

Listing 2.2: Beispiel Anfragekörper zu einem invaliden Helden

```

1 {
2   "name": "UncleBob",
3   "experience_level": "Legendary",
4   "race": "Elfen",
5   "culture": "Auelfen",
6   "profession": "Söldner",
7   "talents": {
8     "Körperbeherrschung": 2,
9     "Kraftakt": 3,
10    "Selbstbeherrschung": 4,
11    "Zechen": 5,
12    "Menschenkenntnis": 3,
13    "Überreden": 3,
14    "Orientierung": 4,
15    "Wildnisleben": 3,
16    "Götter & Kulte": 3,
17    "Kriegskunst": 6,
18    "Sagen & Legenden": 5,
19    "Handel": 3,
20    "Heilkunde Wunden": 4
21  },
22  "combat_techniques": {
23    "Armbrüste": 10,
24    "Rauen": 10,
25    "Stangenwaffen": 9,
26    "Zweihandschwerter": 10
27  },
28  "advantages": [
29    ["Begabung", "Singen", 1],
30    ["Begabung", "Musizieren", 1],
31    ["Beidhändig", "", 1],
32    ["Dunkelsicht", "", 2]
33  ],
34  "disadvantages": [
35    ["Körpergebundene Kraft", "", 1],
36    ["Lästige Mindergeister", "", 1],
37    ["Wahrer Name", "", 1]
38  ]
39 }

```

Listing 2.3: Beispiel Antwort zu Listing 2.2 auf S. 23

```

1 {
2   "valid": false,
3   "errors": [
4     {
5       "type": "missing_level",
6       "message": "Heros 'profession' is missing minimum level '3' for 'talent' of
7       'Körperbeherrschung'.",
8       "parameter": {
9         "caused_feature": "profession",
10        "caused_feature_value": "Söldner",
11        "referred_feature": "talent",
12        "referred_feature_value": "Körperbeherrschung",
13        "min_level": 3
14      }
15    },
16   "warnings": []
17 }

```

2.5 Regelwerk-Interpretation

Dieses Kapitel setzt Kenntnisse im Bereich *LP* voraus, insb. die Syntax und Semantik von *Clingo*. Eine Einstieghilfe liegt mit Anhang A.2 auf S. 60, auch für die Mitwirkenden im quelloffenen Aufbewahrungsort (englisch: *repository*), vor. Es basiert auf den ebenso Praxis-orientierten „*Potassco User Guide*“ [61].

Die Formalisierung des Basisregelwerks zu *RPG-P&P-DSA* mit *ASP*, unter Berücksichtigung möglicher weiterer Regelbücher und -erweiterungen, ist mit der Realisierung dessen Fokus dieser Arbeit. Dazu werden die verschiedenen Regelvarianten ausgearbeitet und anschließend, im Sinne eines „Kochbuches“ rezeptartig, formal beschrieben.

2.5.1 Konventionen

- KV.1** Es wird unterschieden zwischen den Charakter-Fakten, Spielwelt-Fakten, Spielwelt-Regeln und Ergebnis-Fakten. Fakten sind immer *Clingo-Funktionen*. Spielwelt-Regeln sind immer *Clingo-Regeln*, welche Ergebnis-Fakten ableiten.
- KV.2** Bei Fakten ist als erstes Argument das Merkmal zu platzieren, welches das „Verursachende“ ist, sofern vorhanden. Verursachend gilt jenes Merkmal, dessen Ausprägung die Regel festlegt, wie z. B., dass eine bestimmte andere Merkmalsausprägung benötigt wird. Letzteres ist damit das referenzierte Merkmal. Dies macht es möglich die dazugehörige Regel schneller im echten Regelbuch nachzuschlagen.
- KV.3** Merkmale werden als *Clingo-Konstante* oder *-Funktion* formuliert. Bei Letzterem ist das erste Argument immer der Name der Merkmalsausprägung. Diese sind immer in Form eines *Clingo-Textes* (also in Hochkommata), da Leerzeichen und Sonderzeichen möglich sind. Dies macht es einfach die Merkmalsausprägung im echten Regelbuch wiederzufinden.
- KV.4** *Clingo-Variable*-Namen sind prägnant, also möglichst kurz, aber unverwechselbar. Abkürzungen sind erlaubt, wenn sie sich direkt aus dem verwendeten Atom ableiten lassen.
- KV.5** Funktionen mit (mindestens) zwei Argumenten, im Speziellen Spielwelt-Regeln und Ergebnis-Fakten, sind so zu lesen/verstehen, dass der „Faktname“ die Beziehung zwischen den beiden Argumenten (meist Merkmale) beschreibt. Semantisch wird aus „`relates_to(caused_feature, referred_feature)`“ dann „`caused feature relates to referred feature`“.

1 %% Spielwelt-Fakt:

2 `has_usual(race("Elfen"),culture("Auelfen")).` %% Spezies 'Elfen' hat üblich Kultur 'Auelfen'

3 %% Ergebnis-Fakt:

4 `missing_usual(race(R),culture(C)).` %% Spezies <R> fehlt die übliche Kultur <C>

- KV.6** Um Missverständnisse vorzubeugen ist bei der Verwendung des Operators „=“ auf folgendes zu achten: wenn es als Vergleich (auf Gleichheit) genutzt wird, ist der Operator zwischen Leerzeichen zu platzieren (Beispiel: Punkt AR.1 auf S. 27); wenn dieser hingegen als Vereinigung (englisch: *unification*) genutzt wird, also wie eine Variablenzuweisung zu verstehen ist, dann werden vor und nach dem Operator keine Leerzeichen gesetzt (Beispiel: Punkt HR.3 auf S. 28).
- KV.7** Innerhalb von *Clingo-Funktionen* sind bei der Auflistung der Argumente keine Leerzeichen zu setzen, sodass schnell zwischen einer Auflistung von Argumenten oder Bedingungen unterschieden werden kann.

