet den Syntaxbaum von oben nach unten ab. Um seine Funktionsweise zu veranschaulichen, werden die Abläufe an dieser Stelle beispielhaft dargestellt:

$$\begin{aligned} S &\rightarrow ABd \\ A &\rightarrow aBc \\ B &\rightarrow b \end{aligned}$$

Produktionsregeln 2.9: Funktionsweise der Parser

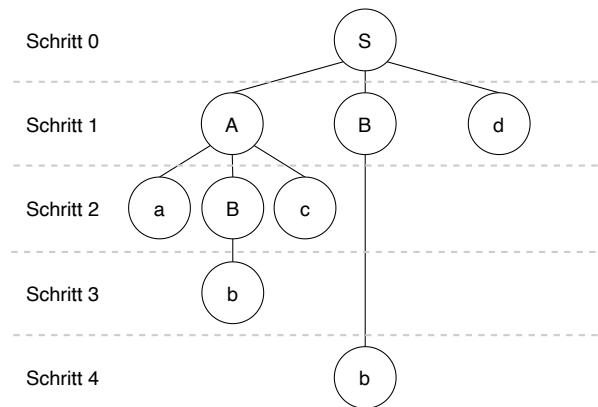


Abbildung 2.10: Baum für Top-Down Parsing (eigene Darstellung)

Im Beispiel aus Abbildung 2.10 ist eine Sequenz *abcd* gegeben. Der Parser muss überprüfen, ob sie zur Grammatik aus den Produktionsregeln 2.9 passt. Der Parser läuft also den Baum ab, der durch die Grammatik entsteht, und versucht, auf den Input-Text zu kommen [101, S. 19ff].

Jede Reihe steht dabei für einen Schritt in den Ableitungen. Die Ableitungen sehen wie folgt aus:

$$S \Rightarrow ABd \Rightarrow aBcBd \Rightarrow abcBd \Rightarrow abcbd \quad (\text{F-2.4})$$

Der Top-Down Parser heißt also auch LL-Parser, weil er den Input von links nach rechts bearbeitet und Nichtterminalsymbole von links nach rechts ersetzt (auch *expand* genannt). Top-Down Parser können keine Grammatiken behandeln, die linksrekursiv oder mehrdeutig sind. Folgende Regeln wären linksrekursiv:

$$\begin{aligned} S &\rightarrow BA \\ A &\rightarrow Aa \mid \epsilon \end{aligned}$$

Produktionsregeln 2.10: Linksrekursive Grammatik

Diese Grammatik erlaubt das Ersetzen von A durch Aa . Auf der linken Seite findet also eine Rekursion statt. Ein LL-Parser hätte hier Probleme, eine Lösung zu finden, da er immer zuerst die linken Pfade des Baums abläuft. Die folgenden Produktionsregeln zeigen, wie eine mehrdeutige Grammatik aussieht.

$$\begin{aligned} E &\rightarrow A \\ A &\rightarrow AA \mid a \end{aligned}$$

Produktionsregeln 2.11: Mehrdeutige Grammatik

Diese Grammatik erlaubt verschiedene Bäume, die ein Parser unterschiedlich behandeln kann. Abbildung 2.11 zeigt zwei gültige Bäume für den Input aaa . LL-Parser können das Problem hier nicht lösen, da sie nicht unbedingt den „richtigen“ Baum auswählen.

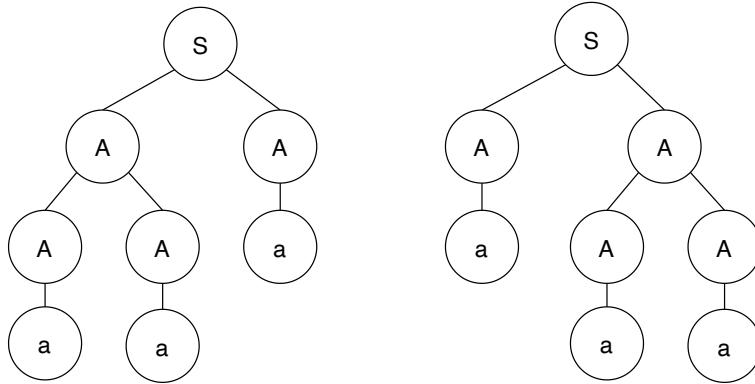


Abbildung 2.11: Bäume für mehrdeutige Grammatik [101, S. 9]

In beiden Fällen muss die Grammatik so umgeschrieben werden, dass sie weder linksrekursiv noch mehrdeutig ist [60][101, S. 9]. Top-Down Parser können weiter unterteilt werden in:

- **Tabellenbasierte Top-Down Parser (LL(1))**

Ein tabellenbasierter Top-Down Parser erstellt anhand der Produktionsregeln eine Tabelle. Sie hält für alle Variablen fest, was beim Ersetzen als erstes Terminalsymbol möglich ist und welches Terminalsymbol auf die Variable folgen kann. Daraus entsteht eine zweite Tabelle. In ihr wird ein Terminalsymbol aus der gegebenen Sequenz und einer zu ersetzenen Variable einer konkreten Regel zugeordnet, die der Parser in diesem Fall verwenden soll. Die Konstruktion dieser Tabelle und der Algorithmus des Parsers ist für diese Arbeit nicht weiter relevant [28, S. 20ff]. Die 1 in LL(1) bezieht sich hier auf die Folge-Symbole, die bei der Erstellung der Tabelle betrachtet werden. Sobald die Zahl größer als 1 ist, ist die zugehörige Grammatik linksrekursiv oder mehrdeutig und somit nicht lösbar [28, S. 27].

Der Vorteil tabellenbasierter Top-Down Parser besteht darin, dass sie sehr effizient arbeiten. Allerdings ist die Erstellung der Tabellen für Menschen nicht trivial. Für die Erstellung der Tabellen gibt es einige Werkzeuge [28, S. 6][50].

- **Rekursive Top-Down Parser**

Rekursive Top-Down Parser laufen den Baum ab und rufen Funktionen rekursiv auf, um zu überprüfen, ob die Sequenz mit der Grammatik übereinstimmt. Der

Ablauf soll anhand des Baums in Abbildung 2.10 erklärt werden [29, S. 20ff]:

1. Der Parser erhält die Sequenz $abcbd$. Die definierte Grammatik aus den Produktionsregeln 2.9 startet mit S . Dementsprechend ruft der Parser S auf und lässt es die Sequenz überprüfen. Zusätzlich hält der Parser einen Pointer fest, der auf das aktuell zu überprüfende Terminalsymbol in der Sequenz zeigt. Der Startindex des Pointers ist 0.
2. $S \rightarrow ABd$ definiert, dass beim Ersetzen von S zunächst die Variable A eingesetzt werden muss. Deswegen ruft S zunächst A auf, damit A überprüfen kann, ob die Sequenz valide ist.
3. A überprüft, ob das erste Symbol ein a ist. Dies ist der Fall. A bewegt den Pointer weiter. Er zeigt jetzt auf Index 1 - also auf das b .
4. A überprüft, ob das zweite Symbol ein B ist. Da es sich hier um eine Variable handelt, ruft es B auf.
5. B nutzt den Pointer und überprüft, ob bei Index 1 ein b steht. Dies ist der Fall. Also erhöht B den Index um 1 und gibt die Kontrolle an A zurück.
6. A muss schließlich noch prüfen, ob auf Index 2 ein c gegeben ist. Dies ist der Fall. Also erhöht A den Index des Pointers auf 3. Anschließend gibt A die Kontrolle an S .
7. S weiß nun, dass A valide ist. Es muss allerdings noch zwei weitere Symbole überprüfen. Da B eine Nichtterminalsymbol ist, ruft es B auf.
8. B agiert wie in Schritt 5. Allerdings erhöht sich hier der Index auf 4.
9. S muss nur noch d überprüfen. Auch dies ist valide. Daraus kann der Parser schließen, dass die Sequenz zu der Grammatik passt.

Passt beim Durchlaufen der Sequenz ein Terminalsymbol nicht zum aktuellen Symbol, auf das der Pointer zeigt, entsteht ein Fehlerzustand. Hat eine Variable mehrere Optionen oder ist optional, wird der Index des Pointers wieder zurückgesetzt und der Prozess läuft normal weiter. Gegeben seien die Produktionsregeln 2.12.

$$\begin{array}{l} S \rightarrow A \\ A \rightarrow abb \mid aba \end{array}$$

Produktionsregeln 2.12: Grammatik für rekursive Fehlerbehandlung

1. S erhält die Sequenz aba . S ruft A auf.
2. A hat zwei Optionen für valide Sequenzen: abb und aba . Es überprüft zunächst abb . Das erste Symbol a passt. Der Index erhöht sich auf 1.
3. Das zweite Symbol b passt auch. Der Index erhöht sich auf 2.
4. Das dritte Symbol passt allerdings nicht. A setzt also den Index wieder auf den ursprünglichen Wert 0 zurück und überprüft seine zweite valide Sequenz.
5. Die zweite Sequenz ist valide.
6. S hat alle Variablen überprüft und bestätigt, dass der Input zur Grammatik passt.

Das rekursive Top-Down Parsing hat einige Vorteile. Die Implementierung ist einfach und nicht zur Grammatik passende Inputsequenzen werden zuverlässig gefunden [28, S. 6].

Bottom-Up Parser

Neben Top-Down Parsern gibt es Bottom-Up Parser (auch LR-Parser). Sie setzen im unteren Teil eines Syntaxbaums an. Dabei versuchen sie, Terminalsymbole im Input zu ersetzen (auch reduce genannt), um den Baum weiter nach oben ablaufen zu können.

Abbildung 2.12 zeigt einen typischen Verlauf, der die Unterschiede zum Top-Down Parsing aus Abbildung 2.10 deutlich macht.

Die zugehörigen Ableitungen (von oben nach unten) sehen folgendermaßen aus:

$$S \Rightarrow ABd \Rightarrow Abd \Rightarrow aBcbd \Rightarrow abc bd \quad (\text{F-2.5})$$

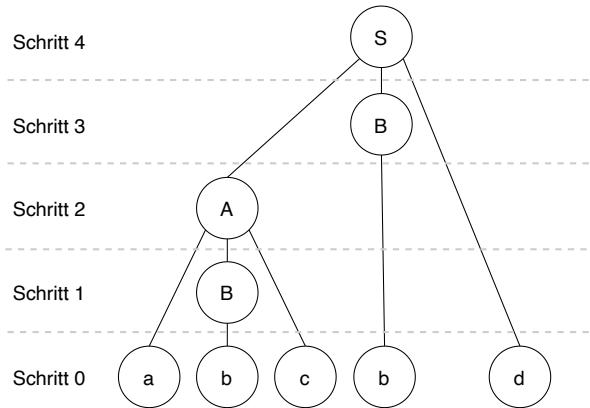


Abbildung 2.12: Baum für Bottom-Up Parsing (eigene Darstellung)

Die Variablen werden hier von rechts nach links abgeleitet. Daher kommt auch der Name **LR**-Parser. Im Gegensatz zu LL-Parsern arbeiten alle LR-Parser tabellenbasiert. Sie sind generell komplizierter als Top-Down Parser, da sich hier immer die Frage stellt: Wann und zu was muss ein Symbol reduziert werden? Allerdings sind LR-Parser mächtiger als LL-Parser. Sie können beispielsweise mit Mehrdeutigkeit und linksrekursiven Grammatiken arbeiten. Die Tabellenerstellung ist auch hier für Entwickelnde eine Herausforderung [30, S. 4ff]. Der Algorithmus und die Erstellung der Tabellen für LR-Parser sind für diese Arbeit nicht relevant.

Auch bei Bottom-Up Parsern gibt es verschiedene Vorgehensweisen, die hier der Vollständigkeit halber erwähnt sind [30, S. 5]:

- **Simple LR (kurz SLR)**

Hierbei handelt es sich um die einfachste Vorgehensweise, die allerdings nicht für eigene Grammatiken anwendbar ist.

- **Kanonisches LR**

Es ist das mächtigste Verfahren, allerdings auch sehr schwer zu implementieren.

- **Lookahead LR (LALR)**

Liegt in der Komplexität zwischen den beiden anderen Verfahren. *yacc* und *bison* verwenden es.

Verschiedene Tools, etwa *yacc*, ermöglichen es, einen Parser automatisch zu generieren [16, S. 339].

2.5.3 Type-Checking

Nach dem Parsing überprüfen Compiler von typisierten höheren Programmiersprachen die Typen verschiedener Werten, wie Variablen oder Funktionen. Dabei kontrolliert der Compiler auch die Verwendung von Operatoren und Operanden [101, S. 1f].

2.5.4 Code-Generator

Bei der Code-Generierung handelt es sich um den letzten Schritt des Compilings. Hier verwendet der Compiler den AST des Parsers, um die Implementierung (für eine untergeordnete Programmiersprache) zu generieren. Es ist meistens der komplizierteste Schritt des Compilings [101, S. 1f]. Die Aufgaben eines Code-Generators sind [16, S. 333][94][101, S. 1f]:

- **Umwandlung von AST:** Der AST muss in Instruktionen umgewandelt werden. Hierbei ist es wichtig herauszufinden, welche Instruktionen notwendig sind und in welcher Reihenfolge sie ausgeführt werden müssen.
- **Generierung von validem Code:** Der entstehende Code muss valide sein und verwendet werden können. Entwickelnde sollten in der Lage sein, den generierten Code zu verstehen, wenn sie Anpassungen vornehmen wollen.
- **Performance:** Die Generierung muss effizient hinsichtlich CPU und Speicher sein.

Hierbei werden verschiedene Techniken angewendet, die auch in Mischformen auftreten [31, S. 124ff][51, S. 156].

Die Technik, die wohl am weitesten verbreitet ist, ist die Benutzung von **Templates**. Hier nutzt man Vorlagen, die mit Werten befüllt werden, wie folgendes Beispiel zeigt (entnommen aus [82, S. 146]):

<Anrede>,

Wir bitten Sie die Rechnung <Rechnungsnr.> vom <Rechnungsdatum> in Höhe von <Rechnungsbeitrag> zu überweisen.

Mit freundlichen Grüßen,

<Signatur>

In diesem Template gibt es verschiedene Platzhalter, die durch gegebene Werte ersetzt werden. So würde hier z.B. *Rechnungsnr.* mit der gegebenen Nummer der Rechnung gefüllt. Dementsprechend wäre der Text nun *Wir bitten Sie die Rechnung 123 vom....* Im Bereich der Code-Generierung enthalten die Templates quasi den Rahmen für den zu generierenden Code.

Der zu generierende Code kann sich von Programmiersprache zu Programmiersprache unterscheiden. Das Template kann den Rahmen für eine einzelne Sprache enthalten. Ein Beispiel dafür liefert Listing 2.4. Es zeigt ein Template für die Definition einer If-Bedingung in Python.

```
1 if <conditions>:  
2     <children>
```

Listing 2.4: Beispiel eines Templates für eine If-Bedingung in Python (vereinfacht)

Zusätzlich gibt es noch die Möglichkeit, Templates so zu verwenden, dass sie abstrakte Definitionen für den Code enthalten. Das hat den Vorteil, dass das Template für verschiedene Programmiersprachen verwendbar ist. Listing 2.5 zeigt, wie ein solches Template aussehen könnte.

```
1 <if> <cond_start> <conditions> <cond_end> <if_start>  
2     <children>  
3 <if_end>
```

Listing 2.5: Beispiel eines Templates für eine If-Bedingung in beliebiger Sprache (vereinfacht)

Bei der Generierung mithilfe von Templates orientiert man sich an der Struktur des Outputs. Für Templates gibt es viele verschiedene Bibliotheken und Tools, die erweiterte Funktionen in den Templates ermöglichen. So kann man z.B. Schleifen direkt in das Template integrieren, um die Generierung zu vereinfachen. Ein Beispiel hierfür sind Java Server Pages - kurz JSP [31, S. 125][51, S. 156ff].

Eine zweite Technik ist die Transformation. Dabei liest der Generator ein Modell (z.B. ein AST) und wandelt dieses in einzelne Statements um. Dieses Modell kann man vorab erstellen (z.B. als UML) oder im Code mithilfe von Methoden und Funktionen. Oft sind bei der Transformation auch kleine Templates vorhanden, die es erlauben, das Modell zu Code umzuwandeln [31, S. 126f][51, S. 161f][82, S. 149ff]. Listing 2.6 zeigt ein Beispiel für die Transformations-Technik.

```
1 my_class = Class(name="User")
2 my_attribute = Attribute(name="foo", default_value=None)
3 my_class.add_attribute(my_attribute)
4 my_class.generate_code()
```

Listing 2.6: Beispiel einer Code-Generierung per Transformation in Python

Es werden einzelne Instanzen von verschiedenen Klassen erstellt, die somit ein Modell des zu generierenden Codes abbilden. In Zeile 4 wird diese Struktur dann zu Code umgewandelt. Listing 2.7 zeigt eine beispielhafte Implementierung der verwendeten Klasse *Class* aus Zeile 1 in Listing 2.6. Mithilfe der Methode *add_attribute* speichert die Klasse *Attribute*. In Zeilen 9 bis 15 gibt die Methode *generate_code* den Code der zu generierenden Klasse zurück.

Zeile 10 enthält die Klassendefinition. Im Anschluss werden Attribute der zu generierenden Klasse untersucht. Wenn diese vorhanden sind, gibt die *Attribute*-Klasse den Code für einzelne Attribute zurück. Gibt es keine Attribute wird stattdessen ein *pass* verwendet. *pass* ist in Python ein Schlagwort für einen leeren Code-Block. Der generierte Code aus Listing 2.6 ist in Listing 2.8 dargestellt.

```

1  class Class:
2      def __init__(self, name):
3          self.name = name
4          self.atts = []
5
6      def add_attribute(attribute):
7          self.atts.append(attribute)
8
9      def generate_code():
10         code = 'class ' + self.name + ':\n    '
11         if self.atts:
12             code += '\n        '.join(att.generate_code() for att in self.atts)
13         else:
14             code += 'pass'
15
16         return code

```

Listing 2.7: Beispielhafte Implementierung der Generierung einer Klasse in Python

```

1  class User:
2      foo = None

```

Listing 2.8: Generierter Code aus Listing 2.6

Generierung über Templates und über Transformation haben verschiedene Anwendungsbereiche: Templates zählen sich besonders bei Generierung von Code mit überwiegend statischen Anteilen aus. Sind große Bereiche des Outputs dynamisch, werden Templates schnell unübersichtlich. Transformationen dagegen zählen sich bei großen Mengen an dynamischem Code aus. Sie sind flexibler und komplexer. Dies verursacht einen Overhead, wenn große Teile des Codes im Vorhinein klar sind [31, S. 124ff][82, S. 149, 151].

2.6 Natural Language Processing

Ziel dieser Arbeit ist es, die Übersetzung von Anforderungen zu Tests durch Verwendung natürlicher Sprache zu vereinfachen. Der Prototyp muss dabei nicht nur die Syntax von Gherkin verarbeiten können (s. Abschnitt 2.4), sondern auch die Bedeutung der in der Feature-Datei festgehaltenen Texte (s. Abbildung 2.13).

Dafür lassen sich Techniken und Methoden nutzen, die zum Bereich des Natural Language Processing (kurz NLP) gehören [48][102]. „Natural language“ (natürliche Sprache) steht dabei für die Sprache, die Menschen in der täglichen Kommunikation verwenden. NLP analysiert sie und wandelt sie in strukturelle, maschinenverwertbare Datenformate um. Typische Anwendungsbereiche sind Suchmaschinen, maschinelle Übersetzung von Texten oder Textvervollständigung [12, S. xi].

Dieser Abschnitt erläutert Techniken und Funktionen, mit deren Hilfe der Prototyp Inhalte einer solchen Datei nachvollziehen kann - unter anderem die Tokenizaton und die Lemmatization.

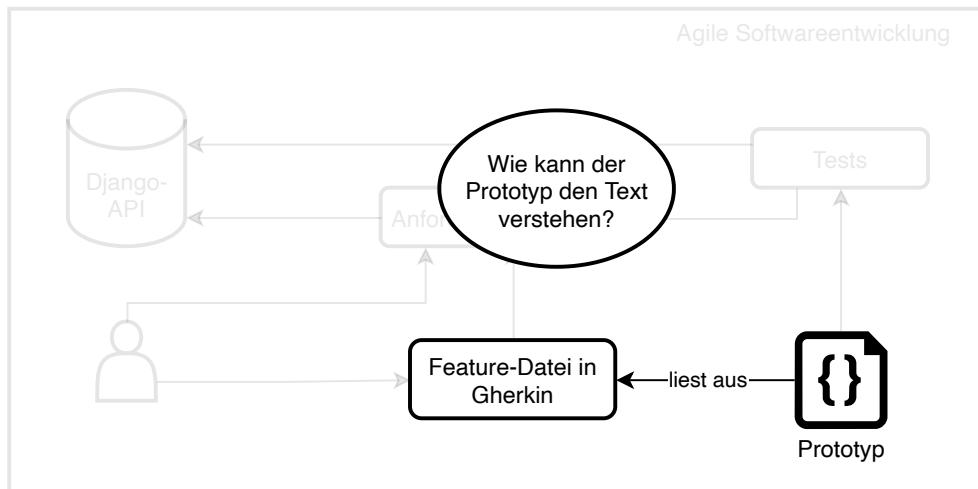


Abbildung 2.13: Abschnitt 2.6 im Ablauf der Generierung (eigene Darstellung)

2.6.1 Tokenization

NLP macht Tokenization erforderlich: Text (auch Corpus genannt) wird dabei in Tokens umgewandelt. Ein **Token** ist im Kontext von NLP der Name für eine Folge von

Charakteren, die als Gruppe behandelt werden. Von der Zielsetzung her ähneln sie Tokens, die bei der lexikalischen Analyse eines Compilers entstehen (s. Abschnitt 2.5.1). Beispiele für Tokens sind: „Hund“, „my-email@local.local“, „;“ oder „;:)“ [12, S. 7]. Für die Erstellung bzw. Identifikation von Tokens stellen Libraries verschiedene Methoden bereit [76].

Tokens erlauben die genauere Analyse einzelner Wörter. Sie wird durch ein trainiertes Model durchgeführt [76]. Trainierte Models reproduzieren Entscheidungsprozesse. Sie sind mathematische Algorithmen, die mithilfe großer Datenmengen trainiert wurden und sich immer wieder nutzen lassen [3]. Die Erstellung eines Models ist für diese Arbeit nicht relevant, da diese Thesis kein eigenes Model verwendet.

2.6.2 Stop Words

Einige Wörter kommen sehr oft in der natürlichen Sprache vor, obwohl sie nicht sonderlich viele zusätzliche Informationen liefern. Diese Wörter nennen sich **Stop Words** [59, S. 27]. Beispiele aus der deutschen Sprache sind „oft“, „doch“, „sehr“ oder „z.B.“ [83]. Wenn man diese Wörter aus Texten entfernt, bleibt die Bedeutung des Satzes in einer kompakten Version bestehen:

Sie spielen doch sehr oft Fußball. → *Sie spielen Fußball.*

2.6.3 Lemmatization

Um beim Analysieren von Texten zusammengehörige Worte gruppieren zu können, nutzt NLP die **Lemmatization** (deutsch Lemmatisierung). Den im Fließtext verwendeten gebeugten Formen eines Wortes wird dabei ihre Grundform zugeordnet. Aus *singt* wird also *singen*, aus *Männer* wird *Mann*. Welche Bedeutung das hat, zeigt sich am Beispiel der Suchmaschine. Lemmatization macht es hier möglich, Texte mit dem Begriff *Bäume* sinnvollerweise auch dann anzuzeigen, wenn in der Suchmaske das Wort *Baum* eingegeben wird. Lemmatization nutzt üblicherweise ein Lexikon und Informationen zur Gruppierung verschiedener Wörter. Eine der Lemmatization verwandte Methode ist das Stemming. Es ist für diese Arbeit nicht relevant [12, S. 107f][59, S. 32f].

2.6.4 POS Tagging

Das POS (Part-of-speech) Tagging stellt den Prozess der Kategorisierung einzelner Wörter bzw. Tokens dar. Dabei ist sowohl das Wort selber, als auch der Kontext wichtig [12, S. 179f]. Das Labeln der Worte geschieht über Tags, die es für verschiedene Sprachen gibt. Universelle Tag-Varianten funktionieren über mehrere Sprachen, sind aber recht ungenau. Spezifizierte Tags, die auf eine einzelne Sprache ausgerichtet sind, erlauben eine genauere Kategorisierung [64][96].

Tabelle 2.1 und Tabelle 2.2 zeigen Beispiele für solche Tags. Hier ist deutlich erkennbar, dass Tags aus Tabelle 2.2 detaillierter auf die Kategorisierung der Tokens eingehen. Tabelle 2.3 zeigt, welche Tags die einzelnen Wörter im Satz „Sie programmieren gerne Zuhause“ erhalten würden.

Tag	Beschreibung	Beispiel
ADV	Adverb	Sie gehen bald
NOUN	Nomen	der Abend
PRON	Pronomen	Sie gehen bald
PUNCT	Interpunktionszeichen	.
VERB	Verb	Sie gehen bald

Tabelle 2.1: Beispiele für universelle POS Tags [95]

Tag	Beschreibung	Beispiel
ADV	Adverb	Sie gehen bald
NN	Nomen (keine Adjektive, die als Nomen verwendet werden)	der Abend
PPER	Personalpronomen	Sie gehen bald
VMFIN	finites Modalverb	Sie sollte gehen
VVFIN	finites Vollverb	Sie gehen bald

Tabelle 2.2: Beispiele für POS Tags für Deutsch [85]

Wörter	Sie	programmieren	gerne	Zuhause
Universelle Tags	PRON	VERB	ADV	NOUN
Tags für Deutsch	PPER	VVFIN	ADV	NN

Tabelle 2.3: Tagging an einem deutschen Satz

2.6.5 Dependencies

Neben der Bedeutung einzelner Tokens sind Satzstrukturen bzw. Abhängigkeiten zwischen Tokens interessant. Um diese abzubilden, gibt es zwei Hauptmethoden: **Constituency** und **Dependency** Parsing. Constituency Parsing analysiert Texte, indem die Grammatik der jeweiligen Sprache kontextfrei definiert wird (s. Abschnitt 2.4). Dependency Parsing hingegen konzentriert sich mehr auf Abhängigkeiten und Interaktionen verschiedener Wörter [25][52]. Für diese Arbeit ist nur Dependency Parsing relevant.

Texte, in denen jeder Satz syntaktische Informationen erhält, nennt man auch **Treebank**. Das Erstellen einer solchen Treebank kann manuell erfolgen oder mithilfe der genannten Methoden durch einen Parser. Bei der Erstellung einer Treebank werden häufig bereits mit POS Tags versehene Texte verwendet [61].

Um Abhängigkeiten darzustellen, startet das Dependency Parsing beim **head**. Dabei handelt es sich meistens um ein Verb. Es wird auch **root** genannt, weil sich von ihm alle **dependants** (auch **children** genannt) ableiten. Jede Beziehung bzw. Abhängigkeit zwischen ihnen erhält eine Bezeichnung, die einer Kategorie zugeordnet ist [19]. Abbildung 2.14 zeigt die Abhängigkeiten in einem einfachen Satz.

In der Darstellung ist *Sie* das Subjekt von *programmieren*. *mo* steht für modifier. Modifier verändern die Bedeutung anderer Wörter [99]. *oa* steht für die Verwendung eines Objekts im Akkusativ. Da *programmiert* der head aller Tokens ist, ist es das root. Einige Libraries bieten Funktionen, die die Bezeichnungen näher erläutern [91].

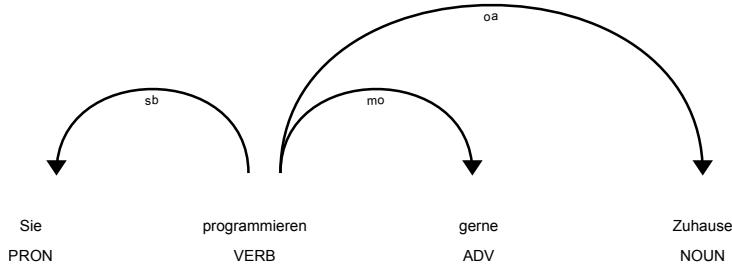


Abbildung 2.14: Dependencies eines Satzes (generiert von Displacy [98])

2.6.6 Similarity

Für diese Thesis ist das Vergleichen von Wörtern bzw. kurzen Textabschnitten essentiell. Im Kern geht es um die Frage: *Wann sind zwei Wörter bzw. mehrere Wörter ähnlich?* Diese Ähnlichkeit heißt auch **Similarity**. Dabei ist vor allem die semantische Ähnlichkeit wichtig - also die Bedeutung von Wörtern und Textabschnitten [12, S. 71f]. Im NLP gibt es verschiedene mathematische Methoden zur Bestimmung der Similarity [41][45, S. 2ff]. Davon werden hier zwei näher erläutert.

