Carl von Ossietzky
Universität
Oldenburg

Bachelorarbeit
Wirtschaftsinformatik

Design und Implementierung einer webbasierten und quelloffenen Heldenverwaltung für das Rollenspiel „Das Schwarze Auge“ (Backend)

Wissenschaftliche Arbeit zur Erlangung
des akademischen Grades Bachelor of Science
vorgelegt am 22. Mai 2023 von:

Björn Nowak
geboren 10.07.1993 in Bremen
(*3562757*)

Erstprüferin:

Prof. Dr.-Ing. Astrid Nieße

Professorin für Digitalisierte Energiesysteme
Carl von Ossietzky Universität Oldenburg
Fakultät II, Department für Informatik

Zweitprüfer und Betreuer:

Dr. rer. nat. Martin Tröschel

Gruppenleiter „Verteilte Künstliche Intelligenz“
OFFIS e. V. - Institut für Informatik
FuE-Bereich Energie

Digitalisierte Energiesysteme

OFFIS

Kurzzusammenfassung

Die Problemstellung bzw. die Motivation dieser Arbeit ist es aufzuzeigen wie das Regelwerk des erfolgreichsten deutschen Stift-und-Papier-Rollenspiels *"Das Schwarze Auge"* in der aktuellen 5ten-Ausgabe so formalisiert werden kann, dass ein möglichst wartbares und quelloffenes Hilfsmittel für die Charakter-Erstellung und perspektivisch eine Basis für die Charakter-Verwaltung entsteht. Das Regelwerk zeichnet sich insb. durch den Umfang und die Komplexität der Regeln und Kombinationen der Charakter-Merkmale und Merkmalsausprägungen aus sowie der Vielzahl an Regelwerkserweiterungen. Aufgrund dessen hat die Spielenden-Gemeinschaft Hilfsmittel entwickelt, um die Aufgabe der Charakter-Erstellung und -Verwaltung zu vereinfachen. Keines der bisher verbreiteten Hilfsmittel, welches Charakter-Validierungen durchführt, unterstützt die aktuellste Ausgabe. Oft wurde als Grund die weitreichenden notwendigen Anpassungen an der Software und die fehlenden Kapazitäten dafür genannt. In dieser Arbeit werden zunächst die Rahmenbedingungen für quelloffene Projekte erörtert und ein geeignetes Formalisierungsverfahren samt Framework und Programmiersprache ermittelt. Anschließend wird ein „Kochbuch" aufgestellt, wie bestimmte Arten von Regeln des Regelwerks, unter festgelegten Konventionen, formalisiert werden könnten. Abschließend wird dieses Konzept als Proof-of-Concept umgesetzt und beides kritisch bewertet. Es wird zum Ergebnis kommen, dass eine Formalisierung möglich und einfach ist sowie die gut durchdachte Integration dessen in eine Software als Hilfsmittel eine wartbare quelloffene Lösung für die Gemeinschaft ist.

Abstract

The problem and motivation of this work is to demonstrate how the rulebook of the most popular pen-and-paper role-playing game in Germany, *"Das Schwarze Auge"*, can be formalized in its latest 5th edition, creating a maintainable, open-source tool for character creation and ultimately also character management. The rulebook is known for its complexity by the combination of character traits and trait characteristics as well as the extensive number of rulebook extensions. As a result, the community has developed tools to simplify the task of character creation and management. However, none of the widely used tools, doing character validation, support the latest edition due to self-reported extensive required customizations and lack of development capacity to tackle this. This paper discusses the general conditions for open-source projects and identifies a suitable formalization procedure, including the framework and programming language. It provides a „cookbook" outlining how to formalize different rules of the rulebook under specified conventions. Finally, the paper implements a proof-of-concept and evaluates both critically. It concludes that a formalization is achievable and quite simple, and the well-designed integration of this into a software created a great maintainable and open-source solution for the community.

Inhaltsverzeichnis

Abkürzungen

API	Application Programming Interface
AP	Abenteuerpunkt*e
ASP	Answer Set Programming
DB	Datenbank
DIP	Dependency Inversion Principle
DLP	Disjunktive Logikprogrammierung
DLV	DataLog with Disjunction
DPLL	Davis-Putnam-Logemann-Loveland
DSA	“Das Schwarze Auge“
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LP	Logik-Programm
OOP	Objekt-orientierte Programmierung
OS	Betriebssystem
P&P	“Pen & Paper“
Potassco	POTsdam Answer Set Solving COllection
Prolog	PROgrammation en LOGique
RPG	Rollenspiel
SAT	boolean SATisfiability problem
YAML	YAML Ain’t Markup Language (ursprünglich “Yet Another Markup Language“)

1 Einleitung

Das erfolgreichste bzw. bekannteste deutsche *“Pen & Paper“ (P&P)-Rollenspiel (RPG)* ist *“Das Schwarze Auge“ (DSA)*. Es ist im Jahr 1984 erschienen und befindet sich aktuell in der 5ten-Ausgabe aus 2015 [vgl. 1; 2]. Ein *P&P-RPG* zeichnet sich insb. dadurch aus, dass die Teilnehmenden physisch am selben Ort sind und vor allem mit Stift und Papier gearbeitet wird. Grundsätzlich wird in einem *P&P-RPG* eine Geschichte von fiktiven Charakteren durch die Teilnehmenden interaktiv konstruiert und erzählt. Eine teilnehmende Person besetzt die Rolle der Spielleitung; die Restlichen sind Spielende. Die Spielleitung führt anhand einer fertig beschriebenen Spielwelt und einem passenden Regelwerk die Spielenden durch die Geschichte. Die Spielenden besetzen jeweils die Rolle eines Charakters und beschreiben diesen auf einem sog. Charakterbogen, auf dem seine Merkmale, wie u. a. dessen Beruf und Fähigkeiten, stehen. [vgl. 3; 4]

1.1 Motivation

Das *P&P-RPG*-Regelwerk zu *DSA* zeichnet sich insb. durch den Umfang und die Komplexität der Regeln und Charakter-Merkmale aus. Dies wird durch zahlreiche Regelerweiterungsbänder verschärft. Dadurch ist u. a. die Charakter-Erstellung und -Verwaltung für die Teilnehmenden deutlich erschwert, zeitintensiv und fehleranfällig. Um dieser Komplexität, dem Zeitaufwand und unbeabsichtigten Regelverstößen entgegenzuwirken, bedarf es Hilfsmittel. Bis heute nutzen die Teilnehmenden verschiedenste Hilfsmittel; diese können selbsterstellt sein oder von der *DSA*-Gemeinschaft (englisch: *community*), teils quelloffen (englisch: *open source*), veröffentlicht sein. Sie reichen von Charakterbögen-Layouts, über Tabellenkalkulationen bis hin zu Programmen und Internetseiten [vgl. 5; 6; 7].

Im Jahr 1993 und 1995 veröffentlichte der damalige Herausgeber von *DSA* eine Software mit dem Namen „*DSA*-Tools“ zur Erstellung und Verwaltung von Charakteren für die damalige Ausgabe [vgl. 8; 9]. Die aktuellen und umfangreichsten veröffentlichten Hilfsmittel kommen aus der Gemeinschaft selbst, stehen kostenlos zur Verfügung und sind teils quelloffen; nachfolgend die populärsten Softwares aufgelistet, die einen ähnlichen Zweck, wie der dieser Arbeit, erfüllen [vgl. 10]:

Optolith Heldengenerator erschienen am 8. April 2018, zuletzt aktualisiert am 22. März 2022, dient der Charakter-Erstellung und -Verwaltung in der 5ten-*DSA*-Ausgabe. Es ist seit August 2019 quelloffen und wird hauptsächlich von einer Person gewartet. Genutzt werden die Sprachen TypeScript[1] und ReasonML[2]. Die Regeln werden in *YAML*-Dateien festgehalten und ein Zugriff wird, aufgrund von Lizenzgebungen, nur auf Anfrage und Notwendigkeit ermöglicht. [vgl. 11]

Helden-Software erschienen im Jahr 2012, zuletzt aktualisiert am 23. Juli 2017, dient der Charakter-Erstellung und -Verwaltung in der 4ten-*DSA*-Ausgabe. Es ist nicht quelloffen und wurde in Java[3] ohne bemerkenswerte Frameworks entwickelt. [vgl. 12]

Gepflegt wird es von einem Team aus fünf Entwickelnden und zwei Testenden [vgl. 13]. Als Besonderheit bietet diese Lösung das Speichern der Charaktere auf deren Server und eine Schnittstelle zur Entwicklung von Erweiterungen (englisch: *plugins*) an, mit denen Informationen zu den Charakteren abgerufen werden können [vgl. 14; 15].

DSA Excel Heldenblatt erschienen im Jahr 2003, zuletzt aktualisiert am 15. Mai 2021, dient der Charakter-Erstellung und -Verwaltung in der *DSA*-4.1-Ausgabe. Es handelt sich hierbei um eine Tabellenkalkulation basierend auf Excel[4], weshalb dieses als quelloffen zu betrachten ist. Die Regelwerk-Werte hingegen sind, wie auch bei den bereits genannten Programmen, aufgrund von Lizenzen verschlüsselt und damit nicht frei zugänglich. Das Besondere hier ist, dass eigene Merkmals-Ausprägungen, wie z. B. Zauber, Rituale und Liturgien, integriert werden können. [vgl. 16; 17]

DSA MeisterGeister erschienen im Jahr 2010, zuletzt aktualisiert am 6. Juni 2022, unterstützt die Spielleitung in verschiedenen Bereichen, darunter auch der Charakter-Erstellung und -Verwaltung, in der *DSA*-4.1-Ausgabe [vgl. 18]. Aufgrund fehlender personeller Kapazitäten im Team, bestehend aus zwei Personen, wurde der Quellcode mit der letzten Aktualisierung veröffentlicht [vgl. 19; 20]. Es ist in C#[5] geschrieben und daher auf das *OS* „Microsoft Windows“ ausgelegt [vgl. 21].

Die meisten Hilfsmittel unterstützen die aktuelle 5te-*DSA*-Ausgabe nicht. Dies ist auch nicht geplant; genannt wird als Begründung meistens hohe (zeitliche) Umsetzungsaufwände zusammen mit fehlenden Kapazitäten [vgl. 22; 23]. Das letztgenannte Hilfsmittel wurde aufgrund dessen quelloffen gelegt, in der Hoffnung, dass die *P&P*-Gemeinschaft sich um die Weiterentwicklung kümmert [vgl. 19]. Gründe für die hohen Umsetzungsaufwände, also die

[1] Typisierter Superset von JavaScript; https://www.typescriptlang.org/
[2] OCaml (https://ocaml.org/) in JavaScript-angelehnter Notation; https://reasonml.github.io/
[3] *Betriebssystem (OS)*-unabhängige *Objekt-orientierte Programmierung (OOP)* Sprache; https://www.oracle.com/de/java/
[4] Vom Hersteller Microsoft; https://www.microsoft.com/de-de/microsoft-365/excel
[5] Typsichere *OOP* Sprache; https://learn.microsoft.com/de-de/dotnet/csharp/tour-of-csharp/

Probleme dieser Hilftsmittel, könnten sein: **(a)** eine starre Software-Architektur; **(b)** stark gekoppelte und komplizierte Implementierung des Regelwerks; **(c)** schlecht gewachsene und gewartete Code-Basis (technische Schuld).

1.2 Grundlagen

Das *DSA*-Basis-Regelwerk kategorisiert seine Regeln wie folgt [vgl. 24]: **(a) Grundregeln:** sind immer einzuhalten [vgl. 25]; **(b) Optionale Regeln:** können zu Beginn der Geschichte zugeschaltet werden [vgl. 26]; **(c) Fokusregeln:** können nach Bedarf während des Geschehens zu- und abgeschaltet werden und vertiefen Regeln eines speziellen Bereiches, z. B. die Jagd oder den Hausbau [vgl. 27; 28]; **(d) Hausregeln:** sind inoffiziell und werden unter den Teilnehmenden ausgemacht [vgl. 29]. In den nachfolgenden Kapiteln werden ausschließlich die Grundregeln des Basis-Regelwerks beachtet.

1.2.1 Charakter-Erstellung

Ein Charakter definiert sich durch folgende Merkmale, welche auf dem Charakterbogen einzutragen sind: Spezies, Kultur, Profession, Eigenschaften, Fertigkeiten sowie Vor- und Nachteile. Einige Merkmale können Stufen bzw. eine Wertigkeit haben, u. a. Eigenschaften, Fertigkeiten sowie Vor- und Nachteile. Fertigkeiten bestehen aus Talenten, Zaubern, magischen Handlungen und Liturgien. Jeder dieser Merkmale, oder die Steigerung einer Stufe/Wertigkeit, kostet dem Spielenden *Abenteuerpunkt*e (AP)*. Bei der Charakter-Erstellung hat jeder Spielende eine, von der Spielleitung vorgegebene, Anzahl an Start-AP, welche durch die Wahl von Nachteilen gesteigert werden kann. Zunächst wird die Spezies, Kultur und Profession ausgewählt. Mit den restlichen AP wird für Fertigkeiten, Vorteile und andere Spielmechaniken bezahlt oder durch Nachteile erworben. Anschließend sind acht Basiswerte nach vorgegebenen Formeln zu berechnen. Diese werden ggf. durch die Spezies, den Eigenschaften, Vor- und Nachteilen sowie Sonderfertigkeiten beeinflusst. [vgl. 30]

Dabei ist aus folgenden und teils durch Regeln eingeschränkten Umfängen zu wählen (dies stellt einen Auszug und keine komplette Aufstellung dar): vier Spezien, 25 Kulturen, 33 Professionen, acht Eigenschaften, 59 Talenten, 45 Zaubern, sechs Ritualen, 16 magischen Handlungen, 34 Liturgien, 50 Vorteilen, 52 Nachteilen und weiteren Spielmechaniken, wie z. B. 222 Sonderfertigkeiten[6]. [vgl. 31]

[6] Sonderfertigkeiten der Kategorie „Sprachen und Schriften“ wurden aufgrund schwieriger Zuordnung ausgeschlossen

Die Komplexität entsteht durch die Vielzahl der Merkmals-Kombinationen und Regeln. Alle Regeln, ohne weitere Hilfsmittel, einzuhalten ist, aufgrund der Anzahl, für die Teilnehmenden eine zeitintensive und fehleranfällige Aufgabe. Die o. g. Merkmale können sich bspw. gegenseitig beeinflussen oder ausschließen:

- Eine Spezies gibt Boni und Mali für Eigenschaften vor [vgl. 32].
- Spezies und Professionen können übliche und unübliche Kulturen sowie Vor- und Nachteile vorgeben [vgl. 32; 33].
- Professionen und Sonderfertigkeiten haben Voraussetzungen, u. a. ggf. an Eigenschaften, Fertigkeiten, Vor- und Nachteilen und Sonderfertigkeiten [vgl. 33; 34].
- Eine Profession gibt Fertigkeiten (Talente, Zauber, Rituale, Liturgien) vor [vgl. 33].
- Einige Fertigkeiten stehen nur bestimmten Professionen zur Verfügung (dies sind sog. Berufsgeheimnis*se) [vgl. 35].

Übliche und unübliche Merkmals-Kombinationen sind Vorgaben der Spielwelt bzw. des Regelwerks, um eine stimmige und logische Spielwelt zu erhalten [vgl. 30, Schritt 6]. Unübliche Merkmals-Kombinationen sind nicht verboten, müssen jedoch bekannt und mit der Spielleitung abgesprochen werden [vgl. 32]. Weitere Regeln, welche bei der Charakter-Erstellung zu beachtet sind, werden nachfolgend (und nicht abschließend) aufgezeigt:

- Eine vorher festgelegte Grenze an aktivierten Zaubern, Liturgien sowie „Fremdzauber“ (der Profession nicht zugeordnet) darf nicht überschritten werden [vgl. 30, Schritt 2].
- Eigenschaften, Fertigkeiten und Kampftechniken können nur bis zu einem vorab festgelegten maximalen Wert verbessert werden [vgl. 30, Schritt 2].
- Die erste Kultur ist gratis [vgl. 30, Schritt 4].
- Zauber und Liturgien sind vor der Wert-Steigerung zu aktivieren; dafür sind Aktivierungskosten zu zahlen [vgl. 30, Schritt 8].
- Es darf nur bis zu einer bestimmten Anzahl *AP* Vorteile gekauft oder mit Nachteilen erworben werden [vgl. 36].
- Am Ende darf nur eine maximale Anzahl *AP* übrig bleiben [vgl. 30, Schritt 11].

1.2.2 Charakter-Verwaltung

Bei der Charakter-Verwaltung geht es um Zukäufe und Steigerungen von Merkmalen, welche mit, im Laufe der Spielgeschichte erhaltenen, *AP* getätigt werden können (inkl. der Nachpflege im Charakterbogen). Die meisten der bereits zur Charakter-Erstellung

genannten Regeln und einhergehenden Probleme gelten auch bei der Verwaltung. Als Beispiel sind Merkmals-Ausschlüsse zu nennen, bei dem ein bereits gewähltes Merkmal ein Anderes nicht erlaubt (sinngemäß nach unüblichen Merkmalen); oder das Einhalten der Regeln ohne Hilfsmittel eine zeitintensive und fehleranfällige Aufgabe ist. Zusätzlich zu den bekannten kauf- bzw. steigerbaren Merkmalen können drei der acht Basiswerte gesteigert werden. [vgl. 37]

Weitere Regeln, die zu beachten sind, werden nachfolgend und nicht abschließend aufgezeigt: **(a)** Fertigkeiten können maximal zwei Stufen über der höchsten beteiligten Eigenschaft gesteigert werden. **(b)** Kampftechniken sind maximal zwei Stufen über der Leit-Eigenschaft zu steigern. **(c)** Der Basiswert „Lebensenergie“ kann maximal um die Höhe der Eigenschaft „Konstitution“ gesteigert werden. **(d)** Zauber und Liturgien können maximal bis Stufe 14 erhöht werden; dies kann durch die Sonderfertigkeit „Merkmalskenntnis“ bzw. „Aspektkenntnis“ aufgehoben werden. **(e)** Nur in Abstimmung mit der Spielleitung können Vorteile gekauft und Nachteile abgelegt werden. [vgl. 37]

1.2.3 Technische Betrachtung

Die nachstehende Gleichung 1.1 verdeutlicht vereinfacht und beispielhaft die Vielzahl der Merkmals-Kombinationen mit der ein Hilfsmittel umgehen muss. Es werden die in Absatz 2 von Kapitel 1.2.1: Charakter-Erstellung auf S. 3 genannten Umfänge für Merkmale genutzt und komplexe Regeln, wie z. B. Voraussetzungen von Merkmalen, nicht berücksichtigt. Gemäß Regelwerk gilt für einen Spielenden mit dem Erfahrungsgrad „Erfahren“, dass jede Eigenschaft um bis zu sechs Stufen gesteigert werden kann, aber maximal 36 Eigenschaftsstufen erkauft werden können [vgl. 30, Schritt 2 und 5]. Es wird angenommen, dass zwei Professionen gewählt werden, sechs Eigenschaften maximal gesteigert werden, 20 Fertigkeiten aktiviert und gesteigert werden, 10 Vorteile und fünf Nachteile gewählt werden und sich die Hälfte der Vor- und Nachteile sowie ein Viertel der Fertigkeiten, von denen nur 1/10 der Sonderfertigkeiten auswählbar sind, ausschließen. Unter Fertigkeiten wurden folgende Spielmechaniken zusammengefasst: Talente, Zauber, Rituale, magische Handlungen und Liturgien.

$$\underset{[\mathrm{S}]pezien}{\binom{\mathrm{S}}{1}} * \underset{[\mathrm{K}]ulturen}{\binom{\mathrm{K}}{1}} * \underset{[\mathrm{P}]rofessionen}{\binom{\mathrm{P}}{2}} * \underset{[\mathrm{E}]igenschaften}{\binom{\mathrm{E}}{6}} * \underset{\substack{[\mathrm{F}]ertigkeiten \\ [\mathrm{SF}]Sonderfertigkeiten}}{\binom{\frac{\mathrm{F}}{4}+\frac{\mathrm{SF}}{10}}{20}} * \underset{[\mathrm{V}]orteile}{\binom{\frac{\mathrm{V}}{2}}{10}} * \underset{[\mathrm{N}]achteile}{\binom{\frac{\mathrm{N}}{2}}{5}} \tag{1.1}$$

Damit ergeben sich **ca. 3,160780 Quintilliarden (10^{33}) Merkmals-Kombinationen**. Diese Zahl steigt auf ca. 3,409411 Oktilliarden (10^{51}), wenn man alle vom Herausgeber

online veröffentlichte Merkmale, welches Regelerweiterungsbänder und nicht Grundregeln voraussetzt, inkludiert. Dies sind 32 Kulturen, 270 Professionen, 189 Liturgien, 251 Zauber, 74 Rituale, 184 magische Handlungen, 1755 Sonderfertigkeiten, 130 Vorteile und 77 Nachteile [vgl. 31]. Es wurde angenommen, dass sich 1/3 der Professionen ausschließen und 1/6 der Fertigkeiten und 1/20 der Sonderfertigkeiten auswählbar sind.

1.3 Zielsetzung

In der vorliegenden Arbeit wird konzeptionell und praktisch, in Form einer lauffähigen Software, dargelegt, wie die nachstehenden Herausforderungen in einer quelloffenen Software zur Erstellung und Verwaltung von *DSA*-Helden für die 5te-Ausgabe gelöst werden können. Dabei wird ausschließlich das Backend der Software und dessen Schnittstelle (*Application Programming Interface (API)*) zu einem möglichen Frontend[7] betrachtet. Das Frontend ist also nicht Teil dieser Arbeit.

Technische Herausforderungen Die technische Herausforderung kann in zwei Teile gefasst werden: Zum einen muss die Vielzahl der Regeln in „kleinere Probleme“ zerteilt und formal dargestellt werden, sodass diese zur automatischen Überprüfung von Charakteren, effizient zur Laufzeit, genutzt werden können. Zum anderen muss das Regelwerk möglichst modular, also leicht erweiter- und wartbar, abgebildet werden, sodass Regeln einfach angepasst, hinzugefügt und getestet werden können. Dazu gehört auch die konzeptionelle Berücksichtigung der Einbindung von Regelerweiterungsbändern.