2.5.2 Eingabemodell (Kontext): der Charakter

Clingo bietet verschiedene Möglichkeiten an, wie man einem *LP* Informationen/Werte bereitstellen kann. Zum einen kann ein *LP* um weitere *LPs* erweitert werden; direkt aus dem *LP* mit der Direktive „#include“ oder dynamisch über die Python-API mit „Control.load(...)²⁵. Dieses *LP* könnte zur Laufzeit mit den Werten des Charakters generiert werden. Zum anderen besteht die Möglichkeit zur Laufzeit mit „Control.add(...)²⁶ ein sog. *Programmteil* (englisch: *program part*) (Direktive „#program“) als Zeichenfolge hinzuzufügen, welche entsprechend die Charakter-Fakten enthält.

Die beste Variante, da diese die loseste Kopplung, bezogen auf das sonst in Python notwendige Wissen über die Syntax von Clingo, bietet, ist das Nutzen der „externen Funktionen“ (englisch: *external functions*). Damit können aus einem *LP* Python-Funktionen, welche im Rahmen eines Kontext-Objektes bereitgestellt wurden, aufgerufen werden. Diese liefern dann entsprechende Clingo-Objekte zurück.

Das Ziel, also Endprodukt, ist ein definiertes formales Modell der Charakter-Fakten; dieses wird nachfolgend kurz beschrieben und in Listing 2.4: Merkmalsfakten eines Charakters auf S. 25 syntaktisch vorgestellt: Jedes Charakter-Merkmal ist ein eigener Fakt, dargestellt als *Funktion* mit dem Charakter-Merkmal als Namen und mindestens als erstes Argument den Namen der Merkmalsausprägung (als Text). Daneben bekannt sind die Argumente „Stufe“ (als Ganzzahl) und eine Referenz auf andere Merkmalsausprägungen (Name als Text). Entsprechend des Charakter-Schemas vom Schnittstellenvertrag (Kapitel 2.4.1: Schemata auf S. 20) können bestimmte Merkmalsfakten mit unterschiedlichen Ausprägungen (Argumenten) mehrfach vorkommen.

Listing 2.4: Merkmalsfakten eines Charakters

```

1 %% Strukturelle Darstellung mit Konventions-konformen          1 %% Beispiel, wobei bestimmte Fakten
2   ↪ Argumenten-Variablen-Namen                                ↪ mehrfach vorkommen dürfen z.B. Talente
2 experience_level(EL).                                         2 experience_level("Average").
3 race(R).                                                       3 race("Elfen").
4 culture(C).                                                    4 culture("Auelfen").
5 profession(P).                                                 5 profession("Söldner").
6 talent(T,LVL). %% LVL := Stufe (englisch für 'level')      6 talent("Kraftakt",3). talent("Zechen",3).
7 combat_technique(CT,LVL).                                     7 combat_technique("Armbrüste",10).
8 advantage(A,USES,LVL). %% USES := Referenz (Text)           8 advantage("Begebung","Singen",1).
9 disadvantage(DA,USES,LVL).                                    9 disadvantage("Wahrer Name","",1).
10 %% weitere Ähnliche wie Eigenschaften,                         ↪ Sonderfertigkeiten, Zauber und Liturgien möglich
    ↪

```

²⁵<https://potassco.org/clingo/python-api/5.6/clingo/control.html#clingo.control.Control.load>

²⁶<https://potassco.org/clingo/python-api/5.6/clingo/control.html#clingo.control.Control.add>

2.5.3 Validierungsvoraussetzung

Die Validierung eines Charakters hat eine wichtige Voraussetzung: Alle angegebenen Merkmalsausprägungen müssen unter den verwendeten Regelbüchern bekannt sein.

- VV.1** Für diese Überprüfung ist es erforderlich, dass Regelbücher bekannte Merkmalsausprägungen deklarieren. Dabei kann und sollte ein Regelbuch eine Abhängigkeit zu einem Anderen definieren, um dessen deklarierte Merkmalsausprägungen wiederzuverwenden und nicht doppelt zu deklarieren.

```

1 %% Eigene Existenz bekannt machen.
2 rulebook("dsa5_aventurisches_götterwirken_2").
3 %% Anforderung im Regelbuch 'DSA5 Aventurisches Götterwirken II':
4 rulebook_depends("dsa5_aventurisches_götterwirken_2", "dsa5").
5 %% Allgemeine Überprüfung der Abhängigkeiten:
6 rulebook_missing(RB,D) :- rulebook(RB), rulebook_depends(RB,D), not rulebook(D).

```

- VV.2** Deklariert werden Merkmalsausprägungen mit dem Präfix „known_“ und den Merkmalsnamen. Die Stufen bei z. B. Talenten und Kampftechniken werden nicht deklariert, da diese beliebig sein können; hingegen sind diese bei Vor- und Nachteilen nicht beliebig und daher zu deklarieren.

```

1 known_profession("Händler").
2 known_combat_technique("Armbrüste";"Raufen").
3 known_advantage("Dunkelsicht","",,(1..2)).
4 known_advantage("Begabung",("Singen";"Musizieren"),1).

```

- VV.3** Diese somit bekannten Merkmalsausprägungen werden dann überprüft.

```

1 %% Finde unbekannte Ausprägungen
2 unknown(profession(P)) :- profession(P), not known_profession(P).
3 unknown(combat_technique(CT)) :- combat_technique(CT,_), not known_combat_technique(CT).
4 unknown(advantage(A,USES,LVL)) :- advantage(A,USES,LVL), not known_advantage(A,USES,LVL).

```

2.5.4 „Kochbuch“ zur Regelformalisierung

Wie bereits in Kapitel 1.2: Grundlagen auf S. 3 beschrieben, gibt es nach [24] folgende Regelkategorien: Grundregel, optionale Regel, Fokusregel und Hausregel. Hausregeln können nicht betrachtet werden, da diese unter den Spielenden frei definierbar sind. Aufgrund des sonst entstehenden Umfangs dieser Arbeit wird folgendes nicht beachtet: optionale Regeln, Fokusregeln und Grundregel bezogen auf AP oder Charakter-Eigenschaften bzw.