Eine der beiden Methoden ist das Repräsentieren von Wörtern mithilfe eines Vektors. Als Vektoren für die Wörter *brown*, *lightseagreen* und *orange* eignen sich beispielsweise die RGB-Werte, die den Rot-, Grün- und Blauanteil einer Farbe beschreiben. Die drei Farben im RGB-Schema sind 165, 42, 42 (*brown*), 32, 178, 170 (*lightseagreen*) und 255, 165, 0 (*orange*). Mittels ihrer Vektoren ließen sich die drei Wörter dann in einem dreidimensionalen Raum abbilden (s. Abbildung 2.15) und mit anderen Farben vergleichen, um Ähnlichkeiten zu finden. Dies ist ein einfaches Beispiel, das zur Veranschaulichung dient.

Echte Vektoren sind komplexer und vielschichtiger. Durch die Darstellung kann man auf Wörter mathematische Operationen anwenden, etwa Addieren oder den Durchschnitt nehmen. Betrachtet man das Beispiel der Farben, könnte man mithilfe der Vektoren das Ergebnis von *Blau + Rot* berechnen. Beispiele für Methoden, die Vektoren verwenden, sind die **Cosine Distance**, die den Winkel der Wörter-Vektoren vergleicht, und die **Euclidean Distance**, bei der der Abstand der Wörter-Vektoren zueinander entscheidend ist [41].

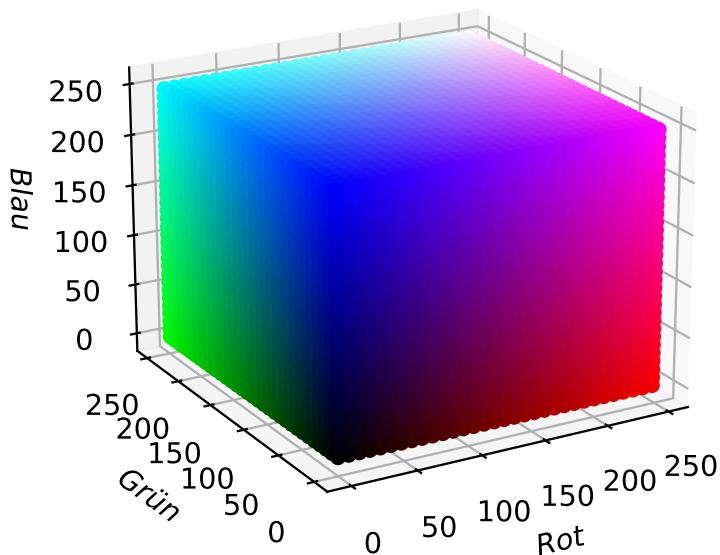


Abbildung 2.15: Farben in 3-dimensionalen Raum (eigene Darstellung, erstellt mit Matplotlib [44])

Andere Methoden kommen ohne Vektoren aus und vergleichen stattdessen die Worte selbst. Beim Vergleich von *Ente* und dem falsch geschriebenen *Etne* beispielsweise berechnet die **Levenshtein Distance** die Anzahl der Operationen (Einfügen, Ändern oder Löschen von Buchstaben), die nötig sind, um von Wort A zu Wort B zu kommen. In dem genannten Beispiel ist die Anpassung von zwei Buchstaben notwendig. Die Levenshtein Distance findet oft bei Suchen Verwendung, da man mit ihrer Hilfe Tippfehler ausgleichen kann. Es geht hier also nicht um die Kontrolle der semantischen Bedeutung [41][59, S. 58].

2.7 Django

Der Prototyp soll die Tests einer Django-API generieren (s. Abbildung 2.16). Aus diesem Grund erläutert dieser Abschnitt einige Begriffe, die im Kontext des Python Web-Frameworks gebräuchlich sind.

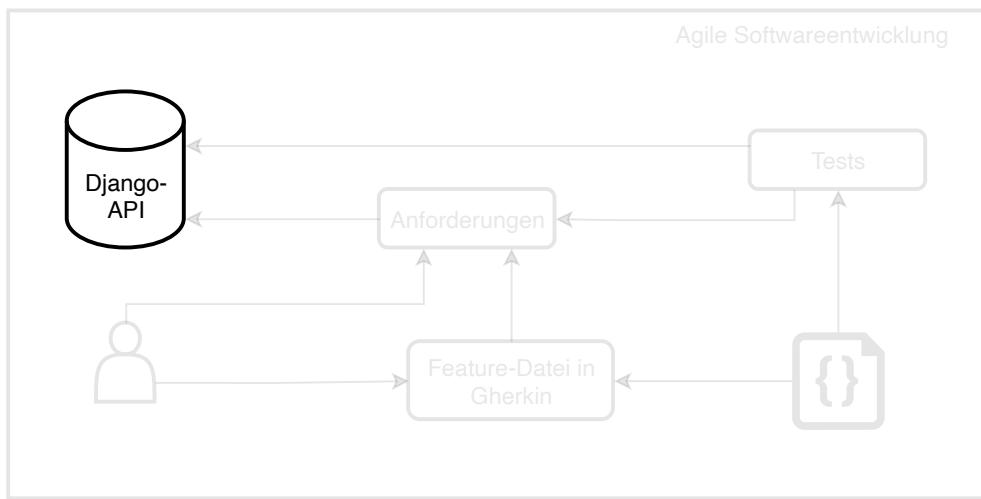


Abbildung 2.16: Abschnitt 2.7 im Ablauf der Generierung (eigene Darstellung)

2.7.1 Apps

Django-Applikationen sind in verschiedene Teil-/ Themenbereiche aufgeteilt. Diese nennen sich **Apps**. Sie enthalten zusammenhängende Features und können normalerweise von verschiedenen Projekten wiederverwendet werden [5].

2.7.2 MVT

Innerhalb jeder App lässt sich das **MVT** (oder auch Model-View-Template) Design Pattern finden. Gegeben sei folgendes Beispiel: eine Person ruft die Webseite auf. Die Django-Applikation muss nun Inhalte über eine URL wiedergeben. Entwickelnde können diese URLs definieren und sie **Views** zuweisen. Diese Views verarbeiten Anfragen und nutzen das **Model**, um entsprechende Daten aus der Datenbank zu holen. Jedes Model steht für eine Tabelle in der Datenbank und hat Felder, die die Spalten der Tabelle repräsentieren. Die Daten aus der Datenbank kann man in Django mit sogenannten **Querysets** abgreifen, in denen sich Model-Instanzen befinden. Anschließend befüllt der View die **Templates** (z.B. HTML-Dateien). Das befüllte Template können benutzende Personen nun im Browser sehen. Es übernimmt eine ähnliche Rolle wie Templates bei der Code-Generierung (s. Abschnitt 2.5.4) [14].

2.7.3 API

Traditionell hat Django keine API-Schnittstelle. Allerdings gibt es eine sehr verbreitete Library hierfür: das Django REST Framework. Die wichtigsten Begriffe im Kontext dieser Library sind **Serializer** und **Viewset**. Ein Viewset ähnelt einer View-Klasse, nimmt allerdings speziell API-Anfragen entgegen. Es ist eine Sammlung von einzelnen Schnittstellen, die zueinander passen. Ein einfaches Beispiel hierfür sind die CRUD-Methoden (create, read, update, delete) für ein Model. Neben dem Viewset gibt es für jeden Endpunkt einen Serializer. Er garantiert, dass die Daten im Request ein valides Format haben. Gleichzeitig wird er verwendet, um das Format für die Antwort festzulegen [42].

Wenn also ein Viewset eine Anfrage bearbeitet, nutzt es zunächst den Serializer, um die Daten zu validieren. Anschließend werden Daten aus dem Model geholt. Zuletzt gibt das Viewset diese Daten an den Serializer, der die Daten in das vorgegebene Format bringt. Im Anschluss gibt es die formatierten Daten als Response zurück [42].

3. Anforderungsanalyse

Dieses Kapitel beschreibt detailliert, welche Anforderungen der im Rahmen dieser Arbeit implementierte Prototyp erfüllen sollte, um die in Abschnitt 1.4 gesteckten Ziele zu erreichen. Die Anforderungen sind in verschiedene Kategorien eingeteilt. Der Prototyp hat den Namen **Ghengo** (IPA: ['geŋgou]). Der Name entsteht durch eine Kombination der Wörter *Django*, *Gherkin* und *Generierung*.

Zunächst listet dieser Abschnitt einige allgemein an Ghengo gestellte Anforderungen auf. Die restlichen Anforderungen orientieren sich an den verschiedenen Schritten des Compilingprozesses: dem Auslesen, der Analyse und der Generierung.

3.1 Allgemein

Kategorie 1 (K1) fasst die Ansprüche zusammen, die sich an den gesamten Prototyp richten.

K1-1 Erweiterbarkeit

Da der Prototyp auf Django spezialisiert ist, soll er Code für Python generieren, der für ein einzelnes, ausgewähltes Python Testframework ausführbar sein soll. Ein wesentliches Ziel dabei ist es, den Prototyp so zu bauen, dass er sich in der Zukunft erweitern lässt. Der Prototyp soll eine Architektur haben, in der sich vor der Generierung Zielsprache und Ziel-Testframework auswählen lassen.

K1-2 Library

Der Prototyp muss unabhängig von einem konkreten Django-Projekt entwi-

ckelt sein. Er soll als eine Art Library fungieren, die man in beliebige Django-Projekte einbinden kann.

K1-3 Sprachen

Gherkin unterstützt verschiedene natürliche Sprachen (s. Abschnitt 2.2.10). Der Prototyp soll in der in dieser Thesis entwickelten Version Deutsch und Englisch unterstützen. Da die Auftraggebenden von Ambient zum Großteil Deutsch sprechen und es für sie angenehm ist, Anforderungen in ihrer Muttersprache zu definieren, liegt der Fokus dabei auf Deutsch. Der Prototyp muss also Englisch als Input akzeptieren, aber es ist akzeptabel, wenn er die Daten nicht hundertprozentig korrekt verarbeitet.

K1-4 Performance

Der Prototyp muss eine gute Performance haben und Entwickelnden erlauben, schnell und effizient damit zu arbeiten. Die Generierung eines Testfalls sollte nicht länger als 5 Sekunden beanspruchen. So lange hat die Generierung von Testfällen in vorherigen Arbeiten gedauert [48, S. 58]. Eine optimierte Version von Kirby kann Tests innerhalb von 1 Sekunde generieren [48, S. 59].

3.2 Auslesen von Gherkin

Kategorie 2 (K2) enthält die Ansprüche, die das Auslesen der Feature-Dateien bzw. Gherkin betreffen.

K2-1 Gherkin

Der Prototyp soll die Gherkin-Versionen 6 und höher unterstützen.

K2-2 Fehlerbehandlung

Für Entwickelnde und Auftraggebende, die nicht mit Gherkin vertraut sind, soll es detaillierte Meldungen geben, wenn die Syntax des Inputs inkorrekt ist. Die Fehlermeldung muss den Ort des Fehlers und mögliche Verbesserungen enthalten.

3.3 Vorbereitung der Generierung

In Kategorie 3 (K3) geht es um den Schritt zwischen Auslesen von Gherkin und Ausgabe der Tests. Der Prototyp muss den Text analysieren und die Generierung des Codes vorbereiten.

K3-1 Unterstützte Aktionen

Beim Schreiben von Tests für eine Django-API gibt es verschiedene **Aktionen**, die immer wieder Verwendung finden. Dazu zählen...

- ...das Erstellen von Model-Instanzen mit ihren Relationen in der Datenbank (OneToMany, ManyToMany, OneToOne).
- ...die Instanziierung von Klassen, die bei der Erstellung von Models relevant sein könnten (z.B. eine Datei, die eine benutzende Person hochgeladen hat). Ein Beispiel dafür wäre ein Profilbild, das einem Model zugeordnet ist.
- ...Anfragen bzw. Requests, die man an die API sendet.
- ...die Analyse der Datenbank.
- ...die Analyse der Response nach einem Request. Jede Response beinhaltet einen Statuscode (z.B. 200 steht für „Hat alles geklappt“), Daten und Informationen zu eventuellen Fehlern. Ghengo sollte all diese Dinge kontrollieren können.
- ...die Analyse zuvor erstellter Model-Instanzen.

Ghengo sollte mit diesen Aktionen arbeiten können. Dazu muss der Prototyp anhand des Textes erkennen, welche dieser Aktionen am besten passt. Der Satz „Gegeben sei ein Auftrag“ würde auf das Erstellen einer Model-Instanz hindeuten. „Dann sollte die Antwort den Status 200 haben“ würde zur Analyse einer Response führen.

K3-2 Analyse Django-Projekt

Der Prototyp sollte ein bestehendes Django-Projekt analysieren und alle Informationen abgreifen können, die zur Erstellung der Tests notwendig sind.

K3-3 Referenzierten Code erkennen

Input kann die bestehende Code-Basis referenzieren. Ghengo sollte in der Lage sein, die Referenzen zu erkennen und daraus Konsequenzen für den resultierenden Code ableiten zu können (s. dafür auch K3-2). Beispiel: Gegeben sei ein Model *Order* mit einem Feld *number*. Erhält der Prototyp nun den Text „Gegeben sei ein Auftrag mit der Nummer 1“, dann muss er erkennen, dass damit das Order-Model und dessen Feld *number* gemeint ist.

K3-4 Daten extrahieren

Aufgabe des Prototyps ist es auch, Daten aus einem Text zu extrahieren. Nimmt man das Beispiel aus K3-3, beschreibt die Zahl 1 den Wert für das Feld *number*. Ghengo muss diesen Wert und den korrekten Datentyp zuverlässig erkennen. Für die Analyse des Textes soll der Prototyp NLP verwenden.

K3-5 Unterstützung von Test-First

Der Prototyp soll in einem agilen Umfeld eingesetzt werden, in dem die Test-First Strategie präferiert wird. Man kann sich hier nicht darauf verlassen, dass referenzierte Code schon existiert (s. den Workflow in Abschnitt 2.2.7). Ghengo soll auch dann funktionieren, wenn die Implementierung zu einem Szenario bzw. Feature noch fehlt. Beispiel: Bei einem Step „Gegeben sei ein Auftrag“, sollte der Prototyp erkennen, dass *Auftrag* ein Model referenziert, auch wenn dieses noch nicht existiert.

K3-6 Übersetzung

Bei Ambient ist die übliche Sprache für eine Code-Basis Englisch. Folglich muss Ghengo Texte übersetzen können, um Referenzen zu verstehen.

3.4 Generierter Code

Kategorie 4 (K4) befasst sich mit dem Code, den der Prototyp generieren soll. Sie definiert die Anforderungen an die generierten Tests.

K4-1 Dynamischer Output

Der Code-Output sollte dynamisch sein, da durch die Verwendung natürlicher Sprache damit zu rechnen ist, dass sich Inhalte stark unterscheiden. Das hat zur Folge, dass der zu generierende Code flexibel sein muss.

K4-2 Lesbarkeit

Der zu generierende Code von Ghengo sollte lesbar sein, damit Entwickelnde im Anschluss Änderungen vornehmen können. Dies ist praktisch in all den Fällen, in denen der Prototyp einen Wert falsch erkannt hat oder es einen Fehler gab.

K4-3 Validität

Der zu generierende Code muss valide sein. Das bedeutet nicht, dass der generierte Test durchläuft. Es bedeutet, dass das Testframework und der Python Compiler den Code ausführen können, ohne dass es zu Fehlern kommt.

K4-4 Error Handling

Wenn Ghengo Problem beim Extrahieren von Daten hat (s. Anforderung K3-4), dann sollte im generierten Code ein Hinweis stehen. Der Entwickelnde kann im Anschluss die jeweilige Stelle anpassen bzw. ergänzen.

K4-5 Korrekte Umwandlung

Ghengo sollte Test-Code ausgeben, der auch das kontrolliert, was in Gherkin beschrieben wurde.

4. Konzeptionierung

In diesem Kapitel werden Konzepte vorgestellt, die es ermöglichen, die in Kapitel 3 formulierten Anforderungen zu erfüllen. Es geht also um Fragen der praktischen Umsetzung und die Architektur des Prototyps.

Ghengo erhält die Anforderungen an die Django-API in einer Feature-Datei und muss diese Anforderungen in Tests übersetzen. Dies ist mit einem Compiler möglich, der an dieser Stelle sinnvollerweise eingesetzt werden soll. Das Compiling in Ghengo wird also, wie in Abschnitt 2.5 beschrieben, aus der lexikalischen Analyse durch einen Lexer, der syntaktischen Analyse durch einen Parser und der Code-Generierung bestehen (s. Abbildung 4.1). Type Checking ist nicht erforderlich, da es sich bei Gherkin um eine Sprache ohne Typisierung handelt [36].



Abbildung 4.1: Schritte des Ghengo-Compilers (eigene Darstellung)

Bevor konzeptionelle Entscheidungen zu den einzelnen Schritten des Compilings näher erläutert werden, ist festzulegen, für welches Test-Framework der zu generierende Code ausführbar sein soll.

4.1 Test-Framework

Ein weit verbreitetes Test-Framework für Python ist die Library PyTest [55]. Der Code-Output des Prototyps soll hierfür ausführbar sein, da PyTest und Gherkin einige ähnliche Funktionen anbieten. Ein Beispiel hierfür sind Gherkin-Tags. Sie geben den Entwickelnden die Möglichkeit, nur bestimmte Teile einer Feature-Datei auszuführen. In PyTest gibt es einen Decorator, der das Gleiche macht. In Listing 4.1 und Listing 4.2 ist die Ähnlichkeit deutlich zu erkennen. Auch für Datentabellen bieten Gherkin und PyTest einander ähnelnde Funktionen.

```
1  @pytest.mark.foo
2  def test_only_sometimes():
3      pass
```

Listing 4.1: Pytest Testfall mit Decorator

```
1  # language: de
2  Funktionalität:
3  @foo
4  Szenario:
5      Gegeben sei ein Auftrag 1
```

Listing 4.2: Tag in Feature-Datei

Abbildung 4.2 zeigt den aktualisierten Ablauf des Compilers nach der Wahl des Test-Frameworks.

Für die Erstellung von Models innerhalb von Tests bietet sich die Library *pytest-factoryboy* an [27]. Es wird an dieser Stelle vorausgesetzt, dass diese Library bei der späteren Arbeit mit Ghengo zur Verfügung steht.



Abbildung 4.2: Schritte des Ghengo-Compilers nach Wahl des Test-Frameworks (eigene Darstellung)

4.2 Lexer

Im ersten Schritt des Compilings muss der Lexer des Prototyps Gherkin-Text aus einer Feature-Datei in Tokens umwandeln. Da in Gherkin jede Textzeile typischerweise mit einem Keyword beginnt, bietet es sich an, die für die Umwandlung erforderliche lexikalische Analyse zeilenbasiert durchzuführen [36].

Der Lexer erstellt dabei Tokens, die Schlagwörter bzw. Patterns zugeordnet bekommen. Die Tokens geben auch an, welche Abschnitte des Textes zu ihnen gehören. Diese Abschnitte heißen auch Lexemes (s. Abschnitt 2.5.1).

Tabelle 4.1 zeigt alle Tokens, die Schlagwörter aus Gherkin repräsentieren. Ihre Bedeutungen sind in der Gherkin Dokumentation beschrieben [36].

Ghengo unterstützt Deutsch und Englisch als Input (s. Anforderung K1-3). Die Patterns für die einzelnen Tokens unterscheiden sich je nach gewählter Sprache. Würde im Englischen der Begriff „Scenario“ verwendet, wäre die Entsprechung in deutscher Sprache „Szenario“. Tokens müssen auf beides gleichermaßen korrekt reagieren.

Neben Tokens für Keywords aus Gherkin verwendet Ghengo Tokens, die keine expliziten Schlagwörter sind:

- **Empty** - eine leere Zeile
- **Description** - Freitext
- **EndOfLine** (kurz EOL) - das Ende einer Zeile
- **EndOfFile** (kurz EOF) - das Ende der Datei

Der Ablauf der lexikalischen Analyse soll anhand der Feature-Datei aus Listing 4.3 dargestellt werden.

Tag	Feature	Background	Examples
ScenarioOutline	Scenario	And	But
Given	When	Then	DataTable
DocString	Language	Comment	Rule
Language			

Tabelle 4.1: Tokens für Keywords aus Gherkin

Zunächst betrachtet der Lexer Zeile 1 bzw. den Text „# language: de“ und überprüft, welcher Token zu diesem Text passt. In diesem Fall ist es der *Language*-Token, dem der komplette Text dieser Zeile als Lexeme zugeordnet werden kann. Dementsprechend erstellt der Lexer den Token. In der ersten Zeile hat er somit den gesamten Text bearbeitet. Um das Ende der Zeile zu markieren, fügt er einen *EOL*-Token hinzu.

```

1 # language: de
2 Funktionalität:
3 Szenario:
4 Gegeben sei ein Auftrag 1

```

Listing 4.3: Beispiel einer deutschen Feature-Datei

Diesen Prozess wiederholt der Lexer für die Zeilen 2 und 3. Durch den *Language*-Token kennt er die gewählte Sprache und entdeckt die deutschen Schlagwörter *Funktionalität* und *Szenario*. Aus den beiden Zeilen resultieren die Tokens *Feature*, *EOL*, *Scenario* und *EOL*.

In Zeile 4 erkennt der Lexer zunächst einen *Given*-Token. Das Lexeme des Tokens ist allerdings nur „Gegeben sei“. Im Anschluss findet sich Freitext, dem noch kein Token zugeordnet wurde. Dementsprechend entstehen aus der Zeile die Tokens *Given*, *Description* und *EOL*.

Die lexikalische Analyse ist beendet. Um das Ende der Datei anzuzeigen, erstellt der Lexer noch ein *EOF*-Token. Abbildung 4.3 zeigt alle entstandenen Tokens mit ihren Lexemes. Der Lexer erstellt aus ihnen eine Liste, die weiterverwendet werden kann.

Language <i># language: de</i>	EOL
Feature <i>Funktionalität:</i>	EOL
Scenario <i>Szenario:</i>	EOL
Given <i>Gegeben sei</i>	Description <i>ein Auftrag 1</i>
	EOL
	EOF

Abbildung 4.3: Tokens mit Lexemes für Listing 4.3 (eigene Darstellung)

4.3 Parser

Im nächsten Schritt muss der Parser des Prototyps die syntaktische Analyse durchführen. Er erhält dafür die im Verlauf der lexikalischen Analyse entstandenen Tokens (s. Abschnitt 4.2), prüft deren Syntax und erzeugt einen AST. Grundlage für die Prüfung ist die für Gherkin geltende kontextfreie Grammatik mit den Produktionsregeln A.1 [36]. Da auf der linken Seite dieser Regeln in jedem Fall ein einzelnes Nichtterminalsymbol steht und Doppeldeutigkeit sowie Linksrekursion auszuschließen sind, wird Ghengo einen Top-Down Parser verwenden.

Einen solchen Gherkin analysierenden Parser gibt es bereits für Python [35]. Da dieser sich in ersten Versuchen nicht bewährt hat, verwendet Ghengo jedoch einen eigenen. Die Wahl fiel hierbei auf einen rekursiven Parser, da dieser leichter zu implementieren ist und eine bessere Fehlerlokalisierung ermöglicht als ein tabellenbasierter Parser (s. Anforderung K2-2). Das rekursive Ablaufen des Syntaxbaums ist in Abschnitt 2.5.2 detailliert erläutert.

Um den eigenen Parser implementieren zu können, braucht Ghengo Klassen für die **Operatoren** der Regeln, die **Nichtterminale** und die **Terminale**.

Für **Operatoren** werden folgende Klassen gebraucht:

- **Optional** entspricht [] in EBNF. Die Klasse steht für ein optionales Symbol.
- **Repeatable** entspricht { } in EBNF. Kindes-Symbole der Klasse können wiederholt auftreten.

- **OneOf** entspricht | in EBNF. Die Klasse steht für das logische Oder.
- **Chain** gibt an, welche Symbole aufeinander folgen müssen.

Die Klasse für **Nichtterminale** gibt vor, womit man sie ersetzen kann. **Terminale** sind Wrapper um die Tokens, die in Tabelle 4.1 zu sehen sind. Mit den genannten Klassen kann die Grammatik von Gherkin im Code abgebildet und die in Abschnitt 2.5.2 beschriebene, rekursiv-syntaktische Analyse durchgeführt werden.

Nachdem die Analyse abgeschlossen ist, steht ein AST zur Verfügung. Abbildung A.3 zeigt das zugehörige Klassendiagramm. Tabelle 4.2 enthält die wichtigsten Informationen und hält fest, wofür sie später verwendet werden.

Datensatz aus Parser-AST	Wird gebraucht für...
Name des Features	Name der Test Datei bzw. Suite
Name jeder ScenarioDefinition	Name für die Testfälle
Texte der Steps	Generierung der Statements
Datentabellen	Generierung der Statements

Tabelle 4.2: Daten aus Parser-AST und deren Verwendungszweck für Generator

Ghengo muss jede ScenarioDefinition aus dem AST durchlaufen. Für jede Definition entsteht ein Test Case, dessen Name vom Namen der ScenarioDefinition abhängig ist. Den Inhalt bzw. die Statements der Tests entstehen mithilfe der Steps. Alle Klassen aus Abbildung A.3 müssen verschiedene Funktionen zur Verfügung stellen, um an diese Steps zu kommen. Ein Feature ist durch eine Test Suite repräsentiert.

4.4 Code-Generierung

Der Code-Generator übernimmt den größten und kompliziersten Teil der Übersetzung von Gherkin zu Test-Fällen. Er erhält den AST des Parsers (ab hier **Parser-AST** genannt), muss aus ihm Informationen entnehmen und daraus Testfälle für PyTest erstellen (s. Abbildung 4.4).

Der Parser-AST gibt dabei die Struktur der Tests vor, z.B. aus einem Feature entsteht eine Test-Suite und aus einem Scenario bildet sich ein Test-Case. Außerdem enthalten

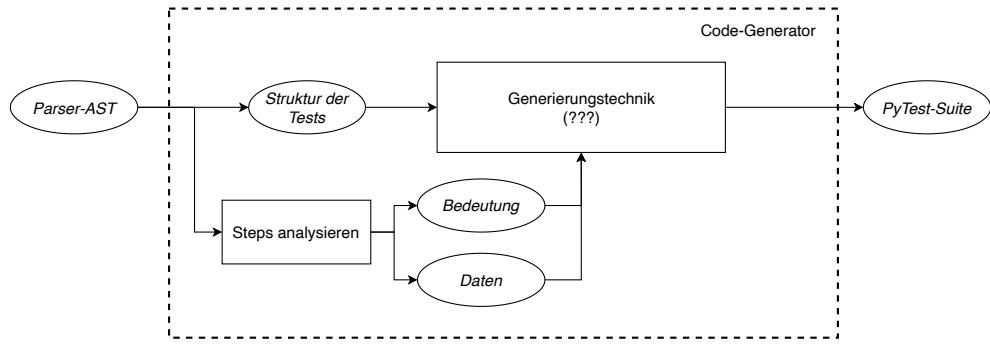


Abbildung 4.4: Ablauf der Codegenerierung mit Ghengos Generator (eigene Darstellung)

die Steps im Parser-AST Informationen, die Ghengo analysieren muss. Diese Analyse ist in zwei Bereiche unterteilt:

- **Analyse der Bedeutung:** Was bedeutet eine Formulierung? Welche der in Anforderung K3-1 definierten Aktionen ist gemeint?
- **Analyse der Daten:** Welche Daten sind im Text enthalten?

Die Struktur der Tests, die Bedeutung und die Daten werden mithilfe einer Generierungstechnik in Suites übersetzt. Da die Art der gewählten Generierungstechnik Auswirkungen auf alle anderen an dieser Stelle genannten Schritte hat, ist zunächst die Frage zu klären, welche Technik Ghengos Generator verwenden soll. Die auf 4.4.1 folgenden Abschnitte behandeln dann die einzelnen Schritte der Codegenerierung ihrer logischen Abfolge entsprechend. Konzeptionelle Entscheidungen werden dabei erläutert und begründet.

4.4.1 Generierungstechnik auswählen

Der Code-Generator erhält den Parser-AST und muss ihn zu Code umwandeln. Da der zu generierende Code sehr dynamisch (s. Anforderung K4-1) sein soll, verwendet Ghengo die Generierungstechnik der Transformation. Der Generator wandelt den AST des Parsers dabei um. Es entsteht ein zweiter Baum, der den zu generierenden Code repräsentiert. Um die beiden Bäume zu unterscheiden, wird der zweite Baum von nun an als **Output-AST** bezeichnet.