Quelloffen-bezogene Herausforderungen Eine quelloffene Software lebt und fällt mit den sog. Mitwirkenden (englisch: *contributor*), welche letztlich die Tätigkeiten klassischer Softwareentwicklung durchführen, z. B. das Anpassen, Testen und Veröffentlichen der Software. Auch wenn das Interesse der Nutzenden einer quelloffenen Software groß ist, besteht die Gefahr, dass die Software veraltet und letztlich „stirbt“, wenn diese nicht genügend aktiv Mitwirkende hat.

1.4 Aufbau

Zur vorliegenden Arbeit: Alle Fußnoten werden fortlaufend und über Kapitelgrenzen durchgehend nummeriert, sodass diese einfach referenziert werden können. Tabellen, Abbildungen und Listings hingegen werden fortlaufend innerhalb eines Oberkapitels nummeriert, um so den Kontext der Tabelle einfacher erfassen zu können. Quellenangaben werden nummeriert

[7] z. B. eine Webseite; diese wurde parallel in einer anderen Arbeit erarbeitet

in eckigen Klammern, entsprechend dem „Institute for Electrical and Electronics Engineers“ (IEEE)-Stil, gemacht. Daher orientiert sich die Sortierung im Literaturverzeichnis ausschließlich nach Reihenfolge der Verwendung. Dies ist angelehnt an der Vorlage der universitären Abteilung für Digitalisierte Energiesysteme (DES).

Der Aufbau der nachfolgenden Kapitel ist wie folgt: In Kapitel 2: Konzept auf S. 9 werden die nachfolgenden Aufgaben (a) bis (d) und in Kapitel 3: Realisierung auf S. 31 (e) bis (h) aufeinander aufbauend behandelt. Anschließend wird in Kapitel 4: Diskussion und Ausblick auf S. 47 die entstandene Software bewertet, sowie ein Ausblick auf bevorstehende Aufgaben gegeben. Abschließend wird die Arbeit in Kapitel 5: Fazit auf S. 51 bündig zusammengefasst.

a) **Anforderungen an quelloffener Software identifizieren:** die Erfolgs-wichtigsten Anforderungen an quelloffener Software werden identifiziert und im Konzept beachtet.

b) **Ansatz zur Problemlösung ermitteln:** es wird u. a. ein geeignetes Verfahren zur Formalisierung des Regelwerks samt Programmiersprache ermittelt und einhergehende Implikationen benannt.

c) **Schnittstelle definieren:** die Schnittstelle zum Frontend wird erarbeitet, ggf. mit dem Ersteller des Frontends diskutiert, und in einem Schnittstellenvertrag festgehalten.

d) **Regelwerk interpretieren:** anhand des ausgewählten Verfahrens wird das *DSA*-Basis-Regelwerk interpretiert und konzeptionell formalisiert, sodass Vorlagen, z. B. für die Implementierung eines Merkmals, entstehen.

e) **Projektstruktur und Architektur festlegen:** vorbereitend zur Konzeptumsetzung wird die grundlegende Projektstruktur und Architektur festgelegt, umgesetzt und dokumentiert.

f) **Design-Vorgaben festlegen:** um am Projekt mitzuwirken (dies schließt die initiale Konzeptumsetzung ein), werden u. a. der Code-Style, Namensgebungen, Testabdeckung, Dokumentationspflichten und ggf. vom Standard, z. B. der Programmiersprache oder gängigen ingenieurmäßigen Software-Entwicklung, abweichende Vorgaben definiert und dokumentiert. Teils werden elementare, aber dennoch wichtige Vorgaben explizit festgehalten.

g) **Konzept umsetzen:** beinhaltet die letztliche Implementierung der Kern-Funktionalitäten, wie die Frontend-Schnittstelle, Charakter-Validierung und Regelwerk-Formalisierung.

h) **Qualität sicherstellen:** gemäß der Design-Vorgaben muss die Qualität der Software sichergestellt werden und soll zum Projektende abschließend kontrolliert und ggf. verbessert werden.

2 | Konzept

In diesem Kapitel wird die Herleitung des Konzeptes aufgezeigt und letztlich damit vorgestellt. Zuerst werden die Anforderungen an einer quelloffenen Software identifiziert, um sie in den nachfolgenden Kapiteln berücksichtigen zu können. Anschließend wird der Ansatz zur Problemlösung ermittelt, welches das Verfahren und die Programmiersprache festlegt. Unabhängig dessen wird danach die *API* zum Frontend definiert. Das Kapitel wird abgeschlossen mit der konzeptionellen und formalen Interpretation des *DSA*-Regelwerks.

2.1 Anforderungen an quelloffene Software

In diesem Kapitel werden nur kurz die wissenschaftlich belegten Motivationsfaktoren von an quelloffener Software Mitwirkenden dargelegt, sodass anschließend, u. a. aus diesen, technische Bedingungen an eine quelloffene Software abgeleitet werden können.

2.1.1 Motivationsfaktoren der Mitwirkenden

Die in [38] zusammengefassten Motivationsfaktoren, ermittelt aus verschiedenen empirischen Studien ([39],[40],[41],[42],[43],[44],[45],[46],[46],[47]), werden nachfolgend sinngemäß prägnant wiedergegeben: **(a) Nutzen:** die Mitwirkenden können, insb. nach eigener Anpassung, einen Nutzen aus der Software ziehen, um z. B. ein eigenes Problem zu lösen. **(b) Ansehen:** durch die Mitwirkung kann die Person in verschiedenen Personenkreisen Ansehen erlangen. Unter Entwickelnden (ein Fachkreis) könnte dies z. B. die Qualität und Quantität der Mitwirkung aufzeigen. Wenn dieses Ansehen in Fachkreisen anerkannt wird, kann dies z. B. zu besseren Arbeitsangeboten führen. **(c) Identifikation:** wenn sich die Mitwirkenden stark mit den Zielen der Projektgruppe identifizieren können, steigt das (zeitliche) Engagement mitzuwirken. **(d) Wissen:** Mitwirkende können die eigenen Fähigkeiten, durch das Erlernen von Wissen bei der Lösung von Problemen und Anwendung neuer Technologien, verbessern. **(e) Altruismus:** das Mitwirken kann abstrakt mit einem Gefühl das Richtige zu tun verbunden sein oder als selbstlose und uneigennützige Spende gesehen werden, ggf. nach dem eigenen Nutzen von quelloffener Software. **(f) Freude:** die kreativen Freiheiten und lockeren Rahmenbedingungen fördern Spaß beim Mitwirken.

2.1.2 Abgeleitete technische Anforderungen

Aus den in Kapitel 1.3: Zielsetzung auf S. 6 genannten Herausforderungen für quelloffene Software-Projekte und den zuvor genannten Motivationsfaktoren der Mitwirkenden lassen sich einige wichtige technische Anforderungen ableiten und stellen damit wichtige Voraussetzungen für ein langfristig erfolgreiches quelloffenes Software-Projekt dar.

In der frühen Phase einer quelloffenen Software, unter Beachtung des vorliegenden (eher kleineren) Kreises möglicher Mitwirkenden, ist es von Bedeutung die Einstiegshürden für Mitwirkende möglichst gering zu halten, sodass eine möglichst große Basis an Mitwirkenden gewonnen werden kann.

Die Motivationsfaktoren *Nutzen* und *Identifikation* sind aus Projektsicht nicht steuerbar, sondern als gegeben zu sehen, da auszugehen ist, dass vorwiegend Mitwirkende aus der *P&P*-Gemeinschaft kommen werden. Als gegeben kann ebenfalls der Faktor *Ansehen*, aufgrund der Verwendung eines wissenschaftlich basierten Verfahrens, gewertet werden. Hierdurch und durch die zu erlernenden Sprachen und Frameworks kann der Motivationsfaktor *Wissen* bedient werden. Für den Faktor *Freude* ist zum Teil bereits dadurch vorgesorgt, dass es sich bei der Fachlichkeit, also dem Anwendungsbereich der Software, um etwas grundsätzlich Kreatives handelt und es, außer dem festen *DSA*-Regelwerk, kaum eingrenzende Rahmenbedingungen gibt.

Folgende technische Anforderungen werden daher abgeleitet definiert:

- Die Dokumentationen müssen stets aktuell, leicht auffindbar und verständlich sein; dazu gehören u. a. Code- und Architektur-Dokumentationen sowie Anleitungen und Richtlinien zur Mitwirkung, Einrichtung der Entwicklungsumgebung und Testdurchführung; umso prägnanter diese sind, desto zuträglicher ist es für die Wartbarkeit und insb. neuen Mitwirkenden, sodass diese nicht erschlagend sind.
- Die Projektstruktur, Architektur und der Quellcode entsprechen weitestgehend den „Branchen-“ (z. B. CleanCode[8]), Programmiersprachen- und Frameworks-Standards; Abweichungen sind erlaubt, wo es insb. dem Verständnis und der Wartbarkeit hilft. Dies soll einen leichteren Wissensaufbau ermöglichen, welches z. B. durch den Wiedererkennungswert zwischen Framework-Dokumentation und der Verwendung im Code, durch Nutzen gleicher Benamungen, gegeben wird.
- Es existiert eine hohe und vor allem qualitative Testabdeckung, durch welche insb. neue Mitwirkende frühzeitig Fehler während der Entwicklung erkennen können.

[8] `https://clean-code-developer.de/`

- Es werden möglichst weit verbreitete Programmiersprachen und Frameworks genutzt, um die Zugänglichkeit des Projekts indirekt zu erhöhen, als auch eine gewisse Langlebigkeit dieser und damit einen einhergehenden niedrigeren Wartungsaufwand sicherzustellen. Die Zugänglichkeit wird hier als das Finden von Hilfsmaterialien (z. B. Dokumentationen und Anleitungen (englisch: *guides*)) und Hilfestellungen Dritter definiert.
- Der Quellcode wird in der Sprache Englisch verfasst. Dies soll den möglichen Kreis der Benutzenden und Mitwirkenden maximal groß halten. Außerdem wird damit der Wiedererkennungswert des Quellcodes hinsichtlich möglicher Hilfsmaterialien gesteigert, da diese meistens auf Englisch sind. Konkrete Namen von Merkmalsausprägungen hingegen werden nicht übersetzt, um Verwirrungen und Inkonsistenzen zu vermeiden, da eine vollständige Übersetzung durch den *DSA*-Veröffentlichenden fehlt.

2.2 Verfahrens-Evaluation

Für die, im Kapitel 1.3: Zielsetzung auf S. 6 genannten, Herausforderungen muss eine geeignete Lösung (Framework/Algorithmus) ermittelt werden. Dazu werden in den nachfolgenden Kapiteln bekannte Verfahren/Varianten von Regel-basierten Systemen und (Bedingungs-basierten) Logiksystemen vorgestellt und nach einer Gegenüberstellung dieser eins ausgewählt.

2.2.1 Bekannte Verfahren

PROgrammation en LOGique (Prolog) hat die formalisierte Darstellung von Wissen und die Verwendung von logischen Regeln als Grundlage, um dieses Wissen zu verarbeiten und Probleme zu lösen. *Prolog* basiert auf der *Horn-Klausel*-Semantik, bei der eine Aussage durch eine Menge von Horn-Klauseln dargestellt und von einem *Prolog*-Interpreter verarbeitet wird. Eine Horn-Klausel ist eine logische Formel, die aus einem *Kopf* und einer Menge von *Füßen* besteht, wobei der Kopf ein positives Literal ist und die Füße negative Literale sind. Eine Aussage ist dann wahr, wenn es eine Übereinstimmung (Unifikation) zwischen dem Kopf und einer der Füße gibt. Um die Horn-Klauseln zu verarbeiten, wird ein Übereinstimmung-Algorithmus genutzt. Dieser versucht die Variablen in den Horn-Klauseln so zu substituieren, dass der Kopf und einer der Füße übereinstimmen. Nur wenn eine solche Substitution möglich ist, gilt die Horn-Klausel als wahr. Dabei kommt ein sog. Backtracking-Mechanismus zum Einsatz, bei dem zu einem früheren Punkt im Suchprozess zurückgekehrt wird, um alternative Lösungen zu finden oder wenn keine Übereinstimmung möglich ist. Durch das Navigieren durch verschiedene Möglichkeiten kann *Prolog* komplexe Probleme lösen. [vgl. 48; 49]

boolean SATisfiability problem (SAT) ist ein Erfüllbarkeitsproblem der Aussagenlogik, das darin besteht, festzustellen, ob es eine Boolean-Vereinbarung für eine gegebene *Boolean-Formel* gibt, welche die Formel als wahr evaluiert. Eine Boolean-Formel besteht nur aus den Operatoren „*und*“, „*oder*“ und „*nicht*“. Sie kann als Konjunktion von Disjunktionen von Literalen dargestellt werden, wobei ein Literal entweder eine Variable oder deren Negation ist. *SAT* ist ein NP-vollständiges Problem, was bedeutet, dass es keine effiziente algorithmische Lösung gibt, die für alle Fälle eine korrekte Lösung findet. Es gibt jedoch algorithmische Ansätze, die in vielen Fällen effizient arbeiten und erfolgreich eingesetzt werden. Nennenswerte und weit verbreitete Ansätze sind der *Davis-Putnam-Logemann-Loveland (DPLL)*-Algorithmus und Backtracking-Algorithmen, wie Conflict-Driven Clause Learning (CDCL). Letzterer vermeidet unnötiges Backtracking und verkürzt damit die Laufzeit des Algorithmus, indem es Konflikt- und Klausel-Lernstrategien nutzt. Das *SAT*-Solver-Verfahren wird häufig mit anderen Techniken, wie z. B. Formalisierungs- und Beweisverfahren, kombiniert, sodass Probleme in einer formalen Sprache beschrieben und mögliche Lösungen effizient verifiziert werden können. Es ist ein wichtiges und weit verbreitetes (Entscheidungs-)Problem in der theoretischen Informatik und der künstlichen Intelligenz, das viel Anwendung findet und weiterhin intensiv erforscht wird, um effizientere und leistungsfähigere Lösungsmethoden zu entwickeln. [vgl. 50; 51; 52]

Answer Set Programming (ASP) ist ein deklaratives Problemlösungsparadigma. Es basiert auf der Logikprogrammierung (*Prolog*), dem Erfüllbarkeitsproblem der Aussagenlogik (*SAT*) sowie der Wissensrepräsentation und Argumentation (englisch: *knowledge representation and reasoning*). Ein *ASP*-System besteht aus einer Menge von *Regeln* und *Fakten*. Die Regeln sind logische Implikationen in der Form „*wenn A dann B*“, wobei *A* und *B* logische *Aussagen* sind. Ein Beispiel wäre: „*Wenn es regnet, dann nehme ich einen Regenschirm mit.*“. Dies kann interpretiert werden als: wenn die Bedingung „*es regnet*“ erfüllt ist, dann ist die Konsequenz „*ich nehme einen Regenschirm mit*“ auch erfüllt. Fakten sind einfache logische Aussagen, die bereits als *wahr* bekannt sind; sie haben keine Bedigung. Unbekannte Aussagen werden, gemäß der Closed World Assumption (CWA), als *falsch* gewertet. Das Ziel des *ASP*-Systems ist es, eine Menge von Aussagen zu finden, die, unter den gegebenen Regeln und Fakten, *wahr* sind. Diese Menge von Aussagen wird als *Answer Set* (auch *stable models*) bezeichnet. Um ein Answer Set zu finden, verwendet das *ASP*-System (*Answer Set Solver*) einen Suchalgorithmus, um den Raum möglicher Lösungen zu durchsuchen. Es beginnt mit der Menge der Fakten und wendet die Regeln an, um zu sehen, welche anderen Aussagen logisch aus ihnen abgeleitet werden können. Wenn ein Widerspruch gefunden wird, geht das System zurück und versucht

einen anderen Weg. Wenn es keine weiteren Schlussfolgerungen mehr findet, gibt es die Menge der abgeleiteten Aussagen als Answer Set zurück. Weitere Eigenschaften von *ASP* sind u. a.: **(a)** Es ist möglich mehrere Lösungen (Answer Sets) zu finden. **(b)** Regeln und Fakten können negiert oder als „unbekannt/unvollständig" gekennzeichnet werden, weshalb *ASP* als nicht-deterministisch angesehen werden kann. **(c)** Regeln können mit Gewichtungen versehen werden, was es ermöglicht unterschiedliche Arten von Unvollständigkeiten zu berücksichtigen. **(d)** Die formale Modellierung ist besonders flexibel, weil Regeln dynamisch hinzugefügt und entfernt werden können. [vgl. 53; 54]

2.2.2 Gegenüberstellung und Auswahl

In der Tabelle 2.1: Verfahrensgegenüberstellung auf S. 14 werden die vorgestellten Verfahren mit den folgenden Kriterien verglichen: das Prinzip des Verfahrens, die Sprache, welche die grundlegende Syntax und Semantik vorgibt, die Technik der Lösungssuche, mögliche Anwendungsbereiche, Verbreitung (in der Wissenschaft) sowie Stärken und Schwächen bei der Anwendung.

Die Verbreitung wird rudimentär über die Anzahl der gefundenen Ergebnisse über Google Scholar bestimmt. Es wird nach der expliziten (in Hochkommata gestellten) englischen Langform der Abkürzungen *Prolog*, *SAT* und *ASP* gesucht, wobei diese die Kurzform als Präfix hat. Das Suchformat entspricht damit: <Kurzform> "<englische Langform>". Besonderes hohe Bedeutung werden den Kriterien Verbreitung, Stärken und Schwächen zugesprochen. Die Verbreitung stellt einen sehr einfachen Indikator für die Verbreitung des Verfahrens und somit für das Finden möglicher hilfreicher Ressourcen dar. Dies ist auch in Hinblick der Langlebigkeit der Software relevant. Die beiden Kriterien Stärken und Schwächen hingegen zeigen insb. die Eignung des Verfahrens zur Formalisierung eines *P&P*-Regelwerks und der Modellierung einer Charakterüberprüfung.

Demnach ist das **Verfahren *ASP*** für die Umsetzung **am besten geeignet** und somit auch zu wählen. *ASP* ist ein modernes und verbreitetes Verfahren und erfüllt damit am ehesten den vorletzten Punkt der abgeleiteten technischen Anforderungen auf S. 11. Es bringt die notwendige Effizienz bei großen Problemen auf, welches dem umfangreichen *DSA*-Regelwerk gerecht wird. Es bringt ebenfalls ein großes Set an Möglichkeiten mit sich, wie z. B. Negationen und komplexe Kontroll- und Flussstrukturen. Des Weiteren erlaubt es bereits vom Prinzip und der Technik her mehrere mögliche Lösungen, was der Realität, also dem Anwendungsbereich, am ehestem entspricht. Die Schwäche einer steilen Lernkurve, aufgrund der komplexen Syntax und Semantik, kann und wird in der Realisierung durch umfangreiche Dokumentation beachtet und entgegengewirkt.

Tabelle 2.1: Verfahrensgegenüberstellung

	Prolog	***SAT***	***ASP***
Prinzip	Logikprogrammierung	Wahrheitsbelegung für boolesche Variablen finden	Lösungen nicht-deterministisch finden, die Bedingungen (englisch: *constraints*) erfüllen
Sprache	Horn-Klauseln	Boolesche Formeln	Normalform Horn-Klauseln
Technik	Unifikation, Backtracking	Backtracking, *DPLL*-Algorithmus	Stable Model Semantics, Grounder, Solver
Anwendung	Künstliche Intelligenz, Expertensysteme, natürliche Sprache	Logische Schaltungen, Beweisverfahren, Scheduling- und Optimierungsprobleme	Bedingungs-basierte Modellierung, kombinatorische Optimierung, Planungsprobleme, Entscheidungsunterstützung
Verbreitung	4320 Google Scholar Treffer mit: *PROLOG "programming in logic"*	5170 Google Scholar Treffer mit: *SAT "boolean satisfiability problem"*	7970 Google Scholar Treffer mit: *ASP "answer set programming"*
Stärken	Natürliche Ausdrucksweise, einfache Handhabung von Unsicherheiten	Effiziente Lösungstechniken, geeignet konsistente Lösungen zu finden	Präzise und vollständige Repräsentation von Wissen, unterstützt Negation und Unbestimmtheit, komplexe Programmflusskontrolle
Schwächen[(1)]	Ineffiziente Lösungstechniken, begrenzte Skalierbarkeit, ungeeignet für komplexe Kontrollstrukturen[(2)]	Schwierigkeiten bei Unsicherheiten und Wissens-Abhängigkeiten, konstante Lösungssuche ggf. ineffizient bei großen Problemen	Komplexe Syntax und Semantik (Lernkurve)

[(1)] um die Schwächen zu kompensieren, werden diese Verfahren oft mit Anderen kombiniert.
[(2)] wie Schleifen und bedingte Anweisungen.

2.3 Auswahl der Programmiersprache

Mit der Wahl des Verfahrens grenzt sich, im Allgemeinen und insb. im Speziellen, wie im Folgenden gezeigt wird, die Auswahl möglicher Programmiersprachen für die quelloffene Software ein. In diesem Kapitel werden zunächst die populärsten Frameworks für *ASP* und ihre wichtigsten unterstützten Programmiersprachen vorgestellt und anschließend, unter Beachtung der Rahmenbedingungen der quelloffenen Software, wie die aus Kapitel 2.1.2 auf S. 10, bewertet und ausgewählt.

Wrapper- und High-Level-*API*s (Dritter), welche diese Frameworks für weitere Programmiersprachen verfügbar machen oder die Nutzung vereinfachen, werden nicht beachtet. Diese würden eine weitere Komplexitäts- und Abhängigkeitsschicht bringen, welche im Kontext von langlebiger und wartungsarmer quelloffener Software ein größerer Nachteil ist als die erlangten Vorteile. Wenn es z. B. relevante Updates des eigentlichen Frameworks gibt, aber diese „Zwischen-*API*s" dieses Update nicht unterstützten, kann die Entwicklung blockiert sein. Noch kritischer ist dieser Fall, wenn diese „Zwischen-*API*s", welche zumeist ebenfalls quelloffen sind, nicht mehr gewartet werden und man somit gezwungen ist, diese selbst zu warten oder sie auszubauen, was in beiden Fällen zu (erheblichen) Mehraufwand führen würde. Beispiele sind clingo-haskell[9], Clyngor[10], Asp4J[11], jclingo[12], clingo-wasm[13] und DLVHEX[14].