-Basiswerte. Beachtet werden Grundregeln außerdem nur mit Relevanz bei der Charakter-Erstellung. In Kapitel 4: Diskussion und Ausblick (Punkt AB.4 auf S. 50) wird ein Ausblick auf die Verwaltung gegeben.

Die beachteten Regeln werden gruppiert nach dem Anwendungsbereich bzw. Einhaltungsgrad: allgemeingültig (Regelbuch-übergreifend) sowie Regelbuch-spezifisch hart (immer einzuhalten) und weich (Abweichung nach Absprache mit der Spielleitung möglich [vgl. 30, Schritt 4 und 6]). Wie bereits in Kapitel 2.4.2: Endpunkte, wird „Regelbuch“ als Synonym für Regelbuch-Erweiterung benutzt.

Das Kochbuch setzt sich aus Rezepten zusammen, welche, beispielhaft für einen Fall, konkrete Regeln formalisiert. Dabei wird die Regel (Ableitung eines Atoms), als auch die notwendige Modellierung (Fakten) aufgezeigt. Ein Beispiel endet immer mit der Regel.

Allgemeingültige Regeln:

Jeder Charakter gehört nur einer Spezies, Kultur und Profession an [vgl. 30, Schritt 4 und **AR.1** 6]. Wobei weitere Regelbücher diese Begrenzung auflockern könnten.

```

1 max_count(culture,1). %% nur Fakt mit höchstem Wert entscheidend
2 max_count_exceeded(culture,MAX) :- COUNT=#count{C:culture(C)}, MAX=#max{MC:max_count(culture,MC)},
   → COUNT > MAX.

```

Die Mindeststufe für alle Fertigkeiten, wenn aktiviert (also angegeben), ist null [vgl. 30, **AR.2** Schritt 8]. Somit sind negative Zahlen ausgeschlossen. Es ist nicht vorgesehen, dass andere Regelbücher diese Grenze verschieben.

```
1 missing_min_lvl(talent(T,LVL),MIN) :- talent(T,LVL), MIN=0, LVL < MIN.
```

Der gewählte Erfahrungsgrad des Charakters begrenzt die maximalen Stufen von Eigenschaften, Fertigkeiten (Talente, Sonderfertigkeiten) und Kampftechniken, die maximale Gesamtanzahl an Eigenschaftspunkten, die maximale Anzahl an Zauber und Liturgien sowie davon die Anzahl an Fremdzauber. Wobei nach Charakter-Erstellung nur noch die maximalen Stufen gelten. [vgl. 30, Schritt 2, 5 und 8]

```

1 %% Model des Erfahrungsgrades: known_experience_level(<name>,<ES>,<FS>,<KTS>,<EP>,<ZL>,<FZ>).
2 %% ES := max. Eigenschaftsstufe %% EP := max. Eigenschaftspunkte
3 %% FS := max. Fertigkeitsstufe %% ZL := max. Anzahl Zauber und Liturgien
4 %% KTS := max. Kampftechniksstufe %% FZ := davon max. Fremdzauber
5 known_experience_level("Average",13,10,10,98,10,1).
6 %% Finde Talente, welche die maximale Stufengrenze überschreiten
7 max_lvl_exceeded(talent(T,LVL),MAX) :- experience_level(EL),
   → known_experience_level(EL,-,MAX,-,-,-), talent(T,LVL), LVL > MAX.

```

Regelbuch-spezifische harte Regeln:

- HR.1** Eine Merkmalsausprägung benötigt eine Ausprägung eines anderen Merkmals, das nur eine Ausprägung haben kann [vgl. 80]. Das geforderte Merkmal hat keine dynamische Stufe.

```

1 %% Eine bestimmte Spezies ist gefordert. Spezies kann nur eine Ausprägung haben.
2 requires(culture("Auelfen"),race("Elfen")).
3 unusable_by(culture(C),race(R)) :- culture(C), requires(culture(C),race(_)), race(R),
   ↳ not requires(culture(C),race(R)).

```

- HR.2** Eine Merkmalsausprägung benötigt eine Ausprägung eines anderen Merkmals, das mehrere Ausprägungen haben kann [vgl. 32]. Das geforderte Merkmal hat keine dynamische Stufe. Der entscheidende Unterschied bei der Formalisierung zur vorherigen Regel ist, dass die genutzte *Variable* für die geforderte Ausprägung aus der Forderung kommt und nicht vom Charakter-Fakt.

```

1 %% Ein bestimmter Vorteil ist gefordert. Es kann mehrere Vorteile geben.
2 requires(race("Elfen"),advantage("Zauberer","",1)).
3 missing(race(R),advantage(A,USES,LVL)) :- race(R), requires(race(R),advantage(A,USES,LVL)),
   ↳ not advantage(A,USES,LVL).

```

- HR.3** Eine Merkmalsausprägung benötigt eine Ausprägung eines anderen Merkmals, das mehrere Ausprägungen haben kann, auf eine Mindeststufe [vgl. 81].

```

1 requires(profession("Söldner"),talent("Kriegskunst",6)).
2 %% logisches 'oder' durch '#count': das Merkmal fehlt oder die Stufe ist nicht erreicht
3 missing_level(profession(P),talent(T,MIN_LVL)) :- profession(P),
   ↳ requires(profession(P),talent(T,MIN_LVL)),
   ↳ 1 = #count{ x: not talent(T,_); x: talent(T,LVL), LVL < MIN_LVL }.