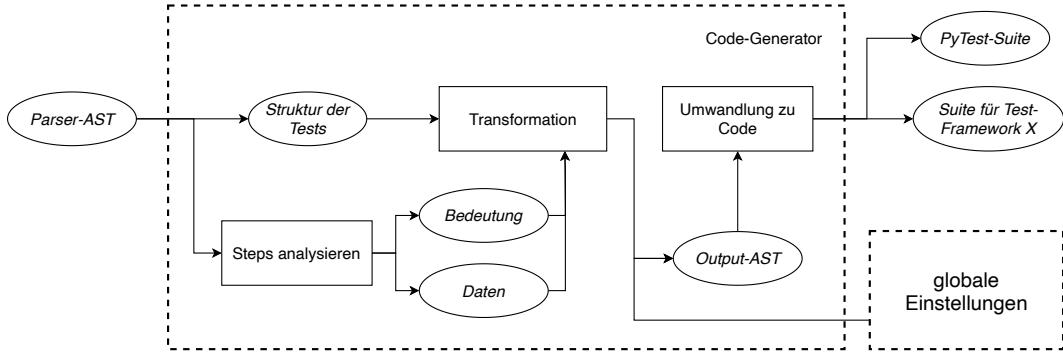


Abbildung 4.5: Ablauf der Code-Generierung mit Ghengo nach Wahl der Generierungstechnik (eigene Darstellung)

Für die Umwandlung verwendet Ghengo die in der Einleitung erwähnten Informationen. Im letzten Schritt erfolgen die Umwandlung zu Code und die Ausgabe der Suite unter Berücksichtigung der globalen Einstellungen (s. Abbildung 4.5).

4.4.2 Bestimmung der Bedeutung

Damit Ghengo die Test Cases bzw. den Output-AST befüllen kann, soll sein Generator die Bedeutung der Texte aus den Steps des Parser-ASTs entnehmen. Es geht hier um die Frage, was die benutzende Person mit einer Formulierung meint. Eine erste Kategorisierung wird erforderlich.

Die Art eines Steps gibt dem Generator hierfür schon einige nützliche Informationen. Zur Erinnerung: *Given* steht für die Überführung des Systems in einen gewünschten Zustand, *When* repräsentiert eine Aktion der nutzenden Person und *Then* markiert die Kontrolle des Systems. Tabelle 4.3 zeigt, welche Steps zu den unterstützten Aktionen in Django aus Anforderung K3-1 passen.

Für jede Step-Art erstellt der Generator zugehörige Tiler-Instanzen (z.B. *GivenTiler* oder *WhenTiler*). Tiler wenden auf ihren Step NLP mithilfe der Library Spacy an, die Models in verschiedenen Sprachen enthält [43]. Ghengo verwendet die Models *de_core_news_lg* (Deutsch) und *en_core_web_lg* (Englisch) [26][33]. Es handelt sich um große Models, die vergleichsweise lange laden müssen, aber dafür sehr präzise arbeiten. Sie enthalten auch Vektoren, die zur Bestimmung der Similarity notwendig sind.

Unterstützte Aktion in Ghengo	Step-Art
Erstellen von Model-Instanzen	Given
Instanziierung von Klassen, die für Models relevant sind	Given
Senden von Requests an die API	When
Analyse der Datenbank	Then
Analyse der Antwort nach einer Anfrage	Then
Analyse zuvor erstellter Model-Instanzen	Then

Tabelle 4.3: Zuordnung unterstützter Aktionen zu Step-Arten

Jedem Tiler sind verschiedene Converter-Klassen zugeordnet. Jeder Converter repräsentiert eine Aktion und kann einen Step zu Statements für diese Aktion konvertieren. Tiler suchen den passendsten Converter und entnehmen ihm Statements, die der Code-Generator an den Test Case hängt. Nachdem der Prototyp alle Steps eines Szenarios durchlaufen hat, ordnet er den Test Case der Suite zu. Abbildung A.2 zeigt die Abhängigkeiten der Klassen.

4.4.3 Daten extrahieren

Die in Abschnitt 4.4.2 beschriebene Kategorisierung hat geklärt, welche Aktion die nutzende Person meint. Der zugehörige Converter muss jetzt Daten aus dem Text extrahieren und diese in Statements einbetten. Dabei fließen Informationen über ein bestehendes Django-Projekt ein. Ghengo erhält diese Informationen durch die Angabe eines Pfades zu einem Projekt und erfüllt damit Anforderung K1-2.

Ghengo nutzt zwei Arten von Daten: Namen und Werte. Mithilfe von Namen erstellt Ghengo Referenzen auf bestehende oder geplante Implementierungen in Django (z.B. Models). Werte fließen direkt in die entstehenden Tests ein. Diese Werte sind meistens in Datentypen der zu generierenden Programmiersprache festgehalten, in dieser Arbeit also in Python. Beispiele hierfür sind Boolean, Integer oder String. Im Folgenden werden sie **native Werte** genannt.

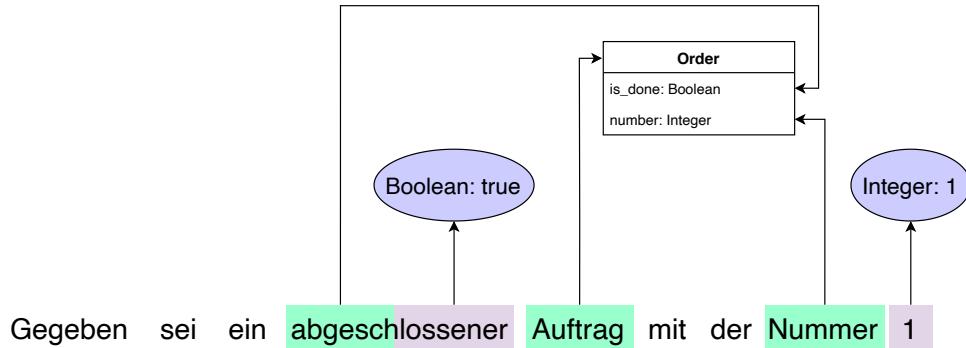


Abbildung 4.6: Daten in einem Beispielsatz (eigene Darstellung)

Abbildung 4.6 zeigt die Daten, die Ghengo aus dem Satz „Gegeben sei ein abgeschlossener Auftrag mit der Nummer 1“ zieht. Die Wörter *abgeschlossener*, *Auftrag* und *Nummer* sind Namen. *Auftrag* referenziert das Model *Order* und die Wörter *abgeschlossener* und *Nummer* die Felder *is_done* und *number*. Das Wort *abgeschlossener* impliziert einen Wert - den Boolean *True*. Die Zahl *1* steht für einen Integer.

An diesem Beispiel lässt sich erkennen: Tokens können Namen und/ oder Werte repräsentieren. Converter funktionieren immer nach dem gleichen Grundprinzip: Sie durchlaufen alle von NLP erstellten Tokens und analysieren diese. Converter suchen nach Tokens, die für eine Aktion Daten enthalten. Jeder Converter weiß durch seine Aktion, welche Daten notwendig und welche optional sind.

Extrahieren von Namen

Zunächst versucht der Converter Namen im Text zu finden. Soeken, Wille und Drechsler werten Nomen als Referenzen zu Klassen, Adjektive als Referenzen zu Attributen und Verben als Referenzen zu Methoden (s. Abschnitt 1.2) [77]. Da Ghengo auf Django spezialisiert ist und Converter auf einzelne Aktionen, lässt sich ein leicht abgeänderter Ansatz verwenden. Beim Identifizieren interessanter Tokens für Namen passiert folgendes:

1. Der Converter entfernt alle Tokens, die Stop Words sind. Sie enthalten keine nützlichen Informationen.
2. Der erste Token ist immer ein Keyword von Gherkin (Given, When etc.) und wird immer ignoriert.

3. Es gibt „blockierte“ Tokens. Sie beschreiben normalerweise verpflichtete Daten, die Ghengo an anderen Stellen nutzt. Im genannten Beispiel wäre das Wort *Auftrag* blockiert, da es schon das Model beschreibt.
4. Converter suchen (mit einigen Ausnahmen) nach Nomen, Adjektiven, Verben, Eigennamen und Adverbien. Sie alle beschreiben Namen, die Ghengo weiterverarbeitet. Bei dem Satz „Gegeben sei ein abgeschlossener Auftrag, der aus einem anderen System kommt“ untersucht Ghengo die Wörter *abgeschlossener* (Adjektiv) und *System* (Nomen). Obwohl *anderen* ein Adjektiv ist, ignoriert der Converter es, da es sich um ein Stop Word handelt. Bei der leicht abgeänderten Formulierung „Gegeben sei ein Auftrag, der abgeschlossen ist und aus einem anderen System kommt“ erkennt Ghengo das Wort *abgeschlossen* als Verb. Dementsprechend zählt es auch hier als Feld. Die Information über die Wort-Arten erhält Ghengo durch POS Tagging von Spacy.

Im Anschluss verfügt der Converter über eine Liste aus Tokens, die Namen repräsentieren. Ghengo versucht jetzt, die Referenzen zu finden. Beispiel: *Auftrag* soll dem Model *Order* und *Nummer* dem Model-Feld *number* zugeordnet werden. Dazu verwendet Ghengo zwei Klassen: **Lookout** und **Translator**.

Translator übersetzen Texte und erfüllen damit Anforderung K3-6. Sie verwenden die DeepL-API [18]. Die Ergebnisse speichert der Prototyp in einem Cache, so dass er weniger Anfragen senden muss und damit Zeit sparen kann. Der Cache sorgt auch für gute Performance bei wiederholten Übersetzungen (s. Anforderung K1-4).

Lookout-Klassen sind für Suchen zuständig. Sie vergleichen verschiedene Wörter und Texte, bestimmen ihre Similarity und verwenden dabei Translator in Kombination mit Lemmatization. Ghengo nutzt sie für das **Finden von Wörtern** (z.B. zur Bestimmung der CRUD-Methode) und das **Heraussuchen von passenden Referenzen in der Implementierung**. Lookout-Klassen bieten die Möglichkeit, ein Fallback-Objekt zurückzugeben, wenn sie kein passendes Ergebnis finden.

Ein Beispiel: Bei der Analyse einer bestehenden Implementierung des gegebenen Django-Projekts (s. Anforderungen K3-2 und K3-3) verwendet der Lookout den Translator. Beim Satz „Gegeben sei ein Auftrag“ übersetzt der Lookout mithilfe des Translators das Wort *Auftrag* (engl. order) und durchsucht alle implementierten

Models. Gibt es ein Model mit ähnlichem Namen, wird es zurückgegeben. Existiert kein passendes Model, gibt der Lookout ein Fallback-Objekt zurück, das den Namen rät. Gäbe es noch keine passende Implementierung, würde Ghengo den Namen *Order* raten. Dies erfüllt Anforderung K3-5.

Lookout-Klassen geben fast ausschließlich sogenannte **Wrapper** zurück. Sie umhüllen das eigentlich interessante Objekt. Wenn ein Fallback notwendig ist, sind Wrapper-Objekte leer und füllen sich von selbst mit wichtigen Informationen (z.B. den Namen). Abbildung 4.7 zeigt ein passendes Klassendiagramm. Dort haben die Wrapper-Klassen die Funktion *get_fields*, die alle Felder des Models zurückgibt. Bei einem bereits implementierten Model, können sie nach dessen Feldern suchen. Der *ModelWrapper* hingegen gibt aufgrund fehlender Implementierung eine leere Liste zurück.

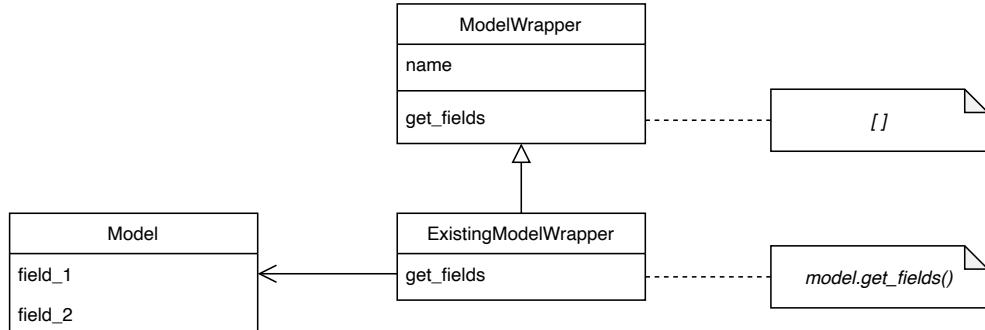


Abbildung 4.7: Klassendiagramm für Wrapper (eigene Darstellung)

Ghengo arbeitet mit Lookouts für Models, Model-Felder, Django-Apps, API-Endpunkte, Serializer-Felder und Viewsets. Das Auffinden von Tokens ermöglichen Lookouts für CRUD-Methoden, Dateiendungen (z.B. txt oder png) und Vergleichsooperatoren (z.B. == oder <=).

Extrahieren von Werten

Mithilfe von Lookout-Klassen kann Ghengo bestimmen, welche Objekte die Namen referenzieren. Für das Bestimmen von Werten werden **ExtractorOutput**-Klassen verwendet. Sie erhalten einen Token (auch **Quelle** genannt), der einen Namen repräsentiert (z.B. aus den vorherigen Beispielen *Nummer* oder *abgeschlossen*) und erstel-

len damit einen nativen Wert (z.B. 1 oder True). ExtractorOutput-Klassen können sich auf verschiedene Datentypen spezialisieren und dann besonders gute Ergebnisse erzielen.

Converter verwenden ausschließlich Extractor-Klassen, die über Informationen zum Anwendungsbereich des nativen Wertes verfügen (z.B. ein Boolean für ein Feld in einem Model). Extractor nutzen eine ExtractorOutput-Klasse und erhalten einen nativen Wert, den sie auf den Anwendungsbereich anpassen können.

Würde Ghengo den Wert für das Wort *abgeschlossener* suchen, besäße er Informationen zum Boolean-Field aus dem Model. Der Converter würde sich den passenden *ModelFieldExtractor* für den Typ Boolean holen. Der Extractor könnte den Text untersuchen und den *BooleanExtractorOutput* nutzen, um einen Boolean-Wert zu erhalten. Hat Ghengo keine Informationen über den Datentyp, muss der Extractor-Output anhand der Quelle Rückschlüsse auf den Datentyp ziehen (s. Anforderung K3-5).

Folgende Liste beschreibt die Datentypen, die die von Ghengo unterstützten nativen Werte annehmen können.

- **String**

ExtractorOutput-Klassen erkennen Strings, wenn Wörter nach der Quelle mit Anführungszeichen markiert sind. Ein Beispiel ist *Gegeben sei ein Auftrag mit dem Namen „foo“*.

- **Boolean**

Boolean-Werte erkennen die Klassen anhand von Negation. Ist die Quelle oder das zugehörige Verb negiert, so ist der Wert *False*. In dem Satz *Gegeben sei ein Auftrag, der nicht abgeschlossen ist*, ist *abgeschlossen* mit dem Wort *nicht* negiert. Hier wäre der Output also *False*. Beim Raten sucht Ghengo nach Wörtern wie *wahr* oder *False*. Wenn die Quelle beim Raten ein Verb oder Adjektiv ist, dann überprüft Ghengo dessen Negation.

- **Zahlen (Float, Integer, Decimal)**

Zahlen sind immer Kindes-Elemente der Quelle im Dependency-Tree. Das Wort *Nummer* im Satz *Gegeben sei ein Auftrag mit der Nummer 1* ist also der head von *1*. Hat Ghengo keine Information über den Datentyp, sucht er nach Text,

den er von einem String in eine Zahl umwandeln kann. Ghengo verarbeitet auch ausgeschriebene Zahlen wie *zwei*, *zwölf* oder *keine*.

- **Variablen**

Weiß der Prototyp, dass aus der Quelle eine Variable entstehen muss, sucht er nach Eigennamen, Zahlen oder Text in Anführungszeichen. Daraus entsteht eine Variable mit passendem Namen. Im Satz *Gegeben sei ein Benutzer Bob* wäre *Bob* ein Eigenname, der zu der Variable *bob* führt. Weiß Ghengo nichts über den Typ, sucht er nach Text mit Anführungszeichen und den Zeichen <> (z.B. „<foo>“).

Einige Converter erlauben auch die Definition einer ID. Im Satz „Gegeben sei ein abgeschlossener Auftrag 1“ interpretiert Ghengo die *1* als ID und erstellt hierfür eine Variable im Output-AST. Referenziert ein Satz in einem anderen Step diesen Auftrag (z.B. „Wenn Auftrag 1 gelöscht wird“), erkennt Ghengo den Kontext des Satzes (das Model *Order*) sowie die ID und erstellt eine zugehörige Referenz zu dieser Variable.

- **Auflistung von Strings, Zahlen oder Variablen**

Der Prototyp unterstützt Auflistungen, also die Verwendung mehrerer Strings, Zahlen oder Variablen. Auflistungen führen immer zu Abhängigkeiten, die Ghengo im Dependency-Tree ablaufen kann. Abbildung 4.8 zeigt beispielhaft Teile der Abhängigkeiten für den Satz *Gegeben sei ein Auftrag mit den Dateien 1, 2 und 3*. Das Ganze funktioniert nicht, wenn Ghengo die Typen erraten muss.

- **Datenstrukturen (dict, set, tuple)**

Datenstrukturen unterstützt Ghengo nur teilweise. Ihre Beschreibung ist in natürlicher Sprache schwer, die Formulierung verständlicher Steps deshalb eine Herausforderung. Output-Klassen erkennen die Datenstrukturen nur dann zuverlässig, wenn diese mit Anführungszeichen versehen sind (z.B. „(1, 2)“). Bei fehlender Information über den Datentyp funktioniert dies auf die gleiche Art und Weise. Verschachtelte Datenstrukturen unterstützt Ghengo nicht.

- **GenerationWarning**

Entdeckt ein Extractor, dass ein ExtractorOutput Probleme beim Erstellen eines nativen Werts hatte, erstellt er ein sogenanntes **GenerationWarning** und gibt es an den Converter weiter. Die entwickelnde Person sieht die Warnung im generierten Code und kann Verbesserungen durchführen (s. Anforderung K4-4).

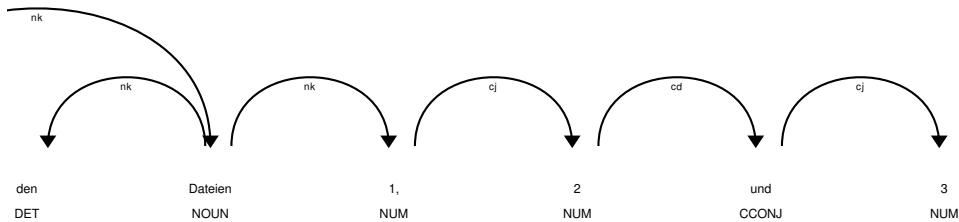


Abbildung 4.8: Abhängigkeiten bei Listen von Zahlen (generiert von Displacy [98])

Exemplarische Datenextrahierung

Abbildung A.4 zeigt die Architektur aller Klassen, die im folgenden Beispiel enthalten sind. Sie gibt auch einen Einblick in die Abhängigkeiten aller beteiligten Klassen in diesem Abschnitt. Gegeben seien folgende Steps:

- **Gegeben sei** ein Auftrag 1, der abgeschlossen ist
- **Wenn** Auftrag 1 gelöscht wird

Der Converter zum Erstellen von Models übernimmt den ersten Schritt. Er erkennt *Auftrag* und identifiziert das Wort mithilfe von Translator und Lookout als Referenz zum Order-Model. Darauf folgt eine 1. Da es eine Zahl ist, interpretiert Ghengo dies als ID, die sich der Converter merkt. Zusätzlich findet er das Wort *abgeschlossen*. Mithilfe von Translator und Lookout kommt er zu dem Ergebnis, dass es sich um ein BooleanField von Django handelt. Er muss also mithilfe der Quelle *abgeschlossen* auf einen Boolean kommen und ruft dafür den passenden Extractor bzw. ExtractorOutput auf. Es wird klar, dass das Wort nicht negiert ist. Folglich ist der native Wert *True*.

Aus all diesen Informationen erstellt der Converter ein Statement im Output-AST. Er muss einen Auftrag mit dem Feld `is_done` und dessen Wert `True` anlegen. Der Auftrag ist der Variable zugewiesen, die Ghengo mithilfe der ID erstellt hat. Listing 4.4 zeigt, wie der generierte Code aussehen könnte, nachdem der Code-Generator den Output-AST zu Code umgewandelt hat.

```
1 order_1 = order_factory(is_done=True)
```

Listing 4.4: Generierter Code aus dem Text „Gegeben sei ein Auftrag 1, der abgeschlossen ist“

Den zweiten Step übernimmt ein Converter, der auf Requests spezialisiert ist. Er erkennt zunächst anhand des Wortes *Auftrag*, dass es sich um einen Request handelt, der mit dem Order-Model zu tun hat. Er entdeckt auch die ID *1* und sucht in den Statements im Output-AST nach einer passenden Variable im Kontext *Auftrag*. Die CRUD-Methode kann der Converter anhand des Wortes *gelöscht* erkennen. Er findet mithilfe von Lookout und Translator eine DELETE-Route für das Order-Model und kontrolliert, welche Daten der Endpunkt erwartet. In diesem Fall braucht die Route einen Primary Key, der den zu löschen Auftrag identifiziert. Der Converter nutzt dafür die Variable des Auftrags. Eine Möglichkeit für den entstehenden Code zeigt Listing 4.5. In Zeile 1 entsteht ein Client, der die Requests beim Testing vornimmt, und Zeile 2 enthält den Request, der die Referenz zur Variable `order_1` aus Listing 4.4 enthält.

```
1 client = APIClient()
2 client.delete(reverse('orders-detail', {'pk': order_1.pk}))
```

Listing 4.5: Generierter Code aus dem Text „Wenn Auftrag 1 gelöscht wird“

4.4.4 Erstellung des Output-ASTs

Python bietet die Möglichkeit, einen AST für Python-Code zu erstellen [6]. Diesen verwendet Ghengo allerdings nicht, da dies die Erweiterbarkeit für verschiedene Programmiersprachen und Libraries einschränken würde (s. Anforderung K1-1).

Der Output-AST wird stattdessen mithilfe verschiedener **Transformationsklassen** aufgebaut, die übliche Schlagwörter aus Programmiersprachen beschreiben. Es folgt eine Auflistung einiger dieser Klassen. Ihre Abhängigkeiten sind in Abbildung A.1 zu sehen.

- **TestSuite**

Die Klasse umfasst alles, was in die jeweilige Suite gehört. Der erste Schritt beim Erstellen des Output-ASTs ist also die Instanziierung dieser Klasse.

- **TestCase**

Hiermit kann der Generator einzelne Test Cases erstellen. Diese sind normalerweise einer Test Suite zugeordnet.

- **Statements**

Sie stehen für einzelne Instruktionen. Test Cases bestehen normalerweise aus mehreren Statements. Ein Beispiel wäre *foo = 1*. Hierbei handelt es sich um eine Zuweisung. Aus jedem Step entsteht ein Statement oder entstehen mehrere Statements.

- **Expression**

Expressions stehen für Werte. Im Beispiel der Zuweisung wäre *1* die Expression.

- **Variable**

Eine Variable speichert einen Wert. Im oben genannten Beispiel wäre *foo* eine Variable.

- **VariableReference**

Diese Klasse ermöglicht die Referenzierung einer vorher definierten Variable. Die Verwendung einer solchen Klasse eröffnet die Option, am Ende der Generierung nicht referenzierte Variablen aus dem Output-AST zu entfernen, um den Code lesbarer zu machen (s. Anforderung K4-2).

- **Import**

Oft muss man andere Dateien importieren, um bestimmte Funktionen zu verwenden. Diese Klasse repräsentiert einen solchen Import.

- **Parameter**

Ein Parameter beschreibt, welche Daten eine Funktion erhält. Bei der folgenden Definition einer Funktion in Python (`def foo(a, b):`) sind *a* und *b* die Parameter.

- **Decorator**

Ein Decorator kann z.B. eine Klasse oder eine Funktion annotieren und sie erweitern. Ein Beispiel für einen Decorator in PyTest zeigt Listing 4.1 (s. Seite 56).

Der folgende Abschnitt beschreibt die wichtigsten Funktionen, die sich die Transformationklassen teilen.

4.4.5 Umwandlung zu Code

Die Umwandlung zu Code ist die zentrale Funktion der Transformationsklassen. Die Klassen enthalten kleine Templates, die die Syntax des zu generierenden Codes bestimmen. Transformationsklassen verbessern die Lesbarkeit des Codes (s. Anforderung K4-2), indem sie lange Zeilen aufteilen und überflüssige Variablen entfernen. Listing 4.6 zeigt ein Beispiel für zu lange Zeilen und nicht verwendete Variablen.

```

1 def test_all_users():
2     user_1 = create_user(email="foo@local.local", first_name="Max")
3     user_2 = create_user()
4     assert len(get_all_users()) == 2

```

Listing 4.6: Nicht verwendete Variablen und lange Zeilen

In den Zeilen 2 und 3 stehen mit *user_1* und *user_2* zwei Variablen, die der Test nicht weiterverwendet. Es gibt also keine Referenzen. Der Code wäre ohne sie besser lesbar. Ein besseres Beispiel zeigt Listing 4.7. Dort gibt es die Variablen nicht und die lange Zeile ist in mehrere kürzere Zeilen aufgeteilt.

Auch Einrückungen (engl. Indentation) verbessern die Lesbarkeit des Codes. Solche Einrückungen können verpflichtend (z.B. beim Erstellen einer Funktion in Listing 4.6 von Zeile 1 zu 2) oder optional (s. den formatierten Code in den Zeilen 3 und 4 aus Listing 4.7) sein.

```

1  def test_all_users():
2      create_user(
3          email="foo@local.local",
4          first_name="Max",
5      )
6      create_user()
7      assert len(get_all_users()) == 2

```

Listing 4.7: Verbesserung von Listing 4.6

Abbildung 4.9 zeigt den Ablauf der Template-Befüllung einiger Transformationsklassen. In der Darstellung sind einem TestCase mehrere Statements zugeordnet. Die TestCase-Instanz soll ihren Code ausgeben. Dazu muss sie zunächst die Einrückung bedenken und anschließend ihr Template befüllen. Sie benötigt den Code der Statements. Die TestCase-Instanz durchläuft die Statements und erhält ihren Code. Statements machen dabei weitere Anfragen an Kindes-Elemente. Am Ende hat die TestCase-Instanz ihr eigenes Template befüllt und gibt es aus.

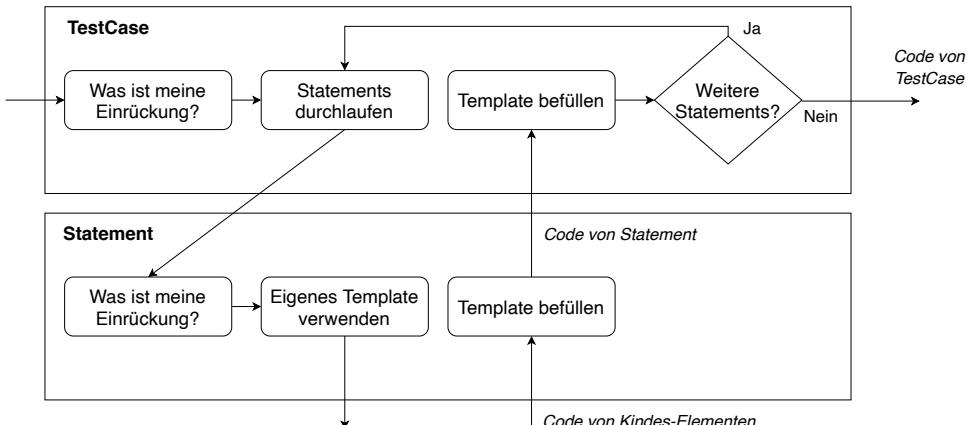


Abbildung 4.9: Beispielhafter Ablauf der Template-Befüllung in Ghengo (eigene Darstellung)

Der Prozess der Code-Umwandlung wird so angelegt, dass Ghengo künftig nicht nur die Library PyTest, sondern auch andere Test-Frameworks unterstützen kann (s. Anforderung K1-1). Das bedeutet: Ghengo muss bei der Umwandlung zu Code reagieren, wenn bestimmte Klassen zu den globalen Einstellungen passen und diese stattdessen verwenden (s. auch Abbildung 4.5 auf Seite 62).

Ein Beispiel: Gegeben sei eine Klasse zum Erstellen einer Expression, mit der man einen Model-Eintrag anlegt (*ModelExpression*). Mit Test-Library A funktioniert dies allerdings anders als mit Test-Library B. Dementsprechend gäbe es verschiedene Klassen, die auf Einstellungen reagieren können. Ghengo muss also beim Anlegen des Output-ASTs flexibel reagieren und die zu den jeweiligen Einstellungen passenden Klassen verwenden. Für Test-Library A wäre das *TestAModelExpression* und für Test-Library B *TestBModelExpression*. Abbildung 4.10 zeigt ein entsprechendes Klassendiagramm.