2.3.1 Populäre Frameworks

DataLog with Disjunction (DLV) ist seit dem Jahr 1996 verfügbar. Über die Jahre wurde das System signifikant verbessert und die Sprache erweitert, wodurch es im Bereich *Disjunktive Logikprogrammierung (DLP)* zum Standard wurde und vergleichbar, insb. hinsichtlich der Effizienz, mit fortgeschrittenen *ASP*-Systemen ist. [vgl. 55, Kapitel 1]

Ein Vorteil von *DLV*, im Vergleich, ist die Breite der Anwendbarkeit. Neben seiner Kernstärke in *Datenbank (DB)*-orientierten Anwendungen mit großen Datenmengen, kann es mit ausreichender Effizienz *NP*-Such- und Optimierungsprobleme bis hin zu komplexen Problemen in der zweiten Ebene der Polynomialzeithierarchie lösen. Andere Systeme hingegen sind eher auf eine Problemklasse spezialisiert, wie z. B. *NP*-vollständige Probleme. [vgl. 55, Kapitel 3 und 7]

[9] Clingo-*API* für Haskell: https://github.com/tsahyt/clingo-haskell
[10] Wrapper-*API* für Clingos Python-*API*: https://github.com/Aluriak/clyngor
[11] Clingo-High-Level-*API* für Java: https://github.com/hbeck/asp4j
[12] Clingo Java Bindings: https://github.com/kherud/jclingo
[13] Clingo-*API* für JavaScript: https://github.com/domoritz/clingo-wasm
[14] Python Interface für HEX-Programme (Erweiterung von *ASP*): https://github.com/hexhex/core; http://www.kr.tuwien.ac.at/research/systems/dlvhex/

DLVSYSTEM S.R.L. bewirbt und vertreibt *DLV* mit dem Fokus auf industrielle Bereiche. Diese Firma wurde 2005 von Professoren der Universität von Calabria gegründet und 2009 als Ausgründung dieser Universtität anerkannt. [vgl. 56; 57]

Ein *DLV*-Programm kann ausgeführt werden, indem es über das „command-line interface“ (CLI) ausgerufen wird. Dies kann manuell oder gesteuert durch eine Software beliebiger Programmiersprache geschehen; die naheliegendste ist C++, da das Framework in C implementiert wurde. Um die Entwicklung zu vereinfachen, wurde von DLVSYSTEM S.R.L. eine eigene *Integrated Development Environment (IDE)* entwickelt (ASPIDE[15]). Diese bietet, neben einer „graphical user interface“ (GUI), wichtige Funktionen wie „unit testing“, *DB*-Integration und eine Plattform zum Austausch von Plugins, erstellt von Anderen. [vgl. 58]

Es existiert zudem eine Java-Schnittstelle (englisch: *interface*) (JDLV[16]), wobei die Integration über die Meta-Spezifikation JASP realisiert wird [vgl. 59].

DLV wird auch weiterhin direkt von der Universtität von Calabria veröffentlicht. Sie bieten zusätzlich eine Variante an, die den Aufruf von Python-Funktionen als *external atoms*, aus einem *DLV*-Programm, unterstützt. [vgl. 60]

Clingo ist ein *ASP*-System von *POTsdam Answer Set Solving COllection (Potassco)*, ein Projekt der Universtität Potsdam, das sowohl den sog. *grounder* (gringo) als auch den *solver* (clasp) des selbigen Herstellers *Potassco* vereint [vgl. 61]. Clingo selbst und alle seine Komponenten sind seit 2008 quelloffen und seit 2016 auf GitHub [vgl. 62]. Eine native Anbindung an Programmiersprachen wird durch die *API*s für Python[17], C[18] und Rust[19] gewährleistet. Aus einem Clingo-*Logik-Programm (LP)* kann man außerdem sehr einfach externe Funktionen, z. B. Python-Funktionen, aufrufen [vgl. 61, Kapitel 3.1.14].

Zwei wichtige Features für die Modellierung von komplexen Problemen sind: erstens die Möglichkeit Teile des *LPs* in separat aufrufbare Unterprogramme (Funktionen) zu strukturieren; und zweitens Regeln und Fakten mit noch unbekannten Aussagen zu formulieren, welche erst später, z. B. in Unterprogrammen, definiert werden [vgl. 63]. Ebenfalls nennenswert ist die Unterstützung von Theorien-basierter Argumentation (englisch: *theory-specific reasoning*), womit komplexe Regeln, Fakten und Aussagen modular (innerhalb einer Theorie) definiert und entstehende Aussagen außerhalb dieser wiederverwendet werden können [vgl. 64]. Auch zu nennen ist das sog. „multi-shot

[15] https://www.dlvsystem.it/dlvsite/aspide/
[16] https://www.dlvsystem.it/dlvsite/jdlv/
[17] https://potassco.org/clingo/python-api/5.6/clingo/
[18] https://potassco.org/clingo/c-api/current/
[19] https://github.com/potassco/clingo-rs

solving“, bei dem ein *LP* nicht einmal in Gänze gelöst wird, sondern in Teilprobleme separiert wird, welche wiederholt und aufeinander aufbauend gelöst werden, um so eine optimale Lösung, bei sich verändernden Aussagen, zu finden [vgl. 65].

Mit der Clingo-Erweiterung clingcon können in der Problemstellung auch Bedingungen auf endliche Ganzzahlen beachtet werden [vgl. 66]. Hingegen ergänzt die Erweiterung Clingo ORM (clorm) die Python *API* um Object Relational Mapping (ORM), welches die (statische) Modellierung der Fachlichkeit vereinfacht [vgl. 67].

2.3.2 Unterstützte Programmiersprachen

C++ ist eine Objekt-orientierte, imperative und Template-basierte Programmiersprache, die von Bjarne Stroustrup 1991 als eine Allzweck-Programmiersprache entwickelt wurde. Es ist eine Erweiterung oder Nachfolger der Programmiersprache C, um dessen Mängel in Bezug auf *OOP* und die Wiederverwendbarkeit von Code zu beheben. So wurden klassische *OOP*-Konzepte, wie Klassen, Vererbung, Polymorphismus und Abstraktion, als auch Paradigmen, wie generische Typen, eingebaut, die es Entwickelnden ermöglicht, komplexe Software effizient zu entwickeln und übersichtlich zu organisieren. [vgl. 68]

Besondere Eigenschaft sind u. a.: **(a)** die Möglichkeit Vorlagen (englisch: *templates*) zu erstellen, um z. B. einen sich oft ähnlich wiederholenden Code generisch zu schreiben und wiederverwendbar zu machen, **(b)** die Standard Template Library (STL), eine umfangreiche Sammlung gebrauchsfertiger Vorlagen, **(c)** Mehrfachvererbung, bei der eine Klasse mehr als eine Elternklasse haben kann, und **(d)** die Möglichkeit die Vererbungshierarchie zur Laufzeit zu verändern. [vgl. 69]

Wie Java und der Vorgänger C, ist C++ eine kompilierte Sprache. Es ist aber nicht so einfach portabel wie Java, da, je nach ausführender Maschine und *OS*, andere Kompilatoren und Bibliotheken zu nutzen sind. C++ wird als Programmiersprache niederer Ebene betrachtet. Grund dafür sind die vielen Möglichkeiten die Software ins kleinste Detail zu steuern und zu optimieren, wie z. B. die Speicherverwaltung, welches oft als „Nähe zur Maschine/Hardware“ bezeichnet wird. Diese Vielseitigkeit macht es zugleich anfälliger für Fehler durch Entwickelnde. [vgl. 70]

Aufgrund der langen Historie und vielseitigen Spracheigenschaften von C++, wurde und wird es in nahezu sämtlichen erdenkbaren Bereichen eingesetzt; einige Beispiele sind Banken und das Finanzwesen, die Systementwicklung von *OS*, *DB* und Kompilatoren, eingebettete Systeme, wie in Kameras, Flugzeugen und medizinischen Geräten, Spiele und wissenschaftliche Simulationen [vgl. 68].

Python ist eine High-Level-Programmiersprache, die ursprünglich 1989 von Guido van Rossum entwickelt wurde. Seitdem hat sie sich zu einer der am häufigst verwendeten Sprachen in der Wissenschaft, Datenanalyse, künstlichen Intelligenz und Webentwicklung entwickelt. Eines der wichtigsten Merkmale ist seine Les- und Schreibbarkeit, die es Entwickelnden ermöglicht Code einfach, verständlich und schnell zu schreiben. Es verwendet eine natürliche und lesbare Syntax, welche zugleich sparsam ist. Für Englisch-sprachige ist sie besonders angenehm beim Verstehen und Pflegen, aufgrund der Ähnlichkeiten der Schlüsselwörter und des Aufbaus von Kontrollstrukturen. Python ist eine interpretierte Sprache, was bedeutet, dass der Code nicht vor der Ausführung kompiliert werden muss. Dieser wird stattdessen zur Laufzeit vom sog. Interpretierer (englisch: *interpreter*) Befehl für Befehl ausgeführt. Ein ggf. langwieriger Kompilierungsprozess entfällt daher. Ein weiteres wichtiges Merkmal von Python ist seine Dynamik. Es ermöglicht Entwickelnden, Variablen und Datentypen während der Laufzeit des Programms zu ändern, anstatt sie vorab deklarieren und kompilieren zu müssen. Dies erleichtert die Entwicklung von Prototypen und ermöglicht es, flexibler auf Änderungen im Code zu reagieren. Ein Nachteil gegenüber kompilierten Sprachen ist, für gewöhnlich, eine längere/langsamere Ausführungszeit/Laufzeit. Python hat eine große und aktive Gemeinschaft, die eine Vielzahl von Bibliotheken und Modulen bereitstellt, die es Entwickelnden erlaubt sich beim Programmieren auf ihren Anwendungsfall (ihre Problemstellung) zu fokussieren und so leichter und schneller fertig zu werden. Beispiele für solche Bibliotheken sind NumPy[20] und Pandas[21] für Datenanalyse, TensorFlow[22] für künstliche Intelligenz und Django[23] für Webentwicklung. Python ist eine vielseitige und leistungsfähige Programmiersprache, die sich sowohl für Personen ohne Vorkenntnisse als auch für erfahrene Entwickelnde eignet und in einer Vielzahl von Bereichen eingesetzt werden kann. [vgl. 71; 72; 73]

2.3.3 Auswahl Framework-Sprachen-Kombination

Entscheidend ist zunächst die Wahl des Frameworks. Erst dann kann eine Programmiersprache ausgewählt werden. Beide der genannten Frameworks sind für den nicht-kommerziellen Gebrauch, bezogen auf die Lizenz, frei verfügbar. Clingo bietet im Vergleich zu *DLV* eine größere Auswahl relevanter Features (inkl. der Erweiterungen) an, wie die Unterteilung und Wiederverwendung von Teilen des *LPs*. Auch die breitere Unterstützung von Programmiersprachen und deutlich bessere Integration in diesen spricht für Clingo. Die genannten Erweiterungen und ungenannten Projekte, welche auf Clingos Komponenten aufbauen, sind

[20] https://numpy.org/
[21] https://pandas.pydata.org/
[22] https://www.tensorflow.org/
[23] https://www.djangoproject.com/

ein Indiz für eine Plattform, die in der Zukunft relevante Erweiterungen und weitere solide Integrationen versprechen. Ebenfalls entscheidend ist die Verfügbarkeit und Qualität von Dokumentationen. Hier bietet Clingo, aufgrund der veröffentlichten Materialien, welche teils in einem universitären Kontext zur Bildung genutzt werden, eine bessere Grundlage. **Clingo ist** daher **das geeignetere Framework**.

Die von Clingo nativ, also offiziell (out-of-the-box), unterstützten und relevanten (aufgrund der Popularität) Programmiersprachen sind C(++) und Python. Die freundlichste für ungeübte und angehende Programmierende ist Python. Dies, also der Einstieg in die Programmierung abhängig der Sprache, wurde in Ansätzen in [vgl. 74] für Python und C++ aufgezeigt. Das Potenzial möglicher Fehlerquellen ist in Python ebenfalls geringer, da diese entgegen C(++) eine Programmiersprache der höheren Ebene ist und daher den Entwickelnden bestimmte Aufgabenbereiche abnimmt. Die Syntax von Python ist außerdem sehr natürlich, was diesem zuträglich ist und so gut zum formalen und ebenfalls natürlichen Sprachbild von Clingo-*LPs* passt. Eine größere Komplexität wird durch die Verwendung von Python vermieden, da nur mit Python[24] die Möglichkeit besteht aus einem Clingo-*LP* heraus externe und Kontext-bezogene (Python) Funktionen aufzurufen und daher keine weitere Programmiersprache erlernt werden muss. Somit stellt **Python** die **sinnvollste** Wahl der **Programmiersprache** dar.

2.4 Frontend-Schnittstelle

In diesem Kapitel wird der Schnittstellenvertrag zwischen Front- und Backend, also eine interne *API*, beschrieben. Dieser spiegelt die konkreten Aufgaben der zu entwickelnden Software wider. Es beinhaltet u. a. die konkreten Endpunkte, die Pflicht- und optionalen Parameter (inkl. des Wertetyps und -bereichs), die erwarteten Antworten (inkl. Wertetypen und -bereiche), eine Beschreibung und Beispielwerte der Parameter. Durch den Umfang ist es zugleich eine gute Dokumentation der Schnittstelle. Dazu wird eine der populärsten Spezifikationen **OpenAPI** eingesetzt und im Format ***YAML*** bereitgestellt, da es eine gut lesbare Syntax hat und Kommentare erlaubt [vgl. 75, Kapitel 4].

Der Vorteil einer frühzeitigen Definition ist, dass man unabhängig „gegen“ diesen Schnittstellenvertrag entwickeln kann, sodass keine (zeitlichen) Entwicklungsabhängigkeiten zwischen Front- und Backend entstehen [vgl. 75, Kapitel 3]. Basierend auf dem Schnittstellenvertrag können sog. Mocks zur Entkopplung erstellt werden. Dies sind Nachbildungen der jeweils anderen Schnittstellen-Seite, welche das erwartete Verhalten imitiert und so ermöglicht das eigene System „gegen“ die Schnittstelle zu testen.

[24] eine weitere Sprache währe LUA, wobei diese nicht zur Auswahl steht

Im Laufe der Zeit, nach dieser Arbeit und der Veröffentlichung der Software, wird es vorkommen, dass Anpassungen am Schnittstellenvertrag gemacht werden und so verschiedene Versionen entstehen [vgl. 75, Kapitel 6]. Um den Wartungsaufwand, z. B. durch *API*-Versionierung im Pfad, gering zu halten und weil es sich hierbei weder um eine öffentliche *API* handelt, noch eine große Anwenderzahl hat, wird stets nur die aktuellste Version vom Backend unterstützt. Dies bedeutet, dass eine neue *API*-Version nur veröffentlicht werden kann, wenn diese mit der aktuell veröffentlichten Frontend-Version kompatibel ist.

2.4.1 Schemata

Das Konzept der Schemata wird am wichtigsten Schema, das des Charakters, erklärt. Nachfolgende Zeilen-Referenzen beziehen sich auf das Listing 2.1.

Darin kann direkt abgelesen werden, dass die einfachen Merkmale (`experience_level`, `race`, `culture`, `profession`) Zeichenfolgen (englisch: *strings*) sind; diese spiegeln den Namen der Merkmalsausprägung aus einem Regelbuch wider. Das Konzept der Schemata sieht vor, dass sämtliche konkrete Werte, neben Zahlen, Zeichenfolgen (Text) sind. Grund ist, dass konkrete Werte nicht statisch modelliert werden können, da sie sich erst dynamisch zur Laufzeit, je nach zu verwendenden Regelbüchern (gemäß der Anfrage), ergeben.

Auch die Merkmale `talents` und `combat_techniques` geben die Merkmalsausprägung (Name) als Freitext an; dies ist jedoch schwerer abzulesen. Hinter der Typen-Definition (z. B. Zeile 356 bis 359) steht ein sog. *Wörterbuch* (englisch: *Dictionary*) von Python; dies kann man als Map von Schlüssel-Wert-Paaren begreifen, wobei ein Schlüssel, aufgrund des *JavaScript Object Notation (JSON)*-Formats, stets ein Text ist und der Wert als positive Ganzzahl definiert wurde. Der Wert ist bei diesen Merkmalen die Stufe. Ein Schlüssel kann nur einmalig in einem solchen Wörterbuch (und auch im JSON-Type `object`) existieren. Dies deckt sich mit der Modellierung des Charakters, da bei einer mehrfachen Existenz eines Merkmals mit Stufe nur der mit der höchsten Stufe relevant ist, da alle Stufen darunter ableitbar sind.

Bei den Merkmalen `advantages` und `disadvantages` ist es etwas leichter zu erkennen (siehe Zeile 368 bis 377). Dies sind Listen von drei-elementigen Tuple, dargestellt durch eine zwei-dimensionale Liste, wobei die Verschachtelte genau drei Elemente verlangt. Das erste Element ist (wie bisher) der Name der Merkmalsausprägung. Das letzte Element ist die Stufe. Das Mittlere ist ein Text, welches insb. auch leer sein darf. Dies wird benötigt, da einige Vor- und Nachteile dahingehend mehrfach vorkommen können, dass sie auf andere Merkmalsausprägungen oder gar nicht modellierbare, frei wählbare Definitionen referenzieren können und daher nur in der Kombination spezifisch sind [vgl. 76; 77; 78; 79].

Auf eine tiefere Unterscheidung wird zunächst verzichtet. Dies führt dazu, dass, wenn ein Name einer Merkmalsausprägung auch in anderen Merkmalen vorkommt, nicht differenziert werden kann welches Merkmal referenziert wird. Diese Ausnahmefälle können jedoch leicht umgangen werden, indem das vermeintliche Duplikat bei der Modellierung im Regelbuch z. B. als Präfix den Namen des Merkmals bekommt.

Listing 2.1: Charakter Schema in api-contract.yaml

```yaml
    Hero:
      title: Hero
      required:
        - name
        - experience_level
        - race
        - culture
        - profession
        - talents
        - combat_techniques
        - advantages
        - disadvantages
      type: object
      properties:
        name:
          title: Name
          type: string
        experience_level:
          title: Experience Level
          type: string
        race:
          title: race
          type: string
        culture:
          title: Culture
          type: string
        profession:
          title: Profession
          type: string
        talents:
          title: Talents
          type: object
          additionalProperties:
            minimum: 0.0
            type: integer
        combat_techniques:
          title: Combat Techniques
          type: object
          additionalProperties:
            minimum: 0.0
            type: integer
        advantages:
          title: Advantages
          type: array
          items:
            maxItems: 3
            minItems: 3
            type: array
            items:
              - type: string
              - type: string
              - minimum: 0.0
                type: integer
        disadvantages:
          title: Disadvantages
          type: array
          items:
            maxItems: 3
            minItems: 3
            type: array
            items:
              - type: string
              - type: string
              - minimum: 0.0
                type: integer
      additionalProperties: false
```

2.4.2 Endpunkte

Der vollständige Schnittstellenvertrag liegt im Anhang A.1 auf S. 53 vor. Im Generellen sind die Anfrage-Körper (englisch: *request bodies*) und Antworten (englisch: *responses*) im Format ***JSON***. Nachfolgend werden die vorgesehenen Endpunkte erklärt; dabei wird unter „Regelbuch“ auch Regelbuch-Erweiterung verstanden.

Beim Entwurf wurde darauf geachtet, dass Eingabe-Parameter nach Möglichkeit stets am gleichen „Ort“ anzugeben sind, um die Verwendung der *API* konsistent und einfach zu halten. So ist z. B. die Liste der zu verwendenden Regelbücher stets als sog. Query-Parameter anzugeben.

/api/meta/rulebook/list gibt die Liste der verfügbaren Regelbücher zurück. Diese Information soll dynamisch ermittelt werden, um den Wartungsaufwand manueller Pflege fester Listen zu verhindern und um insb. zur Laufzeit „Kaputte“ herauszufiltern.

Beispiel Anfrage:
```
GET /api/meta/rulebook/list
```
Beispiel Antwort:
```
["dsa5","dsa5_aventurisches_kompendium_2"]
```

/api/meta/feature/list gibt für ein Merkmal die Liste der verfügbaren Ausprägungen, unter Beachtung zu verwendender Regelbücher, zurück. Diese Information muss dynamisch, entsprechend der auszuwertenden Regelbücher, ermittelt werden.

Beispiel Anfrage:
```
GET /api/meta/feature/list?feature=race&
↪ rulebooks=dsa5
```
Beispiel Antwort:
```
["Elfen","Halbelfen","Mensch","Zwerg"]
```

/api/hero/validate stellt den wichtigsten Endpunkt dar. Es wird geprüft, ob der gegebene Charakter, unter Beachtung der zu verwendenden Regelbücher, gültig ist. Dabei werden etwaige Regelverstöße detailliert als Fehler bzw. Warnung zurückgegeben. Nur Fehler führen dazu, dass ein Charakter für ungültig erklärt wird.

Beispiel Anfrage:
```
POST /api/hero/validate?rulebooks=dsa5
BODY <Listing 2.2>
```
Beispiel Antwort:
```
<Listing 2.3>
```

Folgende Endpunkte wurden bereits für die Verwendung im Frontend vorgesehen, werden jedoch vom Backend nicht ausimplementiert, da diese nicht Fokus dieser Arbeit sind:

/api/hero/save persistiert den Charakter für den Benutzenden privat in der Datenbank. Zur Identifikation dient das Paar aus den Namen des Charakters und Benutzenden.

Beispiel Anfrage:
```
PUT /api/hero/save?rulebooks=dsa5
BODY <Listing 2.2>
```
Beispiel Antwort:
```
<leeres HTTP OK>
```

/api/hero/export exportiert einen gespeicherten Charakter in ein Format, dass auch von anderen Softwares unterstützt wird. Das Antwort-Format ist daher nicht *JSON*.

Beispiel Anfrage:
```
GET /api/hero/export?hero_name=UncleBob
```
Beispiel Antwort:
```
<Zielformat>
```

/api/hero/delete löscht den Charakter für den Benutzenden aus der Datenbank. Zur Identifikation dient das Paar aus den Namen des Charakters und Benutzenden.