```

- HR.4** Eine Merkmalsausprägung benötigt aus einer Liste eine Anzahl von Ausprägungen eines anderen Merkmals, das mehrere Ausprägungen haben kann, auf eine Mindeststufe [vgl. 81]. Aufgrund der leichteren Implementierung, Wartbarkeit und Verständlichkeit wird die Zählung jener Ausprägungen, welche die Mindeststufe erreicht haben, in Python umgesetzt. Dessen Entwurf wird mit Listing 2.5 aufgezeigt. So kann die Auswahlliste als Tuple modelliert werden.

```

1 %% 'Söldner' benötigt aus mehreren Möglichen eine (!) Kampftechnik auf Stufe 10
2 requires(profession("Söldner"),any_of(1,combat_[])
   ↳ technique,("Hiebwaffen","Schwerter","Stangenwaffen"),10)).
3 missing_level(profession(P),combat_technique(any_of(CHOICES,CTS),MIN_LVL)) :- profession(P),
   ↳ requires(profession(P),any_of(CHOICES,combat_technique,CTS,MIN_LVL)),
   ↳ CHOICES > @count_by("combat_techniques",CTS,MIN_LVL).

```

Listing 2.5: Entwurf der count_by Methode (Python Pseudo-Code)

```

1 from clingo import Symbol, Number
2 ## Method der Clingo-Kontext-Klasse, welche als Klassenfeld den Helden hat (self.hero)
3 def count_by(self, feature: Symbol, options: Symbol, min_lvl: Symbol) -> Symbol:
4     """
5         :return: count of feature values ('options') of 'feature' passing minimum level
6         """
7     # dynamically get class field (with 'getattr') instead of manual mapping with a switch-case
8     # requires 'feature.string' to be exactly the field name of the actual hero model
9     values_lvl: dict[str, int] = {fv.name: fv.level for fv in getattr(self._hero, feature.string)}
10    # count all features having the minimum level
11    passed = sum(1 for opt in options.arguments if min_lvl.number <= values_lvl.get(opt.string, 0))
12    return Number(passed)

```

Regelbuch-spezifische weiche Regeln:

Eine Merkmalsausprägung definiert eine Ausprägung eines anderen Merkmals für üblich **WR.1** [vgl. 30, Schritt 4 und 6; 32].

```

1 has_usual(race("Elfen"), culture("Auelfen")) .
2 missing_usual(race(R), culture(C)) :- race(R), has_usual(race(R), culture(_)), culture(C),
   ↳ not has_usual(race(R), culture(C)).

```

Eine Merkmalsausprägung definiert eine Ausprägung eines anderen Merkmals als typisch **WR.2** oder untypisch [vgl. 32]. Es werden beide Regeln in einem Beispiel gezeigt.

```

1 has_typical(race("Elfen"), advantage("Dunkelsicht", "", 2)) .
2 missing_typical(race(R), advantage(A, USES, LVL)) :- race(R),
   ↳ has_typical(race(R), advantage(A, USES, LVL)), not advantage(A, USES, LVL).
3
4 has_atypical(race("Elfen"), disadvantage("Blutrausch", "", 1)) .
5 atypical(race(R), disadvantage(DA, USES)) :- race(R), has_atypical(race(R), disadvantage(DA, USES, _)),
   ↳ disadvantage(DA, USES, _).

```

Anmerkung

Wie zu sehen ist, ist die Struktur der formalen Regeln in **WR.1** und **WR.2** gleich. Diese kann auch auf die Semantik von „empfohlen“ und „ungeeignet“ übertragen werden, weshalb sie nicht zusätzlich aufgeführt werden [vgl. 82].

Darüber hinaus wird im Regelwerk der Einhaltungsgrad der weichen Regeln mit „dringend empfohlen“ verschärft dargestellt, aber wie bisher sind Abweichungen nach Absprache erlaubt [vgl. 32]. Letzteres hat daher lediglich eine Relevanz bei der Auswertung/Interpretation der Ergebnis-Fakten.

2.5.5 Interpretation des *ASP*-Ergebnisses

Wie in Abbildung 2.1 gezeigt wird, muss das Ergebnis vom *LP* (konkret vom *Clingo-Solver*) interpretiert werden. Dies ist die Aufgabe der Software, konkreter die Aufbereitung der Ergebnis-Fakten zur Verwendung im Frontend. Typischerweise sucht man mit *ASP* mögliche Lösungen für ein Problem; hier hingegen nicht: es werden keine möglichen Charaktere (*Answer Set*) unter Beachtung gewählter Regelbücher (Problem) gesucht, sondern für einen gegebenen Charakter mögliche Regelverstöße. Diese Regelverstöße müssen mit ihren Informationen gesammelt werden. Dies geht nur, wenn das *LP* erfüllbar (englisch: *satisfiable*) ist, da es bei einer Unerfüllbarkeit kein *Answer Set* zum Interpretieren gibt. Daher leitet das zuvor beschriebene „Kochbuch“ immer konkrete Atome als Regelverstöße ab. Das Nutzen von z. B. „integrity constraints“ ist daher nicht möglich. Dadurch bleibt, im Sinne von *ASP*, das Problem erfüllbar und das *LP* gibt für einen Charakter ein *Answer Set* an möglichen Regelverstößen zurück.

Die für die Interpretation relevanten Atome sind entsprechend dem Erfüllungsgrad (hart: muss, weich: kann) als Validierungsfehler oder -warnung zu interpretieren. So gehören zu den Fehlern die Atome:

- `unknown`
- `max_count_exceeded`
- `max_lvl_exceeded`
- `unusable_by`
- `missing_min_lvl`
- `missing_level`

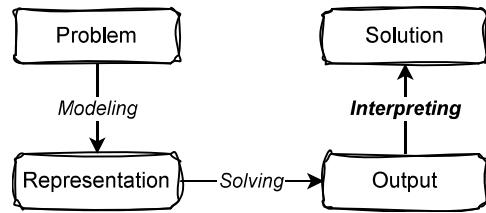
Warnungen sind:

- `missing_usual`
- `missing_typical`
- `atypical`

Diese Interpretation wird nicht in die Namen der Atome aufgenommen, damit die Applikation (und nur diese) ohne weiteren Aufwand darüber entscheiden kann und somit z. B. wartungsarm Gruppen hinzufügen, entfernen und ändern kann.