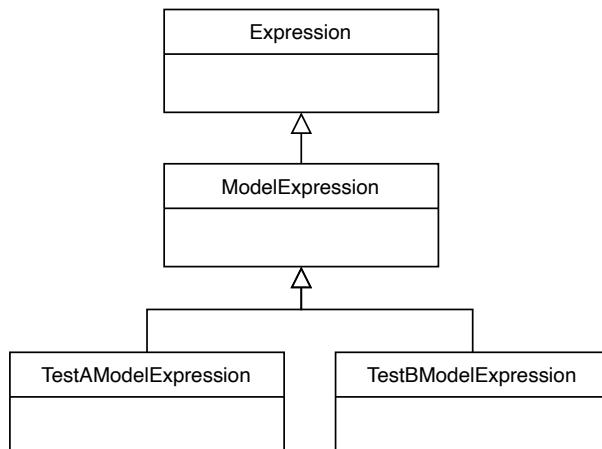


Abbildung 4.10: Klassen für die Erstellung eines Models (eigene Darstellung)

Bei der Umwandlung zu Code sollen Transformationsklassen schließlich auch auf Events hören - insbesondere dann, wenn ein TestCase oder eine TestSuite zu erweitern ist. Ein Beispiel dafür wäre die Verwendung eines Code-Schnipsels, der einen Import erforderlich macht. Sobald das Statement mit diesem Code-Schnipsel dem TestCase hinzugefügt wird, kann es darauf reagieren und einen Import im zu generierenden Code hinzufügen. Ein weiterer Anwendungsfall ist die Verwendung von Parametern.

Listing 4.8 zeigt ein passendes Beispiel. Der Test verwendet in Zeile 4 die Referenz auf eine Variable *par* und die Funktion *get_user*. Würde man während der Generierung nicht in Zeile 1 den Import und in Zeile 3 den Parameter hinzufügen, wäre der Code nicht valide und würde damit Anforderung K4-3 widersprechen.

```
1  from core.user import get_user
2
3  def test_foo(par):
4      assert par == get_user()
```

Listing 4.8: Code mit Parameter und Import

5. Implementierung

Nachdem Kapitel 4 den Ablauf der Generierung und alle beteiligten Klassen vorgestellt hat, wird in diesem Kapitel der Code des Prototyps beschrieben und sein Aufbau begründet. Außerdem geht es an dieser Stelle um Probleme, die bei der Entwicklung aufgetreten sind.

Abschnitt 5.1 befasst sich zunächst mit dem Design-Pattern *Mixin*, das Ghengo an einigen Stellen verwendet. Die Implementierung des Parsers ist kurz gehalten, denn es ist davon auszugehen, dass der interessante Teil der Implementierung in der Code-Generierung liegt. Abschnitt 5.3 beschäftigt sich mit dem Code für NLP. Die restlichen Abschnitte orientieren sich an den in Kapitel 4 erwähnten Klassen.

5.1 Mixin

Ghengo verwendet an mehreren Stellen sogenannte Mixins. Python erlaubt das Erben von mehreren Klassen. Dies können Entwickelnde nutzen, um optionale Funktionen in eine Vererbungskette einzubinden oder um eine Funktion gleichzeitig vielen verschiedenen Klassen hinzuzufügen.

Abbildung 5.1 zeigt beispielhaft ein Vererbungsschema für eine Klasse, die Endpunkte einer API beschreibt. Es gibt zwei Mixins: das erste repräsentiert eine GET-Route und das zweite eine POST-Route. Jedes erbende Kind der *ApiRoute* kann von den Mixins erben und ihre Funktionen integrieren. Listing 5.1 zeigt, wie die Implementierung von *MyApiRoute1* und *MyApiRoute2* in Python aussieht.

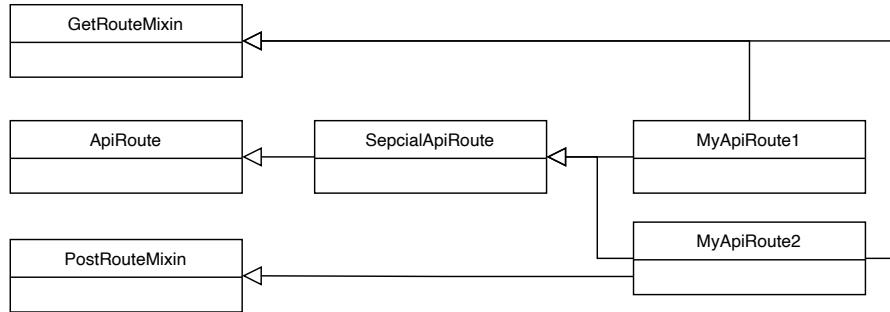


Abbildung 5.1: Beispiel für Anwendung von Mixin (eigene Darstellung)

```

1  class MyApiRoute1(GetRouteMixin, SpecialApiRoute):
2      pass
3
4  class MyApiRoute2(PostRouteMixin, GetRouteMixin, SpecialApiRoute):
5      pass
  
```

Listing 5.1: Erben von Mixins in Python

5.2 Gherkin-Parser

Dieser Abschnitt gibt Einblick in die Implementierung der Validierung einer Sequenz von Tokens. Hier sind vor allem drei Arten von Klassen interessant:

- RuleOperator
- TerminalSymbol
- NonTerminalSymbol

TerminalSymbol-Klassen kontrollieren Tokens an einem gewählten Index. Wenn der Index nicht existiert oder der Token in der Sequenz nicht stimmt, werfen die Klassen einen Fehler. Ist der Token valide, erhöht sich der Index des Pointers um 1. Listing 5.2 zeigt die zugehörige Implementierung.

```

1  class TerminalSymbol(RecursiveValidationBase):
2      def _validate_sequence(self, sequence, index):
3          try:
4              token_wrapper = sequence[index]
5          except IndexError:
6              raise SequenceEnded()
7
8          if not self.token_wrapper_is_valid(token_wrapper):
9              raise RuleNotFulfilled()
10
11         return index + 1

```

Listing 5.2: Validierung der Token-Sequenz der TerminalSymbol-Klasse

RuleOperator-Klassen agieren je nach Art der in Abschnitt 4.3 aufgeführten Operatoren unterschiedlich. Die Methode `_validate_sequence` der Operatoren nutzt immer die `_validate_sequence`-Methode der Kindes-Symbole, um die Validität der Sequenz zu bestimmen. Es kommt zu rekursiven Aufrufen. Treten dabei Fehler auf, werden sie je nach Operator unterschiedlich behandelt. Die Implementierungen der Operatoren befinden sich in den Listings A.15, A.16, A.17 und A.18.

Listing 5.3 zeigt, wie RuleOperator-Klassen verwendet werden. Bei der Instanziierung gibt man die Kindes-Symbole mit. Die RuleOperator-Klassen referenzieren diese dann mit `self.child` oder `self.children`.

```

1  chain = Chain([TerminalSymbol(Token1), TerminalSymbol(Token2)])
2  rep = Repeatable(TerminalSymbol(Token1), minimum=1)
3  oneof = OneOf([TerminalSymbol(Token1), TerminalSymbol(Token2)])
4  opt = Optional(TerminalSymbol(Token1))

```

Listing 5.3: Verwendung der RuleOperator-Klassen

Zu jeder **NonTerminalSymbol**-Klasse gehört jeweils eine Regel. Diese Regel speichert Informationen zu der Frage, wie das Nichtterminalsymbol zu ersetzen ist, und

verwendet sie, um zu überprüfen, ob die Token-Sequenz valide ist. Listing A.19 zeigt die zugehörige Implementierung. Zu jeder NonTerminalSymbol-Klasse gehört ein Attribut *criterion_terminal_symbol*. Ghengo verwendet es, um bei der Validierung verschiedene NonTerminalSymbol-Klassen besser erkennen zu können. In Zeile 12 wird überprüft, ob das Terminalsymbol in der Sequenz vorhanden ist. So kann Ghengo der benutzenden Person genauere Informationen über fehlerhafte Syntax ausgeben.

Alle genannten Fehler, die die Klassen erzeugen (*NonTerminalNotUsed*, *NonTerminalInvalid*, *SequenceEnded* und *RuleNotFulfilled*), enthalten Informationen über die Art des aufgetretenen Fehlers. Diese Informationen werden von den genannten Klassen befüllt, sind aber nicht in den Listings aufgeführt, um sie übersichtlicher zu halten. Die zu Fehlern gegebenen Informationen beantworten folgende Fragen:

- Welcher Token hat den Fehler ausgelöst?
- An welchem Index in der Token-Sequenz ist der Fehler aufgetreten?
- Wo im Text befindet sich der Token, der den Fehler ausgelöst hat (z.B. um die Zeile in einer Fehlermeldung anzuzeigen)?
- Welcher andere Token ließe sich nutzen, um die Syntax zu korrigieren?
- Wie sähe die Fehlermeldung als String aus, die der nutzenden Person ausgegeben wird?

5.3 NLP

Spacy bietet eine einfache Schnittstelle, die es erlaubt, NLP auf einen Text anzuwenden. Listing 5.4 zeigt, dass in Zeile 3 das Model geladen wird, um in Anschluss den Text zu analysieren.

```
1 import spacy
2
3 Nlp = spacy.load('de_core_news_lg')
4 Nlp('Mein Text')
```

Listing 5.4: Anwenden von NLP mit Spacy

Diese Art der Implementierung bereitet Ghengo allerdings Probleme. Es kommt immer wieder vor, dass NLP den gleichen Text im Verlauf der Analyse mehrfach aufbereiten muss. Dies geschieht beispielsweise bei Lookout-Klassen, die nach speziellen Wörtern suchen und dabei Übersetzungen verwenden. Der Code aus Listing 5.4 würde hier zu einer Verlangsamung führen, da Spacy NLP anwenden muss.

Ghengo verwendet hier einen Cache, der dem des Translators aus Abschnitt 4.4.3 ähnelt. Den Code dafür zeigt Listing 5.5.

```

1 import spacy
2
3 class CacheNlp:
4     def __init__(self, name):
5         self.nlp = spacy.load(name)
6         self.cache = {}
7
8     def __call__(self, text):
9         if text not in self.cache:
10             self.cache[text] = self.nlp(text)
11
12     return self.cache[text].copy()
13
14 Nlp = CacheNlp('de_core_news_lg')
15 Nlp('Mein Text')

```

Listing 5.5: Anwenden von NLP mit Spacy und eigenem Cache

Der Prototyp nutzt dabei eine Klasse, die am Anfang das gewählte Model lädt (Zeile 5). Mithilfe der `__call__`-Methode ist der Cache implementiert. Die Klasse überprüft in Zeile 9, ob Spacy den Text bereits einmal verarbeitet hat. Ist dies nicht der Fall, wendet Ghengo das Model an und speichert das Ergebnis im Cache. In Zeile 11 gibt die Klasse immer eine Kopie des Ergebnisses von Spacy zurück. Durch diese Implementierung ist eine gute Performance gewährleistet (s. Anforderung K1-4).

5.4 Tiler

Der Tiler kann NLP mithilfe der in Abschnitt 5.3 beschrieben Klasse an den Steps anwenden. Das daraus entstehende Objekt nennt sich bei Spacy auch **Document**. Jetzt muss der Tiler die beste Aktion bzw. den besten Converter finden. Listing 5.6 zeigt den zugehörigen Code.

```

1  class Tiler:
2      @property
3      def best_converter(self) -> Converter:
4          if self._best_converter is None:
5              highest_compatibility = 0
6
7          for converter_cls in self.converter_classes:
8              converter = converter_cls(**self.get_converter_kwargs())
9              compatibility = converter.get_document_compatibility()
10
11         if compatibility > highest_compatibility:
12             highest_compatibility = compatibility
13             self._best_converter = converter
14
15     return self._best_converter

```

Listing 5.6: Tiler sucht den besten Converter

Der Tiler speichert den besten Converter in einem Attribut, um künftige Verwendungen schneller zu machen. Der Wert ist in `self._best_converter` gespeichert. Beim ersten Durchlauf iteriert der Tiler über alle möglichen Converter-Klassen (Zeile 7) und instanziert diese in Zeile 8. Die Funktion `get_converter_kwargs` übergibt unter anderem das Document des Steps. Der Converter enthält eine Funktion `get_document_compatibility`. Sie gibt einen Wert zwischen 0 und 1 zurück, der zeigt, wie gut ein Document zu diesem Converter bzw. der Aktion passt. Die Methode ist in Abschnitt 5.9.1 genauer erläutert. In Zeile 11 überprüft der Tiler, ob dieser Wert bisher der höchste war. Ist dies der Fall, speichert er den Wert. Am Ende gibt die Property den besten Converter aus.

Tiler, die auf bestimmte Step-Arten spezialisiert sind, müssen somit nur noch die Converter angeben, die für die Step-Art möglich sind (s. Tabelle 4.3 auf Seite 63). Listing 5.7 zeigt die Implementierung des *GivenTilers*.

```

1  class GivenTiler(Tiler):
2      converter_classes = [
3          ModelVariableReferenceConverter,
4          FileConverter,
5          ModelFactoryConverter,
6      ]

```

Listing 5.7: GivenTiler

5.5 Transformationsklassen

Der Aufbau dieses Abschnittes orientiert sich an den drei in Abschnitt 4.4.5 genannten Fähigkeiten der Transformationsklassen: Umwandlung zu Code, Anpassungsfähigkeit und Events.

5.5.1 Umwandlung zu Code

Jede Transformationsklasse muss ihren Code definieren können und nutzt dafür zwei Methoden: *to_template* und *get_template_context*. Die erste Methode ist für die Umwandlung zuständig, während die zweite die Daten für das Template liefert. Listing A.3 zeigt die Klasse, von der alle Transformationsklassen erben. Der Parameter *line_indent* gibt an, wie weit die aktuelle Zeile eingerückt ist. *at_start_of_line* zeigt, ob diese Instanz am Anfang einer Zeile steht. So erhält jede Instanz die Möglichkeit, auf Indentation zu reagieren.

Listing 5.8 zeigt die Implementierung der Umwandlung zur Transformationsklasse für ein Attribut. In Zeile 2 ist das Template mit Platzhaltern definiert. Die Daten der Platzhalter sind in Zeile 10 gegeben. Das *TemplateMixin* befüllt das Template anschließend mit der *format*-Funktion, die Python anbietet.

```

1  class Attribute(TemplateMixin):
2      template = '{variable_ref}.{attribute_name}'
3
4      def __init__(self, variable_ref, attribute_name):
5          super().__init__()
6          self.variable_ref = variable_ref
7          self.attribute_name = attribute_name
8
9      def get_template_context(self, line_indent, at_start_of_line):
10         return {'variable_ref': self.variable_ref,
11             'attribute_name': self.attribute_name}

```

Listing 5.8: Die Transformationsklasse für ein Attribut

5.5.2 Anpassungsfähigkeit

Für die Anpassungsfähigkeit von Transformationsklassen sorgt eine zweite Klasse: **Replaceable** (s. Listing A.4). Erbende Klassen erhalten darüber die Möglichkeit, Eltern in bestimmten Fällen zu ersetzen. Diese geschieht mithilfe der `__new__`- und `__subclasses__`-Methoden. Mit `__new__` lässt sich kontrollieren, welches Objekt bei der Instanziierung entstehen soll. Die `__subclasses__`-Methode gibt alle erbbenden Klassen zurück. Erstellt man im Code eine Instanz der Elternklasse, lässt sich somit an ihrer Stelle das Objekt eines Kindes zurückgeben.

Listing 5.9 zeigt hierfür ein Beispiel in Zeile 2 und 3. In diesem Fall handelt es sich um eine Expression zum Erstellen einer Model-Instanz. Damit dies in PyTest funktioniert, müssen einige Änderungen am TestCase vorgenommen werden. Erstellt Ghengo bei der Generierung von PyTest an einer beliebigen Stelle im Code eine *ModelFactoryExpression*-Instanz, entsteht stattdessen ein *PyTestModelFactoryExpression*-Objekt (Zeilen 11 und 12). Wird kein PyTest generiert, geschieht dies nicht (Zeilen 8 und 9).

Diese Form der Implementierung hat entscheidende Vorteile. Die Basis-Klasse benötigt keine Informationen über das Kind. Beim Erstellen des Output-ASTs muss der Converter nicht Ghengos Einstellungen kontrollieren und kann darauf verzichten, verschiedene Transformationsklassen zu instanziieren. Das Ersetzen übernehmen die Transformationsklassen selbst. Man könnte an dieser Stelle auch Factories verwenden. Dies ist unterblieben, da die Implementierung von Factories für alle Transformationsklassen in der aktuellen Version von Ghengo zu einem Overhead führen würde. Die gewählte Version ist kompakt und funktioniert.

```

1  class PyTestModelFactoryExpression(ModelFactoryExpression):
2      replacement_for = ModelFactoryExpression
3      for_test_type = GenerationType.PY_TEST
4
5      def on_add_to_test_case(self, test_case):
6          test_case.add_parameter(Parameter(self.factory_name))
7
8      exp_1 = ModelFactoryExpression()
9      print(isinstance(exp_1, PyTestModelFactoryExpression)) # -> False
10     Settings.generate_test_type = GenerationType.PY_TEST
11     exp_2 = ModelFactoryExpression()
12     print(isinstance(exp_2, PyTestModelFactoryExpression)) # -> True

```

Listing 5.9: Die Transformationsklasse PyTestModelFactoryExpression

5.5.3 Events

Transformationsklassen müssen Code ausführen können, wenn sie zu einem TestCase hinzugefügt werden. Auch hierfür gibt es eine Basis-Klasse, die Listing A.1 zeigt. Jede Transformationsklasse kann mithilfe der *get_children*-Methode ihre Kindeselemente im Output-AST angeben. Ist z.B. ein Statement Teil eines TestCases geworden, kann es von allen Kindes-Elementen die *on_add_to_test_case*-Methode aufrufen. Jedes Kind erhält eine Benachrichtigung. Zeile 6 in Listing 5.9 erstellt beispielsweise einen Parameter im TestCase, wenn Ghengo eine Model-Instanz für PyTest hinzufügt.

Ghengo verwendet im Moment nur dieses eine Event. Informationen über Kindselemente lassen prinzipiell weitere Events zu, auf die Transformationsklassen hören können. Der Anwendungsbereich von Ghengo ließe sich damit künftig erweitern.

5.6 Lookout

Neben Transformationsklassen verwenden Converter Lookout-Klassen. Sie sind für das Auffinden von Objekten mithilfe eines Text-Inputs zuständig. Dieser Abschnitt macht ihre Implementierung anhand eines Beispiels Schritt für Schritt nachvollziehbar. Der zugehörige Code findet sich in Listing A.2.

5.6.1 Möglicher Output

Jede Lookout-Klasse hat einen festgelegten, möglichen Output - also Objekte die aus dem Text resultieren können. Sucht ein Lookout nach bestimmten Wörtern (*TokenLookout*), dann ist der mögliche Output jeder Token im Document. Versucht der Lookout ein Model zu ermitteln (*ModelLookout*), kämen alle Models des Django-Projekts in Frage. Sie bestimmen also Referenzen (s. Abschnitt 4.4.3). Das beste Ergebnis nennt sich *fittest_output_object*. Listing 5.10 zeigt den ersten Teil der *locate*-Methode. Sie durchläuft alle möglichen Output-Objekte. Alle nicht relevanten Objekte überspringt der Lookout.

```

1  class Lookout:
2      def locate(self, *args, raise_exception=False, **kwargs):
3          if self.fittest_output_object is not None:
4              return self.fittest_output_object
5
6          for output_object in self.get_output_objects(*args, **kwargs):
7              if not self.output_object_is_relevant(output_object):
8                  continue
9              # s. Schritt 2...
10             return self.fittest_output_object

```

Listing 5.10: Schritt 1 der Suche der Lookout-Klasse

5.6.2 Keywords

Im nächsten Schritt entnimmt der Lookout den Output-Objekten Schlagwörter. Diese Schlagwörter repräsentieren oder beschreiben die bereits erwähnten Objekte. Es findet also eine Umwandlung der Objekte zu Strings statt. Listing 5.11 zeigt den zugehörigen Code.

```

1  def locate(self, *args, raise_exception=False, **kwargs):
2      for output_object in self.get_output_objects(*args, **kwargs):
3          # s. Schritt 1...
4          keywords = self.get_keywords(output_object)
5          for keyword in keywords:
6              # s. Schritt 3...
7      return self.fittest_output_object

```

Listing 5.11: Schritt 2 der Suche der Lookout-Klasse

Jede Klasse hat die Methode *get_keywords* implementiert und wandelt dort ein Output-Objekt in eine Liste von Schlagwörtern um. Der ModelLookout verwendet ModelWrapper (s. Abschnitt 4.4.3) und gibt deren Namen zurück. Ghengo nutzt in diesem Beispiel den Klassennamen und den *verbose_name* - ein Attribut, welches den Namen ausführlicher beschreibt. Django gibt für jedes Model dieses Attribut vor, das Entwickelnde einfach nutzen können.

5.6.3 Variationen

Damit Lookout-Klassen zu verlässlicheren Ergebnissen kommen, benutzen sie Variationen des Textes. Listing 5.12 zeigt, in welcher Form die Lookout-Klasse diese Variationen erhält.

Variationen geben eine Liste mit zu vergleichenden Documents zurück. Ein zum jeweiligen Lookout passender Translator übersetzt die in Abschnitt 5.6.2 genannten Schlagwörter und den erhaltenen Text-Input. Darüber hinaus verwenden einige der Lookouts Lemmatization. Übersetzungen machen es möglich, Code mit Freitext zu vergleichen.

```

1  def locate(self, *args, raise_exception=False, **kwargs):
2      # s. Schritt 2...
3      for keyword in keywords:
4          variations = self.get_compare_variations(output_object, keyword)
5
6      # s. Schritt 4...
7      return self.fittest_output_object

```

Listing 5.12: Schritt 3 in der Suche der Lookout-Klasse

Übliche Variationen sind:

- der Input auf Englisch übersetzt und das Keyword in einem englischen Document
- der nicht übersetzte Input und das Keyword in einem deutschen Document
- der nicht übersetzte Input und das **zu Deutsch übersetzte** Keyword in einem deutschen Document

Würde der Lookout einen Input-Token *Auftrag* prüfen und nach einem Model suchen, würde er das Keyword *Order* finden, das er aus dem *verbose_name* entnommen hat. Die entstehenden Variationen wären: (*Order* [en], *Order* [en]), (*Auftrag* [de], *Order* [de]) und (*Auftrag* [de], *Auftrag* [de]), wobei [] die Sprache des entstehenden Dokuments angibt. Diese Variationen bieten mehrere Vorteile:

- Es ist möglich, dass z.B. ein deutscher Step auch ein englisches Wort enthält. Ghengo versteht dies immer noch.
- In manchen Fällen funktioniert die Übersetzung in die eine Richtung nicht so gut wie in die andere. Übersetzungen in beide Richtungen führen zu besseren Ergebnissen.
- Die Sprache des Documents ist relevant beim Vergleichen der Texte. Ein englisches Wort in einem deutschen Document (oder umgekehrt) kann beim Bestimmen der Similarity zu Problemen führen. Die erstellten Variationen sind immer in Documents der gleichen Sprache gruppiert. Somit kann man sie besser vergleichen.

Durch diese Variationen entstehen sehr viele neue Documents. Bei den ersten Versionen des Prototyps hat dies zu massiven Performance-Problemen geführt. Die in Abschnitt 5.3 beschriebene Strategie hat das Problem gelöst. Das Gleiche galt für wiederholte Übersetzungen. In beiden Fällen hat ein Cache geholfen.

5.6.4 Similarity

Im nächsten Schritt untersuchen Lookout-Klassen die Similarity der entstandenen Variationen (s. Listing 5.13). Bei dem genannten Beispiel vergleicht Ghengo *Order* [en] mit *Order* [en], *Auftrag* [de] mit *Order* [de] und *Auftrag* [de] mit *Auftrag* [de].

Wie die Berechnung der Similarity funktioniert, unterscheidet sich dabei von Klasse zu Klasse. Ghengo verwendet die Cosine Distance, die Spacy anbietet, und die Levenshtein Distance aus einer importierten Library [66]. Zusätzlich kontrolliert Ghengo in einigen Fällen, ob ein Wort Bestandteil eines anderer Wortes ist, also ob z.B. das Wort *Auftrag* im Wort *Auftragsliste* enthalten ist.

```

1  def locate(self, *args, raise_exception=False, **kwargs):
2      # s. Schritt 3...
3      for value_1, value_2 in variations:
4          similarity = self.get_similarity(value_1, value_2)
5
6          if self.is_new_fittest_output_object(similarity,
7              ↳ output_object, value_1, value_2):
7              self._highest_similarity = similarity
8              self._fittest_output_object = output_object
9              self._fittest_keyword = keyword
10
11     # s. Schritt 5...
12
13     return self.fittest_output_object

```

Listing 5.13: Schritt 4 in der Suche der Lookout-Klasse

Die Methode *get_similarity* gibt einen Wert zwischen 0 und 1 zurück. Ist dieser höher als die bisher beste Similarity, speichert der Lookout diese Daten.

5.6.5 Unsicherheit

Im letzten Schritt befasst sich die Lookout-Klasse mit Unsicherheiten: Wie sicher ist es, dass der beste Output gefunden wurde? Dazu betrachtet die Lookout-Klasse in Listing 5.14 den höchsten Similarity-Score. Ist dieser zu niedrig, behandelt die Klasse dies so, als ob sie kein Ergebnis gefunden hätte. In diesem Fall gibt sie einen Fehler aus oder das Fallback-Objekt, normalerweise eine *Wrapper*-Instanz, zurück.

```

1  def locate(self, *args, raise_exception=False, **kwargs):
2      # s. Schritt 4...
3      if self.has_invalid_fittest_output():
4          self._fittest_output_object = self.get_fallback()
5          self._fittest_keyword = None
6
7      if raise_exception:
8          raise LookoutFoundNothing()
9
10     return self.fittest_output_object

```

Listing 5.14: Schritt 5 in der Suche der Lookout-Klasse

Listing 5.15 zeigt ein Beispiel für einen solchen Fall. Gegeben ist der *ModelLookout*. Die *get_fallback*-Methode nutzt den Translator, um den eigenen Text (z.B. *Auftrag*) zu übersetzen. Diese Übersetzung ist der geratene Name des Models.

```

1  class ModelLookout(DjangoProjectLookout):
2      def get_fallback(self):
3          return ModelWrapper(
4              name=self.translator_to_en.translate(self.text)
5          )

```

Listing 5.15: Beispiel für Fallback der Lookout-Klasse

5.7 ExtractorOutput

Converter benutzen Extractors. Bevor Abschnitt 5.8 auf die Implementierung der Extractor-Klasse eingeht, soll es hier um den Code des ExtractorOutputs gehen. Dieser Abschnitt erklärt den Ablauf der Umwandlung von Text zu nativem Wert und gibt Code-Beispiele.

Jede ExtractorOutput-Klasse durchläuft bei der Umwandlung mehrere Schritte. Der zugehörige Code ist in Listing A.5 zu sehen. Hier steht *native_value* für den nativen Wert und *output_token* für den Token, der den nativen Wert repräsentiert. Im Beispielsatz „Gegeben sei ein Auftrag mit der Nummer 1“ ist *self.source* (Quelle) der Token des Wortes *Nummer* und *output_token* der Token der Zahl 1.

Im Listing sind folgende Schritte zu erkennen:

1. Zunächst werden Tokens zu nativen Werten umgewandelt (Zeilen 7 bis 12). Hat die ExtractorOutput-Instanz keinen Token als Quelle (sondern z.B. einen String), überspringt sie diesen Schritt. Dies kann passieren, wenn im Gherkin-Input Datentabellen enthalten sind. Die Implementierung hierfür findet sich in Abschnitt 5.7.1.
2. Beim Erstellen einer Output-Instanz kann angegeben werden, dass die Quelle der Output ist. Dafür lässt sich das Attribut *source_represents_output* nutzen. Der Ersteller der Klasse kann also vorgeben: bitte suche nicht nach dem *output_token*, sondern nehme stattdessen die Quelle.
3. Der Output-Token wird gesetzt (Zeile 14).
4. Der entstandene native Wert (z.B. die Nummer 1) kann unter Umständen eine Variable repräsentieren (s. Abschnitt 4.4.3). Ist dies der Fall, gibt der ExtractorOutput in Zeile 17 die Instanz einer Variable-Transformationsklasse zurück (s. Abschnitt 4.4.4).
5. Ist dies nicht der Fall, wird der native Wert in Zeile 19 vorbereitet. Dies gibt ableitenden Klassen Spielraum für Änderungen.
6. In Zeile 20 findet eine zweite Transformation statt: vom nativen Wert zum Output. Der Grund: Manchmal ist es erforderlich, native Werte noch einmal

anzupassen. Ist z.B. eine Zahl als String gegeben, muss der *ExtractorOutput* diese unter Umständen erst in einen Integer umwandeln. Die zweite Transformation ist in Abschnitt 5.7.2 beschrieben.