Beispiel Anfrage:
```
DELETE /api/hero/delete/hero_name=UncleBob
```
Beispiel Antwort:
```
<leeres HTTP OK>
```

Listing 2.2: Beispiel Anfragekörper zu einem invaliden Helden

```
{
  "name": "UncleBob",
  "experience_level": "Legendary",
  "race": "Elfen",
  "culture": "Auelfen",
  "profession": "Söldner",
  "talents": {
    "Körperbeherrschung": 2,
    "Kraftakt": 3,
    "Selbstbeherrschung": 4,
    "Zechen": 5,
    "Menschenkenntnis": 3,
    "Überreden": 3,
    "Orientierung": 4,
    "Wildnisleben": 3,
    "Götter & Kulte": 3,
    "Kriegskunst": 6,
    "Sagen & Legenden": 5,
    "Handel": 3,
    "Heilkunde Wunden": 4
  },
  "combat_techniques": {
    "Armbrüste": 10,
    "Raufen": 10,
    "Stangenwaffen": 9,
    "Zweihandschwerter": 10
  },
  "advantages": [
    ["Begabung", "Singen", 1],
    ["Begabung", "Musizieren", 1],
    ["Beidhändig", "", 1],
    ["Dunkelsicht", "", 2]
  ],
  "disadvantages": [
    ["Körpergebundene Kraft", "", 1],
    ["Lästige Mindergeister", "", 1],
    ["Wahrer Name", "", 1]
  ]
}
```

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

```
{
  "valid": false,
  "errors": [
    {
      "type": "missing_level",
      "message": "Heros 'profession' is missing minimum level '3' for 'talent' of
↪  'Körperbeherrschung'.",
      "parameter": {
        "caused_feature": "profession",
        "caused_feature_value": "Söldner",
        "referred_feature": "talent",
        "referred_feature_value": "Körperbeherrschung",
        "min_level": 3
      }
    }
  ],
  "warnings": []
}
```

2.5 Regelwerk-Interpretation

Dieses Kapitel setzt Kenntnisse im Bereich *LP* voraus, insb. die Syntax und Semantik von Clingo. Eine Einstiegshilfe 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

```
%% Strukturelle Darstellung mit Konventions-konformern Argumenten-Variable-Namen
experience_level(EL).
race(R).
culture(C).
profession(P).
talent(T,LVL). %% LVL := Stufe (englisch für 'level')
combat_technique(CT,LVL).
advantage(A,USES,LVL). %% USES := Referenz (Text)
disadvantage(DA,USES,LVL).
%% weitere Ähnliche wie Eigenschaften, Sonderfertigkeiten, Zauber und Liturgien möglich
```

```
%% Beispiel, wobei bestimmte Fakten mehrfach vorkommen dürfen z.B. Talente
experience_level("Average").
race("Elfen").
culture("Auelfen").
profession("Söldner").
talent("Kraftakt",3). talent("Zechen",3).
combat_technique("Armbrüste",10).
advantage("Begabung","Singen",1).
disadvantage("Wahrer Name","",1).
```

[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.

```
%% Eigene Existenz bekannt machen.
rulebook("dsa5_aventurisches_götterwirken_2").
%% Anforderung im Regelbuch 'DSA5 Aventurisches Götterwirken II':
rulebook_depends("dsa5_aventurisches_götterwirken_2", "dsa5").
%% Allgemeine Überprüfung der Abhängigkeiten:
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.

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

VV.3 Diese somit bekannten Merkmalsausprägungen werden dann überprüft.

```
%% Finde unbekannte Ausprägungen
unknown(profession(P)) :- profession(P), not known_profession(P).
unknown(combat_technique(CT)) :- combat_technique(CT,_), not known_combat_technique(CT).
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 6]. Wobei weitere Regelbücher diese Begrenzung auflockern könnten. **AR.1**

```
max_count(culture,1). %% nur Fakt mit höhstem Wert entscheidend
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, Schritt 8]. Somit sind negative Zahlen ausgeschlossen. Es ist nicht vorgesehen, dass andere Regelbücher diese Grenze verschieben. **AR.2**

```
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] **AR.3**

```
%% Model des Erfahrungsgrades: known_experience_level(<name>,<ES>,<FS>,<KTS>,<EP>,<ZL>,<FZ>).
%%   ES  := max. Eigenschaftsstufe        %%   EP  := max. Eigenschaftspunkte
%%   FS  := max. Fertigkeitsstufe         %%   ZL  := max. Anzahl Zauber und Liturgien
%%   KTS := max. Kamptechnikstufe         %%   FZ  := davon max. Fremdzaumber
known_experience_level("Average",13,10,10,98,10,1).
%% Finde Talente, welche die maximale Stufengrenze überschreiten
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.

```
%% Eine bestimmte Spezies ist gefordert. Spezies kann nur eine Ausprägung haben.
requires(culture("Auelfen"),race("Elfen")).
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.

```
%% Ein bestimmter Vorteil ist gefordert. Es kann mehrere Vorteile geben.
requires(race("Elfen"),advantage("Zauberer","",1)).
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].

```
requires(profession("Söldner"),talent("Kriegskunst",6)).
%% logisches 'oder' durch '#count': das Merkmal fehlt oder die Stufe ist nicht erreicht
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.

```
%% 'Söldner' benötigt aus mehreren Möglichen eine (!) Kampftechnik auf Stufe 10
requires(profession("Söldner"),any_of(1,combat_technique,("Hiebwaffen","Schwerter","Stangenwaffen"),10)).
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)

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

Regelbuch-spezifische weiche Regeln:

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

```
has_usual(race("Elfen"),culture("Auelfen")).
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 oder untypisch [vgl. 32]. Es werden beide Regeln in einem Beispiel gezeigt. **WR.2**

```
has_typical(race("Elfen"),advantage("Dunkelsicht","",2)).
missing_typical(race(R),advantage(A,USES,LVL)) :- race(R),
↪  has_typical(race(R),advantage(A,USES,LVL)), not advantage(A,USES,LVL).

has_atypical(race("Elfen"),disadvantage("Blutrausch","",1)).
atypical(race(R),disadvantage(DA,USES)) :- race(R), has_atypical(race(R),disadvantage(DA,USES,_)),
↪  disadvantage(DA,USES,_).
```

Anmerkung

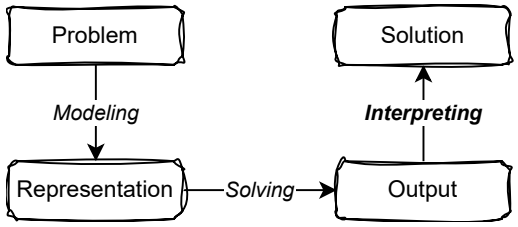
Wie zu sehen ist, ist die Struktur der formalen Regeln in WR.1 und WR.2 gleich. Diese kann auch auf die Semantik von „empfohlen“ und „ungeeignet“ übertragen werden, weshalb sie nicht zusätzlich aufgeführt werden [vgl. 82].

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.

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



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_lvl_exceeded`
- `missing_min_lvl`
- `max_count_exceeded`
- `unusable_by`
- `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.

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). 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] **PS.1**

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

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

Die zweitwichtigste Datei ist das sog. „`Makefile`“ (Anhang A.4 auf S. 69). Es dient 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.4**

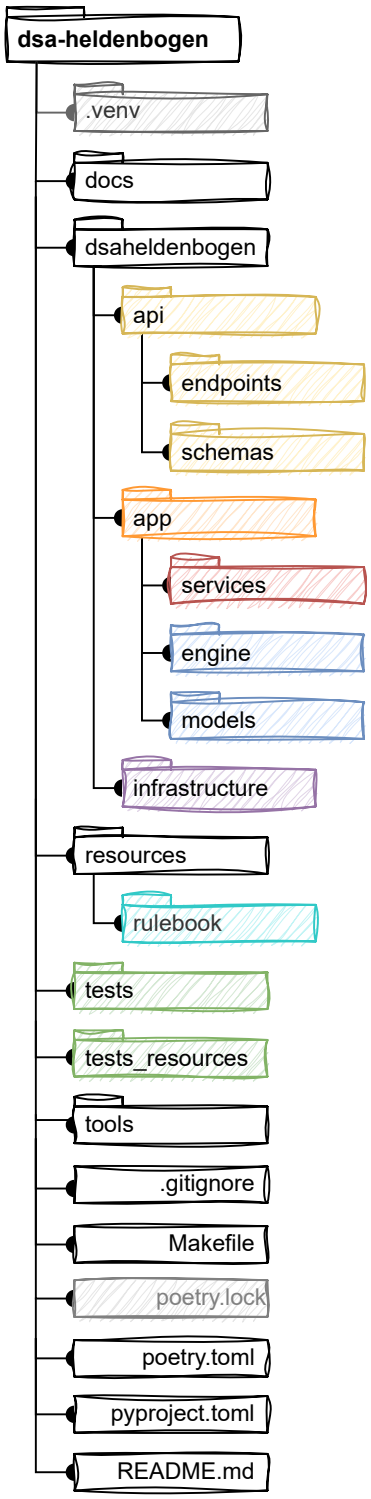
PS.5 Zur Wahrung der Portabilität und Vorbeugung von Versionskonflikten mit bereits installierten Abhängigkeiten, wird eine sog. „virtuelle Umgebung“ geschaffen; diese liegt unter „`.venv`“. Darin werden, anstatt global, Projekt-spezifisch die Abhängigkeiten installiert. Genutzt wird dazu das sehr verbreitete, funktionsreiche und leicht zu bedienende poetry[27]. 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 Container[28], 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`
[28] z. B. als Docker: `https://www.docker.com/`

Abbildung 3.1
Projektstruktur und -architektur
(Eigene Darstellung)

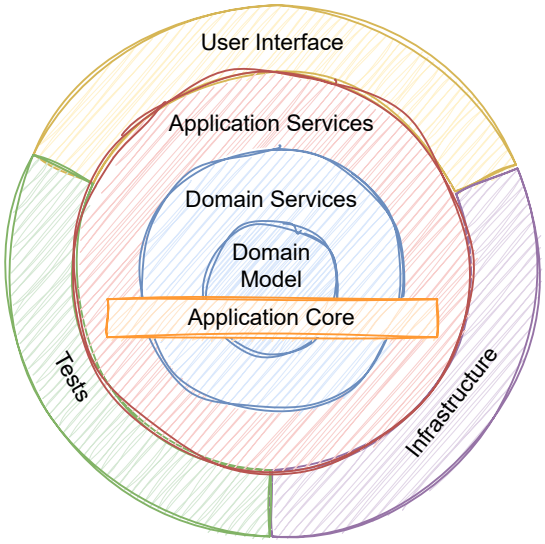


Zur Versionsverwaltung des Quellcodes wird das verbreitete und quelloffene GiT[29] eingesetzt. Mit der Datei „`.gitignore`“, werden Pfade und Dateien, welche z. B. durch Frameworks oder die eingesetzte Entwicklungsumgebung erstellt werden und keine Relevanz für die Software selbst haben, als zu ignorieren markiert. Dazu gehören „`.venv`“ und „`poetry.lock`“. Zur Veröffentlichung der Software wird die größte und bekannteste GiT-Plattform GitHub[30] 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`
[31] `https://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 äußeren 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 Benutzender, 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. 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“. **AT.6**

Das Package „engine“ enthält Klassen, welche solche „Operation“-Methoden bereitstellen. 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. **AT.7**

Unter „`infrastructure`“ befindet sich der Python-Code, welcher als Schnittstelle zwischen 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. **AT.8**

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. **AT.9**

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:

[32] https://clean-code-developer.de/die-grade/roter-grad/#Integration_Operation_Segregation_Principle_IOSP
[33] https://clean-code-developer.de/die-grade/roter-grad/#Keep_it_simple_stupid_KISS
[34] https://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 Klassenname 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 „excpetions.py“ oder „errors.py“ zu sammeln. Dies ist in der Python-Gemeinschaft bei Bibliotheken und Frameworks üblich.

[35] https://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 zwei Design-Entscheidungen vorgeschlagen: Kommentare, welche einen nicht temporär auskommentierten Quellcode darstellen, wie Dokumentationen, werden mit einem zusätzlichen (doppelten) Kommentar-Zeichen (%%) gekennzeichnet. **DE.7**

Die zweite Design-Entscheidung zu *LPs* ist, dass es keine Zeichenbegrenzung für Zeilen 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 konkretem Lesebedarf können Zeilen natürlich temporär, also nicht in die Versionsverwaltung übertragen, umgebrochen werden. **DE.8**

Die strukturelle Freiheit bei der Implementierung von Regelbüchern aus Punkt AT.3 auf 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) **DE.9**

Weiter wird die Freiheit durch die, der `Engine` (siehe nachfolgendes Kapitel), vorgegebene 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. **DE.10**

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). **DE.11**

Dadurch können Regelbücher weitere (Zwischen-)Validierungsschritte frei definieren. Dazu 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.

Tabelle 3.1: Von der `Engine` direkt genutzte *LP*-Unterprogramme

Name	**Beschreibung**
`base`	Wird immer ausgeführt. Im `common.lp` werden z. B. genutzte Atome vor-deklariert, um Warnungen zu vermeiden, wenn diese fehlen.
`rulebook_usable`	Wird vor jeder Nutzung eines Regelbuchs ausgeführt und beschreibt, entsprechend dem Punkt VV.1 auf S. 26, die Abhängigkeiten zu anderen Regelbüchern.
`meta`	Wird vor den Charakter-Validierungsschritten ausgeführt, um z. B. Regelbuch-spezifische Schritte, gemäß Punkt DE.11 auf S. 37, zu finden.
`world_facts`	Enthält alle Spielwelt-Fakten, wie bekannte Merkmalsausprägungen und dessen Regeln (Voraussetzungen).
`hero_facts`	Enthält bzw. generiert die Charakter-Fakten aus dem Kontext-Objekt.
`validate_hero_step_50`	Charakter-Validierungsschritt zur Vorprüfung der Validierungsvoraussetzung gemäß Punkt VV.3 auf S. 26.
`validate_hero_step_100`	Charakter-Validierungsschritt zur Prüfung, ob die Spezies nutzbar ist.
`validate_hero_step_150`	Charakter-Validierungsschritt zur Prüfung der Anforderungen der Spezies.
`validate_hero_step_200`	Charakter-Validierungsschritt zur Prüfung, ob die Kultur nutzbar ist.
`validate_hero_step_250`	Charakter-Validierungsschritt zur Prüfung der Anforderungen der Kultur.
`validate_hero_step_300`	Charakter-Validierungsschritt zur Prüfung, ob die Profession nutzbar ist.
`validate_hero_step_350`	Charakter-Validierungsschritt zur Prüfung der Anforderungen der Profession.
`validate_hero_step_400`	Charakter-Validierungsschritt zur Prüfung, ob die Vor- und Nachteile nutzbar sind.
`validate_hero_step_450`	Charakter-Validierungsschritt zur Prüfung der Anforderungen der Vor- und Nachteile.
`validate_hero_step_500`	Charakter-Validierungsschritt zur Prüfung, ob die Fähigkeiten (Talente und Kampftechniken) nutzbar sind.
`validate_hero_step_550`	Charakter-Validierungsschritt zur Prüfung der Anforderungen der Fähigkeiten (Talente und Kampftechniken).

3.4 Die wichtigsten Komponenten

Nach der Formalisierung eines Regelwerks muss, entsprechend der Abbildung 2.1 auf S. 30, das *LP* ausgeführt und interpretiert werden. Die Orchestrierung dessen ist die Aufgabe der `Engine` und wird nachfolgend, anhand der Charakter-Validierung, aufgezeigt. Dabei wird Bezug auf konkrete Klassen genommen, welche daher in Tabelle 3.2 aufgelistet werden.

Tabelle 3.2: Übersicht wichtigster Komponenten
(Reihenfolge nach nachfolgender Nennung)

Name	Schicht	Anhang
`Engine`	Domänenservice	A.7 auf S. 73
`ClingoExecutor`	Infrastruktur	A.8 auf S. 76
`common.lp`	Ressource	A.6 auf S. 71
`HeroWrapper`	Infrastruktur	A.9 auf S. 77
`Collector`	Domänenservice	A.10 auf S. 79
`HeroValidationInterpreter`	Domänenservice	A.11 auf S. 80

Die `Engine` ist die den formalisierten Regelbüchern übergeordnete Steuerungseinheit. Sie definiert z. B. ein geordnetes Standard-Set an Charakter-Validierungsschritten (siehe Tabelle 3.1), welches gemäß Punkt DE.11 erweitert werden kann. Eine Charakter-Validierung läuft wie folgt ab: Jeder Schritt wird einzeln ausgeführt. Dabei werden Validierungswarnungen Schritt-übergreifend gesammelt. Bei einem Validerungsfehler beendet die `Engine` die Charakter-Validierung mit der Exception `HeroInvalidError`, welche neben dem Fehler auch die bisher gesammelten Warnungen beinhaltet. Ansonsten werden die potenziellen Validierungswarnungen zurückgegeben.

Dazu greift die `Engine` auf den `ClingoExecutor` zu. Dieser gehört der Infrastrukturschicht an und abstrahiert die Verwendung der Clingo-*API*. Er erhält konkrete Anweisungen, welche *LPs* zu laden und welche Unterprogramme auszuführen sind. Für jeden Durchlauf werden von Clingo die *LP*-Dateien neu geladen, sowie die gewünschten Unterprogramme geerdet[36] (englisch: *grounded*) und gelöst. Dabei ist anzumerken, dass, wie bereits in Kapitel 2.5.5 auf S. 30 beschrieben, bisher alle „Probleme“ (*LPs*) generell als lösbar modelliert sind.

Neben den frei-wählbaren Regelbücher-*LPs*, wird das `common.lp` immer zusätzlich mitgeladen. Dieses enthält Regelbuch-übergreifende, also allgemeine, Definitionen wie Spielwelt-Regeln und Charakter-Fakten. Dadurch wäre es möglich andere Basisregelbücher zu implementieren, ohne Gemeinsamkeiten erneut zu implementieren, wie z. B. die allgemeingültigen Regeln auf S. 27.

[36] Das Erden ist ein normaler Prozess bei *ASP*, bei dem alle Variablen „weg“ transformiert werden.

Jeder Charakter-Valididierungsschritt ist ein eigener Durchlauf und bekommt als Kontext die Klasse `HeroWrapper`. Dieses stellt die Felder des Charakter-Domänenmodells als Clingo-*Symbole* (Charakter-Fakt), wie in Kapitel 2.5.2 auf S. 25 dargestellt, bereit. Das aus dem jeweiligen Durchlauf resultierende *answer set* wird dann von der `Engine` an den `Collector` übergeben, um die entsprechend gesuchten Ergebnis-Fakten herauszufiltern. Anschließend werden diese vom `HeroValidationInterpreter` in entsprechende Domänenmodelle (Validierungswarnungen oder -fehler) konvertiert. **Anmerkung:** Diese Klasse ist von kritischer Bedeutung, da es die genaue Interpretation des *ASP*-Ergebnisses beherbergt.

3.5 Qualitätssicherung

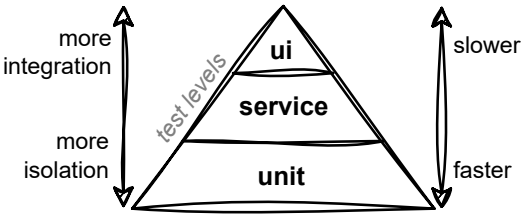
Die automatisierte Qualitätssicherung ist ein unerlässliches Werkzeug bei der Entwicklung nachhaltig korrekter und wartbarer Software. Zusätzlich ist es aufgrund des dritten Punktes der abgeleiteten technischen Anforderungen an quelloffene Software auf S. 10 von hoher Bedeutung für dieses Projekt. Die Qualitätssicherung in der Softwareentwicklung ist ein vielfältiges Feld und in der Breite wie Tiefe komplex; es gibt verschiedene Standards wie [93] und [94], als auch Standards und Definitionen indirekt geprägt durch Zertifizierungen von Testenden durch ISTQB [95].

Dabei hat sich die weltweite Gemeinschaft der Software-Entwickelnden bis heute nicht auf eine Terminologie im Bereich der Tests festlegen können. Es gibt durchaus Definitionen, welche eher verbreitet sowie allgemein anwendbar sind und auf mehr Zustimmung stoßen als andere. Entscheidend ist, dass es innerhalb der Software eine klare Definition dessen gibt. [vgl. 96]

So kann unterschieden werden zwischen der Testintention, dem Testverfahren und der Teststufe. Ersteres beschreibt welchen Zweck der Test verfolgt, wie z. B. das Testen von (nicht-)funktionalen Anforderungen oder Leistungen. Das Testverfahren gibt an, ob dieser Test statisch oder dynamisch ist, also ob die Software ausgeführt wird, und ob der Test Kenntnisse über die Implementierung hat, also ein „black/white box“ Test ist. Letzteres kategorisiert in welchem Rahmen sich der Test bewegt bzw. welchem Umfang der Test hat. So sind i.d.R. z. B. Komponententests (englisch: *unit tests*) schnelle und kleinteilige Tests, welche einfach zu entwickeln sind und von keiner anderen Komponente abhängen. Währenddessen haben Ende-zu-Ende Tests meist eine längere Ausführungszeit und tendenziell einen komplexeren Aufbau, bei der die Applikation, in der Gesamtheit und dem Zusammenspiel seiner Komponenten, zur Laufzeit über die *API*, mit gegebenen Anfragen und erwarteten Antworten, überprüft wird. [vgl. 97]

Die beschriebene Kategorisierung von Teststufen entspricht dem gängigen Modell der Testpyramide (siehe Abbildung 3.3). Es besagt im Wesentlichen, dass eher mehr schnelle und isolierte Komponententests und weniger langsame und integrative Ende-zu-Ende (ui) Tests zu schreiben sind. [vgl. 98]

Abbildung 3.3
Testpyramide (Anlehnung an [98])

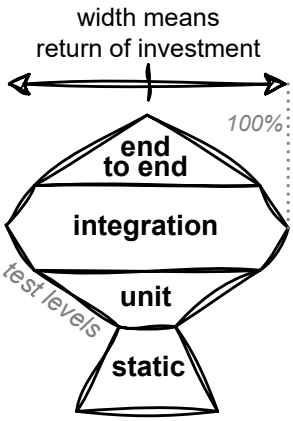


Welches Testmodell zu wählen und damit welche Tests (in welchen Teststufen) letztlich geschrieben werden, hängt von unterschiedlichsten Faktoren ab, wie z. B. dem Entwicklungsaufwand, der Testbarkeit der Software bzw. des Quellcodes selbst und Laufzeitanforderungen.