Mehr dazu im Kapitel 3.4: Die wichtigsten Komponenten auf S. 39.

Abbildung 2.1: Deklarative Problemlösung
(Anlehnung an [83, Seite 14])



3 | Realisierung

Mit diesem Kapitel wird aufgezeigt wie die Realisierung der Software, unter den Anforderungen und Vorgaben der vorangegangen konzeptionellen Betrachtung, erfolgte und welche Entscheidungen dabei getroffen worden sind. Zunächst wird der Aufbau des quelloffenen Aufbewahrungsortes und die Architektur der Software erklärt. Anschließend folgt eine Darstellung der getroffenen Design-Entscheidungen und der wichtigsten Komponenten der Umsetzung. Abgeschlossen wird mit den qualitätssichernden Maßnahmen und der Betrachtung von etwaigen Schwierigkeiten während der Implementierung der Software.

3.1 Projektstruktur

Die Projektstruktur definiert den Aufbau des Projekts und damit z. B. die Ablageorte für bestimmte Dateien und wie, mit welchen Abhängigkeiten, das Projekt zu bauen ist. Es ist also maßgeblich für die Orientierung im quelloffenen Projekt und ist für den ersten Punkt der abgeleiteten technischen Anforderungen auf S. 10 relevant. Nachfolgend wird Bezug auf die erste Ebene in Abbildung 3.1 auf S. 32 genommen.

Eine der wichtigsten Bestandteile ist die sog. „lies mich Datei“ (Anhang A.3 auf S. 68). **PS.1** Diese wird oft von Betreibern quelloffener Aufbewahrungsorte automatisch als Einstiegsseite angezeigt und ist damit das „Aushängeschild“ des Projekts. Inhalte sind u. a. eine Kurzbeschreibung des Projektes, wie man mitwirken und Probleme melden kann sowie ggf. Lizenz- und Kontakt-Hinweise. [vgl. 84]

Von dort wird auf weitere Dokumentationen verwiesen. Diese befinden sich unter „docs“. **PS.2** Dazu gehört z. B. eine Anleitung wie man eine lokale Entwicklungsumgebung aufsetzt und die Software startet, aber auch wie man konkret Regelbücher erstellt.

Diese Regelbücher, genauer die entsprechenden Clingo-LPs, und ggf. künftig weitere, zur **PS.3** Laufzeit relevante, Ressourcen (nicht Python-Dateien) befinden sich unter „resources“.

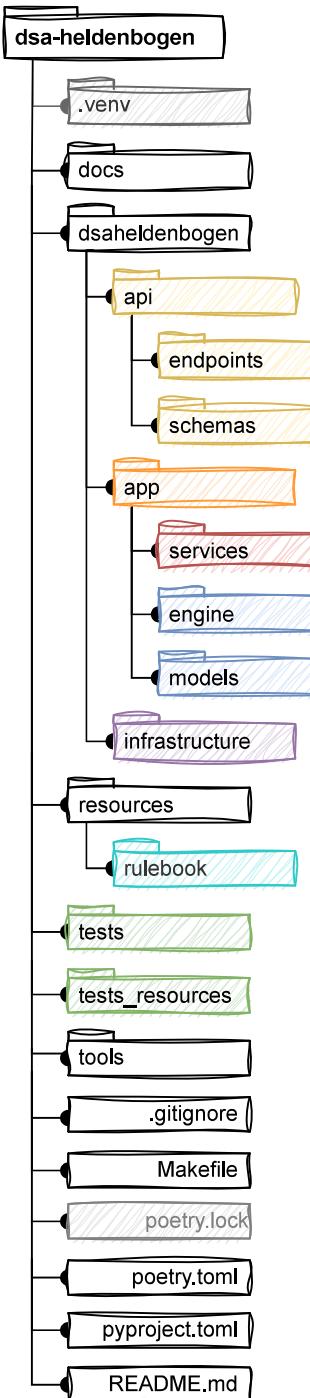
Die zweitwichtigste Datei ist das sog. „Makefile“ (Anhang A.4 auf S. 69). Es dient **PS.4** zur Automatisierung häufiger Aufgaben bzw. Schritte von Mitwirkenden, wie z. B. das Installieren von Abhängigkeiten, Durchführen aller Tests und Starten des Backends. [vgl. 85, Kapitel 1]

- PS.5** Zur Wahrung der Portabilität und Vorbeugung von Versionskonflikten mit bereits installierten Abhängigkeiten, wird eine sog. „virtuelle Umgebung“ geschaffen; diese liegt unter „.venv“. Darin werden, anstatt global, Projekt-spezifisch die Abhängigkeiten installiert. Genutzt wird dazu das sehr verbreitete, funktionsreiche und leicht zu bedienende poetry²⁷. Es kann u. a. auch die zu verwendende Python-Version steuern, falls weitere installiert sind, und die Software für eine mögliche Verteilung, z. B. als ausführbare Datei oder Container²⁸, verpacken. Projekt-Meta-Informationen und Abhängigkeiten werden in „pyproject.toml“ dokumentiert (siehe Anhang A.5 auf S. 70). Poetry wertet dies aus und erzeugt eine „poetry.lock“ Datei, welche die exakt geladenen Versionen der Abhängigkeiten festhält.
- PS.6** Hilfreiche Werkzeuge und Skripte für die Entwicklung werden unter „tools“ abgelegt. Während der Entwicklung hat es sich z. B. bewährt, Ideen zur Formalisierung und Modellierung zunächst auf eine sog. „grüne Wiese“, also ohne eventuell hinderliche Einflüsse der eigentlichen Implementierung, auszuprobieren. Dazu liegt entsprechend ein ausführbares Python-Programm bereit, welches ein leeres *LP* ausführt. Der Quellcode der eigentlichen Implementierung kann dabei nach Bedarf genutzt werden; so kann z. B. das Modell des Charakters weiterhin genutzt werden.
- PS.7** Dieser Quellcode, also das Python-Backend inkl. Schnittstelle, befindet sich unter „dsaheldenbogen“. Dies ist das oberste Python-Package der Software und trägt, entsprechend dem gängigen Standard, den Namen des Projekts. Jegliche Art von Tests sind in „tests“ abgelegt; mehr dazu in Punkt AT.2 auf S. 34.