7. Die Methode *prepare_output* bietet ableitenden Klassen erneut die Möglichkeit, Anpassungen vorzunehmen (Zeile 22).

Ist einer dieser Schritte fehlerhaft (*ExtractionError*), wird der *output_token* in Zeile 24 überschrieben und der Fehler weitergegeben. Normalerweise behandelt ihn der zugehörige *Extractor* und hängt ein GenerationWarning an den Output-AST (s. Abschnitt 4.4.3).

5.7.1 Token zu nativem Wert

Dieser Schritt ist durch die Methode *token_to_native_value* repräsentiert. Der *ExtractorOutput* (die Basis-Klasse) versucht, anhand der Quelle den nativen Wert zu erraten. Der zugehörige Code steht in Listing A.6.

In den Zeilen 3 bis 9 behandelt der *ExtractorOutput* anhand von Adjektiven, Verben oder Adverbien zu erkennende Boolean-Werte. Ghengo kontrolliert, ob die Tokens negiert sind und ob der native Wert *True* oder *False* ist. Im Anschluss sucht der Prototyp in den Zeilen 11 bis 13 nach abhängigen Tokens, die Variablen sein könnten. Steht der nächste Token in Anführungszeichen, übernimmt Ghengo diesen Wert in den Zeilen 15 bis 17. Im letzten Schritt sucht er nach Eigennamen (engl. proper noun) vor der Quelle. Es fällt auf, dass die Methode in den meisten Fällen einen String als nativen Wert zurück gibt. Der native Wert muss dann nochmals zu Output verarbeitet werden.

Falls der *ExtractorOutput* keinen nativen Wert entnehmen kann, entsteht ein *ExtractionError*. Der Code in Zeile 21 (z.B. *NO_VALUE_FOUND_CODE*) hilft Ghengo, die richtigen *GenerationWarnings* mit passender Beschreibung im entstehenden Test-Code anzuzeigen.

Klassen, die auf gewisse Datentypen spezialisiert sind, überschreiben diese Methode. Listing 5.16 zeigt beispielhaft die Implementierung einer *BooleanOutput*-Klasse, die von *ExtractorOutput* erbt. Die Klasse kann Verben und die Negierung des gegebenen Tokens fokussieren und erzielt dadurch gute Ergebnisse für diesen Datentyp.

```

1  class BooleanOutput(ExtractorOutput):
2      def token_to_native_value(self, token):
3          if self.string_represents_variable(token):
4              return str(token), token
5
6          verb = get_verb_for_token(token)
7          token_value_true = not token_is_negated(token)
8
9          if not verb:
10             return token_value_true, token
11
12         verb_value_true = not token_is_negated(verb)
13         boolean_value = verb_value_true and token_value_true
14         return boolean_value, verb

```

Listing 5.16: Token zu nativem Wert in BooleanOutput-Klasse

5.7.2 Nativer Wert zu Output

Die Methode *native_value_to_output* wandelt native Werte zu Output um. Die Implementierung der Basis-Klasse ist in Listing A.7 zu sehen. In den Zeilen 3 und 4 fängt sie einige native Datentypen, wie Integer, Float oder Boolean ab. Handelt es sich bei dem nativen Wert um einen String, versucht Ghengo, diesen ab Zeile 10 in verschiedene Zahlen umzuwandeln. Schlägt dies fehl, versucht der ExtractorOutput in den Zeilen 16 bis 21, den String in eine Datenstruktur, etwa Listen oder Dictionaries, umzuwandeln. Im letzten Versuch sucht die Klasse nach Wörtern, die einen positiven oder negativen Boolean-Wert in der gegebenen Sprache repräsentieren (Zeilen 23 bis 26). Ist dies nicht möglich, gibt die Methode den String aus.

Auch hier können ableitende Klassen Änderungen vornehmen, die zu spezifischen Ergebnissen führen. Listing 5.17 zeigt die Implementierung der *BooleanOutput*-Klasse. Die Spezialisierung auf den Boolean-Datentyp verkürzt die Implementierung im Vergleich zu der in Listing A.7.

```

1  class BooleanOutput(ExtractorOutput):
2      def native_value_to_output(self, native_value):
3          if isinstance(native_value, bool):
4              return native_value
5
6          return native_value in
    ↳ POSITIVE_BOOLEAN_INDICATORS[self.document.lang_]

```

Listing 5.17: Zum Generieren von Output erforderliche Schritte der BooleanOutput-Klasse

5.8 Extractor

Dieser Abschnitt soll zeigen, wie Extractor-Klassen die ExtractorOutput-Klassen verwenden. Ein Beispiel macht deutlich, dass Extractor-Klassen Auflistungen von nativen Werten ermöglichen.

5.8.1 Verwendung von Output

Zu jedem Extractor gehört ein ExtractorOutput. Listing A.8 zeigt die verschiedenen Methoden der Extrahierung. Converter verwenden die Methode *extract_value* als public-Methode, um an den Output zu kommen. Intern verwendet der Extractor *_extract_value*. Methoden, Funktionen und Variablen, die in Python mit einem *_* anfangen, gelten per Konvention als private.

Der Extractor erstellt eine Instanz der zugehörigen ExtractorOutput-Klasse (Zeile 17 und 18) und ruft anschließend deren *get_output*-Methode auf. Entsteht ein *ExtractionError* (s. Abschnitt 5.7), gibt der Extractor einen *GenerationWarning* zurück.

5.8.2 ManyExtractorMixin

Ghengo unterstützt die Aufzählung von Strings, Zahlen oder Variablen (s. Abschnitt 4.4.3). Der Extractor muss dafür Aufzählungen, wie in Abbildung 4.8 (s. Seite 69) dargestellt, ablaufen.

Hierfür verwendet Ghengo das *ManyExtractorMixin*. Jeder Extractor kann von ihm erben und Aufzählungen möglich machen. Listing A.9 zeigt die wichtigste Methode des Mixins (*_extract_many*). Sie holt sich mithilfe des Dependency Trees alle Kinder der Quelle (Zeile 6). Das Mixin durchläuft jedes Kind und erstellt den Extractor für diesen Input. Passt das Kind nicht zum Output, wird es übersprungen (Zeilen 10 und 11).

Den passenden Kindes-Token passt man in der entstehenden Output-Instanz an. Mit dem Attribut *source_represents_output* wird er in Zeile 14 als Output-Token definiert (s. Abschnitt 5.7). Im Anschluss speichert das Mixin alle nativen Werte und GenerationWarnings in einer Liste (Zeilen 16 bis 22), die es am Ende zurück gibt.

5.9 Converter

Nachdem vorherige Abschnitte dieses Kapitels die Implementierung aller Klassen um die Converter beschrieben haben, geht es nun um den Code, der alles verbindet. Zunächst zeigt Abschnitt 5.9.1, wie die Kompatibilität zu einzelnen Texten berechnet wird. Diese Kompatibilität nutzen Tiler, um die beste Aktion zu bestimmen. Im Anschluss geht es um die Frage, wie Converter Daten mithilfe von Lookout und Extractor bestimmen. Zuletzt illustrieren einige Beispiele die Verwendung der Transformationsklassen.

5.9.1 Kompatibilität

Zur Erinnerung: jeder Converter ist einer durch Ghengo unterstützten Aktion zugeordnet (s. Abschnitt 4.4.2). Tiler müssen den Converter finden, der am besten zu einem Text passt (s. Abschnitt 5.4). Dafür stehen in Ghengos aktueller Version 13 verschiedene Converter mit der Methode *get_document_compatibility* zur Verfügung. Diese Methode analysiert den Text und überprüft, wie gut er zur Aktion des Converters passt. Die Converter...

- ...suchen mithilfe von Lookout-Klassen nach gewissen Wörtern.
- ...analysieren die POS-Tags spezifischer Wörter.
- ...überprüfen, ob Wörter im Singular oder Plural stehen.

- ...verwenden den Dependency Tree, um Abhängigkeiten zu erkennen.
- ...prüfen, wie der bisherige Output-AST aussieht.
- ...nutzen den Output-Token von ExtractorOutput-Klassen (s. Abschnitt 5.7).

Jeder Converter verwendet unterschiedliche Techniken, um auf einen Wert zwischen 0 und 1 zu kommen. Tiler verarbeiten diesen Wert anschließend (s. Abschnitt 5.4).

An diesem Punkt stellte die Entwicklung des Prototyps eine echte Herausforderung dar. Es ging um die Frage: Wie kann man sicherstellen, dass Ghengo einer in natürlicher Sprache formulierte Botschaft tatsächlich die von der auftraggebenden Person intendierte Aktion zuweist. Das Problem dabei: In manchen Fällen können Formulierungen ähnlich klingen, aber doch unterschiedliche Dinge bedeuten.

Ein Beispiel: Die Sätze „Dann sollte der Auftrag 1 die Nummer 1 haben“ und „Dann sollte ein Auftrag die Nummer 1 haben“ klingen sehr ähnlich. Ghengo interpretiert sie allerdings unterschiedlich. Im ersten Step weisen die Wörter **der Auftrag** auf einen spezifischen, zuvor erstellten Auftrag hin. Im zweiten Step gibt es diesen Bezug nicht.

Listing A.10 zeigt die Implementierung aus dem *AssertPreviousModelConverter*. Er ist für Aktionen zuständig, die vorher erstellte Model-Instanzen überprüfen - also den ersten der zwei genannten Sätze. In Zeile 5 überprüft der Converter zunächst, ob es eine Referenz zu einer bereits im Output-AST vorhandenen Variable gibt. Im vorher genannten Step wäre dies die Nummer 1 nach dem Wort *Auftrag*. Findet sich diese Referenz nicht, vermindert dies die Kompatibilität. Außerdem überprüft der Converter, ob der Token, der das Model repräsentiert, im Plural steht. Eine direkte Referenz auf eine Model-Instanz stünde vermutlich im Singular.

Listing A.11 zeigt den Code des *ObjectQuerysetConverters*. Er untersucht die Datenbank und wäre damit für den zweiten Satz zuständig. Der Converter überprüft, ob das Wort vor dem Token, der das Model repräsentiert, ein bestimmter (der, die, das) oder unbestimmter (ein, eine) Artikel ist (Zeilen 8 bis 13). Auch dieser Converter kontrolliert, ob der Model-Token im Plural steht (Zeilen 15 und 16).

5.9.2 Datenbestimmung

Um den Output-AST befüllen zu können, brauchen Converter Namen, Objekte, die die Namen referenzieren, und native Werte (s. Abschnitt 4.4.3). Converter verwenden zur Bestimmung der Referenzen einen festgelegten Algorithmus und durchlaufen alle möglichen Tokens (s. Listing A.12 in Zeile 5). Tokens, die keine Namen repräsentieren können, sortiert Ghengo in Zeile 6 und 7 aus. Im Anschluss bestimmt der Converter mithilfe von Lookout-Klassen in Zeile 9 und 10 das referenzierte Objekt im Code. Liegen doppelte Referenzen vor, wird erneut gefiltert (s. Zeile 12 und 13). Zuletzt speichert der Converter einen Wrapper, der den Token und das referenzierte Objekt festhält. Ghengo bestimmt für jeden dieser *ReferenceTokenWrapper* den optimalen Extractor und den zugehörigen nativen Wert.

Listing A.13 zeigt die Implementierung des Algorithmus, der überprüft, ob ein Token einen Namen repräsentieren kann. Da der Algorithmus bereits in Abschnitt 4.4.3 beschrieben ist, wird der Code hier nicht weiter erläutert.

5.9.3 Statements erstellen

Im letzten Schritt erstellt der Converter Statements mithilfe der Extractors. Das Erstellen geschieht in drei Phasen, deren Implementierung Listing 5.18 zeigt:

1. Erstellung einer Liste aus Statements:

Diese Liste ist initial leer. Manche Converter erstellen hier allerdings bereits Statements, die sie in späteren Phasen verwenden. In dieser Phase wird die Methode *prepare_statements* ausgeführt.

2. Durchlaufen der Extractors:

Der Converter durchläuft alle erstellten Extractors und passt die Liste der Statements aus Phase 1 an. Er ruft für jeden Extractor die Methode *handle_extractor* auf.

3. Aufräumen der Statements

Diesen Schritt nutzen einige Converter, um der Liste nach dem Durchlaufen der Extractors weitere Statements hinzuzufügen. Dazu ruft der Converter die Methode *finish_statements* auf.

```

1  class Converter:
2      def get_statements_from_extractors(self, extractors):
3          statements = []
4          prepared_statements = self.prepare_statements(statements)
5
6          for extractor in extractors:
7              self.handle_extractor(extractor, prepared_statements)
8
9      return self.finish_statements(prepared_statements)

```

Listing 5.18: Phasen der Erstellung von Statements in Converter-Klasse

Listing A.14 zeigt die Implementierung dieser Methoden am Beispiel der *AssertPreviousModelConverter*-Klasse, die Abschnitt 5.9.1 bereits vorgestellt hat. Zur Erinnerung: der Converter ist dafür zuständig, eine Model-Instanz, die im TestCase erstellt wurde, zu überprüfen.

In den Zeilen 3 bis 5 bereitet der Converter Statements vor. In Django ist es dabei manchmal erforderlich, die Daten der Model-Instanz, die ursprünglich aus der Datenbank kommen, zu aktualisieren. Dafür ist Methode *refresh_from_db* zuständig. Der Converter nutzt die Referenz auf die zuvor erstellte Model-Instanz im TestCase in Zeile 4 (*self.variable_ref*), erstellt mithilfe der Transformationsklassen ein Statement und fügt dieses Statement einer Liste hinzu.

Jeder Extractor, den der Converter durchläuft, steht für die Überprüfung von Werten der Model-Instanz. Erhält der Converter den Satz „Dann sollte der Auftrag 1 die Nummer 1 haben“, hat er im entstehenden Code zu prüfen, ob das Feld *number* der Instanz den Wert 1 hat. Dazu passende Statements erstellt die Methode *handle_extractor*. Zunächst sucht *handle_extractor* in den Zeilen 10 und 11 nach der Art des Vergleichs (z.B. größer als oder kleiner als). Anschließend nutzt die Methode den Extractor und entnimmt ihm den nativen Wert (Zeile 12). In den Zeilen 14 bis 19 baut der Converter das Statement zusammen.

Zeile 15 beschreibt das Model-Attribut, das überprüft werden soll. Zeile 16 steht für den Operator. Zeile 17 enthält den Wunschwert - also im Beispielsatz den Wert 1. Abschließend packt der Converter die Expression in ein *AssertStatement*, das in PyTest zum Überprüfen von Expressions verwendet wird. Listing 5.19 zeigt, wie der generierte Code für den Beispielsatz aussehen könnte.

```
1 order_1.refresh_from_db()  
2 assert order_1.number == 1
```

Listing 5.19: Beispiel für generierten Code vom AssertPreviousModelConverter

6. Anwendungsanalyse

Nach der Implementierung des Prototyps stellen sich drei Fragen: Funktioniert die Übersetzung von Gherkin zu PyTest (s. Anforderung K4-5)? Ist der entstehende Test-Code valide (s. Anforderung K4-3)? Wie ist die Performance bei der Generierung der Tests und was dauert dabei am längsten (s. Anforderung K1-4)? Dies lässt sich im Rahmen einer Simulation klären.

Die Simulation imitiert den Ablauf aus Abbildung 1.1 (s. Seite 5). Um die Generierung anhand bestehender Implementierung zu überprüfen, startet die Simulation mit einem minimalen Django-Projekt, das Abschnitt 6.1 vorstellt. Die Anforderungen an das System werden in mehreren Schritten erweitert. Durch die Änderung der Anforderungen entstehen neue Feature-Dateien. Ghengo soll in der Simulation die Dateien übersetzen und Tests generieren. Die entstandenen Tests werden analysiert, um die oben genannten Fragen zu beantworten.

6.1 Aufbau

In der Simulation soll die Django-API für einen kleinen Online-Shop erstellt werden. Dazu gibt es bereits eine Datenbankstruktur, die Abbildung 6.1 zeigt. Es existiert ein Model, das für die User der Applikation steht. Sie haben einen Vornamen, einen Nachnamen und eine Email-Adresse. Daneben gibt es auch das Model für Aufträge: *Order*. Jedem Auftrag ist ein User über einen ForeignKey zugeordnet. Außerdem hat jeder Auftrag eine Nummer.

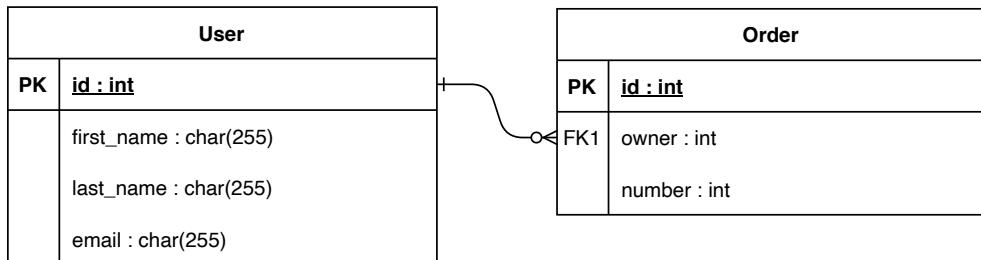


Abbildung 6.1: Aufbau der Datenbank bei Start der Simulation (eigene Darstellung)

Das Projekt enthält bereits einige API-Endpunkte für Aufträge, die Tabelle 6.1 beschreibt. In die Endpunkte ist eine Authentifizierung eingebaut, so dass das System erkennt, welcher User die Anfrage verschickt hat. Die GET-Routen geben nur Aufträge zurück, die den User aus der Anfrage als Besitzer haben.

Methode	URL	Beschreibung
GET	api/orders/	Gibt eine Liste aller sichtbaren Aufträge zurück
GET	api/orders/{id}/	Gibt die Details eines Auftrags zurück
POST	api/orders/{id}/	Erstellen eines Auftrags
DELETE	api/orders/{id}/	Löschen eines Auftrags

Tabelle 6.1: API-Endpunkte am Start der Simulation

Holt sich ein User die Details des Auftrags mit der *id* 1, so muss er die Endroute GET - *api/orders/1/* verwenden. Die GET- und POST-Methoden geben alle Felder der Aufträge in der Antwort zurück. Eine mögliche Antwort wäre:

```
{ id : 1, owner : 1, number : 1 }
```

Die DELETE-Route gibt aus, dass alles geklappt hat. Hier könnte die Antwort so aussehen:

```
{ ok : true }
```

Das Projekt besteht bisher lediglich aus der Implementierung und enthält keine Tests. Mit der Simulation kommt es zu einer Weiterentwicklung des Online-Shops im Rahmen des Behavior-Driven Developments (BDD).

6.2 Ablauf

Die Simulation durchläuft mehrere Schritte:

S-1 Integration des Prototyps in das Projekt

Ghengo muss in das Projekt integriert werden, bevor man es im Anschluss zur Generierung von Tests verwenden kann.

S-2 Anforderungen an bestehendes System

Die Weiterentwicklung des Projekts beginnt. Im ersten Schritt werden Anforderungen an das bestehende System festgehalten. Mit diesen Anforderungen sollen Tests entstehen, die überprüfen, ob die bisherige Implementierung des Systems funktioniert.

S-3 Erweiterung um ein Feld

Es kommt zur ersten Erweiterung des Systems. Das Order-Model erhält ein zusätzliches Feld, das beschreibt, ob ein Auftrag aktiv ist. Beendete oder abgebrochene Aufträge zählen als inaktiv und sollen in den GET-Requests nicht enthalten sein.

S-4 Neue API-Calls mit neuem Model

Der Online-Shop erhält nun die ersten Produkte, die durch ein neues Model repräsentiert werden: dem *Product*. Order und Product sind über eine ManyToMany-Beziehung mithilfe des Models *Item* verknüpft. Ein Item steht für ein Produkt in einem Auftrag und enthält zusätzlich die gewählte Menge. Abbildung 6.2 zeigt die neue Datenbankstruktur. Die API erhält auch die CRUD-Methoden für das neue *Product*-Model und eine weitere Route, mit der man einem Auftrag ein Produkt zuweisen kann.

S-5 Permissions

Nicht jeder User soll jede Aktion durchführen können. Aus diesem Grund gibt es die Permissions: „Kann Produkt anlegen“ und „Kann Produkt ändern“. Ausschließlich User mit den zugehörigen Permissions dürfen Produkte anlegen bzw. bearbeiten.

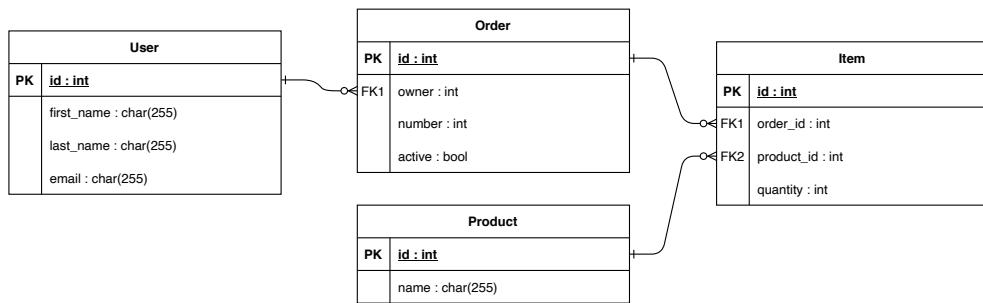


Abbildung 6.2: Aufbau der Datenbank nach Schritt S-4 (eigene Darstellung)

S-6 Anpassung eines API-Calls

Die API erhält eine Erweiterung: Mussten User bisher zunächst einen Auftrag erstellen und dann das jeweilige Produkt hinzufügen, wird nun beides in einem Schritt möglich. Beim Erstellen eines Auftrags soll man zugehörige Produkte und deren Mengen angeben können. Im Anschluss erstellt der Online-Shop automatisch die passenden *Item*-Einträge. Die Daten zum Erstellen eines Auftrags könnten nach den Änderungen folgendermaßen aussehen:

```
{ number : 1, active : true, items : [ { product_id : 1, quantity : 2 } ] }
```

6.3 Durchführung

Im Folgenden werden die zuvor festgelegten Schritte praktisch vollzogen: Eine entwickelnde Person übergibt Feature-Dateien an den Prototyp. Ghengo liest sie aus und generiert daraus Testfälle. Entwickelnde kontrollieren den entstandenen Code und kopieren ihn anschließend, um ihn in die bestehenden Test-Dateien des Projekts einzufügen.

Außerdem beschäftigt sich dieser Abschnitt mit dem Prozess der Generierung und mit dem entstehenden Test-Code, die beide näher untersucht werden sollen. Um einen besseren Einblick in die Performance zu erhalten, wandelt Ghengo jede Feature-Datei 20 Mal um. Mithilfe der dabei ermittelten Durchschnittswerte werden Schwankungen ausgeglichen.

Zum generierten Test-Code stellen sich folgende Fragen:

- Beschreibt der generierte Code den Text aus dem Szenario?
- Ist der entstandene Code valide?

Im Bezug auf Performance wird untersucht:

- Wie lange braucht Ghengo pro Szenario bzw. pro Step?
- Wie viel Zeit pro Szenario erfordert die Anwendung von NLP?
- Welche Klassen benötigen am meisten Zeit?
- Wie lange dauert das Bauen der Statements?

Die Dauer wurde mithilfe des Python-Moduls *time* untersucht [90]. Dabei hat ein Mac-Rechner mit einem 2,8 GHz Quad-Core Intel Core i7 Prozessor und 16GB Arbeitsspeicher Ghengo ausgeführt. Das Modell stammt aus 2017.

6.3.1 Schritt S-1

Ghengo ist zum Zeitpunkt der Erstellung dieser Arbeit noch nicht als Python-Package verwendbar. Für die Simulation wird daher der Ghengo-Ordner in das Django-Projekt kopiert und seine Abhängigkeiten werden installiert. Damit Ghengo Tests generieren kann, müssen folgende Daten verfügbar sein:

- der Pfad zu den Einstellungen des Django-Projekts
- der Pfad zu den Apps des Django-Projekts, damit Python die Dateien laden kann
- der Ordner, in dem Ghengo die Tests anlegen soll
- die Feature-Datei, die Ghengo auslesen soll

Entwickelnde starten Ghengo mithilfe des Befehls aus Listing 6.1. Die Optionen geben die bereits genannten Daten an. Die Informationen können auch in einer *settings.py* in Ghengo angegeben werden. Geschieht dies nicht, muss Ghengo den gesamten Output erraten, da er keine Informationen über das Django-Projekt hat.

```

1 pipenv run python ghengo/main.py \
2   --apps /User/.../apps/ \
3   --settings apps.config.settings \
4   --export-dir generated_tests/ \
5   --feature features/foo.feature

```

Listing 6.1: Start-Befehl für Ghengo

Step-Art	Besten Converter finden	Statements erstellen	Insgesamt
Given	0,13 s	0,09 s	0,22 s
When	0,32 s	0,05 s	0,37 s
Then	0,45 s	0,08 s	0,53 s

Tabelle 6.2: Dauer der verschiedenen Step-Arten in Simulationsschritt S-2

6.3.2 Schritt S-2

Listing A.20 und A.21 zeigen die Feature-Datei, die Ghengo in diesem Schritt erhalten hat. Die entstandenen Tests sind in den Listings A.22, A.23 und A.24 zu sehen.

Da die Implementierung zu diesem Schritt bereits existiert hat, hatte Ghengo keine Probleme bei der Generierung. Der generierte Code war valide und beschrieb die erhaltenen Szenarien. Die Generierung aller Tests dauerte im Schnitt 15,21 Sekunden. Knapp ein Drittel dieser Zeit entfiel auf das Laden der Models von Spacy (4,7 Sekunden). Für die Generierung eines Szenarios mussten durchschnittlich 1,48 Sekunden veranschlagt werden. Dieser Schritt erfüllt somit Anforderung K1-4. Die Anwendung von NLP an Texten hat 63 Prozent der Zeit pro Szenario ausgemacht (0,93 Sekunden).

Tabelle 6.2 vergleicht verschiedene Step-Arten. Sie zeigt, wie lange Tiler für das Auffinden der korrekten Aktion bzw. des korrekten Converters brauchten. Außerdem wird angegeben, wie viel Zeit die Converter benötigten, um Statements mit ihren Daten zu erstellen. Am größten war der Zeitaufwand bei Lookout-Klassen, da sie teilweise große Datenmengen durchsuchen und vergleichen müssen. Pro Szenario haben Lookout-Klassen durchschnittlich 1,33 Sekunden gebraucht.

Step-Art	Besten Converter finden	Statements erstellen	Insgesamt
Given	0,2 s	0,05 s	0,25 s
When	0,33 s	0,01 s	0,34 s
Then	0,2 s	0,04 s	0,24 s

Tabelle 6.3: Dauer der verschiedenen Step-Arten in Simulationsschritt S-3

6.3.3 Schritt S-3

In- und Output sind in Listing A.25 und Listing A.26 erkennbar. Der generierte Code ist valide und entspricht auch der Beschreibung des Textes in Gherkin.

Vor der Implementierung des Features arbeitet Ghengo allerdings nicht einheitlich. Während der Text im ersten und dritten Szenario das Wort *aktiv* verwendet, nutzt das zweite Szenario für den gleichen Sachverhalt das Wort *inaktiv*. Ghengo erkennt nicht, dass hiermit das gleiche Model-Feld gemeint ist. In einem Fall rät Ghengo den Namen *active* und im anderen Fall *more_inactive*. Dies geschieht, weil Ghengo das Wort *inaktiver* für eine Steigerungsform des Wortes *aktiv* hält und an dieser Stelle falsch übersetzt. Nachdem die Implementierung des Feldes *active* im Order-Model erstellt wurde, erkennt Ghengo das Feld (s. Listing A.27).

Das Laden der Spacy-Models dauert bei dieser Datei durchschnittlich 5,06 Sekunden. Jedes Szenario brauchte 1,09 Sekunden. Davon hat die Anwendung von NLP 64 Prozent (0,7 Sekunden) ausgemacht. Tabelle 6.3 vergleicht die benötigte Zeit für die verschiedenen Step-Arten für die Feature-Datei dieses Simulationsschritts. Auch bei dieser Feature-Datei hat Ghengo die meiste Zeit in den Lookout-Klassen verbracht.

6.3.4 Schritt S-4

Die Feature-Datei zu diesem Schritt wurde vor der Implementierung zunächst erstellt, dann umgewandelt und befindet sich in A.28. Listing A.29 und Listing A.30 zeigen den generierten Code. Ghengo hat hier festgestellt, dass es ein Model *Product* geben sollte, aber dafür noch keine Implementierung existiert. Aus diesem Grund gibt es in Zeile 7 (Listing A.30) einen Platzhalter, den Entwickelnde ersetzen müssen.