In diesem Projekt wird das Modell der Testtrophäe nach [99] verfolgt. Diese drückt aus, wie hoch der Nutzen einer Teststufe im Vergleich zum notwendigen Entwicklungsaufwand ist (siehe Abbildung 3.4). Fokussiert werden demnach Tests mit dem besten Verhältnis, was dem dritten Punkt der abgeleiteten technischen Anforderungen auf S. 10 und dem impliziten Ziel einer möglichst wartbaren Software entspricht. **QS.1**

Bei der Testintention unterscheidet das Projekt zwischen der Quellcode-Qualität und der Anforderungseinhaltung. **QS.2**

Abbildung 3.4
Testtrophäe
(Anlehnung an [99])



Die Analyse der Quellcode-Qualität kann statisch und dynamisch durchgeführt werden. Beim Letzteren kann z. B. die Testabdeckung ermittelt werden. Die Testabdeckung drückt aus wie viele Zeilen des Quellcodes durch die vorhandenen Tests abgedeckt sind und gegeben somit einen sehr einfachen Indikator für blinde Flecke. Eine hohe Testabdeckung ist nicht gleichzusetzen mit einer hohen Testqualität. Das Messen der Testqualität ist ein schwieriges Feld; erwähnt sei als bekanntes Mittel das sog. „Mutation Testing“[37]. Bei der statischen Quellcode-Analyse wird z. B. die Einhaltung von Programmierkonventionen und vorgegebener Quellcode-Formatierung, die Existenz von ungenutzten Variablen und Imports sowie Programmierfehler (wie eine falsche Reihenfolge beim Fangen von Exceptions) und schlechter Praktiken (sog. „code smells“) überprüft. Bei der Anforderungseinhaltung geht es um die Erstellung von Tests, welche Funktionalitäten auf bekannte

[37] die Populärsten Python-Frameworks dazu sind mutmut (`https://github.com/boxed/mutmut`) und mutpy (`https://github.com/mutpy/mutpy`)

Anforderungen überprüft. Die Relevanz dessen lässt sich u. a. durch die Verbreitung des „test-driven development“ (TDD) beispielhaft zeigen. Es ist eine Entwicklungsmethode, bei der die Erstellung des Tests vor der Implementierung der Funktionalität gestellt wird. [vgl. 85, Kapitel 8; 73, Kapitel 10]

Dabei können Anforderungen technisch, z. B. eine Methode soll in einem bestimmten Fall eine Exception werfen, oder fachlich, z. B. ein Regelbuch ist verwendbar, wenn all seine Abhängigkeiten gegeben sind, sein.

QS.3 Alle Tests werden automatisch vor dem Start der Applikation ausgeführt und können während der Entwicklung beliebig oft wiederholt werden (vgl. Anhang A.4 auf S. 69).

QS.4 Zur Wahrung der Quellcode-Qualität werden die Werkzeuge pylint[38] (strikte Prüfung von Programmierkonventionen]), bandit[39] (Finden von bekannten Sicherheitslücken]) und mypy[40] (statische Prüfung der Typisierung) in der Standardkonfiguration eingesetzt. Diese können „dem Sockel der Testtrophäe“ zugeordnet werden.

Nachfolgend wird dargelegt, wie die fachliche Anforderungseinhaltung gewährleistet wird.

3.5.1 Prüfung der Regelbücher-Struktur

Um zu gewährleisten, dass alle implementierten Regelbücher von der `Engine` nutzbar sind, werden diese auf entsprechenden Anforderungen durch den `RulebookValidator` überprüft (vgl. Anhang A.12 auf S. 85 und Anhang A.13 auf S. 87). Diese Prüfung wird auch genutzt, wenn per *API* die verfügbaren Regelbücher abgefragt werden. Konkret wird geprüft, dass die vorgeschlagene Dateistruktur (Punkt DE.9 auf S. 37) eingehalten wird, die notwendigen Unterprogramme für die `Engine` vorhanden sind (Punkt DE.10 auf S. 37) und das ein Regelbuch nur sich selbst als Fakt deklariert (sich also nicht als ein Anderes ausgibt), wie es entsprechend im *LP*-Beispiel von VV.1 auf S. 26 als notwendige Gegebenheit gezeigt wird. Dies kann als Komponententest im Testtrophäen-Modell angesiedelt werden.

3.5.2 Prüfung einzelner Funktionalitäten

Gemäß dem Modell der Testtrophäe, wurde der Schwerpunkt auf die Erstellung von Integrationstests gelegt. Das Integrative an den hier beschriebenen Tests ist, dass diese die technische Korrektheit der Funktionalität über die Ebene der Applikationsservice-Schicht prüfen und dabei alle tieferen Schichten durchläuft.

[38] `https://github.com/pylint-dev/pylint`
[39] `https://github.com/PyCQA/bandit`
[40] `https://github.com/python/mypy`

Als Beispiel sei der `TestMetaService` (Anhang A.14 auf S. 87) aufgeführt, welcher den `MetaService` (Anhang A.15 auf S. 88) testet. Dieser Service stellt u. a. die Funktionalität bereit für ein Merkmal alle bekannten Ausprägungen, unter Angabe der zu nutzenden Regelbücher, aufzulisten. Damit solche Tests unabhängig „echter“ Regelbücher durchgeführt werden können und damit wartungsarm bleiben, wurde eine Test-Variante (`TestingRulebook`) der internen Representation von Regelbüchern eingeführt, welche Regelbücher aus dem separaten `tests_resources/rulebook` Ordner lädt. So ist es möglich für verschiedene Testklassen und -fälle unterschiedliche Regelbücher zu definieren, wodurch auch Negativtests ermöglicht werden. Um die Implementierung/Nutzung dieser Test-Regelbücher zu verbessern, wurde die `Engine` um eine `TestingEngine` (Anhang A.16 auf S. 89) erweitert. Diese fügt das `common.lp` nicht automatisch hinzu und führt keine Validierung durch, ob ein Regelbuch nutzbar ist. Das Prinzip von „Favour Composition over Inheritance“[41] (FCoI) ermöglicht es Tests die `TestingEngine` zu nutzen, indem sie bei der Erstellung des zu testenden Service diesen übergeben können.

3.5.3 Prüfung gesamtheitliches Zusammenspiel

Um (un)vermeidliche blinde Flecken, durch fehlende Komponenten- oder Integrationstests und insb. bei vergessenen Testfällen, wahrscheinlicher aufzudecken, werden Ende-zu-Ende Tests durchgeführt. In Gegensatz zu Komponententests werden bei diesen sämtliche Anforderungen, technisch wie fachlich, für einen Testfall gleichzeitig getestet. Bei diesen Ende-zu-Ende Tests wird die Applikation gestartet, um die *API* mit definierten Anfragen anzusteuern und gegen erwartete Ergebnisse zu testen. Alles dazwischen, also die Verarbeitung innerhalb der Applikation, ist im Sinne einer „schwarzen Box“ nicht bekannt und wird daher bei der Testfallerstellung nicht berücksichtigt. Beachtet wird lediglich der Schnittstellenvertrag.

Der Test `TestHeroApi` (Anhang A.18 auf S. 91) als solcher z. B. prüft den `/hero/validate` Endpunkt darauf, dass zu einem gültigen Charakter keine Validierungsfehler erzeugt werden (Warnungen werden ignoriert) und bei ungültigen Charakteren entsprechend korrekte Validierungsfehler zurückgemeldet werden.

Dessen Testfälle sind ausgelagert in separate Klassen definiert, da diese aufgrund der Länge die Testklasse zu unübersichtlich machen würde und damit sie von anderen Tests (sauber) wiederverwendet werden können. Diese Klassen bieten eine praktische Methode an alle definierten Testfälle aufzulisten, sodass beim Entfernen oder Hinzufügen von Fällen die Tests nicht manuell angepasst werden müssen. Der Testfall bei gültigen Charakteren ist

[41] https://clean-code-developer.de/die-grade/roter-grad/#Favour_Composition_over_Inheritance_FCoI

denkbar einfach; es besteht ausschließlich aus den vom Endpunkt angeforderten Daten: dem Charakter und die Regelbücher. Für den Fall von ungültigen Charakteren wird dieses Modell um Informationen zum erwarteten Validierungsfehler ergänzt; dazu gehört der Typ und eine Menge von Parameter. (siehe Anhang A.19 auf S. 92 und A.20 auf S. 93)

3.6 Schwierigkeiten

Während der Umsetzung des Konzeptes sind einige unerwartete oder nennenswerte Schwierigkeiten aufgekommen. Diese werden nachfolgend aufgelistet (die Reihenfolge hat keine Bedeutung):

SK.1 Das Absichern der Architektur ist ebenso wichtig wie klassische Funktionstests. Für Python war die Wahl eines passenden Frameworks jedoch nicht eindeutig. Es wurden mehrere quelloffene Frameworks (import-linter[42], pytest-archon[43], pytestarch[44]) gefunden, dennoch konnte keines vom Funktionsumfang überzeugen (im Vergleich zu ArchUnit[45] für Java). Die Entscheidung wurde daher offen gelassen.

SK.2 Es war nicht möglich für ein Merkmal mehrere Charakter-Fakten, welche mehrere Argumente haben, ohne Hilfsfakten zu erzeugen. Zwar können „external functions“ auch Listen bzw. Clingo-*Tuple* zurückgeben; diese werden aber nicht automatisch als Argumente entpackt. Für z. B. Talente sieht daher die Generierung der Merkmalsfakten wie folgt aus (vgl. Anhang A.6 auf S. 71, Zeile 49-62):

```
__talent_wrapper(@talents).
talent(T,LVL) :- __talent_wrapper((T,LVL)).
```

Die externe Funktion `@talents` ist wie folgt umgesetzt (vgl. Anhang A.9 auf S. 77):

```
% [ ... ]
def _map_skills(skills: List[Skill]) -> List[Symbol]:
    return [Tuple_([String(skill.name), Number(skill.level)]) for skill in skills]
% [ ... ]
class HeroWrapper:
    _hero: Hero
% [ ... ]
    def talents(self) -> List[Symbol]:
        return _map_skills(self._hero.talents)
% [ ... ]
```

[42] https://github.com/seddonym/import-linter
[43] https://github.com/jwbargsten/pytest-archon
[44] https://github.com/zyskarch/pytestarch
[45] https://www.archunit.org/

Zur Überprüfung der Existenz von Pflicht-*LP*-Unterprogrammen (gemäß Kapitel 3.5.1 auf S. 42) musste eine implizite Anforderung an die Deklarierung von *LP*-Unterprogrammen aufgenommen werden. Diese besagt, dass nach jeder Deklaration eines Unterprogramms ein zusätzlicher Fakt mit dessen Namen folgen muss. Um dies zu gewährleisten, sollte die Anforderung im `RulebookValidator` aufgenommen werden. **SK.3**

```
#program rulebook_usable.
program("rulebook_usable").
```

Notwendig ist dies, da die Clingo-Python-*API* keine Möglichkeit bietet auf die Existenz eines Unterprogramms zu prüfen. Selbst beim programmatischen Hinzufügen eines Unterprogramms (über diese *API*) zu einem *LP*, das bereits ein gleichnamiges Unterprogramm definiert hat, wird keine Fehlermeldung geworfen, sondern lediglich die Definition erweitert.

Das Prüfen einzelner (Regelbuch-spezifischer) Spielwelt-Regeln ohne Einflüsse Anderer ist nicht ohne eine unverhältnismäßig hohe Verschlechterung der Wartbarkeit möglich. Um die zu testende Spielwelt-Regel in einem Durchlauf zu erreichen, müssen sämtliche vorherige Charakter-Validierungsschritte positiv ausfallen. Es erfordert also Wissen und die Modellierung von nicht-relevanten Informationen für den Testfall, wie z. B. beim Testen einer Regel zu Kampftechniken, bei dem das Merkmal der Spezies nicht von Bedeutung ist, jedoch angegeben werden muss. Dabei können mögliche und irrelevante Charakter-Validierungswarnungen entstehen. **SK.4**

Der erstellte und nicht mehr verfolgte Test `TestHeroValidation` (Anhang A.17 auf S. 89) zeigt diese Abhängigkeit gut auf. Wie in Listing 3.1 zu sehen ist, wird zum Testen der fachlichen Anforderung „der Beruf muss verwendbar sein“, also einer allgemeinen Spielwelt-Regel, Angaben zu Merkmalen gemacht, welche vorab geprüft werden. Um dies zu vermeiden, müsste jede Spielwelt-Regel in einem separaten Unterprogramm definiert und damit einzeln aufrufbar gemacht werden. Tatsächlich bieten die bereits existierenden Charakter-Validierungsschritte eine solche Trennung, wenn auch nicht so kleinteilig. Die `TestingEngine` könnte derart angepasst werden, dass sie ermöglicht nur bestimmte Schritte auszuführen.

Listing 3.1: Auszug von Anhang A.17

```
    @parameterized.expand([
        (0, 'Mensch', 'Menschlichekultur', 'Händler'),
        (1, 'Mensch', 'Menschlichekultur', ''),
        (1, 'Mensch', 'Menschlichekultur', '_invalid_'),
        (1, 'Mensch', 'Menschlichekultur', 'Zauberweber'),
    ])
    def test_profession_usable(self, error_count: int, race: str, culture: str, profession: str):
```

SK.5 Das Definieren von sinnvollen und ausreichenden Testfällen auf der Teststufe der, in Kapitel 3.5.2 auf S. 42 vorgestellten, Integrations- als auch Ende-zu-Ende Tests war und ist schwierig. So gilt es, unter der Prämisse der Wartbarkeit, ein möglichst kleines Set an Testfällen zu definieren, welches dennoch eine möglichst vollständige Abdeckung der Funktionalitäten bietet. Ferner ist es für die Verständlichkeit von Tests äußerst hilfreich, wenn ein Testfall nur eine Absicht hat, wie z. B. eine bestimmte Funktion/Regel zu testen oder eine spezielle komplexe (und realitätsnahe) Kombination von Regelbüchern und Merkmalen. In einer idealen Welt würden z. B. bei einem Entwicklungsfehler nur wenige, aber dafür für die Ursache relevante, Tests fehlschlagen und so die Lösungsfindung vereinfachen. Das strikte Folgen dessen würde jedoch dafür sorgen, dass viele ähnliche Tests entstehen und damit der eingangs genannten Prämisse widersprechen. Aufgrund der Rahmenbedingungen des Projektes, eine „attraktive“ quelloffene Software für Mitwirkende zu sein und der schier unzähligen Kombinationsmöglichkeiten von Regelbüchern und Charakter-Merkmalen, wurde bisher und sollte auch künftig der Fokus auf die Sicherstellung der einzelnen Funktionalitäten gelegt werden und darüber hinaus nur besondere/wichtige komplexe Fälle getestet werden.

4 | Diskussion und Ausblick

Diskussion

In diesem Teil der Arbeit wird die entstandene Software begutachtet, bei der insb. Entscheidungen des Konzeptes und der Realisierung bewertet werden.

Ein Export-Endpunkt, welches einen Charakter in einem Format zurückgibt, das kompatibel zu anderen Softwares ist, ist zwar im Allgemeinen Benutzenden-freundlich; besser bzw. wichtiger ist jedoch ein Import-Endpunkt, welches es ermöglicht fremde Formate in das Eigene zu konvertieren. Dies würde Benutzenden die Migration von einer anderen Software erleichtern und damit die Zahl der Benutzenden und möglichen Mitwirkenden potenziell vergrößern. **DK.1**

Viele Regeln werden im *LP* für ähnliche Merkmale wiederholt. Zur Vermeidung dessen könnte man diese Merkmale auf eine gemeinsame Clingo-*Funktion* (*Gruppe*) abbilden, wodurch die Regel nur noch einmalig für diese „Gruppe“ definiert werden muss. Die Gruppe kann von den bestehenden Charakter-Fakten abgeleitet werden oder diese ersetzen; Letzteres ist jedoch nicht zu empfehlen, da Ersteres mehr Freiheiten bietet. Das Listing 4.1 zeigt dies beispielhaft. Dadurch müsste jedoch die Interpretation der Ergebnis-Fakten angepasst werden, weil das erste Argument nicht mehr (ausschließlich) eine Clingo-*Funktion* als Merkmal ist, sondern dessen Argumente direkte Argumente des Ergebnis-Fakts wären (vgl. Anhang A.11 auf S. 80). **DK.2**

Listing 4.1: Modellierung von strukturell ähnlichen Merkmalen in einer Gruppe

```
%% Beispiel einer Regel-Wiederholung
missing_min_lvl(talent(T,LVL),MIN) :- talent(T,LVL), MIN = 0, LVL < MIN.
missing_min_lvl(combat_technique(CT,LVL),MIN) :- combat_technique(CT,LVL), MIN = 0, LVL < MIN.
%% Option A: zusätzlich zu bekannten Charakter-Fakten
skill(talent,NAME,LVL) :- talent(NAME,LVL).
skill(combat_technique,NAME,LVL) :- combat_technique(NAME,LVL).
%% Option B: Charakter-Fakten ersetzen
skill(talent,NAME,LVL) :- __talent_wrapper((NAME,LVL)).
skill(combat_technique,NAME,LVL) :- __combat_technique_wrapper((NAME,LVL)).
%% Einzehlne Regel über eine Merkmalsgruppe
missing_min_lvl(FEATURE,NAME,LVL,MIN) :- skill(FEATURE,NAME,LVL), MIN = 0, LVL < MIN.
```

DK.3 Bei den konzipierten formalen Regeln (Punkte AR, HR und WR) wird bei den gesuchten Merkmalsausprägungen zwischen Merkmalen unterschieden, welche eine oder mehrere Ausprägungen annehmen können. Dies wurde aber nicht dynamisch in der Regel formalisiert, sondern lediglich zum IST-Stand während der Modellierung beachtet. Dadurch kann eine Inkonsistenz/Widerspruch entstehen, wenn entsprechende Änderungen der maximalen Anzahl (definiert wie in Punkt AR.1 auf S. 27) gemacht werden, aber betroffene Regeln nicht korrigiert werden. Auch die *API* beachtet dieses nicht; hier müsste die *API* generischer modelliert werden, sodass diese ausschließlich Listen für Merkmale akzeptiert. Wenn z. B. ein Regelbuch implementiert werden soll, welches mehrere Spezies erlaubt, dann wäre auch dies vollständig und wartungsfrei von der *API* unterstützt.

DK.4 Eine einfach nutzbare und gut verständliche Formalisierung der „count_by“ Methode aus Regel HR.4 auf S. 28 konnte nicht gefunden werden. Die Modellierung erfordert, dass die Auswahlliste kein Tuple von freier Länge ist, sodass „pooling“ genutzt werden muss. Damit die daraus generierten Spielwelt-Fakten der entsprechenden Auswahlliste zugeordnet werden können, muss ein Schlüssel vergeben werden. Dieser muss, in Kombination mit dem verursachendem und gesuchten Merkmal, einmalig unter allen Regelbüchern sein, sofern die Liste nicht erweitert werden soll. In Listing 4.2 wird ein Lösungsansatz skizziert.

Listing 4.2: Ansatz zur vollen Formalisierung von Regel HR.4 (Pseudo *LP*)

```
%%%% 'prof := profession' und 'combatT := combat_technique'
%% Auswahlliste mit drei Elementen ergibt drei Spielwelt-Fakten mit dem selben Schlüssel
requires(prof("Söldner"),any_of("uniqueID",1,combatT("Hiebwaffen";"Schwerter";"Stangenwaffen"),10)).
%% Ansatz nutzt "head aggregation", um drei Ergebnis-Fakten mit selbigen Schlüssel zu erzeugen, damit
↪  die Auswahlliste zusammengesetzt und dem Frontend zur Verfügung gestellt werden kann
CHOICES > #count{ KEY,CT: missing_level(prof(P),any_of(KEY,CHOICES,combatT,CT),MIN_LVL)):
↪  requires(prof(P),any_of(KEY,CHOICES,combatT(CT),MIN_LVL)), combatT(CT,LVL), LVL >= MIN_LVL }
↪  :- prof(P), requires(prof(P),any_of(KEY,CHOICES,combatT(_),MIN_LVL)).
```

DK.5 Außerdem sollte, für einen guten Wiedererkennungswert, die Struktur einer *LP*-Funktion beibehalten werden. Bei der Regel HR.4 wird jedoch die *LP*-Funktion „any_of“ im Ergebnis-Fakt anders als in der Spielwelt-Regel aufgebaut, sowie die Konvention KV.3 auf S. 24 gebrochen und stellt damit bereits jetzt eine technische Schuld dar.

DK.6 Der entstandenen Software mangelt es an tiefgehender loser Kopplung zwischen der `Engine` (Domäne) und Clingo (Infrastruktur). Dies ist zwar durchaus beabsichtigt (vgl. Punkt AT.9 auf S. 35), jedoch kann es für die zukünftige Testerstellung oder einem, aktuell unwahrscheinlichen, Wechsel des *ASP*-Frameworks hinderlich sein. Die Packages/Module sind sauber definiert, jedoch wurde das *DIP* und insb. das in Kapitel 3.2: Architektur auf S. 33 genannte „Dependency Injection Pattern“ zu stark vernachlässigt. Dadurch, dass vermehrt

und etwas „un-pythonic“ (also für Python-Verhältnisse unüblich) vorwiegend bzw. grundsätzlich Klassen erstellt wurden, kann auch dies mit überschaubarem Aufwand bei Bedarf nachgepflegt werden. Des Weiteren fehlt es an einem Domänenmodell der Ergebnis-Fakten, welches der `ClingoExecutor`, bzw. eine andere Infrastruktur-Klasse, an die `Engine` zurückgeben müsste, sodass die Applikationskern-Schicht gänzlich vom genutzten *ASP*-Framework unabhängig sein kann.

Als besonders praktisch hat sich die Konvention KV.2 auf S. 24 gezeigt. Bei Umsetzungsfehlern und Unklarheiten konnte man dadurch schnell die Quelle der Regel nachschlagen ohne jede Spielwelt-Regel explizit mit einer Quelle dokumentieren zu müssen. **DK.7**

Auch die Konvention KV.5 auf S. 24 ist positiv anzumerken. Dies hat die Verständlichkeit der Spielwelt-Regel-Formalisierung gefördert und ermöglichte ein einfaches Interpretieren der Ergebnisfakten. **DK.8**

Ausblick

Ähnlich zur Diskussion wird der IST-Zustand der Software betrachtet, jedoch um kommende wichtige Aufgaben zu benennen, welche nicht Fokus dieser Arbeit waren.