²⁷<https://github.com/python-poetry/poetry>

²⁸z. B. als Docker: <https://www.docker.com/>

Abbildung 3.1
Projektstruktur und -architektur
(Eigene Darstellung)

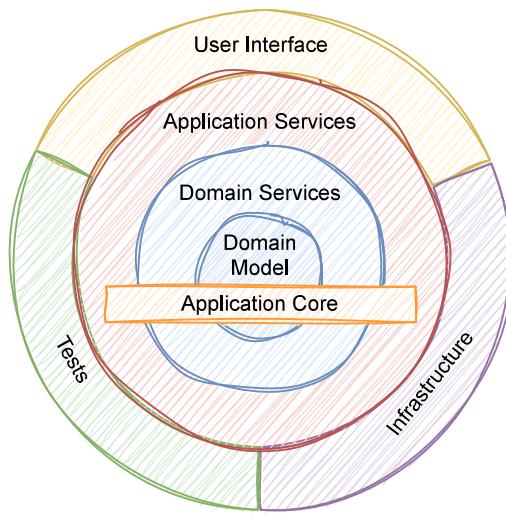


Zur Versionsverwaltung des Quellcodes wird das verbreitete und quelloffene Git²⁹ eingesetzt. Mit der Datei „.gitignore“, werden Pfade und Dateien, welche z. B. durch Frameworks oder die eingesetzte Entwicklungsumgebung erstellt werden und keine Relevanz für die Software selbst haben, als zu ignorieren markiert. Dazu gehören „.venv“ und „poetry.lock“. Zur Veröffentlichung der Software wird die größte und bekannteste Git-Plattform GitHub³⁰ genutzt. PS.8

3.2 Architektur

Die Architektur ist das Fundament des Quellcodes und stellt damit eines der wichtigsten technischen Merkmale dar, um eine qualitative und wartbare Software zu erstellen [vgl. 86, Kapitel 15]. Zwei sehr bekannte und von den Prinzipien ähnliche sind „Clean Architecture“ [86, Kapitel 22] und „Onion Architecture“ [87]. Gewählt wird Letzteres, da es bereits das (Standard-)Pattern „Domain-driven Design“ [88] gut integriert.

Abbildung 3.2: Onion Architecture
(Anlehnung an [87])



Es fokussiert sich auf eine möglichst lose Kopplung zwischen den Schichten und das „Zentrieren“ der fachlichen Logik in die Mitte (siehe Abbildung 3.2). Dabei ist es fundamental, dass jede Schicht nur von Tieferen abhängig sein darf. Die Idee ist, umso tiefer die Schicht ist, desto seltener kommen Änderungen vor, wodurch weniger Wartungsaufwand bei den äußeren Schichten entstehen soll. Dies macht die innerste Schicht „Domain Model“ zur stabilsten. Durch *Dependency Inversion Principle (DIP)*³¹ wird es inneren Schichten ermöglicht indirekt äußere Schichten zu verwenden. Es ist dadurch notwendig, dass ein Mechanismus bereitsteht, wodurch innere Klassen zur Laufzeit Äußere einbinden können. [vgl. 87]

Das „Dependency Injection Pattern“ ist ein solcher Mechanismus [vgl. 73, Seite 195 bis 212; 89].

²⁹<https://git-scm.com/>

³⁰<https://github.com/bjoern-nowak/dsa-heldenbogen>

³¹https://clean-code-developer.de/die-grade/gelber-grad/#Dependency_Inversion_Principle_DIP

Nachfolgendes bezieht sich insb. auf den eingefärbten Teil der Abbildung 3.1 auf S. 32:

- AT.1** Sämtliche Packages der äußereren Schichten und der Applikationsschicht befinden sich direkt unter „dsaheldenbogen“. Die „Application Services“ Schicht und „Domain“ Schichten wurden unter „app“ gebündelt, um eine bessere Übersicht und flachere Struktur zu erhalten. Außerdem dokumentieren die Packages mit „docstring“ ihre Schicht-Zugehörigkeit.
- AT.2** Eine Ausnahme sind die Tests. Diese werden nicht zur Laufzeit benötigt und werden daher von der eigentlichen Applikation getrennt. Das `tests` Package ist in weitere Packages unterteilt; konkret spiegelt es die Package-Struktur von `dsaheldenbogen` wider und wird daher nicht zusätzlich abgebildet. Der Vorteil daran ist, dass schnell erkannt werden kann in welchem Kontext der Test stattfindet. Das Package `tests_resources` verhält sich ähnlich; es spiegelt die Struktur der `resources` ab und enthält Test-exklusive Regelbücher-*LPs*.
- AT.3** Alle Regelbücher-*LPs* befinden sich unter „`resources/rulebook`“ in separaten Unterordnern. Die Applikation lädt für ein Regelbuch jeweils immer nur eine *LP*-Datei als Einstiegspunkt, welche direkt unterhalb des Unterordners sein und den Namen „`_entrypoint.lp`“ tragen muss. Es erlaubt jedem Regelbuch den Aufbau einer eigenen internen Struktur. In der Architektur lassen sich die *LPs* schwer einordnen. Einerseits gehören sie zur Domänen-Modellierung, insb. die allgemeingültigen Regeln. Andererseits ist es kein Python-Code und wird oft unter Veränderungen stehen. Auch stellt es, aufgrund der Einordnung des Clingo-Frameworks in die Infrastruktur-Schicht, ein Implementierungsdetail der eigentlichen `Engine` (im Sinne von *DIP*) dar. Daher werden sie als eine der äußersten Schichten angesiedelt betrachtet.
- AT.4** Die Schnittstelle liegt unter „`api`“ und definiert zusätzlich, neben den Domänenmodells, eigene entsprechende Schemata. Dies ist nicht nur konform zur Architektur, sondern bietet insb. die Möglichkeit die Strukturen und Datentypen so zu wählen, dass diese möglichst einfach für die *API*-Benutzenden (z. B. das Frontend) zu nutzen sind.
- AT.5** Konkrete Endpunkte werden unter „`api/endpoints`“ getrennt nach der Domäne abgelegt und spiegeln im Dateinamen die Domäne wider. So stellt der Charakter eine Domäne dar. Künftig könnte ebenfalls der Benutzende eine Domäne sein, über dessen Endpunkte ein Benutzer, wie beim Charakter, erstellt, bearbeitet und gelöscht werden könnte. Die Funktionalität des An- und Abmeldens hingegen gehört der Applikationsdomäne „`root`“ selbst an.