Ähnlich wie in Abschnitt 6.3.3 übersetzt Ghengo den Namen für das Feld nicht ganz korrekt. Der Prototyp geht vom Namen *names* aus, obwohl das Feld vermutlich eher

Step-Art	Besten Converter finden	Statements erstellen	Insgesamt
Given	0,33 s	0,01 s	0,34 s
When	0,31 s	0,01 s	0,32 s
Then	0,76 s	0,16 s	0,92 s

Tabelle 6.4: Dauer der verschiedenen Step-Arten in Simulationsschritt S-4

name heißen würde (z.B. Zeile 12 in Listing A.29). Durch die fehlende Implementierung sind auch das Anlegen des Auftrags in Zeile 16 und der Vergleich in Zeile 22 nicht korrekt (s. Listing A.30). Hier erkennt Ghengo nicht, dass es sich bei den Produkten um ein ManyToMany-Feld handeln wird.

Auch wenn an einigen Stellen der entstandene Code nochmal angepasst werden muss, hat Ghengo den Inhalt aus der Feature-Datei korrekt übersetzt. Die entstandene Syntax ist ebenfalls fehlerfrei. Nach der Implementierung des Product-Models und des API-Calls, liefert Ghengo bessere Ergebnisse. In Listing A.31 und Listing A.32 wird dies deutlich:

- In Zeile 4 gibt es nun einen Import des *Product*-Models.
- Ghengo erkennt das Feld *name* (z.B. Zeile 9 in Listing A.31).
- Vor der Implementierung hat Ghengo immer den Endpunkt mit der ID *product-detail* verwendet. Nach der Implementierung kann er API-Calls besser unterscheiden. Ghengo nutzt z.B. in Zeile 20 die ID *products-add* (s. Listing A.32).
- Der Vergleich in Zeile 22 aus Listing A.32 arbeitet nun so wie er sollte. Durch die Implementierung erkennt Ghengo, dass man beim Vergleichen die Produktliste verwenden muss. Ghengo erstellt diese Liste mit der Auflistung aus dem zugehörigen Satz „Dann sollte der Auftrag 1 die **Produkte 1, 2 und 3** haben“ (s. Zeile 21 in Listing A.28).

Ghengo brauchte in diesem Schritt zum Laden der Models durchschnittlich 4,95 Sekunden. Jedes Szenario hat 1,9 Sekunden in Anspruch genommen, also etwas länger als bei vorherigen Schritten. Tabelle 6.4 zeigt, dass der Hauptgrund hierfür in den Then-Steps liegt. Sie dauerten länger als bei vorherigen Schritten, weil Ghengo sehr viele verschiedene Converter-Klassen für Then-Schritte prüfen musste. Wurde die

Step-Art	Besten Converter finden	Statements erstellen	Insgesamt
Given	0,29 s	0,12 s	0,41 s
When	0,33 s	0,02 s	0,35 s
Then	0,28 s	0,01 s	0,29 s

Tabelle 6.5: Dauer der verschiedenen Step-Arten in Simulationsschritt S-5

beste Aktion erst am Ende gefunden, dauert dies vergleichsweise lange. NLP hat pro Szenario 67 Prozent ausgemacht (1,27 Sekunden). Die Verwendung von Lookout-Klassen nahm durchschnittlich 1,59 Sekunden ein. Fehlende Implementierung hatte keine Auswirkung auf Ghengos Performance.

6.3.5 Schritt S-5

Listing A.33 zeigt die nächste Feature-Datei der Simulation. Hier wird beschrieben, wie die API die Berechtigungen für User beim Erstellen und Ändern von Produkten überprüft. Der generierte Code befindet sich in den Listings A.34 und A.35. Django stellt bereits die Logik für Berechtigungen zur Verfügung. Ghengo nutzt dies und kann den Inhalt aus der Feature-Datei korrekt übersetzen. Der entstandene Code ist valide. Hier fällt auf, dass der Prototyp den Code an einigen Stellen über mehrere Zeilen verteilt (Zeilen 10 bis 14 in Listing A.34 und Zeilen 7 bis 11 in Listing A.35) und damit lesbarer gemacht hat (s. Anforderung K4-2).

Das Laden der NLP-Models hat durchschnittlich 4,28 Sekunden und jedes Szenario 1,38 Sekunden gedauert. Die Anwendung von NLP an Texten machte pro Szenario zwei Drittel der Zeit aus (0,92 Sekunden). Tabelle 6.5 vergleicht die Zeiten der verschiedenen Step-Arten. Bei der Verwendung von Lookout-Klassen lag der Zeitaufwand bei durchschnittlich 1,12 Sekunden pro Szenario.

Bei diesem Schritt der Simulation haben Then-Steps weniger Zeit in Anspruch genommen als bei vorherigen Schritten. Das ist darauf zurückzuführen, dass Ghengo die Aktionen *Request-Status analysieren* und *Request-Fehler analysieren* sehr einfach und schnell erkennen kann. Mit am längsten hat hier die Verarbeitung der Given-Steps gedauert. Der Grund: Das Durchsuchen aller durch Django bereit gestellten Permissions und die damit verbundenen Anfragen an die Datenbank sind zeitaufwendig.

Step-Art	Besten Converter finden	Statements erstellen	Insgesamt
Given	0,3 s	0,01 s	0,31 s
When	0,89 s	0,35 s	1,24 s
Then	0,55 s	0,14 s	0,69 s

Tabelle 6.6: Dauer der verschiedenen Step-Arten in Simulationsschritt S-6

6.3.6 Schritt S-6

Listing A.36 zeigt die letzte Feature-Datei der Simulation. Sie enthält zwei Szenarien, die es erlauben, den gleichen Sachverhalt auf unterschiedliche Art und Weise zu überprüfen. Dies soll zeigen, dass Ghengo mit verschachtelten Objekten arbeiten kann. Im ersten Fall (Zeile 5) versucht man die Daten in einem Fließtext zu beschreiben. Die zweite Variante (Zeile 11) enthält das verschachtelte Objekt mit Anführungszeichen.

Listing A.37 zeigt den entstandenen Code. Ghengo hat hier offenkundig Probleme, das erste Szenario als Test-Code abzubilden. Der Prototyp kann dem When-Step die Daten nicht korrekt entnehmen (s. Zeile 16), versteht die gewählte Formulierung nicht und macht dies mit einem GenerationWarning erkenntlich (Zeile 6 bis 9). Den When-Step im zweiten Szenario generiert Ghengo korrekt und problemlos, da es ihm möglich ist, alle Informationen direkt aus dem Step zu kopieren. Der generierte Code ist vor und nach der Implementierung des API-Calls identisch.

Die genannten Probleme zeigen sich auch in der Performance. Ghengo braucht für die Generierung der Szenarien vergleichsweise viel Zeit (2,92 Sekunden). Während die meisten Daten mit denen aus vorherigen Schritten übereinstimmen, dauert der When-Step deutlich länger (s. Tabelle 6.6).

7. Diskussion

Dieses Kapitel bewertet den Prototyp auf Grundlage der in der schrittweise erweiterten Anwendungssimulation gemachten Erfahrungen, nennt seine Stärken und seine Schwächen. Dabei geht es um folgende Fragen: Lässt Ghengo sich gut in ein Projekt integrieren? Funktioniert die Transformation von natürlicher Sprache zu Tests? Ist der erstellte Code valide? Und: Stimmt die Performance?

Integration: Die Simulation hat gezeigt, dass Ghengo sich problemlos in ein bestehendes Projekt integrieren lässt. Allerdings müssen Entwickelnde den Ordner des Prototyps dafür in ein Django-Projekt kopieren und seine Abhängigkeiten installieren. Ein automatisierter Prozess wäre an dieser Stelle wünschenswert und würde die Integration vereinfachen. Ghengo zeigt somit die Qualitäten einer Library, aber es fehlt ihm an dieser Stelle noch der Feinschliff.

Transformation von Sprache zu Tests: Die Anwendungsanalyse hat gezeigt, dass Ghengo aus natürlicher Sprache Tests generieren kann. Einfache Formulierungen wurden schon in den ersten Simulationsdurchläufen korrekt abgebildet. Fehler, die zunächst bei Auflistungen und beim Erraten von Namen auftraten (s. Listings A.26 und A.29), ließen sich durch die Implementierung eines neuen Features beheben. Danach konnte Ghengo die Informationen aus dem Projekt korrekt verwenden.

Trotzdem lässt sich nicht ausschließen, dass Ghengo unter Real-Life Bedingungen mit bestimmten Texten Schwierigkeiten haben könnte. Möglicherweise würde der Prototyp dann mit bei seiner Entwicklung nicht bedachten Formulierungen konfrontiert.

Vielleicht müsste er auch Sätze verarbeiten, in denen Namen, Referenzen und native Werte anders vermittelt sind, als bisher angenommen. Dies könnte zu Ungenauigkeiten und inkorrektener Transformation führen. Was bei einer breiten Anwendung des Prototyps zu erwarten wäre, müsste künftig geklärt werden. Dies zu untersuchen, war nicht Gegenstand dieser Arbeit.

Außerdem hat sich bestätigt, was in Abschnitt 4.4.3 bereits angedeutet wurde: Ghengo hat Probleme mit Datenstrukturen und deren Vernestung und zeigt es Entwickelnden mithilfe von GenerationWarnings. Simulationsschritt S-6 macht deutlich, dass die Verständlichkeit der Feature-Datei bei solchen Anforderungen leidet. Dies kann auch zu Verständnisproblemen beim PO führen.

Code: Der entstandene Python Code für die PyTest-Library war in jedem Schritt der Simulation valide. Muss Ghengo Teile des Codes erraten, gleicht er dies mit Platzhaltern aus (s. Listing A.29). In üblichen IDEs entstehen für diese Platzhalter Warnings. Dies ist zu erwarten und sogar erwünscht, denn so erhalten Entwickelnde den Hinweis, dass etwas angepasst werden muss.

Ghengo hat auch in allen Schritten der Simulation für eine gute Lesbarkeit des Codes gesorgt. In keinem generierten Testfall entstanden nicht verwendete Variablen. Lange Zeilen hat Ghengo auf mehrere Zeilen aufgeteilt (s. Listing A.34).

Performance: Der Prototyp kann die maximal erwünschte Dauer von 5 Sekunden pro Szenario aus Anforderung K1-4 unterbieten und arbeitet damit performant. In der Simulation hat die Generierung pro Szenario nur 1 bis 2 Sekunden gedauert. Stößt Ghengo auf vernestete Datenstrukturen braucht er 3 Sekunden pro Szenario (s. Schritt S-6). Innerhalb des insgesamt sehr schnell ablaufenden Generierungsprozesses beanspruchen die Lookout-Klassen, die mehrfach NLP anwenden und Texte übersetzen müssen, die meiste Zeit. Es hat sich auch gezeigt, dass das Extrahieren von Daten bzw. das Erstellen der Statements weniger Zeit in Anspruch nimmt als die Bestimmung der Bedeutung bzw. das Auffinden der besten Aktion (s. Tabellen aus Abschnitt 6.3). Fehlende Implementierung hat keinen Einfluss auf die Performance. Es ist sogar damit zu rechnen, dass Ghengo vor der Implementierung tendenziell schneller ist, da er danach eine größere Code-Basis durchsuchen muss.

Um gute Performance bei Übersetzungen zu erreichen, verwendet der Prototyp einen Cache. Ohne Cache, muss Ghengo mehrfach Requests gegen die DeepL-API ausführen und braucht länger für die Testgenerierung als die angestrebten 5 Sekunden. Bei der Simulation hat sich der Cache bewährt und initiale Probleme gelöst.

Die Simulation hat auch gezeigt, dass das Laden der Spacy-Models durchschnittlich 4 bis 5 Sekunden dauert. Ist die Feature-Datei klein (z.B. Listing A.25), kann das 60 Prozent der benötigten Generierungszeit ausmachen. Bei der Implementierung des Prototyps wurde kein Weg gefunden, der es ermöglicht hätte, die Models schneller zu laden.

Zusammenfassung: Zusammengefasst lässt sich sagen, dass Ghengo in der Lage ist, aus natürlicher Sprache Testfälle in PyTest für Django zu generieren. Es konnten alle Ziele aus Kapitel 3 erreicht werden. Die Simulation hat darüber hinaus gezeigt, dass der Prototyp auch bei fehlender Implementierung weitestgehend korrekt arbeitet und damit genau das bietet, was im Test-First-Umfeld der agilen Softwareentwicklung gebraucht wird. Außerdem wurde deutlich, dass der Prototyp Testfälle mit weniger Informationen generieren kann als in früheren Arbeiten beschriebene Methoden, die auf den Prozess des Red-Green-Refactor Zyklus (s. Abschnitt 2.2.8) setzen [48, S. 26ff]. Durch schnelle Generierung und gute Performance verspricht der Prototyp effizientes Arbeiten. In einigen Bereichen kann Ghengo allerdings noch verbessert werden. Beispiele hierfür sind vernestete Datenstrukturen, das Erkennen von Aufzählungen ohne Implementierung, die Übersetzung von Text ohne externe API und das Laden der Spacy-Models. Es ist auch damit zu rechnen, dass Ghengo gewisse Formulierungen falsch verarbeitet, die bei seiner Entwicklung nicht bedacht wurden.

8. Fazit und Ausblick

Mit der Erstellung des Prototyps wurde das Ziel dieser Arbeit erreicht: Es konnte gezeigt werden, dass es möglich ist, mithilfe von in natürlicher Sprache verfassten Feature-Dateien passende Testfälle zu generieren. Die Spezialisierung auf Django und die impliziten Informationen zu den Step-Arten erlauben es Ghengo, auch ohne bestehende Implementierung Test-Code zu generieren. Damit leistet der Prototyp mehr als in bisherigen Veröffentlichungen beschriebene Generierungskonzepte und eignet sich besonders für agiles Arbeiten.

Gibt es bereits eine Implementierung, kann Ghengo zuverlässig Testfälle generieren. Unter diesen Bedingungen erkennt der Prototyp beispielsweise Auflistungen von Daten. In allen Fällen kann er in punkto Geschwindigkeit mit in vorherigen Arbeiten genannten Generierungsmethoden mithalten und generiert lesbaren Code, der die beschriebenen Anforderungen aus Feature-Dateien abbildet.

Ghengo hat allerdings mit der Umsetzung komplexer Anforderungen Probleme. Und es lässt sich nicht ausschließen, dass sich unter Real-Life Bedingungen weitere Schwächen zeigen könnten. Solche Probleme müssten in weiteren Entwicklungszyklen - den Grundprinzipien der generativen Programmierung entsprechend - angepackt werden (s. Abschnitt 2.3.2).

Für den Einsatz in Real-Life Projekten sollte der Prototyp als Python Package verfügbar sein. Alternativ könnte man Entwickelnden eine Benutzeroberfläche zur Verfügung stellen. Eine weitere Option wäre die Integration in eine IDE. Benutzerober-

fläche bzw. IDE hätten den Vorteil, dass der Prototyp die Spacy-Models nicht bei jeder Generierung neu laden müsste. Aktuell nimmt dies einen Großteil der Generierungszeit in Anspruch.

In Real-Life Projekten ist es üblich, dass POs die Anforderungen aus der Sicht des Frontends definieren [70]. Feature-Dateien beschreiben dann Anforderungen, die sowohl Frontend als auch Backend betreffen und E2E-Tests erforderlich machen. Auch daraus lassen sich Ansätze zur Weiterentwicklung des Prototyps ableiten: Man könnte Ghengo so erweitern, dass er Tests nicht nur für Django, sondern auch für Frontend-Technologien wie React oder Vue generiert. Eine Generierung für beide Plattformen (Front- und Backend) gleichzeitig wäre der nächste Schritt.

Darüber hinaus könnten erweiterte Versionen des Prototyps auch andere Test-Frameworks unterstützen. Es ist damit zu rechnen, dass die notwendigen Anpassungen für neue Test-Frameworks in Python kleiner ausfallen, als dies bei der Unterstützung anderer Programmiersprachen der Fall wäre.

Abschließend soll noch Entwicklungspotential bei der Verarbeitung von Tests angesprochen werden: Die Anwendung des Prototyps ließe sich weiter verbessern, wenn er künftig bereits bestehende Test-Dateien auswerten und die darin enthaltenen Informationen zur Aktualisierung nachfolgender Tests nutzen könnte. Somit wäre das Verwalten von Testfällen mit einer Mischung aus generiertem und von Entwickelnden erstelltem Code einfacher. In der aktuellen Version betrachtet Ghengo bestehenden Test-Code nicht. Der Prototyp nutzt lediglich Informationen aus der Feature-Datei und gibt vollständige Test-Suites aus. Ändern sich Anforderungen bzw. eine bestehende Feature-Datei, muss die generierte Datei mit der bisherigen Version abgeglichen werden. Entwickelnde stehen dabei vor vielen Fragen: Was muss aus der alten Datei beibehalten werden und was kann man aus der neuen Version übernehmen? Welche Teile des Codes stammen von Ghengo und welche nicht? Kann man erkennen, was in der alten Datei angepasst wurde?

Abschließend kann gesagt werden, dass Ghengo die definierten Anforderungen erfüllt. Der Prototyp kann mit wenigen Informationen über ein Projekt Tests generieren. In einer ersten Simulation hat er die an ihn gestellten Erwartungen erfüllt. Und er bietet interessante Ansatzpunkte für eine Weiterentwicklung.

Quellenverzeichnis

- [1] *14th annual State of Agile Report*. 2020. URL: <https://www.qagile.pl/wp-content/uploads/2020/06/14th-annual-state-of-agile-report.pdf>. (Zugriff: 18.08.2021).
- [2] Alfred V. Aho u. a. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.
- [3] *AI/ML Models 101: What Is a Model?* 2021. URL: <https://ospreydata.com/ai-ml-models-101-what-is-a-model/>. (Zugriff: 02.09.2021).
- [4] *Alle Mitarbeiterinnen*. 2021. URL: <https://ambient.digital/menschen/alle-mitarbeiterinnen/>. (Zugriff: 07.09.2021).
- [5] *Applications*. 2021. URL: <https://docs.djangoproject.com/en/3.2/ref/applications/>. (Zugriff: 24.08.2021).
- [6] *ast — Abstract Syntax Trees*. 2021. URL: <https://docs.python.org/3/library/ast.html>. (Zugriff: 10.09.2021).
- [7] *BDD Pros and Cons*. 2020. URL: <http://www.automated-testing.com/bdd/bdd-pros-and-cons/>. (Zugriff: 28.10.2021).
- [8] Kent Beck u. a. *Independent Signatories of The Manifesto for Agile Software Development*. URL: <https://agilemanifesto.org/display/index.html>. (Zugriff: 18.08.2021).
- [9] Kent Beck u. a. *Principles behind the Agile Manifesto*. URL: <https://agilemanifesto.org/principles.html>. (Zugriff: 18.08.2021).

- [10] Koen Van Belle. *Benefits and downsides of Cucumber in BDD / b.ignited*. 2021. URL: <https://bignited.be/2020/01/28/bdd-tests-with-cucumber-and-gherkin/>. (Zugriff: 01.09.2021).
- [11] Eli Bendersky. *Abstract vs. Concrete Syntax Trees*. 2009. URL: <https://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees/>. (Zugriff: 31.08.2021).
- [12] Steven Bird, Ewan Klein und Edward Loper. *Natural Language Processing with Python. Analyzing Text with the Natural Language Toolkit*. O'Reilly Media, Inc, 2009. ISBN: 978-0-596-51649-9.
- [13] Jim Buchan, Ling Li und Stephen MacDonell. „Causal Factors, Benefits and Challenges of Test-Driven Development: Practitioner Perceptions“. In: Dez. 2011, S. 405–413. DOI: [10.1109/APSEC.2011.44](https://doi.org/10.1109/APSEC.2011.44).
- [14] Jay Chaturvedi. *A Brief introduction to Django MVT framework*. 2019. URL: <https://medium.com/@jaychaturvedi18/a-brief-introduction-to-django-mvt-framework-8ef46cc321ab>. (Zugriff: 24.08.2021).
- [15] Charles Consel u. a. „A Generative Programming Approach to Developing DSL Compilers“. In: *Generative Programming and Component Engineering*. Hrsg. von Robert Glück und Michael Lowry. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, S. 29–46. ISBN: 978-3-540-31977-1.
- [16] Krzysztof Czarnecki und Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Boston, MA: Addison-Wesley, 2000. ISBN: 978-0-201-30977-5.
- [17] Krzysztof Czarnecki. „Overview of Generative Software Development“. In: *Unconventional Programming Paradigms*. Hrsg. von Jean-Pierre Banâtre u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, S. 326–341. ISBN: 978-3-540-31482-0.
- [18] DeepL API. URL: <https://www.deepl.com/de/docs-api/>. (Zugriff: 05.10.2021).
- [19] Dependency Parsing. 2021. URL: <https://spacy.io/usage/linguistic-features#dependency-parse>. (Zugriff: 03.09.2021).

- [20] *DevOps & Continuous Integration*. 2021. URL: <https://ambient.digital/leistungen/devops-ci/>. (Zugriff: 07.09.2021).
- [21] *Difference between System Testing and End-to-end Testing*. 2019. URL: <https://www.geeksforgeeks.org/difference-between-system-testing-and-end-to-end-testing/>. (Zugriff: 10.09.2021).
- [22] *Digitalagentur für python & django Programmierung*. 2021. URL: <https://ambient.digital/django-agentur/>. (Zugriff: 07.09.2021).
- [23] *E2E-Tests*. 2021. URL: <https://ambient.digital/wissen/ambipedia/E2E-Tests/>. (Zugriff: 07.09.2021).
- [24] Kate Eby. *Comprehensive Guide to the Agile Manifesto*. 2016. URL: <https://www.smartsheet.com/comprehensive-guide-values-principles-agile-manifesto>. (Zugriff: 18.08.2021).
- [25] Francesco Elia. *Constituency Parsing vs Dependency Parsing*. 2020. URL: <https://www.baeldung.com/cs/constituency-vs-dependency-parsing>. (Zugriff: 03.09.2021).
- [26] *English*. 2021. URL: https://spacy.io/models/en#en_core_web_lg. (Zugriff: 15.09.2021).
- [27] *factory_boy integration with the pytest runner*. 2021. URL: <https://github.com/pytest-dev/pytest-factoryboy>. (Zugriff: 27.09.2021).
- [28] Joachim Fischer, Klaus Ahrens und Ingmar Eveslage. *Compilerbau*. 2014. URL: https://www.informatik.hu-berlin.de/de/forschung/gebiete/sam/Lehre/Compilerbau/vorlesungsfolien/CB14_8. (Zugriff: 22.08.2021).
- [29] Joachim Fischer, Klaus Ahrens und Ingmar Eveslage. *Compilerbau*. 2014. URL: <https://www.informatik.hu-berlin.de/de/forschung/gebiete/sam/Lehre/Compilerbau/vorlesungsfolien/5.-top-down-syntaxanalyse.pdf>. (Zugriff: 22.08.2021).
- [30] Joachim Fischer, Klaus Ahrens und Ingmar Eveslage. *Compilerbau*. 2014. URL: https://www.informatik.hu-berlin.de/de/forschung/gebiete/sam/Lehre/Compilerbau/vorlesungsfolien/CB14_9. (Zugriff: 22.08.2021).
- [31] Martin Fowler und Rebecca Parsons. *Domain-Specific Languages*. Pearson Education, Inc, 2011. ISBN: 978-0-321-71294-3.

- [32] Gordon Fraser und Andrea Arcuri. „EvoSuite: Automatic Test Suite Generation for Object-Oriented Software“. In: ESEC/FSE ’11. Szeged, Hungary: Association for Computing Machinery, 2011, S. 416–419. ISBN: 9781450304436. DOI: 10.1145/2025113.2025179. URL: <https://doi.org/10.1145/2025113.2025179>.
- [33] German. 2021. URL: https://spacy.io/models/de#de_core_news_lg. (Zugriff: 15.09.2021).
- [34] Gherkin. 2021. URL: <https://github.com/cucumber/common/tree/main/gherkin>. (Zugriff: 08.09.2021).
- [35] Gherkin for Python. 2019. URL: <https://github.com/cucumber/gherkin-python>. (Zugriff: 09.09.2021).
- [36] Gherkin Reference - Cucumber Documentation. 2019. URL: <https://cucumber.io/docs/gherkin/reference/>. (Zugriff: 01.09.2021).
- [37] Boris Gloer. „Scrum“. In: *Informatik-Spektrum* 33.2 (Apr. 2010), S. 195–200. ISSN: 1432-122X. DOI: 10.1007/s00287-010-0426-6. URL: <https://doi.org/10.1007/s00287-010-0426-6>.
- [38] Thomas Hamilton. *Levels of Testing in Software Testing*. 2021. URL: <https://www.guru99.com/levels-of-testing.html>. (Zugriff: 01.09.2021).
- [39] Thomas Hamilton. *Sanity Testing Vs Smoke Testing: Introduction and Differences*. 2021. URL: <https://www.guru99.com/smoke-sanity-testing.html>. (Zugriff: 01.09.2021).
- [40] Thomas Hamilton. *What is Test Driven Development (TDD)? Tutorial with Example*. 2021. URL: <https://www.guru99.com/test-driven-development.html>. (Zugriff: 02.09.2021).
- [41] Mahmoud Harmouch. *17 types of similarity and dissimilarity measures used in data science*. 2021. URL: <https://towardsdatascience.com/17-types-of-similarity-and-dissimilarity-measures-used-in-data-science-3eb914d2681>. (Zugriff: 05.09.2021).
- [42] Home - Django REST framework. 2021. URL: <https://www.django-rest-framework.org/>. (Zugriff: 24.08.2021).

- [43] Matthew Honnibal u. a. *spaCy: Industrial-strength Natural Language Processing in Python*. 2020. DOI: 10.5281/zenodo.1212303. URL: <https://doi.org/10.5281/zenodo.1212303>.
- [44] J. D. Hunter. „Matplotlib: A 2D graphics environment“. In: *Computing in Science & Engineering* 9.3 (2007), S. 90–95. DOI: 10.1109/MCSE.2007.55.
- [45] Aminul Islam und Diana Inkpen. „Semantic Text Similarity Using Corpus-Based Word Similarity and String Similarity“. In: *TKDD* 2 (Juli 2008). DOI: 10.1145/1376815.1376819.
- [46] *JavaScript End to End Testing Framework / cypress.io*. 2021. URL: <https://www.cypress.io/>. (Zugriff: 01.09.2021).
- [47] Ron Jeffries und Grigori Melnik. „Guest Editors’ Introduction: TDD—The Art of Fearless Programming“. In: *Software, IEEE* 24 (Juni 2007), S. 24–30. DOI: 10.1109/MS.2007.75.
- [48] Sunil Kamalakar. *Automatically Generating Tests from Natural Language Descriptions of Software Behavior*. 2013. URL: https://vttechworks.lib.vt.edu/bitstream/handle/10919/23907/Sunil_Kamalakar_F_T_2013.pdf. (Zugriff: 30.10.2021).
- [49] Tun Khine. *Red, Green, Refactor!* 2019. URL: <https://medium.com/@tunkhine126/red-green-refactor-42b5b643b506>. (Zugriff: 02.09.2021).
- [50] Zak Kincaid und Shaowei Zhu. *LL(1) Parser Visualization*. URL: <https://www.cs.princeton.edu/courses/archive/spring20/cos320/LL1/>. (Zugriff: 22.08.2021).
- [51] Michael Klar und Susanne Klar. *Einfach generieren. Generative Programmierung verständlich und praxisnah*. Carl Hanser Verlag München Wien, 2006. ISBN: 978-3-446-40448-9.
- [52] Lucas Kohorst. *Constituency vs. Dependency Parsing*. 2019. URL: <https://lucaskohorst.medium.com/constituency-vs-dependency-parsing-8601986e5a52>. (Zugriff: 03.09.2021).
- [53] Thijs Kok. *Test Case, Test Suite, Test Run, What’s the Difference?* 2017. URL: <https://www.testmonitor.com/blog/test-case-test-suite-test-run-whats-the-difference>. (Zugriff: 10.09.2021).