Zur Minimierung von Implementierungsfehlern in den Regelbücher-*LPs* sollte ein allgemeines Unterprogramm hinzugefügt werden, welches überprüft, dass alle Spielwelt-Fakten und -Regeln nur bekannte Merkmalsausprägungen nutzen. Dabei müssen entsprechende Regelbuchabhängigkeiten mit beachtet werden. Dieses Unterprogramm kann dann dem `RulebookValidator` als weiteres Kriterium hinzugefügt werden. **AB.1**

Durch die generierten Validierungsfehler und -warnungen weiß der Benutzende zwar was problematisch ist, aber nicht unbedingt wie es richtig wäre, z. B. bei einer Meldung, dass die gewählte Kultur nicht von der gewählten Spezies nutzbar ist. Es sollte ein Endpunkt bereitgestellt werden, welcher mögliche Merkmalsausprägungen des, anzugebenen, gesuchten Merkmals, unter Beachtung der zu verwendenden Regelbücher und vor allem der bereits gewählten Merkmale, identifiziert und zurückgibt. Das nächst zu wählende, also gesuchte, Merkmal könnte auch hierarchisch, gemäß der Schritte aus [30], automatisch ermittelt werden. Wenn z. B. für einen Charakter bereits eine Spezies und Kultur gewählt wurde, so könnte dieser Endpunkt mögliche Professionen benennen. Dazu muss das Schema des Charakters so angepasst werden, dass diese Merkmale keine Pflichtfelder sind. Entgegen der bisherigen Implementierung von *LPs*, entsprechend Kapitel 2.5.5 auf S. 30, wäre das zu entwickelnde *LP* ein klassisches, bei dem unter Vorgaben (unvollständiger Held) für ein Problem (Füllen eines Merkmals) mögliche Lösungen (nutzbare Merkmalsausprägungen für das Merkmal) ermittelt werden. **AB.2**

AB.3 Des Weiteren werden die Charakter-Validierungswarnungen immer ausgegeben. Es fehlt die Möglichkeit diese zu ignorieren bzw. als „von der Spielleitung akzeptierte Abweichungen“ zu markieren. Dazu kann die *API* angepasst werden, sodass der Validierungsendpunkt zusätzlich eine Liste der konkret zu ignorierenden Warnungen (so wie sie zuvor zurückgegeben wurden) annimmt.

AB.4 Die Charakter-Verwaltung ist zwar nicht Fokus dieser Arbeit, spielt jedoch eine entscheidende Rolle bei der Frage bzw. dem Problem „wie das komplette Regelwerk zu *DSA* formalisiert werden kann“, weshalb nachfolgend ein kurzer Ausblick gegeben wird. Ein herausfordernder Aspekt ist die Beachtung von *AP*, also das Führen eines *AP*-Kontos für einen Charakter und die Beachtung der *AP*-Kosten der Merkmale. Dies könnte durch zusätzliche Spielwelt-Fakten je Merkmalsausprägung modelliert werden. Dadurch muss die bereits existierende Modellierung der Merkmale nicht nachträglich von den Regelbüchern angepasst werden, was einer guten Modularität entspricht und damit einem geringeren Wartungsaufwand. Das *AP*-Konto müsste kumuliert je alle Eingänge und Ausgänge summieren, sodass als allgemeine Spielwelt-Regel modelliert werden kann, dass die Ausgaben nie höher als die Einnahmen sein dürfen. Insb. müsste für die Charakter-Verwaltung, also nach Spielbeginn, alle konkret gewünschten Veränderungen im *LP* bekannt sein, damit darauf Bezug genommen und entsprechend in Gänze validiert werden kann. In Listing 4.3 wird dies in Ansätzen gezeigt. Dafür kann ggf. das sog. „multi-shot solving“ relevant/hilfreich sein, bei dem erst der Charakter ohne Änderungen und anschließend sukzessiv jede Änderung einzeln validiert werden könnte [vgl. 100].

Der zweite Aspekt ist im Prinzip ähnlich zum Ersten. Bei diesem handelt es sich um die Modellierung der Charakter-Eigenschaften und -Basiswerte inkl. des Verfolgens der Veränderungen durch Merkmalsausprägungen und die Einhaltung von Grenzen.

Listing 4.3: Grob-Formalisierung von *AP*

```
%% Regelbuch-spezifische Spielwelt-Fakten (pro Merkmalsausprägung)
ap_cost(<feature:function>,<cost:number>)).
%% Charakter-Fakt
ap_account(<totalIncome>,<totalExpenses>).
%% allgemeine Spielwelt-Regel
ap_account_overdrawn :- ap_account(IN,OUT), IN < OUT.
%% bei Charakter-Verwaltung Änderung deklarieren
change(<type>,<feature>). %% 'type' ist Konstante 'add' oder 'remove'
%% dann: eigentliche Charakter-Fakten ableiten und Regeln prüfen
```

5 | Fazit

In dieser Arbeit konnte erfolgreich aufgezeigt werden, dass die Formalisierung des *DSA*-Regelwerks zur Charakter-Validierung möglich ist. Es hat sich dabei gezeigt, dass die Formalisierung mit dem wissenschaftlich erprobten *ASP* unkompliziert war und leicht verständlich ist, sowohl bei der Formalisierung selbst als auch bei der Integration des genutzten Frameworks clingo. Aufgrund des Umfangs konnten nicht sämtliche relevanten Regeln und Charakter-Merkmale beachtet werden, sodass erst in Zukunft bewertet werden kann, ob nach vollständiger Formalisierung dies so bleibt. Üblicherweise wird mit *ASP* eine Aussage über die Erfüllbarkeit eines Problems und mögliche Lösungen getroffen. Dieses Vorgehen wurde hier „gebrochen“, da (bisher) jedes erstelle *LP* erfüllbar sein musste und nicht mögliche Charaktere (Merkmals-Kombinationen) gefunden werden sollten, sondern vordefinierte abzuleitende Fakten als Validierungsergebnis generiert werden mussten.

Das Problem hoher Migrationsaufwände der bisherigen *DSA*-Hilfsmittel auf die aktuelle 5te-Ausgabe, aufgrund einer (angenommenen) starken Verflechtung der fachlichen Logik (dem Regelwerk) mit der technischen Implementierung oder zu komplexen und schlecht wartbaren Software, wurde in dieser Arbeit beachtet. Mit den im Konzept aufgestellten Konventionen, als auch den getroffenen Architektur- und Design-Entscheidungen sowie den qualitätssichernden Maßnahmen in der Realisierung wurde eine hohe Wartbarkeit der Software erreicht. Auch die Wahl von Python als Programmiersprache war dafür von Vorteil. Ebenfalls konnte in der Realisierung, durch einen modularen Aufbau der Regelbücher-*LPs*, die Umsetzung weiterer Regelbücher und Regelbuch-Erweiterungen berücksichtigt werden. Für die Implementierung anderer Regelwerksausgaben bedarf es jedoch geringfügiger Anpassungen, sofern die Charakter-Modellierung abweicht, wie z. B. durch neue Merkmale. Die entwickelte `Engine` mit dem generischen Ablauf der Charakter-Validierung, klaren Ein- (Charaktermodell) und Ausgabe-Formaten (Ergebnis-Fakten) sowie den ermöglichten Freiheiten bei der strukturellen Umsetzung der *LPs* stellt die zentralste Komponente dar. Die entstandene Software selbst ist insgesamt eine solide Grundbasis für ein langfristiges quelloffenes Projekt. Dies war neben der Formalisierung ein weiteres Ziel der Arbeit.

A | Anhang

Einträge die mit „<repo>/“ beginnen sind Dateien im quelloffenen Aufbewahrungsort.

A.1 <api>/openapi.yaml

API-Endpunkt zum Abrufen des Schnittstellenvertrags. Beispiele innerhalb des Schnittstellenvertrags wurden, aufgrund der sonst entstehenden Länge, entfernt.

```yaml
openapi: 3.0.2
info:
  title: FastAPI
  version: 0.1.0
paths:
  /:
    get:
      tags:
        - root
      summary: Index
      operationId: index__get
      responses:
        '200':
          description: Successful Response
          content:
            application/json:
              schema: {}
  /openapi.yaml:
    get:
      tags:
        - root
      summary: Read Openapi Yaml
      operationId: read_openapi_yaml_openapi_yaml_get
      responses:
        '200':
          description: Successful Response
          content:
            application/json:
              schema: {}
  /api/meta/rulebook/list:
    get:
      tags:
        - meta
      summary: List Known Rulebooks
```

```
      description: Get list of available rulebooks.
      operationId: list_known_rulebooks_api_meta_rulebook_list_get
      responses:
        '200':
          description: Successful Response
          content:
            application/json:
              schema:
                title: Response List Known Rulebooks Api Meta Rulebook List Get
                type: array
                items:
                  type: string
        '500':
          description: Unexpected server error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ServerError'
  /api/meta/feature/list:
    get:
      tags:
        - meta
      summary: List Known Feature Values
      description: Get list of possible values for a feature under given context (like
        active rulebooks).
      operationId: list_known_feature_values_api_meta_feature_list_get
      parameters:
        - required: true
          schema:
            $ref: '#/components/schemas/Feature'
          name: feature
          in: query
        - required: true
          schema:
            title: Rulebooks
            type: array
            items:
              type: string
          example:
            - dsa5
          name: rulebooks
          in: query
      responses:
        '200':
          description: Successful Response
          content:
            application/json:
              schema:
                title: Response List Known Feature Values Api Meta Feature List Get
                anyOf:
                  - type: array
                    items:
                      maxItems: 3
                      minItems: 3
                      type: array
                      items:
                        - type: string
                        - type: string
                        - type: integer
                  - type: array
                    items:
                      type: string
        '400':
          description: Bad Request
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ClientError'
        '500':
```

```yaml
          description: Unexpected server error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ServerError'
        '422':
          description: Validation Error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/HTTPValidationError'
/api/hero/validate:
  post:
    tags:
      - hero
    summary: Validate
    description: Validates hero against given rulebooks.
    operationId: validate_api_hero_validate_post
    parameters:
      - required: true
        schema:
          title: Rulebooks
          type: array
          items:
            type: string
        example:
          - dsa5
        name: rulebooks
        in: query
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Hero'
      required: true
    responses:
      '200':
        description: Successful Response
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/HeroValidationResult'
      '400':
        description: Bad Request
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ClientError'
      '500':
        description: Unexpected server error
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ServerError'
      '422':
        description: Validation Error
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/HTTPValidationError'
/api/hero/save:
  put:
    tags:
      - hero
    summary: Save
    description: Save hero as new or whenever given hero name
  ↪ exists for user override
      it.
    operationId: save_api_hero_save_put
    parameters:
      - required: true
        schema:
          title: Rulebooks
```

```yaml
        type: array
        items:
          type: string
      example:
        - dsa5
      name: rulebooks
      in: query
  requestBody:
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Hero'
    required: true
  responses:
    '200':
      description: Successful Response
      content:
        application/json:
          schema: {}
    '400':
      description: Bad Request
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/ClientError'
    '500':
      description: Unexpected server error
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/ServerError'
    '422':
      description: Validation Error
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/HTTPValidationError'
/api/hero/export:
  get:
    tags:
      - hero
    summary: Export
    description: Export hero of user by given hero name.
    operationId: export_api_hero_export_get
    parameters:
      - required: true
        schema:
          title: Hero Name
          type: string
        name: hero_name
        in: query
    responses:
      '200':
        description: Successful Response
        content:
          application/json:
            schema: {}
      '500':
        description: Unexpected server error
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ServerError'
      '422':
        description: Validation Error
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/HTTPValidationError'
/api/hero/delete:
  delete:
    tags:
      - hero
```

```yaml
      summary: Delete
      description: Delete hero of user by given hero name.
      operationId: delete_api_hero_delete_delete
      parameters:
        - required: true
          schema:
            title: Hero Name
            type: string
          name: hero_name
          in: query
      responses:
        '200':
          description: Successful Response
          content:
            application/json:
              schema: {}
        '500':
          description: Unexpected server error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ServerError'
        '422':
          description: Validation Error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/HTTPValidationError'
components:
  schemas:
    Addon:
      title: Addon
      enum:
        - any_of
      type: string
      description: An enumeration.
    ClientError:
      title: ClientError
      required:
        - type
        - message
        - details
      type: object
      properties:
        type:
          title: Type
          type: string
        message:
          title: Message
          type: string
        details:
          title: Details
          type: object
      additionalProperties: false
    Feature:
      title: Feature
      enum:
        - experience_level
        - race
        - culture
        - profession
        - advantage
        - disadvantage
        - talent
        - combat_technique
      type: string
      description: An enumeration.
    HTTPValidationError:
      title: HTTPValidationError
      type: object
      properties:
        detail:
          title: Detail
```

```yaml
        type: array
        items:
          $ref: '#/components/schemas/ValidationError'
  Hero:
    title: Hero
    required:
      - name
      - experience_level
      - race
      - culture
      - profession
      - talents
      - combat_techniques
      - advantages
      - disadvantages
    type: object
    properties:
      name:
        title: Name
        type: string
      experience_level:
        title: Experience Level
        type: string
      race:
        title: race
        type: string
      culture:
        title: Culture
        type: string
      profession:
        title: Profession
        type: string
      talents:
        title: Talents
        type: object
        additionalProperties:
          minimum: 0.0
          type: integer
      combat_techniques:
        title: Combat Techniques
        type: object
        additionalProperties:
          minimum: 0.0
          type: integer
      advantages:
        title: Advantages
        type: array
        items:
          maxItems: 3
          minItems: 3
          type: array
          items:
            - type: string
            - type: string
            - minimum: 0.0
              type: integer
      disadvantages:
        title: Disadvantages
        type: array
        items:
          maxItems: 3
          minItems: 3
          type: array
          items:
            - type: string
            - type: string
            - minimum: 0.0
              type: integer
    additionalProperties: false
  HeroValidationError:
    title: HeroValidationError
    required:
      - type
```

```yaml
        - message
        - parameter
      type: object
      properties:
        type:
          $ref: '#/components/schemas/app__models__hero_validation_error__HeroValidationError__Type'
        addon:
          $ref: '#/components/schemas/Addon'
        message:
          title: Message
          type: string
        parameter:
          title: Parameter
          type: object
      additionalProperties: false
      description: 'Represents a single hero validation error.
        Field ''message'' uses single quote for used ''parameters''
        Field ''parameters'' contains relevant evaluable data. May not all are used in ''message''.'
    HeroValidationResult:
      title: HeroValidationResult
      required:
        - valid
        - errors
        - warnings
      type: object
      properties:
        valid:
          title: Valid
          type: boolean
        errors:
          title: Errors
          type: array
          items:
            $ref: '#/components/schemas/HeroValidationError'
        warnings:
          title: Warnings
          type: array
          items:
            $ref: '#/components/schemas/HeroValidationWarning'
      additionalProperties: false
    HeroValidationWarning:
      title: HeroValidationWarning
      required:
        - type
        - message
        - parameter
      type: object
      properties:
        type:
          $ref: '#/components/schemas/app__models__hero_validation_warning__HeroValidationWarning__Type'
        message:
          title: Message
          type: string
        parameter:
          title: Parameter
          type: object
      additionalProperties: false
      description: 'Represents a single hero validation warning.
        Field ''message'' uses single quote for used ''parameters''
        Field ''parameters'' contains relevant evaluable data. May not all are used in ''message''.'
    ServerError:
      title: ServerError
      required:
        - type
        - message
        - details
      type: object
      properties:
        type:
```

```
        title: Type
        type: string
      message:
        title: Message
        type: string
      details:
        title: Details
        type: object
    additionalProperties: false
  ValidationError:
    title: ValidationError
    required:
      - loc
      - msg
      - type
    type: object
    properties:
      loc:
        title: Location
        type: array
        items:
          anyOf:
            - type: string
            - type: integer
      msg:
        title: Message
        type: string
      type:
        title: Error Type
        type: string
  app__models__hero_validation_error__HeroValidationError__Type:
    title: Type
    enum:
      - unknown
      - unusable_by
      - missing_level
      - max_lvl_exceeded
    type: string
    description: An enumeration.
  app__models__hero_validation_warning__HeroValidationWarning__
  ↪ Type:
    title: Type
    enum:
      - missing_usual
      - missing_typical
      - atypical
    type: string
    description: An enumeration.
```