Anfragen eines Endpunktes resultieren immer in einem Aufruf eines Applikationsservices. AT.6 Die API-Schicht übernimmt dabei die Übersetzung des Schemas zu und vom Domänenmodell. Die Applikationsservices enthalten entsprechend des „Integration Operation Segregation Principle“³² (IOSP) keine eigene Logik, sondern steuern, mit dem Aufruf anderer Klassen und Methoden, den Ablauf wie die gewünschte Funktionalität erreicht wird. Sie stellen damit eine „Integration“ dar; Methoden mit Logik hingegen eine „Operation“.

Das Package „engine“ enthält Klassen, welche solche „Operation“-Methoden bereitstellen. AT.7 Die gleichnamige Klasse ist, neben der Ausführung und Auswertung von LPs, Kern der Applikation. Mehr dazu in Kapitel 3.4: Die wichtigsten Komponenten auf S. 39.

Unter „infrastructure“ befindet sich der Python-Code, welcher als Schnittstelle zwischen AT.8 der Engine und Clingo-API fungiert. Hier soll die Konvertierung des Charakter-Domänen-Modells zum Clingo-Kontext und die erste Übersetzung der Ergebnis-Fakten in ein Domänenmodell passieren, sodass, entsprechend der Zielarchitektur, bei Änderungen dieses Teils der Applikationskern unberührt bleibt.

Entsprechend der in Clean-Code-Developer aufgeführten (und hier weit ausgelegten) Prinzipien „Keep it simple, stupid“³³ (KISS) und „You Ain´t Gonna Need It“³⁴ (YAGNI), wird die vorgegebene Architektur nicht strikt/blind gefolgt, sondern dort abgewichen, wo es der Wartbarkeit zuträglich ist. Dies ist, auf Ebene der Klassen, der Fall bei der Anwendung von DIP und stellt eine Verletzung der fundamentalen Regel von Onion Architecture dar. Es wird also auf einen gewissen Grad loser Kopplung verzichtet, aber Klassen weiterhin ihrer Schicht zugeordnet. So kann dies bei Bedarf durch steigende Anforderungen nachgeholt werden. Grund ist die Annahme, dass für die Gewinnung von Mitwirkenden eine komplexe und aufwändige Architektur eher hinderlich ist und bei einer (noch) so kleinen Software (wie dieser) der Mehrwert strikten Folgens zu gering ist.

3.3 Design-Entscheidungen

Wie auch bei der Architektur, nur in diesem Fall noch stärker, sind Design-Entscheidungen subjektive Maßnahmen mit dem Ziel einen übersichtlichen, verständlichen und wartbaren Quellcode zu entwickeln. Dies geht von Vorgaben zum Quellcode-Stil (z. B. der Länge von Einrückungen und Position von Klammern) über Namenskonventionen bis hin zu Entwurfsmustern (englisch: *design patterns*). Nachfolgend werden jene vorgestellt, welche der Autor verfolgt hat:

³²https://clean-code-developer.de/die-grade/roter-grad/#Integration_Operation_Segregation_Principle_IOSP

³³https://clean-code-developer.de/die-grade/roter-grad/#Keep_it_simple_stupid_KISS

³⁴https://clean-code-developer.de/die-grade/blauer-grad/#You_Aint_Gonna_Need_It_YAGNI