- [54] Prof. Dr. Ayelt Komus und Moritz Kuberg. *Status Quo Agile*. 2017. URL: https://www.gpm-ipma.de/fileadmin/user_upload/GPM/Know-How/Studie_Status_Quo_Agile_2017.pdf. (Zugriff: 18.08.2021).
- [55] Holger Krekel u. a. *pytest 6.2.4*. 2004. URL: <https://github.com/pytest-dev/pytest>.
- [56] Stefan Luber. *Was ist SQL?* 2017. URL: <https://www.dev-insider.de/was-ist-sql-a-586264/>. (Zugriff: 31.08.2021).
- [57] Stephan Lukasczyk, Florian Kroiß und Gordon Fraser. „Automated Unit Test Generation for Python“. In: *Lecture Notes in Computer Science* (2020), S. 9–24. ISSN: 1611-3349. DOI: [10.1007/978-3-030-59762-7_2](https://doi.org/10.1007/978-3-030-59762-7_2). URL: http://dx.doi.org/10.1007/978-3-030-59762-7_2.
- [58] Alyona Lukina. *Testing Types*. 2015. URL: http://www.althority.com/testing_types/. (Zugriff: 01.09.2021).
- [59] Christopher D. Manning, Prabhakar Raghavan und Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2009. ISBN: 0521865719. URL: https://nlp.stanford.edu/IR-book/pdf/irbookonline_reading.pdf. (Zugriff: 02.09.2021).
- [60] Marc Moreno Maza. *LL(1) Grammars*. 2004. URL: <https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Syntax.html/node14.html>. (Zugriff: 22.08.2021).
- [61] Joakim Nivre. *Corpus Linguistics, Volume 1*: De Gruyter Mouton, 2008, S. 225–242. ISBN: 978-3-11-021142-9. DOI: [doi:10.1515/booksetHSK29](https://doi.org/10.1515/booksetHSK29). URL: <https://doi.org/10.1515/booksetHSK29>.
- [62] *Overview Of The Gherkin Language*. 2015. URL: <https://devzone.channeldadam.com/library/bdd-behaviour-specification-handbook/specs-gherkin-overview/>. (Zugriff: 22.09.2021).
- [63] Carlos Pacheco und Michael D. Ernst. „Randoop: Feedback-Directed Random Testing for Java“. In: *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*. OOPSLA '07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, S. 815–816. ISBN: 9781595938657. DOI: [10.1145/1297846.1297902](https://doi.org/10.1145/1297846.1297902). URL: <https://doi.org/10.1145/1297846.1297902>.

- [64] *Part-of-speech tagging*. 2021. URL: <https://spacy.io/usage/linguistic-features#pos-tagging>. (Zugriff: 03.09.2021).
- [65] Mario A. Pfannstiel, Werner Siedl und Peter F.-J. Steinhoff. *Agilität in Unternehmen. Eine praktische Einführung in SAFe® und Co.* Springer Gabler, Wiesbaden, 2021. DOI: 10.1007/978-3-658-31001-1. URL: <https://doi.org/10.1007/978-3-658-31001-1>.
- [66] *python-Levenshtein*. 2021. URL: <https://github.com/ztane/python-Levenshtein>. (Zugriff: 19.09.2021).
- [67] Eranga Rajapaksha. *Acceptance Criteria in Scrum*. 2018. URL: <https://medium.com/emblatech/acceptance-criteria-in-scrum-6c1770647b79>. (Zugriff: 18.08.2021).
- [68] Gedeon Rauch. *Was ist eine GUI?* 2017. URL: <https://www.dev-insider.de/was-ist-eine-gui-a-651868/>. (Zugriff: 31.08.2021).
- [69] *Red, Green, Refactor*. URL: <https://www.codecademy.com/articles/tdd-red-green-refactor>. (Zugriff: 02.09.2021).
- [70] Max Rehkopf. *User Storys mit Beispielen und Vorlage*. 2021. URL: <https://www.atlassian.com/de/agile/project-management/user-stories>. (Zugriff: 18.08.2021).
- [71] Max Rehkopf. *What is automated testing?* 2021. URL: <https://www.atlassian.com/continuous-delivery/software-testing/automated-testing>. (Zugriff: 01.09.2021).
- [72] Benno Rice, Richard Jones und Jens Engel. *Tutorial*. 2017. URL: <https://behave.readthedocs.io/en/stable/tutorial.html>. (Zugriff: 18.08.2021).
- [73] Uwe Schöning. *Ideen der Informatik. Grundlegende Modelle und Konzepte der Theoretischen Informatik 3. Auflage*. Oldenbourg Wissenschaftsverlag GmbH, 2008. ISBN: 978-3-486-58723-4.
- [74] *Scrum*. 2021. URL: <https://ambient.digital/wissen/ambipedia/scrum/>. (Zugriff: 07.09.2021).
- [75] Shilpa. *How The Testers Are Involved In TDD, BDD & ATDD Techniques*. 2021. URL: <https://www.softwaretestinghelp.com/testers-in-tdd-bdd-atdd-techniques/>. (Zugriff: 02.09.2021).

- [76] Shubham Singh. *How to Get Started with NLP – 6 Unique Methods to Perform Tokenization*. 2019. URL: <https://www.analyticsvidhya.com/blog/2019/07/how-get-started-nlp-6-unique-ways-perform-tokenization/>. (Zugriff: 02.09.2021).
- [77] Mathias Soeken, Robert Wille und Rolf Drechsler. „Assisted Behavior Driven Development Using Natural Language Processing“. In: *Objects, Models, Components, Patterns*. Hrsg. von Carlo A. Furia und Sebastian Nanz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 269–287. ISBN: 978-3-642-30561-0.
- [78] *Software Testing Tutorial*. 2021. URL: <https://www.javatpoint.com/software-testing-tutorial>. (Zugriff: 01.09.2021).
- [79] Christian Spannagel. *Chomsky-Hierarchie der Grammatiktypen*. 2018. URL: <https://www.inf.hs-flensburg.de/lang/theor/chomsky-hierarchie-grammatik.htm>. (Zugriff: 20.08.2021).
- [80] Ruslan Spivak. *Let's Build A Simple Interpreter. Part 7: Abstract Syntax Trees*. 2015. URL: <https://ruslanspivak.com/lsbasi-part7/>. (Zugriff: 21.08.2021).
- [81] Thomas Stahl und Markus Völter. *Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management*. Jan. 2005. ISBN: 978-3-89864-310-8.
- [82] Thomas Stahl u. a. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. dpunkt.verlag GmbH, 2007. ISBN: 978-3-89864-448-8.
- [83] *Stop words list*. 2021. URL: <https://countwordsfree.com/stopwords/german>. (Zugriff: 02.09.2021).
- [84] Tim Storer und Ruxandra Bob. „Behave Nicely! Automatic Generation of Code for Behaviour Driven Development Test Suites“. In: *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2019), S. 228–237.
- [85] *STTS-Tags gemäß Tiger-Annotationsschema*. URL: https://www.linguistikk.hu-berlin.de/de/institut/professuren/korpuslinguistik/mitarbeiter-innen/hagen/STTS_Tagset_Tiger. (Zugriff: 03.09.2021).

- [86] *System Testing Vs End-To-End Testing: Which One Is Better To Opt?* 2021. URL: <https://www.softwaretestinghelp.com/system-vs-end-to-end-testing/>. (Zugriff: 01.09.2021).
- [87] Daniel Terhorst-North. *Introducing BDD*. 2021. URL: <https://dannorth.net/introducing-bdd/>. (Zugriff: 02.09.2021).
- [88] *The Differences Between Unit Testing, Integration Testing And Functional Testing*. 2021. URL: <https://www.softwaretestinghelp.com/the-difference-between-unit-integration-and-functional-testing/>. (Zugriff: 01.09.2021).
- [89] Christopher M. Thomas. „An Overview of the Current State of the Test-First vs. Test-Last Debate“. In: *Morris Undergraduate Journal* 1 (Aug. 2014). URL: <https://digitalcommons.morris.umn.edu/horizons/vol1/iss2/2>.
- [90] *time — Time access and conversions*. 2021. URL: <https://docs.python.org/3/library/time.html>. (Zugriff: 03.10.2021).
- [91] *Top-level Functions*. 2021. URL: <https://spacy.io/api/top-level#spacy-explain>. (Zugriff: 03.09.2021).
- [92] *Types Of Software Testing: Different Testing Types With Details*. 2021. URL: <https://www.softwaretestinghelp.com/types-of-software-testing>. (Zugriff: 01.09.2021).
- [93] *Über Uns*. 2021. URL: <https://ambient.digital/agentur/ueber-uns/>. (Zugriff: 07.09.2021).
- [94] *Übersetzerbau Entwurf - Codegenerierung*. 2021. URL: https://www.tutorialspoint.com/de/compiler_design/compiler_design_code_generation.htm. (Zugriff: 21.08.2021).
- [95] *UD German PUD*. 2021. URL: https://universaldependencies.org/treebanks/de_pud/index.html. (Zugriff: 03.09.2021).
- [96] *Universal POS tags*. 2021. URL: <https://universaldependencies.org/docs/u/pos/>. (Zugriff: 03.09.2021).
- [97] *Unsere Werte*. 2021. URL: <https://ambient.digital/agentur/werte/>. (Zugriff: 07.09.2021).

- [98] *Visualizers*. 2021. URL: <https://spacy.io/usage/visualizers>. (Zugriff: 03.09.2021).
- [99] *What Are Modifiers?* 2021. URL: <https://www.grammarly.com/blog/modifiers/>. (Zugriff: 03.09.2021).
- [100] *What Is Gherkin + How Do You Write Gherkin Tests?* 2019. URL: <https://www.functionize.com/blog/what-is-gherkin-how-do-you-write-gherkin-tests/>. (Zugriff: 01.09.2021).
- [101] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus 3.A.*: Oldenbourg Wissenschaftsverlag, 2012. ISBN: 978-3-486-71974-1. DOI: doi:10.1524/9783486719741. URL: <https://doi.org/10.1524/9783486719741>.
- [102] Jay Jie-Bing Yu und Adam M. Fleming. *Automatic code generation via natural language processing*. 2006. URL: <https://patents.google.com/patent/US7765097B1/en>. (Zugriff: 02.09.2021).

A. Anhang

1.1. Code

```
1  class OnAddToTestCaseListenerMixin:
2      def __init__(self):
3          super().__init__()
4          self.test_case = None
5
6      def get_children(self):
7          raise NotImplementedError()
8
9      def on_add_to_test_case(self, test_case):
10         self.test_case = test_case
11
12         for child in self.get_children():
13             if isinstance(child, OnAddToTestCaseListenerMixin):
14                 child.on_add_to_test_case(test_case)
```

Listing A.1: OnAddToTestCaseListenerMixin für das Hören auf Events

```
1  def locate(self, *args, raise_exception=False, **kwargs):
2      if self.fittest_output_object is not None:
3          return self.fittest_output_object
4
5      for output_object in self.get_output_objects(*args, **kwargs):
6          if not self.output_object_is_relevant(output_object):
7              continue
8
9          keywords = self.get_keywords(output_object)
10         for keyword in keywords:
11             variations = self.get_compare_variations(output_object, keyword)
12
13             for value_1, value_2 in variations:
14                 similarity = self.get_similarity(value_1, value_2)
15
16                 if self.is_new_fittest_output(similarity, output_object,
17                     ↪ value_1, value_2):
17                     self._highest_similarity = similarity
18                     self._fittest_output_object = output_object
19                     self._fittest_keyword = keyword
20
21             if self.has_invalid_fittest_output():
22                 self._fittest_output_object = self.get_fallback()
23                 self._fittest_keyword = None
24
25             if raise_exception:
26                 raise LookoutFoundNothing()
27
28     return self.fittest_output_object
```

Listing A.2: Komplette Suche der Lookout-Klasse

```
1  class TemplateMixin:
2      template = ''
3
4      def get_template(self):
5          return self.template
6
7      def get_template_context(self, line_indent, at_start_of_line):
8          return {}
9
10     @classmethod
11     def get_indent_string(cls, indent):
12         return ' ' * indent
13
14     def to_template(self, line_indent=0, at_start_of_line=True):
15         indent = self.get_indent_string(line_indent) if
16             at_start_of_line else ''
17
18         return indent + self.get_template().format(
19             **self.get_template_context(line_indent, at_start_of_line))
```

Listing A.3: TemplateMixin von Ghengo

```
1  class Replaceable(object):
2      for_test_type = None
3      replacement_for = None
4
5      @classmethod
6      def should_be_replaced(cls, sub_class):
7          return sub_class.should_replace(cls) and
8              Settings.generate_test_type == sub_class.for_test_type
9
10     @classmethod
11     def should_replace(cls, parent):
12         return cls.replacement_for == parent
13
14     def __new__(cls, *args, **kwargs):
15         for sub_class in cls.__subclasses__():
16             if cls.should_be_replaced(sub_class):
17                 return super().__new__(sub_class)
18
19         return super().__new__(cls)
```

Listing A.4: Replaceable-Klasse von Ghengo

```
1  class ExtractorOutput:
2      def _get_output(self):
3          native_value = self.source
4          output_token = NoToken()
5
6          try:
7              if self.source_is_token:
8                  if self.source_represents_output:
9                      output_token = self.source
10                     native_value = str(self.source)
11
12                     native_value, output_token =
13                         self.token_to_native_value(self.source)
14
15
16                     if self.supports_variable_source and
17                         self.string_represents_variable(native_value):
18                         return self.handle_variable_value(native_value)
19
20                     prepared_value = self.prepare_native_value(native_value)
21                     output = self.native_value_to_output(prepared_value)
22
23                     return self.prepare_output(output)
24
25             except ExtractionError as e:
26                 self.set_output_token(NoToken())
27
28                 raise e
```

Listing A.5: Schritte der ExtractorOutput-Klasse, um nativen Wert zu erhalten

```

1  class ExtractorOutput:
2      def token_to_native_value(self, token):
3          if token.pos_ == 'ADJ' or token.pos_ == 'VERB' or token.pos_:
4              ↳ == 'ADV':
5                  if token.pos_ == 'ADV' and token.head.pos_ == 'AUX':
6                      token = token.head
7
8                  children_negated = any([token_is_negated(child) for child in
9                     ↳ get_all_children(token)])
10
11                 boolean_value = not token_is_negated(token) and not
12                     ↳ children_negated
13
14                 return boolean_value, token
15
16
17             for child in token.children:
18                 if token_canRepresent_variable(child):
19                     return str(child), child
20
21
22             next_token = get_next_token(token)
23             if is_quoted(next_token):
24                 return str(next_token)[1:-1], next_token
25
26             previous_token = get_propn_from_previous_chunk(token)
27             if previous_token:
28                 return str(previous_token), previous_token
29
30
31             raise ExtractionError(NO_VALUE_FOUND_CODE)

```

Listing A.6: Token zu nativem Wert in ExtractorOutput-Klasse

```
1  class ExtractorOutput:
2      def native_value_to_output(self, native_value):
3          if isinstance(native_value, (int, float, Decimal, bool)):
4              return native_value
5
6          value_str = str(native_value)
7          if is_quoted(value_str):
8              value_str = value_str[1:-1]
9
10         try:
11             return int(value_str)
12         except ValueError:
13             pass
14         # do the same thing for float and Decimal...
15
16         try:
17             literal_value = ast.literal_eval(value_str)
18             if isinstance(literal_value, (list, tuple, set, dict)):
19                 return literal_value
20             except (ValueError, SyntaxError):
21                 pass
22
23         bool_pos = POSITIVE_BOOLEAN_INDICATORS[self.document.lang_]
24         bool_neg = NEGATIVE_BOOLEAN_INDICATORS[self.document.lang_]
25         if value_str in bool_pos or value_str in bool_neg:
26             return value_str in bool_pos
27         return value_str
```

Listing A.7: Schritte der ExtractorOutput-Klasse, um den Output zu erhalten

```
1  class Extractor:
2      output_class = ExtractorOutput
3
4      def get_output_kwargs(self):
5          return {'source': self.source, 'document': self.document,
6                  'test_case': self.test_case}
7
8      @property
9      def output(self) -> ExtractorOutput:
10         output_class = self.output_class
11         instance = output_class(**self.get_output_kwargs())
12         return instance
13
14     def _get_output_value(self, output_instance):
15         return output_instance.get_output()
16
17     def _extract_value(self, output_instance=None):
18         if output_instance is None:
19             output_instance = self.output
20
21         try:
22             return self._get_output_value(output_instance)
23         except ExtractionError as e:
24             return GenerationWarning(e.code)
25
26     def extract_value(self):
27         return self._extract_value()
```

Listing A.8: Ausschnitt aus Extractor-Klasse

```
1  class ManyExtractorMixin:
2      child_extractor_class = Extractor
3
4      def _extract_many(self, output_instance=None, token=None):
5          output = []
6          children = get_all_children(token)
7          next_token = get_next_token(token)
8
9          for child in children:
10              if not child or self.skip_token_for_many(child):
11                  continue
12              extractor_class = self.child_extractor_class
13              output_instance = self.output
14              output_instance.source_represents_output = True
15              output_instance.source = child
16              extractor =
17                  extractor_class(**self.get_child_extractor_kwargs())
18
19              try:
20                  value = extractor._get_output_value(output_instance)
21              except ExtractionError as e:
22                  value = GenerationWarning(e.code)
23
24              output.append(value)
25
26      return output
```

Listing A.9: Wichtigste Methode der ManyExtractorMixin-Klasse

```
1  class AssertPreviousModelConverter(ModelConverter):
2      def get_document_compatibility(self):
3          compatibility = super().get_document_compatibility()
4
5          if not self.variable_ref.token:
6              compatibility *= 0.2
7
8          if token_is_plural(self.model.token):
9              compatibility *= 0.5
10
11     return compatibility
```

Listing A.10: Dokument-Kompatibilität der AssertPreviousModelConverter-Klasse

```
1  class ObjectQuerysetConverter(QuerysetConverter):
2      def get_document_compatibility(self):
3          compatibility = super().get_document_compatibility()
4          token_before_model = get_previous_token(self.model.token)
5          if not token_before_model:
6              return 0
7
8          if token_is_indefinite(token_before_model):
9              compatibility *= 0.2
10         elif token_is_definite(token_before_model):
11             pass
12         else:
13             compatibility *= 0.5
14
15         if self.model.token and token_is_plural(self.model.token):
16             compatibility *= 0.5
17
18     return compatibility
```

Listing A.11: Dokument-Kompatibilität der ObjectQuerysetConverter-Klasse

```
1  class Converter:
2      def get_reference_wrappers(self):
3          ref_wrappers = []
4
5          for token in self.get_possible_reference_name_tokens():
6              if not self.token_can_be_reference_name(token):
7                  continue
8
9              chunk = get_noun_chunk_of_token(token, self.document)
10             reference = self.search_for_reference(chunk, token)
11
12             if not self.is_valid_search_result(reference) or reference
13                 in [wrapper.reference for wrapper in ref_wrappers]:
14                 continue
15
16             wrapper = ReferenceTokenWrapper(reference=reference,
17                 token=token)
18             ref_wrappers.append(wrapper)
19
20
21         return ref_wrappers
```

Listing A.12: Finden von referenzierten Objekten und den zugehörigen Tokens im Converter

```

1  class Converter:
2      def token_can_be_reference_name(self, token):
3          token_is_blocked = any([
4              blocked_token and tokens_are_equal(blocked_token, token) for
5                  ↳ blocked_token in self._blocked_reference_tokens
6          ])
7          if token_is_blocked or self.document[0] == token:
8              return False
9
10         if token.pos_ != 'ADJ' and token.pos_ != 'NOUN' and token.pos_ != 'VERB' and token.pos_ != 'ADV':
11             if token.pos_ != 'PROPN': # proper noun / Eigenname
12                 return False
13
14         # ignore if head of proper noun is already a reference
15         if self.token_can_be_reference_name(token.head):
16             return False
17
18         # verbs with aux are fine (is done, ist abgeschlossen)
19         if (token.pos_ == 'VERB' or token.pos_ == 'ADV') and token.head.pos_
20             ↳ != 'AUX':
21             return False
22
23         # if there is a verb where the parent is a finites Modalverb (e.g.
24             ↳ sollte), it should not be an argument
25         if tokens_are_equal(token, self.last_document_word) and
26             ↳ token_is_verb(token) and token.head.tag_ == 'VMFIN':
27             return False
28
29         return True

```

Listing A.13: Algorithmus zur Bestimmung von Tokens, die Namen von Referenzen repräsentieren

```
1  class AssertPreviousModelConverter(ModelConverter):
2      def prepare_statements(self, statements):
3          statements.append(
4              Expression(Attribute(self.variable_ref.value,
5                  'refresh_from_db()')).as_statement()
6          )
7
8
9      def handle_extractor(self, extractor, statements):
10         chunk = get_noun_chunk_of_token(extractor.source,
11             self.document)
12         compare_lookout = ComparisonLookout(chunk or self.document,
13             extractor.output.output_token)
14         extracted_value = extractor.extract_value()
15
16         exp = CompareExpression(
17             Attribute(self.variable_ref.value, extractor.field_name),
18             compare_lookout.get_comparison_for_value(extracted_value),
19             Argument(extracted_value),
20         )
21         statements.append(AssertStatement(exp))
```

Listing A.14: Beispiel für die Verwendung von Transformationsklassen im Assert-PreviousModelConverter

```
1  class Chain(RecursiveValidationBase):
2      def _validate_sequence(self, sequence, index):
3          for child in self.children:
4              try:
5                  index = child._validate_sequence(sequence, index)
6              except NonTerminalNotUsed as e:
7                  raise RuleNotFulfilled()
8
9      return index
```

Listing A.15: Validierung der Token-Sequenz der Chain-Klasse

```
1  class Optional(RecursiveValidationBase):
2      def _validate_sequence(self, sequence, index):
3          try:
4              return self.child._validate_sequence(sequence, index)
5          except (RuleNotFulfilled, NonTerminalNotUsed, SequenceEnded):
6              return index
```

Listing A.16: Validierung der Token-Sequenz der Optional-Klasse

```
1  class OneOf(RecursiveValidationBase):
2      def _validate_sequence(self, sequence, index):
3          errors = []
4
5          for child in self.children:
6              try:
7                  return child._validate_sequence(sequence, index)
8              except (RuleNotFulfilled, NonTerminalNotUsed, SequenceEnded)
9                  as e:
10                     errors.append((e, child))
11
12         # if each child has thrown an error, this is invalid
13         raise RuleNotFulfilled()
```

Listing A.17: Validierung der Token-Sequenz der OneOf-Klasse

```
1  class Repeatable(RecursiveValidationBase):
2      def _validate_sequence(self, sequence, index):
3          rounds_done = 0
4          while True:
5              try:
6                  index = self.child._validate_sequence(sequence, index)
7              except (RuleNotFulfilled, NonTerminalNotUsed, SequenceEnded)
8                  as e:
9                      if isinstance(e, NonTerminalNotUsed):
10                          break_error = RuleNotFulfilled()
11                      else:
12                          break_error = e
13                      break
14          rounds_done += 1
15
16          # check if the minimum numbers of rounds were done
17          if break_error and rounds_done < self.minimum:
18              raise break_error
19
20      return index
```

Listing A.18: Validierung der Token-Sequenz der Repeatable-Klasse

```
1  class NonTerminalSymbol(RecursiveValidationBase):
2      criterion_terminal_symbol = TerminalSymbol(Token1)
3      rule = Chain([
4          criterion_terminal_symbol,
5          TerminalSymbol(Token2)
6      ])
7
8      def _validate_sequence(self, sequence, index):
9          try:
10              return self.rule._validate_sequence(sequence, index)
11          except (RuleNotFulfilled, SequenceEnded) as e:
12              if not self.used_by_sequence_area(sequence, index,
13                  e.sequence_index):
14                  raise NonTerminalNotUsed()
15          raise NonTerminalInvalid()
```

Listing A.19: Validierung der Token-Sequenz der NonTerminalSymbol-Klasse

```
1  # language: de
2  Funktionalität: Anfragen für Aufträge
3    Szenario: Auftragsliste ohne Authentifizierung
4      Wenn die Liste der Aufträge geholt wird
5      Dann sollte die Antwort keine Einträge haben
6
7    Szenario: Nutzer holt Auftragsliste
8      Gegeben sei ein Benutzer Alice mit dem Vornamen Alice
9      Und ein Auftrag mit dem Besitzer Alice und der Nummer 1
10     Wenn Alice die Liste der Aufträge holt
11     Dann sollte die Antwort einen Auftrag enthalten
12     Und der erste Eintrag sollte die Nummer 1 haben
13
14   Szenario: nur eigene Aufträge sichtbar
15     Gegeben sei ein Benutzer Alice
16     Und ein Auftrag mit dem Besitzer Alice
17     Und ein Auftrag
18     Wenn Alice die Liste der Aufträge holt
19     Dann sollte die Antwort einen Auftrag enthalten
20
21   Szenario: ohne Authentifizierung Auftrag erstellen
22     Wenn ein Auftrag mit der Nummer 3 erstellt wird
23     Dann sollte die Antwort den Status 400 haben
24     Und den Fehler "Authentifizierung erforderlich" enthalten
25     Und es sollte keine Aufträge geben
26
27   # s. Teil 2...
```

Listing A.20: Teil 1 der Feature-Datei für Schritt S-2 der Simulation

```
1  # language: de
2  Funktionalität: Anfragen für Aufträge
3  # s. Teil 1...
4
5  Szenario: Auftrag erstellen funktioniert
6      Gegeben sei ein Benutzer Bob
7      Wenn Bob einen Auftrag mit der Nummer 3 und dem Besitzer Bob
8          ↪ erstellt
9      Dann sollte ein Auftrag mit der Nummer 3 und dem Besitzer Bob
10         ↪ existieren
11     Und die Antwort sollte den Besitzer Bob und die Nummer 3 enthalten
12
13    Szenario: Auftrag löschen funktioniert
14        Gegeben sei ein Benutzer Bob
15        Und ein Auftrag 1 mit dem Benutzer Bob
16        Wenn Bob einen Auftrag 1 löscht
17        Dann sollte es keinen Auftrag geben
18
19    Szenario: Auftrag löschen nicht existierender Auftrag
20        Gegeben sei ein Benutzer Bob
21        Wenn Bob den Auftrag mit dem PK 1 löscht
22        Dann sollte die Antwort den Status 404 haben
```

Listing A.21: Teil 2 der Feature-Datei für Schritt S-2 der Simulation

```
1  from rest_framework.test import APIClient
2  from django.urls import reverse
3  import pytest
4  from django_sample_project.apps.order.models import Order
5
6  def test_job_list_without_authentication():
7      client = APIClient()
8      response = client.get(reverse('orders-list'))
9      assert len(response.data) == 0
10
11 @pytest.mark.djangoproject
12 def test_user_fetches_order_list(user_factory, order_factory):
13     alice = user_factory(first_name='Alice')
14     order_factory(owner=alice, number=1)
15     client = APIClient()
16     client.force_authenticate(alice)
17     response = client.get(reverse('orders-list'))
18     assert len(response.data) == 1
19     entry_0 = response.data[0]
20     assert entry_0.get('number') == 1
21
22 @pytest.mark.djangoproject
23 def test_only_own_orders_visible(user_factory, order_factory):
24     alice = user_factory()
25     order_factory(owner=alice)
26     order_factory()
27     client = APIClient()
28     client.force_authenticate(alice)
29     response = client.get(reverse('orders-list'))
30     assert len(response.data) == 1
31 # s. Teil 2...
```

Listing A.22: Teil 1 des generierten Codes aus Schritt S-2 der Simulation

```
1  from rest_framework.test import APIClient
2  from django.urls import reverse
3  import pytest
4  from django_sample_project.apps.order.models import Order
5
6  # s. Teil 1...
7
8  def test_create_order_without_authentication():
9      client = APIClient()
10     response = client.post(reverse('orders-detail'), {'number': 3})
11     assert response.status_code == 400
12     assert 'Authentifizierung erforderlich' in str(response.data)
13     qs_0 = Order.objects.all()
14     assert qs_0.exists()
15
16
17 @pytest.mark.django_db
18 def test_create_order_works(user_factory):
19     bob = user_factory()
20     client = APIClient()
21     client.force_authenticate(bob)
22     response = client.post(reverse('orders-detail'), {'number': 3,
23         'owner': bob.pk})
24     qs_0 = Order.objects.filter(number=3, owner=bob)
25     assert qs_0.count() == 1
26     resp_data = response.data
27     assert resp_data.get('owner') == bob.pk
28     assert resp_data.get('number') == 3
29
30 # s. Teil 3...
```

Listing A.23: Teil 2 des generierten Codes aus Schritt S-2 der Simulation

```
1  from rest_framework.test import APIClient
2  from django.urls import reverse
3  import pytest
4  from django_sample_project.apps.order.models import Order
5
6  # s. Teil 2...
7
8  @pytest.mark.djangoproject_db
9  def test_delete_order_works(user_factory, order_factory):
10     bob = user_factory()
11     order_1 = order_factory(user=bob)
12     client = APIClient()
13     client.force_authenticate(bob)
14     client.delete(reverse('orders-detail', {'pk': order_1.pk}))
15     qs_0 = Order.objects.all()
16     assert qs_0.exists()
17
18
19  @pytest.mark.djangoproject_db
20  def test_delete_order_not_existing_order(user_factory):
21     bob = user_factory()
22     client = APIClient()
23     client.force_authenticate(bob)
24     response = client.delete(reverse('orders-detail'), {'id': 1})
25     assert response.status_code == 404
```