A.2 <repo>/docs/clingo-cheatsheet.lp

Einstiegshilfe zur Clingo Syntax und Semantik.

```
%%
%% This is a cheatsheet about clingo/gringo/clasp language
%%  it is not complete nor states all restrictions
%%  but it gives a good and quick overview/understanding
%%
%% See 'Potassco: User Guide' (source of some text and examples) for a
↪  complete guide.
```

```
%%  https://github.com/potassco/guide/releases/tag/v2.2.0
%%




%%--------------------------------------------------------------------
%% PART: basic semantic
%%--------------------------------------------------------------------

%% --- ATOM
%% is a constant or function (see 'language basics' part below)

%% --- RULE: 'HEAD :- BODY.'
%% within the head are atoms
%% the body contains LITERALs which are atoms or negated atoms
%%
%% umbrella if rain and outdoor
umbrella :- rain, outdoor.

%% --- FACT (bodyless rule): 'HEAD.'
%%
%% it rains
rain.

%% --- INTEGRITY CONSTRAINT
%% filters solution candidates: body shall not be satisfied
%%
%% keep candidate if: no umbrella but outdoor
:- umbrella, not outdoor.



%%--------------------------------------------------------------------
%% PART: language basics
%%--------------------------------------------------------------------

%% line comment start with an %
%% block and inline comments starts with %* and ends with *%

%% --- INTEGERS
1

%% --- CONSTANTS
starts_lowercase
_underscorePreventNameClashes

%% --- SPECIAL CONSTANTS
%% greatest and smallest element among all variable-free terms
#sup
#inf

%% --- BOOLEAN CONSTANTS
#true
#false

%% --- VARIABLE
Starts_uppercase
_UnderscorePreventNameClashes

%% --- ANONYMOUS VARIABLE
_

%% --- STRING
"string"
"\\" %% escapes backslash
"\n" %% escapes newline
"\"" %% escapes double quote

%% --- (UNINTERPRETED) FUNCTIONS
%% can be better understood as a named tuple
%%  since there is no self-defined functionality programmed behind
    ↪  these
%%
```

```
%% example with three arguments (elements in a tuple)
functionName(constant,function(123),Variable)

%% --- TUPLES
%% are functions without names
()
(functionName,constant,function(123),Variable)
%% end with a ',' (comma) to declare tuple grade: example is a quadruple
(a,b,c,d,)

%% --- INTERVALS
%%
%% in body: expanded disjunctively
num(1..3).
%% result: num(1). num(2). num(3).
%%
%% in head: expanded conjunctively
grid(1..S,1..S) :- size(S).
%% result: grid(1,1). grid(1,2). grid(2,2). grid(2,1).
%% having: size(2).
%%
%%  which is same as (using unification):
grid(X,Y) :- X = 1..S, Y = 1..S, size(S).
%%  were additional constraints could be used
%%  (remove diagonals: X-Y!=0 and X+Y-1!=S)



%%---------------------------------------------------------------------
%% PART: evaluations and controls
%%---------------------------------------------------------------------


%% --- DISJUNCTION
%% head is derived if at least one atom (of body) is true
%%  (increases computational complexity: use 'choice construct' were possible)
a;b :- c,d.
%% provides answer sets [a] and [b] if c or d is true
a;-b. %% see below for explanation of '-'
%% provides answer sets [a] and [-b] but not [a,-b]


%% --- NEGATION
%%
%% default negation:
%%  'not b' is true until b is derived true
a :- not b.
%%
%% classical/strong negation:
%%  '-b' is only true if b can be derived (to false)
%%  '-b' is complement of 'b' (implicit integrity constraint ':- b, -b')
a :- -b.
%%
%% double negation:
a :- not not b.
a :- not -b.
%%
%% head negation:
%%  can be used in disjunctions
not a :- b.
not not a :- b.
%%  same as integrity constraints
:- b, not not a.
:- b, not a.


%% --- BUILT-IN ARITHMETIC FUNCTIONS
plus    ( L +  R ) :- left(L), right(R). %%  +   := addition
minus   ( L -  R ) :- left(L), right(R). %%  -   := subtraction
uminus  (   -  R ) :-          right(R). %%  -   := unary minus
```

```
times   ( L *  R ) :- left(L), right(R). %%  *   := multiplication
divide  ( L /  R ) :- left(L), right(R). %%  /   := integer division
modulo  ( L \  R ) :- left(L), right(R). %%  \   := modulo
power   ( L ** R ) :- left(L), right(R). %%  **  := exponentiation
absolute(|    -R|) :-          right(R). %%  |.| := absolute value
bitand  ( L &  R ) :- left(L), right(R). %%  &   := bitwise AND
bitor   ( L ?  R ) :- left(L), right(R). %%  ?   := bitwise OR
bitxor  ( L ^  R ) :- left(L), right(R). %%  ^   := bitwise exclusive OR
bitneg  (   ~  R ) :-          right(R). %%  ~   := bitwise complement
%% having: left(7). right(2).


%% --- BUILT-IN COMPARISON PREDICATES
%%
%% integers are compared naturally, which are smaller then constants
%% constants are ordered lexicographically, which are smaller then functions
%% functions both structurally and lexicographically
%%
eq (X,Y) :- X =  Y   , num(X), num(Y). %%  =   := equal
neq(X,Y) :- X != Y   , num(X), num(Y). %%  !=  := not equal
lt (X,Y) :- X <  Y   , num(X), num(Y). %%  <   := less than
leq(X,Y) :- X <= Y   , num(X), num(Y). %%  <=  := less than or equal
gt (X,Y) :- X >  Y   , num(X), num(Y). %%  >   := greater than
geq(X,Y) :- X >= Y   , num(X), num(Y). %%  >=  := greater than or equal
%% having: num(1). num(2).
%% also possible with constants and functions: num(a). num( f(a) ).
%%
%% arithmetic functions are evaluated before comparison literals, see:
all(X,Y) :- X-1 < X+Y, num(X), num(Y).
non(X,Y) :- X/X > Y*Y, num(X), num(Y).


%% --- UNIFICATION or SHORTHANDS for terms.
%% Also possible for functions and tuples.
squares(XX,YY) :- XX  = X*X,   Y*Y = YY,
                  Y'-1 = Y,    Y'*Y' = XX+YY,
                  X<Y, num(X), num(Y).
%% having: num(1). num(2). num(3). num(4). num(5).


%% --- POOLING
%% a set of atoms, function or tuples as an argument
%% intervall '1..3' is same as pool '(1;2;3)'
%%
%% their behavior in head and body is equal to intervals:
grid((1;2),(1;2)).
%% result: grid(1,1). grid(1,2). grid(2,2). grid(2,1).
%%
%% below is same but uses unification which enables us to add constraints
grid(X,Y) :- X = (1;2), Y = (1;2).
%%
%% other variants: first returns tuples, seconds splits arguments
p((1,2;3,4)).
%% result: p((1,2)). p((3,4)).
p(1,2;3,4).
%% result: p(1,2). p(3,4)


%% --- CONDITIONAL LITERAL
%% b is the literal and c is the condition, which may be more than one
%% a yields whenever either c is false (b does not matter) or b and c holds
a :- b : c.
%%
%% usage in head is possible:
```

```
%%  if c then a (literal) only if b (condition)
a : b :- c.
%%  if c then yield a(X) if b(X)
a(X) : b(X) :- c.
%%
%% usages in body with other literals: conditions ends with an ';' (semicolon)
%% a :- b1, b2 : c1, c2, c3; b3, b4
next(X,Z) :- set(X), #false : X < Y, Y < Z, set(Y); set(Z), X < Z.
%% having: set(1..4).
%%
%% important:
%%  variable names within conditions must no match global variable names
%%  a global variable is within a atom which is NOT subject of a condition




%%------------------------------------------------------------------------
%% PART: aggregates
%%------------------------------------------------------------------------
%% see example 3.15 from 'Potassco: user guide' for a complex problem

%% --- BODY AGGREGATES
%% lowerBound <= aggregate-function { elements } <= upperBound
%% '<=' is default and can be omitted or replaced by other comparison predicates
%% 'aggregate-function' default is '#count' and can be omitted or replaced
%% '{ elements }' represent a set, hence duplicate elements are ignored
%%  elements are separated by an ; (semicolon)
%%  an element is: <terms tuple>:<conditional literals tuple>
%%  the 'conditional literals tuple' is optional (colon must be omitted then)
%%
%% aggregate functions are applied to terms (weight) only and are:
#count %% number of elements; used for expressing cardinality constraints
#sum   %% sum of weights; used for expressing weight constraints
#sum+  %% sum of positive weights
#min   %% minimum weight
#max   %% maximum weight
%%
%% weight refers the first element of a term tuple:
%%
%% example: the sum must be below 20
#sum{ 1 ; 2 ; 4 ; 8 } < 20.
%%
%% example with conditional literal:
%%  the sum of credits (weight) by passed modules (mod) must be at least 20
20 #sum{ 4 : mod(a) ; 4 : mod(b) ; 6 : mod(c) ; 6 : mod(d),mod(dExtra) }.
%%  the 'sum' will be '10', due to duplicate terms and hence does not satisfy the constraint
%%  hence non-single tuples can be used make weights (credits) unique (per mod)
20 #sum{ 4,a : mod(a) ; 4,b : mod(b) ; 6,c : mod(c) ; 6,d : mod(d),mod(dExtra) }.
%%
%% usage of unification/shorthands is possible but
%%  discouraged due to potential excessive unwrap of possibilities
%%  (see 'Potassco: user guide' chapter 3.1.12 remark 3.9 for details)
cnt(X) :- X = #count { 2:a ; 3:a }. %% if a hold 'cnt(1)' else 'cnt(0)'
sum(X) :- X = #sum   { 2:a ; 3:a }. %% if a hold 'sum(5)' else 'sum(0)'
pos(X) :- X = #sum+  { 2:a ; 3:a }. %% if a hold 'pos(5)' else 'pos(0)'
```

```
min(X) :- X = #min   { 2:a ; 3:a }. %% if a hold 'min(2)' else 'min(#sup)'
max(X) :- X = #max   { 2:a ; 3:a }. %% if a hold 'max(3)' else 'max(#inf)'
%%
%% #min and #max can be also expressed without aggregation:
:- #min { X,Y : condition_a(Y), condition_b(Y,X) } 2.
%% is same as
:- condition_a(Y), condition_b(Y,X), X <= 2.


%% --- HEAD AGGREGATES
%% syntax is same as with body aggregates
%%  but elements have an 'head literal to be derived' when constraint(s) holds.
%%  hence of form: <terms tuple>:<derivable literal>:<conditional literals tuple>
%%  the 'conditional literals tuple' is still optional
%%
%% get edges having a cost which is higher then 10
10 < #sum { C,X,Y : edge(X,Y) : cost(X,Y,C) }.
%%
%% special case: choice construct/rule
%%  here it is possible to derive any of the atoms (for buy).
{ buy(pizza) ; buy(wine) ; buy(corn) } :- at(grocery).


%%---------------------------------------------------------------------
%% PART: optimization problems
%%---------------------------------------------------------------------


%% --- WEAK CONSTRAINTS
%% syntax is similar to integrity constraints
%%  but associates a weighted term tuple if the body holds
%%  hence answer sets are comparable and will be weight optimized
%%
%% it has the form: (L literal, w weight, p priority, t term)
%%   :~ L1, ..., Ln. [w@p,t1,...,tn]
%%
%% the priority is optional, hence '@p' can be omitted and defaults to 0 (zero)
%%  the greater the p the more important it is
%% the weight can be a term but must be an integer
%%
%% a abstract example:
:~ funcA(X), cost(X,C). [C,X] %% cost C and priority 0 (zero)
:~ funcA(X), funcB(X,Y). [3@1,X,Y] %% cost 3 and a higher priority of 1


%% --- OPTIMIZATION DIRECTIVES
%% above can also be represented like:
#minimize{  C,X : funcA(X), cost(X,C) ,  3@1,X,Y : funcA(X), funcB(X,Y) }.
%% and
#maximize{ -C,X : funcA(X), cost(X,C) , -3@1,X,Y : funcA(X), funcB(X,Y) }.
%% thus multiple weak constraints can be written within one statement


%% --- example 3.16 of 'Potassco: User Guide'
%% given five hotels, find optimal
{ hotel(1..5) } = 1.
%% with information about their stars, cost per night and possible noisiness:
star(1,5). cost(1,170).
star(2,4). cost(2,140).
star(3,3). cost(3,90).
star(4,3). cost(4,75). main_street(4).
star(5,2). cost(5,60).
noisy :- hotel(X), main_street(X).
```

```
%% choose by following prioritization (remember: higher '@p' is more important):
#maximize { Y@1,X : hotel(X), star(X,Y) }. %% (1) the highest stars
#minimize { Y/Z@2,X : hotel(X), cost(X,Y), star(X,Z) }. %% (2) the lowest cost per star
:~ noisy. [ 1@3 ] %% (3) it must not be noisy
%% result: hotel 3 is optimal


%%---------------------------------------------------------------------
%% PART: more features
%%---------------------------------------------------------------------


%% --- OUTPUT CONTROL
%% control output by suppress irrelevant terms and atoms from answer set
%%
%% ONLY show atom n: #show p/n
#show name/2. %% having name(X, Y) show atoms Y
%% ADDITIONALLY show term t if literals hold: #show t : L1, ..., Ln.
#show correct : finished, not error.
%% (optionally) show nothing (except other #show)
#show.


%% --- PROGRAM PARTS
%% a logic programm can be separated into multiple parts
%% per default everything is under the program 'base'
%%  (if not under a '#program <name>' directive other than 'base')
%%  wich is executed per default
a. %% under 'base' program,
#program check_a.
valid_a :- #true. %% under program 'check_a'; not executed per default
#program base. %% below this: rules would be executed by default again
b.
%% result per default to: {a,b}
%%
%% a program can have parameters
#program check_b(x,y).
valid_b :- check(x,y).
%%
%% calling such sub programs:
%%  use parameter 'parts' of 'clingo.Control().ground()'
%%  a list of '(<program>,<list of parameters>)' tuples
%%  e.g.: 'clingo.Control().ground( [ ("check_a",[]) , ("check_b",[2,5]) ] )'


%% --- INCLUDE other logic programs
%% a logic programm can be split into multiple files
%% using '#include "<file_path>"'
#include "held_fakten.lp".
%% looks up 'held_fakten.lp' file path in following order:
%%  1. relative to current working folder
%%  2. relative to the file containing the #include statement
%%
%% it its not affected by #program directive (see above)


%% --- EXTERNAL FUNCTIONS (scripting)
%% (implement and) use simple Python functions out of the logic program
%%  their result MUST BE deterministic
%%
%% on any error the current callee rule is dropped with a warning
%%
%% function parameters are terms of the type 'clingo.Symbol' and are generic
%%  which provide properties for different representations:
%%   number: <parameter>.number
```

```
%%   string: <parameter>.string
%%   function: <parameter>.name
%%   function arguments: <parameter>.arguments
%%
%% constants and tuples act like functions here:
%%  constants have an empty argument list
%%  tuples a empty name
%%
%% #sup and #inf are concrete and unique objects: clingo.Sup and clingo.Inf
%%
%% first declare such function
#script(python) %% till #end everything will be interpreted as python code
import clingo
N = clingo.Number
def increase(a):
    return N(a.number + 1)
#end
%% then use it: get highest step and add one
step(1).
step(@increase(S)) :- #maximize{ S : step(S)}.
%% result: step(2).
%%
%% the python function could also return a list
%%  each element will then be successively inserted.
%% function f with 'return [ N(3) , N(4) ]'
%%  would make 'step(@f())' to 'step(3,4)' (so behave like 3..4)
%%
%% also returnable are:
%%  strings with 'clingo.String(<string>)'
%%  functions with 'clingo.Function(<name>,<arguments>)'
%%  a boolean indirectly with 'clingo.Function(<name>)' as a positive constant
%%
%%
%% python code must not be implemented within a '#script'; using the python api
%%  the 'clingo.Control().ground()' method provides a parameter 'context'
%%  which would be a class instance
%%  which methods can be called by '@' (as before)
%% see example/clingo_context.py


%% --- SOLVING UNDER ASSUMPTIONS
%% an answer set is only valid if the assumption holds (contains the atom)
%%  the 'clingo.Control().solve()' method provides a parameter 'assumptions'
%%  which is a list of '(atom, boolean)' tuples or literals
%%   e.g.: '[(clingo.Function("valid"), True)]'
%%   hence the function valid(<any elements>) must be derived True


%%---------------------------------------------------------------------
%% PART: notes about further features not relevant for our use case
%%---------------------------------------------------------------------


%% TODO write something about #defined, #project


%% --- EXTERNAL STATEMENTS
%% prevents atoms to be discarded for simplifications;
%%  if they are in the body (of an rule, condition, ...)
%%  but not in any rule head
%%
%% form: '#external A:L1,...,Ln'
#external q(X) : p(X). %% q is the external atom, p the condition
%%
```

```
%% its main usage is in extending plain ASP solving like multi-shot solving
%% hence will not be descripted here, more details in 'Potassco: User Guide'
%% under part 'External Statements' in chapter 3.1.15 'Meta-Statements'


%% --- CONSTANTS WITH PARSABLE DEFAULT VALUES
%% a constant having a default value but can be overridden before grounding
%%
%% form: '#const <constant> = <term without variables, pools or intervals>.'
#const x = 42.
#const y = f(x,z).
p(x,y). %% result in p(42,f(42,z))
%% TODO how do override the const using the python api?


%% --- MULTI-SHOT SOLVING
%% TODO may be more to come


%% --- THEORY SOLVING
%% TODO may be more to come

%% --- HEURISTIC-DRIVEN SOLVING
%% TODO may be more to come about #heuristic
```

A.3 <repo>/README.md

Einstiegsseite des quelloffenen Aufbewahrungsortes.

```
# DSA ("Das Schwarze Auge") Heldenbogen - Backend

This project represents a new way of formalizing the rules of germans most popular
RPG-PP [Das Schwarze Auge](https://ulisses-spiele.de/game-system/das-schwarze-auge/) (DSA) including its countless expansions and
making the players characters verifiable at runtime in no time.

It uses "answer set programming" (ASP) which is widely used in scientific and industrial usage but with a quite different purpose
hence the modelling is unusual for ASP.

It is a python 3.11 webserver using ASGI web server [uvicorn](https://www.uvicorn.org/)
with [FastApi](https://fastapi.tiangolo.com/) for API dokumentation and ASP made available trough
framework [clingo](https://potassco.org/clingo/) (from Potassco, the Potsdam Answer Set Solving Collection).

---

* RPG - role-playing game
* PP - pen & paper
* DSA - Das Schwarze Auge
```

```
* ASP - answer set programming

## Documentations

* See how to contribute at [docs/CONTRIBUTE.md](./docs/CONTRIBUTE.md).
  * Always follow as much [https://clean-code-developer.com/](clean code principles) as you can.
  * See how to add a new rulebook (expansion) at [docs/CREATE_A_RULEBOOK.md](./docs/CREATE_A_RULEBOOK.md).
  * Understand ASP modelling (logic program language by clingo) at [docs/CHEATSHEET.lp](./docs/CHEATSHEET.lp).

* See list of open todos at [docs/TODOs.md](./docs/TODOs.md). (there are even more within the code)
```

A.4 <repo>/Makefile

Automatisierung häufiger Aufgabe der Entwicklung.

```
## FYI: .PHONY declares a goal  that does not target a real file

## meta vars
ROOT=./
SRC=$(ROOT)dsaheldenbogen/
TESTS=$(ROOT)tests/
MAIN=$(SRC)main.py

## goals
.DEFAULT_GOAL=help

.PHONY: help
help:
	@echo "Usage 'make <goal>'. Values for <goal> are one of:"
	@echo ""
	@echo "  clean       remove all temporary files"
	@echo "  install     install packages and prepare environment"
	@echo "  lint        run code linters"
	@echo "  typehint    run code typehint checker"
	@echo "  test        run all tests"
	@echo "  prebuild    goals: lint, typehint and test"
	@echo "  format      format code (experimental, check code changes)"
	@echo "  debug       start server for development"
	@echo "  start       start server for production"
	@echo ""

.PHONY: clean
clean:
	rm -rf .venv
	rm -rf .mypy_cache

.PHONY: install
install:
	poetry install

.PHONY: lint
lint: ## TODO may use 'flake8' instead of 'pylint'
	## enforce coding standards
```

```
39          #poetry run python -m --ignore=W503,E501 $(SRC) $(TESTS)
40          ## very strict (coding standards, code smells, simple
    ↪  refactors)
41          poetry run python -m pylint $(SRC) $(TESTS) || true
42          ## find security issues
43          poetry run python -m bandit -r $(SRC) || true
44
45          ## [PRE CHECK]
46          #poetry run python -m isort --profile=black
    ↪  --lines-after-imports=2 --check-only $(TESTS) $(SRC)
47          #poetry run python -m black -check $(TESTS) $(SRC) --diff
48
49  .PHONY: typehint
50  typehint:
51          poetry run python -m mypy $(SRC) $(TESTS) || true
52
53  .PHONY: test
54  test:
55          poetry run python -m pytest --quiet --log-level=WARNING
    ↪  $(TESTS)
56
57  .PHONY: format
58  format: ## TODO currently experimental
59          ## sort and group imports
60          poetry run python -m isort --profile=black
    ↪  --lines-after-imports=2 $(TESTS) $(SRC)
61          ## opinionated formatter
62          poetry run python -m black $(TESTS) $(SRC)
63          ## only fixes pep8 violations
64          poetry run python -m autopep8
65          ## eformats entire code to the best style possible
66          poetry run python -m yapf
67
68  .PHONY: prebuild
69  prebuild: lint test
70
71  .PHONY: debug
72  debug:
73          poetry run python $(MAIN) --reload --loglevel debug
74
75  .PHONY: start
76  start: prebuild
77          poetry run python $(MAIN) --workers 4
```

A.5 <repo>/pyproject.toml

Projekt Meta-Informationen und Abhängigkeiten für Poetry.

```
1  [tool.poetry]
2  name = "dsa-heldenbogen"
3  version = "0.1.0"
4  description = ""
5  authors = ["bjoern-nowak <dsa-heldenbogen@nowakhub.de>"]
6  readme = "README.md"
7  packages = [{ include = "app" }]
8
9  [tool.poetry.dependencies]
10 python = "~3.11"
11 ## python API for anwer set programming (ASP) by Potassco
12 clingo = "5.6.2"
```

```
## web server deps
fastapi = "0.94.1"
uvicorn = { extras = ["standard"], version = "0.21.0" }


[tool.poetry.group.dev.dependencies]
mypy = "^0.991"
pylint = "^2.15.10"
isort = "^5.12.0"
black = "^22.12.0"
autopep8 = "^2.0.1"
yapf = "^0.32.0"
bandit = "^1.7.4"
parameterized = "^0.8.1"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

A.6 <repo>/resources/rulebook/common.lp

LP, das, als Basis für alle Regelbücher, von der Software geladen wird.

```
#program base.
#show. %% hide everything in output but
%% rulebook_usable
#defined rulebook_depends/2.
#defined rulebook_missing/2. #show rulebook_missing/2.
%% world facts
#defined known_experience_level/7.
#defined known_race/1.
#defined known_culture/1.
#defined known_profession/1.
#defined known_advantage/3.
#defined known_disadvantage/3.
#defined known_talent/2.
#defined known_combat_technique/2.
#defined requires/2.
#defined has_usual/2.
#defined has_typical/2.
#defined has_atypical/2.
%% hero facts
#defined race/1. #show race/1.
#defined culture/1. #show culture/1.
#defined profession/1. #show profession/1.
#defined talent/2. #show talent/2.
#defined combat_technique/2. #show combat_technique/2.
#defined advantage/3. #show advantage/3.
#defined disadvantage/3. #show disadvantage/3.
%% hero validation errors
#defined unknown/1. #show unknown/1.
#defined unusable_by/2. #show unusable_by/2.
#defined missing/2. #show missing/2.
#defined missing_level/2. #show missing_level/2.
#defined missing_min_lvl/2. #show missing_min_lvl/2.
#defined max_lvl_exceeded/2. #show max_lvl_exceeded/2.
#defined max_count_exceeded/2. #show max_count_exceeded/2.
%% hero validation warnings
#defined unusual_for/2. #show unusual_for/2.
#defined missing_usual/2. #show missing_usual/2.
#defined missing_typical/2. #show missing_typical/2.
```

```
#defined atypical/2. #show atypical/2.

#program rulebook_usable.
rulebook_missing(RB,D) :- rulebook(RB), rulebook_depends(RB,D), not rulebook(D).

#program rulebook_requirements.
%% TODO add check that rulebook only uses known feature values, to be executed by testcases

#program world_facts.

#program hero_facts.
%% to be used with programs 'validate_hero_step<X>'
%% requires 'hero_wrapper.py' as 'context'
%% TODO these '__<XYZ>_wrapper' are ugly helper facts, try to remove it
__talent_wrapper(@talents).
__combat_technique_wrapper(@combat_techniques).
__advantage_wrapper(@advantages).
__disadvantage_wrapper(@disadvantages).
%
experience_level(@experience_level).
race(@race).
culture(@culture).
profession(@profession).
talent(T,LVL) :- __talent_wrapper((T,LVL)).
combat_technique(CT,LVL) :- __combat_technique_wrapper((CT,LVL)).
advantage(A,USES,LVL) :- __advantage_wrapper((A,USES,LVL)).
disadvantage(DA,USES,LVL) :- __disadvantage_wrapper((DA,USES,LVL)).

#program validate_hero_step_50. %% pre check - values known
unknown(experience_level(EL)) :- experience_level(EL), not known_experience_level(EL,_,_,_,_,_,_).
unknown(race(R)) :- race(R), not known_race(R).
unknown(culture(C)) :- culture(C), not known_culture(C).
unknown(profession(P)) :- profession(P), not known_profession(P).
unknown(advantage(A,USES,LVL)) :- advantage(A,USES,LVL), not known_advantage(A,USES,LVL).
unknown(disadvantage(DA,USES,LVL)) :- disadvantage(DA,USES,LVL), not known_disadvantage(DA,USES,LVL).
unknown(talent(T)) :- talent(T,_), not known_talent(T).
unknown(combat_technique(CT)) :- combat_technique(CT,_), not known_combat_technique(CT).

#program validate_hero_step_100. %% check race usable
max_count_exceeded(race,MAX) :- COUNT=#count{ R:race(R) }, MAX=#max{ MC:max_count(race,MC) }, COUNT > MAX.

#program validate_hero_step_150. %% check race requirements
% WARNINGS
missing_usual(race(R),culture(C)) :- race(R),
    has_usual(race(R),culture(_)), culture(C), not
    has_usual(race(R),culture(C)).

#program validate_hero_step_200. %% check culture usable
max_count_exceeded(culture,MAX) :- COUNT=#count{ C:culture(C) },
    MAX=#max{ MC:max_count(culture,MC) }, COUNT > MAX.
unusable_by(culture(C),race(R)) :- culture(C),
    requires(culture(C),race(_)), race(R), not
    requires(culture(C),race(R)).