- DE.1** In diesem Stadium der Software werden für bestimmte Aufgaben keine Vorgaben zur Anwendung bestimmter gängiger Entwurfsmuster, wie z. B. den sog. „Gang of Four (GOF) Design Patterns“ aus [90], von dieser Arbeit gemacht. Dies soll, entsprechend dem Motivationsfaktor f (Freude) auf S. 9, Freiheiten bei der Mitwirkung gewährleisten.
- DE.2** Entsprechend des zweiten Punktes der abgeleiteten technischen Anforderungen auf S. 10, wird der offizielle „Python Style Guide“ *PEP-8* [91] als Grundlage definiert.
- DE.3** Entgegen diesem wird die maximale Zeilenlänge auf 130 angehoben. Die damalige Begründung sehr kurzer Zeilen lag darin, dass man mehrere Dateien nebeneinander öffnen und vollständig betrachten könne. Die zum Zeitpunkt der Veröffentlichung existierenden technischen Einschränkungen (Bildschirmgröße und -auflösung) sind jedoch heutzutage längst überholt. Außerdem ist der Autor davon überzeugt, dass bei zu vielen gleichzeitig betrachteten Dateien der Fokus verloren geht und dessen Notwendigkeit ein Zeichen schlechter „Separation of Concerns“³⁵ (SoC) ist. Des Weiteren sei unter Verwendung der Namenskonvention und Vermeidung von Abkürzungen die sonst sehr beschränkte Zeile schnell aufgebraucht, welches zu unschönen (Lesefluss-störenden) Zeilenumbrüchen führe.
- DE.4** Weiter unterscheidet *PEP-8* nicht zwischen einfachen und doppelten Anführungszeichen bei Zeichenketten, empfiehlt jedoch die konsistente Anwendung einer frei-wählbaren Regel. In diesem Projekt werden einfache Anführungszeichen verwendet, wenn die Zeichenkette eine Identifikation (Schlüssel) darstellt, wie z. B. der Name einer Merkmalsausprägung, und Doppelte bei freien Texten, wie Fehlermeldungen.
- DE.5** Python-Dateien sind sog. Module, können entsprechend alles enthalten (Variablen, Funktionen, Klassen und auszuführende Befehle) und gruppieren üblicherweise Funktionalitäten eines Kontextes. Dabei können Module von anderen Modulen in Gänze oder in Teile importiert werden. Zusätzlich des eigentlichen Zwecks, werden Klassen stets in separate Module geschrieben (Klassen-Modul). Dies soll die Übersichtlichkeit (Vermeidung von sehr langen Dateien) und Auffindbarkeit von Klassen stärken. [vgl. 92, Kapitel 6]
- DE.6** Das Importieren von Sub-Modulen in Packages, zur Verkürzung von Import-Befehlen und damit der Verschleierung der Implementierungsstruktur, wird vermieden [vgl. 92, Kapitel 5.4.2]. Als Grund ist zu nennen, dass diese Software keine „öffentliche Bibliothek/Framework“ darstellt und die Verwendung daher (beim Importieren) nicht vereinfacht werden muss. Bei reinen Klassen-Modulen wird dies jedoch akzeptiert, da dies Redundanz bei den Import-Befehlen reduziert, da der Klassename i.d.R. dem Modulnamen gleicht. Eine Ausnahme sind Exceptions, aufgrund der oft kurzen Definition. Es ist erlaubt diese in einem Modul, eines ggf. übergeordneten Packages, namens „exceptions.py“ oder „errors.py“ zu sammeln. Dies ist in der Python-Gemeinschaft bei Bibliotheken und Frameworks üblich.

³⁵https://clean-code-developer.de/die-grade/orangener-grad/#Separation_of_Concerns_SoC

Neben den konzeptionellen Konventionen für *LP* aus Kapitel 2.5.1 auf S. 24, werden noch **DE.7** zwei Design-Entscheidungen vorgeschlagen: Kommentare, welche einen nicht temporär aus-kommentierten Quellcode darstellen, wie Dokumentationen, werden mit einem zusätzlichen (doppelten) Kommentar-Zeichen (%%) gekennzeichnet.

Die zweite Design-Entscheidung zu *LPs* ist, dass es keine Zeichenbegrenzung für Zeilen **DE.8** in *LPs* gibt. Damit ist eine schnelle und gute Übersicht des *LP* (bei ausgeschaltetem *IDE* „auto-wrap“) gegeben, da zumeist die wichtigen Informationen am Anfang stehen und zumal sich die Anwendung von Fakten und Regeln hintereinander wiederholt. Bei konkremtem Lesebedarf können Zeilen natürlich temporär, also nicht in die Versionsverwaltung übertragen, umgebrochen werden.

Die strukturelle Freiheit bei der Implementierung von Regelbüchern aus Punkt AT.3 auf **DE.9** S. 34, wird durch eine derzeitig vorgegebene Grundstruktur gering eingeschränkt. Diese schlägt eine Trennung der Spielwelt-Regeln (`rules.lp`) von Regelbuch- und Spielwelt-Fakten (`meta.lp`) vor. (vgl. Kapitel 3.5.1 auf S. 42)

Weiter wird die Freiheit durch die, der **Engine** (siehe nachfolgendes Kapitel), vorgegebene **DE.10** Mindestaufteilung in Unterprogramme verringert. Dadurch wird eine feinere Steuerung und Trennung von irrelevanten Programmteilen für die jeweilige Aufgabe, z. B. das Auflisten bekannter Merkmalsausprägungen, ermöglicht. Die Tabelle 3.1 auf S. 38 gibt eine Übersicht dieser.

Dies ist auch erforderlich, um insb. bei der Charakter-Validierung die Ausgabe von (nicht relevanten) Folgefehlern, also kaskadierenden Spielwelt-Regel-Verletzungen, zu beschränken. Diese Unterprogramme werden Charakter-Validierungsschritte genannt. Pro Charakter-Merkmal gibt es i.d.R. zwei solcher Validierungsschritte: Zuerst die Prüfung, ob die Ausprägung/en genutzt werden kann/können, also ob durch die bereits gewählten anderen Merkmale oder einer Spielwelt-Regel, wie die Überschreitung einer maximalen Stufe, ein Regelverstoß vorliegt. Und anschließend wird geprüft, ob alle Voraussetzungen der Ausprägung/en erfüllt sind, z. B. ob noch eine andere Merkmalsausprägung fehlt.

Jedem Validierungsschritt wird eine Nummer zugeteilt, welche im Namen des Unterprogramms enthalten ist. Die Ausführungsreihenfolge der Schritte wird durch diese Nummer ermittelt, beginnend mit der Kleinsten. Die Standard-Schritte werden mit einem Abstand von 50 Nummern definiert und der Erste fängt bei 50 an (vgl. Anhang A.6 auf S. 71).

Dadurch können Regelbücher weitere (Zwischen-)Validierungsschritte frei definieren. Da-zu müssen sie in einem „meta“-Unterprogramm (`#program meta.`) einen Fakt mit der Schrittnummer (Beispiel mit 175: `extra_hero_validation_step(175).`) definieren und ein entsprechendes Unterprogramm (`#program validate_hero_step_175.`) bereitstellten.