Listing A.24: Teil 3 des generierten Codes aus Schritt S-2 der Simulation

```
1  # language: de
2  Funktionalität: Abgebrochene Aufträge
3    Szenario: Abgebrochene Aufträge nicht in Liste
4      Gegeben sei eine Benutzerin Alice
5      Und ein Auftrag, der nicht aktiv ist
6      Wenn Alice die Liste der Aufträge holt
7      Dann sollte die Liste keine Einträge haben
8
9    Szenario: Abgebrochene Aufträge nicht in Details
10   Gegeben sei eine Benutzerin Alice
11   Und ein inaktiver Auftrag 1
12   Wenn Alice den Auftrag 1 holt
13   Dann sollte die Antwort den Status 404 haben
14
15   Szenario: Abgebrochene Aufträge nicht löschen
16   Gegeben sei eine Benutzerin Alice
17   Und ein Auftrag 1, der nicht aktiv ist
18   Wenn Alice den Auftrag 1 löscht
19   Dann sollte die Antwort den Status 404 haben
```

Listing A.25: Feature-Datei für Schritt S-3 der Simulation

```
1 import pytest
2 from rest_framework.test import APIClient
3 from django.urls import reverse
4
5 @pytest.mark.django_db
6 def test_aborted_jobs_not_in_the_list(user_factory, order_factory):
7     alice = user_factory()
8     order_factory(active=False)
9     client = APIClient()
10    client.force_authenticate(alice)
11    response = client.get(reverse('orders-list'))
12    assert len(response.data) == 0
13
14 @pytest.mark.django_db
15 def test_canceled_orders_not_in_details(user_factory, order_factory):
16     alice = user_factory()
17     order_1 = order_factory(more_inactive=True)
18     client = APIClient()
19     client.force_authenticate(alice)
20     response = client.get(reverse('orders-detail', {'pk': order_1.pk}))
21     assert response.status_code == 404
22
23 @pytest.mark.django_db
24 def test_canceled_jobs_cannot_be_deleted(user_factory, order_factory):
25     alice = user_factory()
26     order_1 = order_factory(active=False)
27     client = APIClient()
28     client.force_authenticate(alice)
29     response = client.delete(reverse('orders-detail', {'pk':
30         ↵ order_1.pk}))
31     assert response.status_code == 404
```

Listing A.26: Generierter Code der Simulation vor Implementierung (Schritt S-3)

```
1 import pytest
2 from rest_framework.test import APIClient
3 from django.urls import reverse
4
5 @pytest.mark.djangoproject_db
6 def test_aborted_jobs_not_in_the_list(user_factory, order_factory):
7     # identisch zu vorher...
8
9 @pytest.mark.djangoproject_db
10 def test_canceled_orders_not_in_details(user_factory, order_factory):
11     alice = user_factory()
12     order_1 = order_factory(active=True)    # <-- korrekt
13     client = APIClient()
14     client.force_authenticate(alice)
15     response = client.get(reverse('orders-detail', {'pk': order_1.pk}))
16     assert response.status_code == 404
17
18 @pytest.mark.djangoproject_db
19 def test_canceled_jobs_cannot_be_deleted(user_factory, order_factory):
20     # identisch zu vorher...
```

Listing A.27: Generierter Code der Simulation nach Implementierung (Schritt S-3)

```
1  # language: de
2  Funktionalität: Produkte und Aufträge
3  Szenario: Produkte löschen
4      Gegeben sei ein Produkt 1 mit dem Namen "Briefmarke"
5      Und ein Benutzer Bob
6      Wenn Bob das Produkt 1 löscht
7      Dann sollten keine Produkte existieren
8
9  Szenario: Produkte erstellen
10     Gegeben sei ein Benutzer Bob
11     Wenn Bob ein Produkt mit dem Namen "Briefmarke" erstellt
12     Dann sollte ein Produkt existieren
13
14 Szenario: Produkte hinzufügen
15     Gegeben sei ein Benutzer Bob
16     Und ein Produkt 1
17     Und ein Produkt 2
18     Und ein Auftrag 1 mit den Produkten 1 und 2
19     Und ein Produkt 3 mit dem Namen "Briefmarke"
20     Wenn Bob dem Produkt 3 zu Auftrag 1 hinzufügt
21     Dann sollte der Auftrag 1 die Produkte 1, 2 und 3 haben
```

Listing A.28: Feature-Datei für Schritt S-4 der Simulation

```
1 import pytest
2 from rest_framework.test import APIClient
3 from django.urls import reverse
4 # TODO: the import for the following values either was not found or does not
5 #       exist. If the import does
6 #       exist in your code, you have to set the value manually. If it does exist,
7 #       try creating the
8 #       import in your code first.
9 Product = None    # <- fix
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31 # s. Teil 2...
```

Listing A.29: Teil 1 des generierten Codes der Simulation vor Implementierung (Schritt S-4)

```
1 import pytest
2 from rest_framework.test import APIClient
3 from django.urls import reverse
4 # TODO: the import for the following values either was not found or does not
5 #       exist. If the import does
6 #       exist in your code, you have to set the value manually. If it does exist,
7 #       try creating the
8 #       import in your code first.
9 Product = None    # <- fix
10
11 @pytest.mark.djangoproject
12 def test_add_products(user_factory, product_factory, order_factory):
13     bob = user_factory()
14     product_1 = product_factory()
15     product_factory()
16     order_1 = order_factory(products=product_1)
17     product_3 = product_factory(names='Briefmarke')
18     client = APIClient()
19     client.force_authenticate(bob)
20     client.post(reverse('product-detail', {'pk': product_3.pk}), {'order':
21                   product_1})
22     order_1.refresh_from_db()
23     assert order_1.products == product_1
```

Listing A.30: Teil 2 des generierten Codes der Simulation vor Implementierung (Schritt S-4)

```
1  import pytest
2  from rest_framework.test import APIClient
3  from django.urls import reverse
4  from django_sample_project.apps.order.models import Product
5
6
7  @pytest.mark.djangoproject_db
8  def test_delete_products(product_factory, user_factory):
9      product_1 = product_factory(name='Briefmarke')
10     bob = user_factory()
11     client = APIClient()
12     client.force_authenticate(bob)
13     client.delete(reverse('products-detail', {'pk': product_1.pk}))
14     qs_0 = Product.objects.all()
15     assert qs_0.exists() is False
16
17
18  @pytest.mark.djangoproject_db
19  def test_create_products(user_factory):
20      bob = user_factory()
21      client = APIClient()
22      client.force_authenticate(bob)
23      client.post(reverse('products-detail'), {'name': 'Briefmarke'})
24      qs_0 = Product.objects.all()
25      assert qs_0.count() == 1
26
27
28  # s. Teil 2...
```

Listing A.31: Teil 1 des generierten Codes der Simulation nach Implementierung (Schritt S-4)

```
1 import pytest
2 from rest_framework.test import APIClient
3 from django.urls import reverse
4 from django_sample_project.apps.order.models import Product
5
6
7 # s. Teil 1...
8
9 @pytest.mark.django_db
10 def test_add_products(user_factory, product_factory, order_factory):
11     bob = user_factory()
12     product_1 = product_factory()
13     product_2 = product_factory()
14     order_1 = order_factory()
15     order_1.products.add(product_1)
16     order_1.products.add(product_2)
17     product_3 = product_factory(name='Briefmarke')
18     client = APIClient()
19     client.force_authenticate(bob)
20     client.post(reverse('products-add', {'pk': product_3.pk}), {'order':
21         order_1.pk})
22     order_1.refresh_from_db()
23     assert list(order_1.products.all()) == [product_1, product_2, product_3]
```

Listing A.32: Teil 2 des generierten Codes der Simulation nach Implementierung (Schritt S-4)

```
1  # language: de
2  Funktionalität: Berechtigungen
3  Szenario: Produkt erstellen mit Berechtigung
4      Gegeben sei ein Benutzer Bob mit der Benutzerberechtigung "Kann
5          ↪ Produkt erstellen"
6      Wenn Bob ein Produkt erstellt
7          Dann sollte die Antwort den Status 200 haben
8
9
10 Szenario: Produkt erstellen ohne Berechtigung
11     Gegeben sei ein Benutzer Bob
12     Wenn Bob ein Produkt erstellt
13     Dann sollte die Antwort den Status 400 haben
14     Und die Antwort sollte den Fehler "Ihnen fehlen die Berechtigungen"
15         ↪ enthalten
16
17 Szenario: Produkt bearbeiten mit Berechtigung
18     Gegeben sei ein Benutzer Bob mit der Benutzerberechtigung "Kann
19         ↪ Produkt bearbeiten"
20     Und ein Produkt 1
21     Wenn Bob in Produkt 1 den Namen "Briefmarke" setzt
22     Dann sollte die Antwort den Status 200 haben
23
24 Szenario: Produkt bearbeiten ohne Berechtigung
25     Gegeben sei ein Benutzer Bob
26     Und ein Produkt 1
27     Wenn Bob in Produkt 1 den Namen "Briefmarke" setzt
28     Dann sollte die Antwort den Status 400 haben
29     Und die Antwort sollte den Fehler "Ihnen fehlen die Berechtigungen"
30         ↪ enthalten
```

Listing A.33: Feature-Datei für Schritt S-5 der Simulation

```
1 import pytest
2 from django.contrib.auth.models import Permission
3 from rest_framework.test import APIClient
4 from django.urls import reverse
5
6
7 @pytest.mark.djangoproject_db
8 def test_create_product_with_authorization(user_factory):
9     bob = user_factory()
10    bob.user_permissions.add(Permission.objects.filter(
11        content_type__model='product',
12        content_type__app_label='order',
13        codename='add_product'
14    ))
15    client = APIClient()
16    client.force_authenticate(bob)
17    response = client.post(reverse('products-detail'))
18    assert response.status_code == 200
19
20
21 @pytest.mark.djangoproject_db
22 def test_create_product_without_authorization(user_factory):
23     bob = user_factory()
24     client = APIClient()
25     client.force_authenticate(bob)
26     response = client.post(reverse('products-detail'))
27     assert response.status_code == 400
28     assert 'Ihnen fehlen die Berechtigungen' in str(response.data)
29
30 # s. Teil 2...
```

Listing A.34: Teil 1 des generierten Codes der Simulation (Schritt S-5)

```
1  # s. Teil 1...
2
3
4  @pytest.mark.djangoproject_db
5  def test_edit_product_with_authorization(user_factory, product_factory):
6      bob = user_factory()
7      bob.user_permissions.add(Permission.objects.filter(
8          content_type__model='product',
9          content_type__app_label='order',
10         codename='add_product'
11     ))
12     product_1 = product_factory()
13     client = APIClient()
14     client.force_authenticate(bob)
15     response = client.put(reverse('products-detail', {'pk':
16         ↪ product_1.pk}), {'name': 'Briefmarke'})
17     assert response.status_code == 200
18
19
20  @pytest.mark.djangoproject_db
21  def test_edit_product_without_authorization(user_factory,
22      ↪ product_factory):
23      bob = user_factory()
24      product_1 = product_factory()
25      client = APIClient()
26      client.force_authenticate(bob)
27      response = client.put(reverse('products-detail', {'pk':
28          ↪ product_1.pk}), {'name': 'Briefmarke'})
29      assert response.status_code == 400
30      assert 'Ihnen fehlen die Berechtigungen' in str(response.data)
```

Listing A.35: Teil 2 des generierten Codes der Simulation (Schritt S-5)

```
1  # language: de
2  Funktionalität: Datenstrukturen
3  Szenario: Variante 1
4      Gegeben sei ein Benutzer Bob
5      Wenn Bob einen Auftrag mit einem Item mit der Produkt-ID 1 und der
6          ↵ Menge 1 erstellt
7          Dann sollte die Antwort den Status 200 haben
8          Und es sollte ein Item mit der Auftrags-ID 1 und der Menge 1
9          ↵ existieren
10
11     Szenario: Variante 2
12     Gegeben sei ein Benutzer Bob
13     Wenn Bob einen Auftrag mit den Items '[{"quantity": 1, "product_id": 1}]' erstellt
14         ↵ Dann sollte die Antwort den Status 200 haben
15         Und es sollte ein Item mit der Auftrags-ID 1 und der Menge 1
16         ↵ existieren
```

Listing A.36: Feature-Datei für Schritt S-6 der Simulation

```

1  import pytest
2  from rest_framework.test import APIClient
3  from django.urls import reverse
4  from django_sample_project.apps.order.models import Item
5
6  class GenerationWarning001:
7      TEXT = 'No value was found for this field. One reason might be that
     ↵ the field does not exist on the ' \
8          'model and therefore it is harder to determine the value of
     ↵ the field. You can try to write ' \
9          'the value after the field name. Like: `Given an order with a
     ↵ number "123"`. '
10
11 @pytest.mark.djangoproject_db
12 def test_variant_1(user_factory):
13     bob = user_factory()
14     client = APIClient()
15     client.force_authenticate(bob)
16     response = client.post(reverse('orders-detail'), {'items':
     ↵ GenerationWarning001(), 'product_id': 1, 'quantity': 1})
17     assert response.status_code == 200
18     qs_0 = Item.objects.filter(order_id=1, quantity=1)
19     assert qs_0.count() == 1
20
21 @pytest.mark.djangoproject_db
22 def test_variant_2(user_factory):
23     # davor identisch zu Variante 1...
24     response = client.post(reverse('orders-detail'), {'items':
     ↵ [{'quantity': 1, 'product_id': 1}]})
25     # danach identisch zu Variante 1...

```

Listing A.37: Generierter Code der Simulation (Schritt S-6)

1.2. Produktionsregeln

```


$$\begin{aligned}
S &= [LANG], [FEAT], "eof" \\
FEAT &= [TAGS], "feature", DESCs, [BACK], (\{RULE\} + | \{SC_DEF\}+) \\
RULE &= [TAGS], "rule", DESCs, [BACK], \{SC_DEF\}+ \\
BACK &= "background", DESCs, \{GIVEN\}+ \\
SC_DEF &= SCENARIO | OUTLINE \\
SCENARIO &= [TAGS], "scenario", DESCs, STEPS \\
OUTLINE &= [TAGS], "outline", DESCs, STEPS, \{EXAMPLES\}+ \\
EXAMPLES &= [TAGS], "examples", DESCs, TABLE \\
STEPS &= G_STEPS | W_STEPS | T_STEPS \\
G_STEPS &= \{GIVEN\}+, \{WHEN\}, \{THEN\} \\
W_STEPS &= \{WHEN\}+, \{THEN\} \\
T_STEPS &= \{THEN\}+ \\
GIVEN &= "given", STEP_END, \{AND | BUT\} \\
WHEN &= "when", STEP_END, \{AND | BUT\} \\
THEN &= "then", STEP_END, \{AND | BUT\} \\
AND &= "and", STEP_END \\
BUT &= "but", STEP_END \\
STEP_END &= DESC, [DOC | TABLE] \\
TABLE &= "dataTable", "eol", \{"dataTable", "eol"\}+ \\
DOC &= "docString", "eol", \{DESC\}, "docstring", "eol" \\
DESCs &= \{DESC\} + | ("eol", \{DESC\}) \\
DESC &= "description", "eol" \\
TAGS &= \{"tag\}+, "eol" \\
LANG &= "language", "eol"
\end{aligned}$$


```

Produktionsregeln A.1: Regeln für Gherkin (abgeleitet aus [36])

1.3. Abbildungen

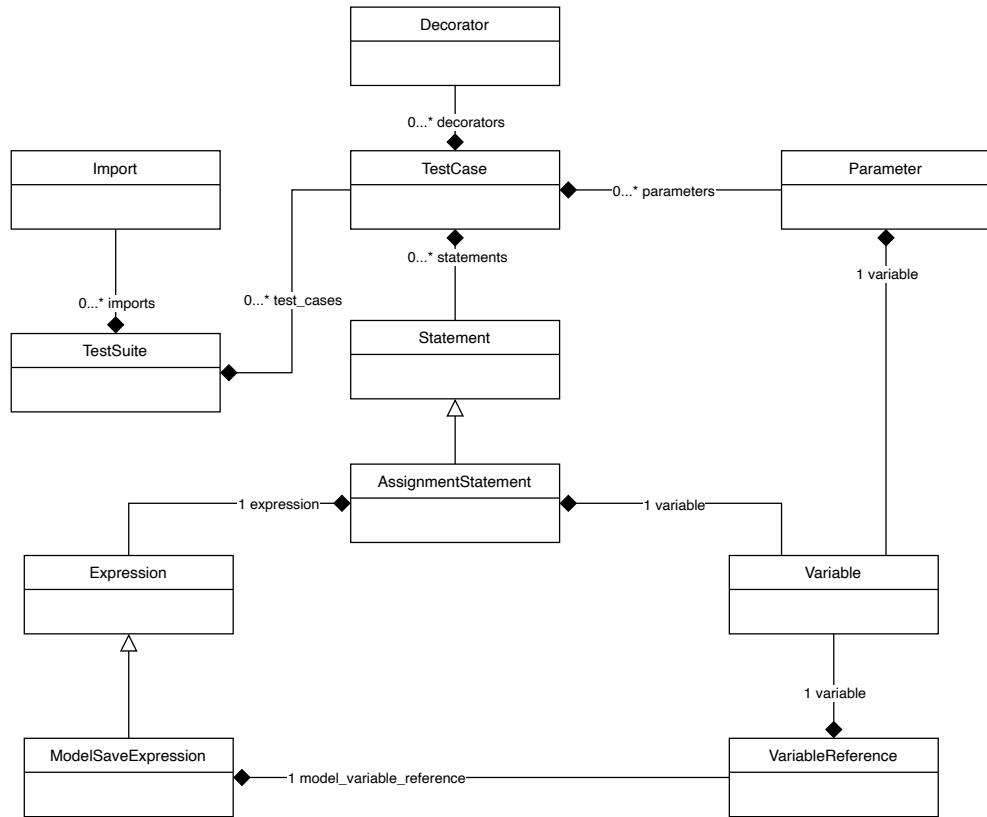


Abbildung A.1: Ausschnitt aus Klassendiagramm des Output-ASTs von Ghengo
(eigene Darstellung)

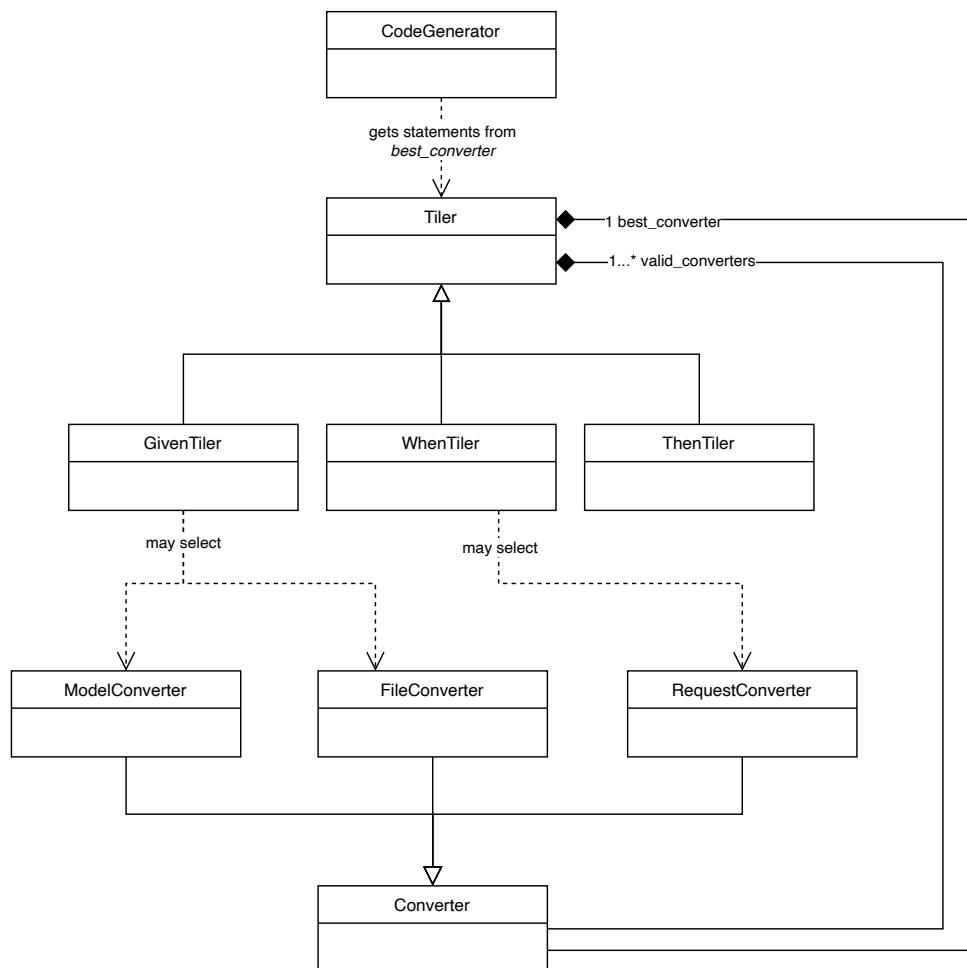


Abbildung A.2: Ausschnitt aus Klassendiagramm für Zusammenspiel Code-Generator, Tiler und Converter (eigene Darstellung)

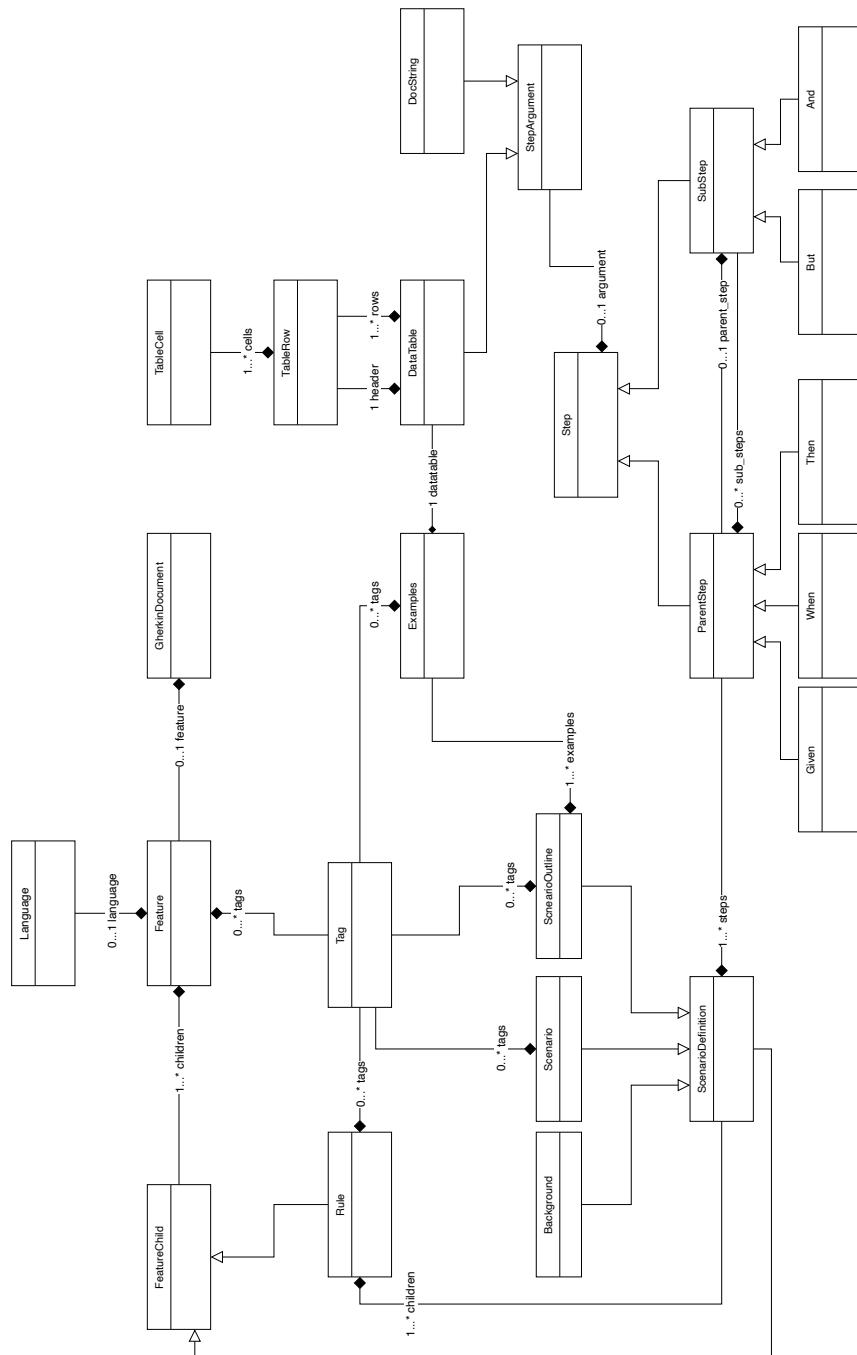


Abbildung A.3: Vereinfachtes Klassendiagramm des ASTs von Gherkin (eigene Darstellung, inspiriert durch [34])

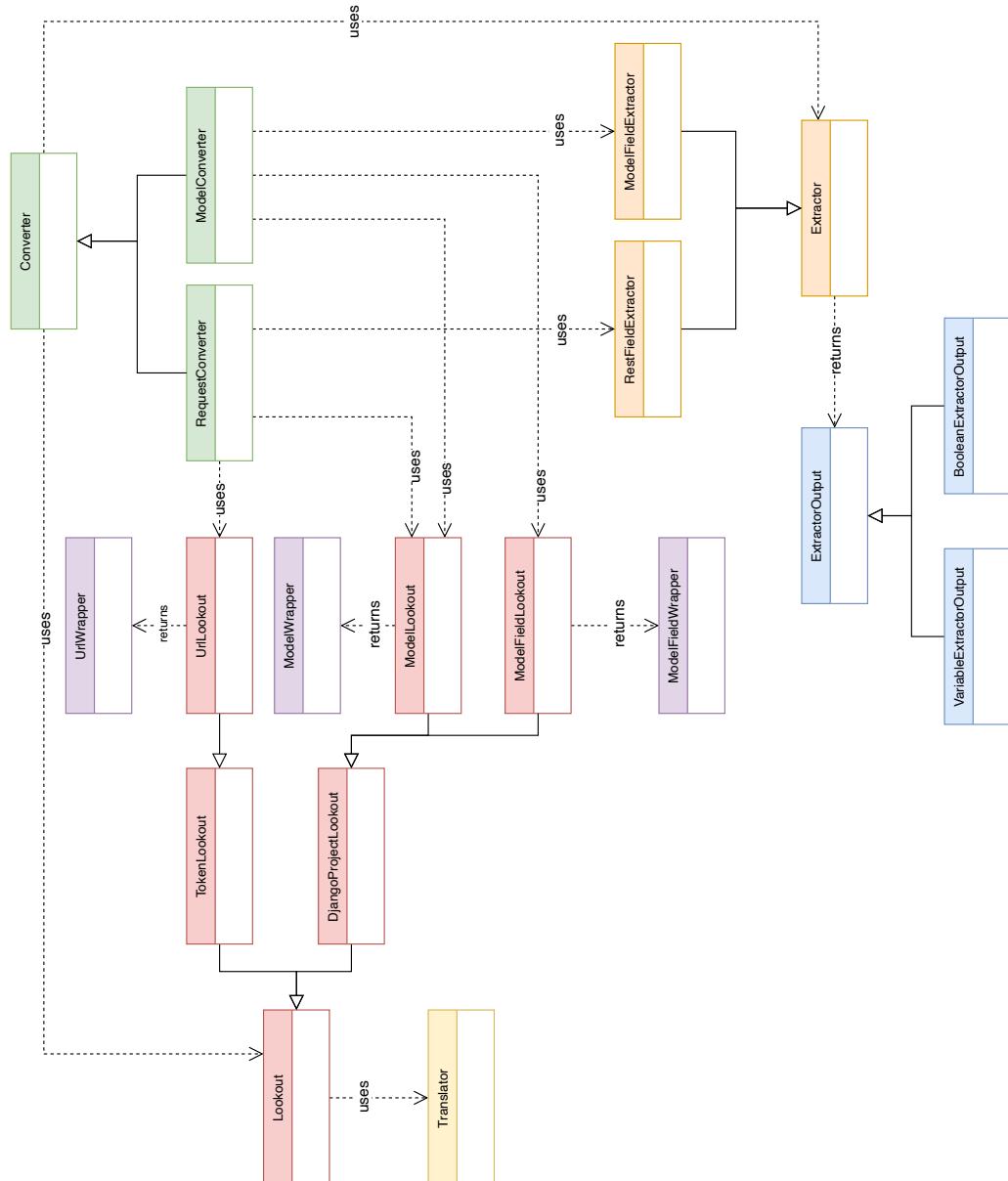


Abbildung A.4: Klassendiagramm des Model- und RequestConverters (eigene Darstellung)