#program validate_hero_step_250. %% check culture requirements

#program validate_hero_step_300. %% check profession usable
max_count_exceeded(profession,MAX) :- COUNT=#count{ P:profession(P) }, MAX=#max{ MC:max_count(profession,MC) }, COUNT > MAX.
unusable_by(profession(P),race(R)) :- profession(P),
    requires(profession(P),race(_)), race(R), not
    requires(profession(P),race(R)).
unusable_by(profession(P),culture(C)) :- profession(P),
    requires(profession(P),culture(_)), culture(C), not
    requires(profession(P),culture(C)).
```

```
94
95 #program validate_hero_step_350. %% check profession requirements
96 missing_level(profession(P),talent(T,MIN_LVL)) :- profession(P),
   ↪ requires(profession(P),talent(T,MIN_LVL)), 1 = #count{ 1: not
   ↪ talent(T,_) ; 1: talent(T,LVL), LVL < MIN_LVL }.
97 missing_level(profession(P),combat_technique(CT,MIN_LVL)) :-
   ↪ profession(P),
   ↪ requires(profession(P),combat_technique(CT,MIN_LVL)), 1 =
   ↪ #count{ 1: not combat_technique(CT,_) ; 1:
   ↪ combat_technique(CT,LVL), LVL < MIN_LVL }.
98 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).
99
100 #program validate_hero_step_400. %% check (dis)advantage usable
101
102 #program validate_hero_step_450. %% check (dis)advantage requirements
103 missing(race(R),advantage(A,USES,LVL)) :- race(R),
   ↪ requires(race(R),advantage(A,USES,LVL)), not
   ↪ advantage(A,USES,LVL).
104 missing(race(R),disadvantage(DA,USES,LVL)) :- race(R),
   ↪ requires(race(R),disadvantage(DA,USES,LVL)), not
   ↪ disadvantage(DA,USES,LVL).
105 % WARNINGS
106 missing_typical(race(R),advantage(A,USES,LVL)) :- race(R),
   ↪ has_typical(race(R),advantage(A,USES,LVL)), not
   ↪ advantage(A,USES,LVL).
107 missing_typical(race(R),disadvantage(DA,USES,LVL)) :- race(R),
   ↪ has_typical(race(R),disadvantage(DA,USES,LVL)), not
   ↪ disadvantage(DA,USES,LVL).
108 atypical(race(R),advantage(A,USES)) :- race(R),
   ↪ has_atypical(race(R),advantage(A,USES,_)), advantage(A,USES,_).
109 atypical(race(R),disadvantage(DA,USES)) :- race(R),
   ↪ has_atypical(race(R),disadvantage(DA,USES,_)),
   ↪ disadvantage(DA,USES,_).
110
111 #program validate_hero_step_500. %% check skills (talents, combat
   ↪ techniques) usable
112 missing_min_lvl(talent(T,LVL),MIN) :- talent(T,LVL), MIN=0, LVL <
   ↪ MIN.
113 missing_min_lvl(combat_technique(CT,LVL),MIN) :-
   ↪ combat_technique(CT,LVL), MIN=0, LVL < MIN.
114 max_lvl_exceeded(talent(T,LVL),MAX_LVL) :- experience_level(EL),
   ↪ known_experience_level(EL,_,MAX_LVL,_,_,_,_), talent(T,LVL), LVL
   ↪ > MAX_LVL.
115 max_lvl_exceeded(combat_technique(CT,LVL),MAX_LVL) :-
   ↪ experience_level(EL),
   ↪ known_experience_level(EL,_,_,MAX_LVL,_,_,_),
   ↪ combat_technique(CT,LVL), LVL > MAX_LVL.
116
117 #program validate_hero_step_550. %% check skills (talents, combat
   ↪ techniques) requirements
```

A.7 <repo>/app/engine/engine.py

Kern-Steuerungseinheit der Applikation. Es orchestriert die Ausführung und Interpretation der *LPs*.

```python
from __future__ import annotations  # required till PEP 563

from typing import List
from typing import Sequence

from clingo import Symbol

from dsaheldenbogen.app.engine import hero_validation_interpreter
from dsaheldenbogen.app.engine.collector import Collector
from dsaheldenbogen.app.engine.exceptions import HeroInvalidError
from dsaheldenbogen.app.engine.exceptions import
↪ UnexpectedResultError
from dsaheldenbogen.app.engine.exceptions import
↪ UnusableRulebookError
from dsaheldenbogen.app.engine.rulebook_program import
↪ RulebookProgram
from dsaheldenbogen.app.logger import getLogger
from dsaheldenbogen.app.models.feature import Feature
from dsaheldenbogen.app.models.hero import Hero
from dsaheldenbogen.app.models.hero_validation_warning import
↪ HeroValidationWarning
from dsaheldenbogen.app.models.rulebook import Rulebook
from dsaheldenbogen.infrastructure.clingo_executor import
↪ ClingoExecutor

logger = getLogger(__name__)


class Engine:
    # TODO one could argue that these step count is execcsive.
    #  To reduce this would require an error/warning filtering to root
    ↪ causes
    #   so that only one step per feature (instead of currently two)
    ↪ is used
    #  Anyway having steps is useful, since it allows rulebooks to
    ↪ intervene
    hero_validation_steps: dict[int, RulebookProgram | int] = {
        50: RulebookProgram.VALIDATE_HERO_STEP_50,
        100: RulebookProgram.VALIDATE_HERO_STEP_100,
        150: RulebookProgram.VALIDATE_HERO_STEP_150,
        200: RulebookProgram.VALIDATE_HERO_STEP_200,
        250: RulebookProgram.VALIDATE_HERO_STEP_250,
        300: RulebookProgram.VALIDATE_HERO_STEP_300,
        350: RulebookProgram.VALIDATE_HERO_STEP_350,
        400: RulebookProgram.VALIDATE_HERO_STEP_400,
        450: RulebookProgram.VALIDATE_HERO_STEP_450,
        500: RulebookProgram.VALIDATE_HERO_STEP_500,
        550: RulebookProgram.VALIDATE_HERO_STEP_550,
    }
    DEFAULT_HERO_VALIDATION_STEPS = hero_validation_steps.keys()

    def __init__(self, rulebooks: List[Rulebook]) -> None:
        self.rulebooks = rulebooks
        self.clingo_executor = ClingoExecutor(
            [Rulebook.common_file()] + [r.entrypoint_file() for r in
            ↪ self.rulebooks],
            [RulebookProgram.BASE]
        )
        self._check_rulebooks_usable()

        self._find_extra_hero_validation_steps()
        self.hero_validation_steps =
        ↪ dict(sorted(self.hero_validation_steps.items()))  # sort
        ↪ by keys

    def _check_rulebooks_usable(self) -> None:
        unusables: List[List[str]] = Collector.unusable_rulebooks(
            self.clingo_executor.run(
                programs=[RulebookProgram.RULEBOOK_USABLE],
                on_fail=UnexpectedResultError("Failed to collect
                ↪ unusable rulebooks.")
            )
        )
        if unusables:
```

```
63              messages = []
64              for sym in unusables:
65                  messages.append(f"Rulebook '{sym[0]}' missing
                    ↪ '{sym[1]}'.")
66              raise UnusableRulebookError(chr(10).join(messages))  #
                ↪ chr(10) := '\n' (line break)
67
68      def _find_extra_hero_validation_steps(self) -> None:
69          steps: List[Symbol] = Collector.extra_hero_validation_steps(
70              self.clingo_executor.run(
71                  programs=[RulebookProgram.META],
72                  on_fail=UnexpectedResultError("Failed to find extra
                    ↪ hero validation steps.")
73              )
74          )
75          if steps:
76              logger.debug(f"Found extra hero validation steps:
                ↪ {steps}")
77
78          for step in [step.arguments[0].number for step in steps]:
79              if step in self.DEFAULT_HERO_VALIDATION_STEPS:
80                  # TODO should be a testcase instead of a runtime
                    ↪ check
81                  logger.warning(f"Some rulebook redeclare default hero
                    ↪ validation step '{step}' as extra. "
82                                 "It does not harm but it is not
                                   ↪ recommended for clarity. "
83                                 f"Used rulebooks: {self.rulebooks}")
84              else:
85                  self.hero_validation_steps[step] = step
86
87      def validate(self, hero: Hero) -> List[HeroValidationWarning]:
88          """
89          If this method passes without an exception the hero has passed
    ↪ the validation positively
90          It breaks validation steps-wise on validation errors, but
    ↪ collect warnings step-wide.
91          :returns: list of warnings, when validation passed
92          :raises HeroInvalidError: whenever any hero validation step
    ↪ has an error
93          :raises UnexpectedResultError: whenever any hero validation
    ↪ step could not be performed
94          """
95          warnings: List[HeroValidationWarning] = []
96          for step in self.hero_validation_steps:
97              program = step if isinstance(step, RulebookProgram) else
                ↪ RulebookProgram.hero_validation_step_for(step)
98              step_errors, step_warnings =
                ↪ self._perform_hero_validation_step(hero, program)
99              warnings += [hero_validation_interpreter.as_warning(w)
                ↪ for w in step_warnings]
100             if step_errors:
101                 # TODO 'return' vs 'raise' is discussable.
102                 #  One could argue that an HeroInvalidError should
                    ↪ only be raised if the input values are e.g.
                    ↪ unknown
103                 #   and 'hero validation errors' are seen as normal
                    ↪ case hence should be returned.
104                 #  In contrast, the main point is to find errors and
                    ↪ should be prominent whenever found.
105                 #   Using a return (tuple or class) could lead to
                    ↪ higher chances of mishandling.
106                 #  What is bad now, is that warnings may be fetched by
                    ↪ return or raise.
107                 raise HeroInvalidError([hero_validation_
                    ↪ interpreter.as_error(e) for e in step_errors],
                    ↪ warnings)
108         return warnings
109
110     def _perform_hero_validation_step(
111             self,
112             hero: Hero,
```

```
            program: tuple[str, Sequence[Symbol]]
    ) -> tuple[List[Symbol], List[Symbol]]:
        """
        :return: tuple of (errors, warnings)
        """
        model = self.clingo_executor.run(
            programs=[RulebookProgram.WORLD_FACTS,
            ↪ RulebookProgram.HERO_FACTS, program],
            context=hero,
            on_fail=UnexpectedResultError(f"Hero validation could not
            ↪ be performed at: {program[0]}")
        )
        errors: List[Symbol] =
        ↪ Collector.hero_validation_errors(model)
        warnings: List[Symbol] =
        ↪ Collector.hero_validation_warnings(model)
        return errors, warnings

    def list_known_for(self, feature: Feature) -> List[tuple[str,
    ↪ str, int]] | List[str]:
        known_values: List[Symbol] = Collector.known_feature_values(
            self.clingo_executor.run(
                programs=[RulebookProgram.WORLD_FACTS],
                on_fail=UnexpectedResultError(f"Value listing for
                ↪ feature '{feature}' failed.")
            ),
            feature
        )
        if feature in [Feature.ADVANTAGE, Feature.DISADVANTAGE]:
            return [(da.arguments[0].string, da.arguments[1].string,
            ↪ da.arguments[2].number) for da in known_values]
        return [k.arguments[0].string for k in known_values]
```

A.8 <repo>/app/infrastructure/clingo_executor.py

Abstraktionsschicht zwischen der Engine und einem *ASP*-Framework, in diesem Fall die Clingo-*API*. Führt gewünschte Unterprogramme von vorgegebenen Regelbücher-*LPs* aus.

```
from __future__ import annotations  # required till PEP 563

from typing import List
from typing import Sequence

from clingo import Control
from clingo import SolveResult
from clingo import Symbol

from dsaheldenbogen.app.logger import getLogger
from dsaheldenbogen.app.models.hero import Hero
from dsaheldenbogen.infrastructure.hero_wrapper import HeroWrapper

logger = getLogger(__name__)


class ClingoExecutor:
    """
    This class is the engines last-mile to call/run clingo.
    """
```

```
21
22      lp_files: List[str]
23      default_programs: List[tuple[str, Sequence[Symbol]]]
24
25      def __init__(self, lp_files: List[str], default_programs:
        ↪  List[tuple[str, Sequence[Symbol]]] = None) -> None:
26          self.lp_files = lp_files
27          self.default_programs = default_programs if default_programs
            ↪  else []
28
29      def run(self,
30              programs: List[tuple[str, Sequence[Symbol]]],
31              context: Hero = None,
32              on_fail: Exception = None) -> List[Symbol]:
33          """
34          Do a clean clingo solve run
35          :param programs: to ground
36          :param context: hero to use
37          :param on_fail: called on unsatisfiable run
38          :returns: all clingo symbols (even not shown ones) when run
    ↪   was satisfied
39          """
40          ctl = self._create_control(self.default_programs + programs,
            ↪  context)
41          symbols: List[Symbol] = []
42          result: SolveResult = ctl.solve(on_model=lambda m:
            ↪  symbols.extend(m.symbols(atoms=True)))
43          if not result.satisfiable:
44              raise on_fail if on_fail else RuntimeError(f"Could not
                ↪  execute clingo programs: {[p[0] for p in programs]}")
45          return symbols
46
47      def _create_control(self, programs: List[tuple[str,
        ↪  Sequence[Symbol]]], context: Hero = None) -> Control:
48          """
49          Creates a fresh clingo control for predefined logic program
    ↪   file
50          :param programs: to be grounded
51          :param context: hero to be used
52          """
53          ctl = Control()
54          for lp_file in self.lp_files:
55              ctl.load(lp_file)
56          ctl.ground(programs, HeroWrapper(context) if
            ↪  isinstance(context, Hero) else context)
57          return ctl
```

A.9 <repo>/app/infrastructure/hero_wrapper.py

Clingo-*LP* Kontext-Klasse, welche die „external functions“ bereitstellt. Konkret konvertiert es die Charakter-Merkmale in Clingo-Symbole

```python
from typing import List

from clingo import Number
from clingo import String
from clingo import Symbol
from clingo import Tuple_

from dsaheldenbogen.app.models.dis_advantage import DisAdvantage
from dsaheldenbogen.app.models.hero import Hero
from dsaheldenbogen.app.models.skill import Skill


def _map_skills(skills: List[Skill]) -> List[Symbol]:
    return [Tuple_([String(skill.name), Number(skill.level)]) for skill in skills]


def _map_dis_advantages(dis_advantages: List[DisAdvantage]) -> List[Symbol]:
    return [Tuple_([String(d_a.name), String(d_a.uses), Number(d_a.level)]) for d_a in dis_advantages]


# TODO may provide a method which returns a list of literals instead of using a extra LP asking each feature
class HeroWrapper:
    """
    provide callables returning hero attributes as clingo symbols
    """
    _hero: Hero

    def __init__(self, hero: Hero) -> None:
        super().__init__()
        self._hero = hero

    def experience_level(self) -> Symbol:
        return String(self._hero.experience_level)

    def race(self) -> Symbol:
        return String(self._hero.race)

    def culture(self) -> Symbol:
        return String(self._hero.culture)

    def profession(self) -> Symbol:
        return String(self._hero.profession)

    def talents(self) -> List[Symbol]:
        return _map_skills(self._hero.talents)

    def combat_techniques(self) -> List[Symbol]:
        return _map_skills(self._hero.combat_techniques)

    def advantages(self) -> List[Symbol]:
        return _map_dis_advantages(self._hero.advantages)

    def disadvantages(self) -> List[Symbol]:
        return _map_dis_advantages(self._hero.disadvantages)

    def count_by(self, feature: Symbol, options: Symbol, min_lvl: Symbol) -> Symbol:
        """
        :return: count of feature values ('options') of 'feature' passing minimum level
        """
        # dynamically get class field (with 'getattr') instead of manual mapping with a switch-case
        #  this requires feature.name (LP function name) to be exactly the field name of the actual hero model
        values_lvl: dict[str, int] = {fv.name: fv.level for fv in getattr(self._hero, feature.string)}
        # count all features having the minimum level
        passed = sum(1 for opt in options.arguments if min_lvl.number <= values_lvl.get(opt.string, 0))
        return Number(passed)
```

A.10 <repo>/app/engine/collector.py

Sammelt bzw. filtert aus dem *answer set* die gesuchte Ergebnis-Fakten (Clingo-Funktionen) heraus.

```python
from typing import List

from clingo import Symbol

from dsaheldenbogen.app.engine import hero_validation_interpreter
from dsaheldenbogen.app.engine.rulebook_function import RulebookFunction
from dsaheldenbogen.app.logger import getLogger
from dsaheldenbogen.app.models.feature import Feature

logger = getLogger(__name__)


class Collector:
    """Collection of methods to collects specific facts (clingo functions)"""

    @classmethod
    def unusable_rulebooks(cls, symbols: List[Symbol]) -> List[List[str]]:
        unusables: List[List[str]] = []
        for func in cls._functions(symbols, [RulebookFunction.RULEBOOK_MISSING]):
            unusables.append([arg.string for arg in func.arguments])
        return unusables

    @classmethod
    def extra_hero_validation_steps(cls, symbols: List[Symbol]) -> List[Symbol]:
        return cls._functions(symbols, [RulebookFunction.EXTRA_HERO_VALIDATION_STEP])

    @classmethod
    def hero_validation_errors(cls, symbols: List[Symbol]) -> List[Symbol]:
        errors: List[Symbol] = cls._functions(symbols, hero_validation_interpreter.ErrorAtom.list())
        if errors:
            logger.trace(f"Model of failed hero validation:\n{symbols}")
        return errors

    @classmethod
    def hero_validation_warnings(cls, symbols: List[Symbol]) -> List[Symbol]:
        return cls._functions(symbols, hero_validation_interpreter.WarningAtom.list())

    @classmethod
    def known_feature_values(cls, symbols: List[Symbol], feature: Feature) -> List[Symbol]:
        return cls._functions(symbols, [RulebookFunction.known(feature)])

    @staticmethod
```

```
    def _functions(symbols: List[Symbol], name_whitelist: List[str] = None) -> List[Symbol]:
        """
        Collects all functions, may filtering by given name whitelist
        :return:
        """
        found = []
        for sym in symbols:
            if not sym.name:  # use 'or not sym.arguments:' to skip constants
                # skip tuples but keep constants
                continue
            if not name_whitelist or sym.name in name_whitelist:
                found.append(sym)
        return found
```

A.11 <repo>/app/engine/hero_validation_interpreter.py

Quellcode zur Interpretation der Ergebnis-Fakten.

```
from __future__ import annotations  # required till PEP 563

from clingo import Symbol
from clingo import SymbolType

from dsaheldenbogen.app.engine.rulebook_function import RulebookFunction
from dsaheldenbogen.app.models.base_enum import BaseEnum
from dsaheldenbogen.app.models.hero_validation_error import HeroValidationError
from dsaheldenbogen.app.models.hero_validation_param import HeroValidationParam
from dsaheldenbogen.app.models.hero_validation_warning import HeroValidationWarning


class ErrorAtom(str, BaseEnum):
  """Result fact to be interpreted as an error"""
  UNKNOWN = 'unknown'
  UNUSABLE_BY = 'unusable_by'
  MISSING = 'missing'
  MISSING_LEVEL = 'missing_level'
  MAX_LVL_EXCEEDED = 'max_lvl_exceeded'
  MAX_COUNT_EXCEEDED = 'max_count_exceeded'
  MISSING_MIN_LVL = 'missing_min_lvl'


class _ErrorAtomAddon(str, BaseEnum):
  ANY_OF = 'any_of'


class WarningAtom(str, BaseEnum):
  """Result fact to be interpreted as a warning"""
  MISSING_USUAL = 'missing_usual'
  MISSING_TYPICAL = 'missing_typical'
  ATYPICAL = 'atypical'


def as_error(error: Symbol) -> HeroValidationError:
  """
```

```
    Converts clingo symbol to a hero validation error
    :param error: expected to be a result fact of a hero validation run
    """
    match error.name:
        case ErrorAtom.UNKNOWN:
            return _unknown_error(error)
        case ErrorAtom.UNUSABLE_BY:
            return _unusable_by_error(error)
        case ErrorAtom.MISSING:
            return _missing_error(error)
        case ErrorAtom.MISSING_LEVEL:
            return _missing_level_error(error)
        case ErrorAtom.MAX_LVL_EXCEEDED:
            return _max_level_exceeded_error(error)
        case ErrorAtom.MAX_COUNT_EXCEEDED:
            return _max_count_exceeded_error(error)
        case ErrorAtom.MISSING_MIN_LVL:
            return _missing_min_lvl_error(error)
        case _:
            raise NotImplementedError(
                f"There is no 'error' parsing definition for given result fact.\n"
                f"Result fact name: {error.name}\n"
                f"Result fact parameters: {[str(a) for a in error.arguments]}."
            )


def as_warning(warning: Symbol) -> HeroValidationWarning:
    """
    Converts clingo symbol to a hero validation warning
    :param warning: expected to be a result fact of a hero validation run
    """
    match warning.name:
        case WarningAtom.ATYPICAL:
            return _atypical_warning(warning)
        case WarningAtom.MISSING_TYPICAL:
            return _missing_typical_warning(warning)
        case WarningAtom.MISSING_USUAL:
            return _missing_usual_warning(warning)
        case _:
            raise NotImplementedError(
                f"There is no 'warning' parsing definition for given result fact.\n"
                f"Result fact name: {warning.name}\n"
                f"Result fact parameters: {[str(a) for a in warning.arguments]}."
            )


def _unknown_error(error: Symbol):
    caused_feature = error.arguments[0]
    caused_feature_value = caused_feature.arguments[0].string
    # TODO may add (dis)advantages as addon for clarification
    if RulebookFunction.is_dis_advantage(caused_feature):
        caused_feature_using = caused_feature.arguments[1].string
        caused_feature_level = caused_feature.arguments[2].number
        return HeroValidationError(
            type=HeroValidationError.Type.UNKNOWN,
            message=f"Heros '{caused_feature.name}' of '{caused_feature_value}' using '{caused_feature_using}'"
                    f" at level '{caused_feature_level}' is not known.",
            parameter={
                HeroValidationParam.C_F: caused_feature.name,
                HeroValidationParam.C_F_VALUE: caused_feature_value,
                HeroValidationParam.C_F_LEVEL: caused_feature_level,
                HeroValidationParam.C_F_USING: caused_feature_using,
            },
        )
    else:
        return HeroValidationError(
```

```
      type=HeroValidationError.Type.UNKNOWN,
      message=f"Heros '{caused_feature.name}' value of '{caused_feature_value}' is not known.",
      parameter={
        HeroValidationParam.C_F: caused_feature.name,
        HeroValidationParam.C_F_VALUE: caused_feature_value,
      },
    )


def _unusable_by_error(error: Symbol):
  caused_feature = error.arguments[0]
  caused_feature_value = caused_feature.arguments[0].string
  referred_feature = error.arguments[1]
  referred_feature_value = referred_feature.arguments[0].string
  return HeroValidationError(
    type=HeroValidationError.Type.UNUSABLE_BY,
    message=f"Heros '{caused_feature.name}' is unusable for heros '{referred_feature.name}'.",
    parameter={
      HeroValidationParam.C_F: caused_feature.name,
      HeroValidationParam.C_F_VALUE: caused_feature_value,
      HeroValidationParam.R_F: referred_feature.name,
      HeroValidationParam.R_F_VALUE: referred_feature_value,
    },
  )


def _missing_error(error: Symbol):
  caused_feature = error.arguments[0]
  caused_feature_value = caused_feature.arguments[0].string
  referred_feature = error.arguments[1]
  if RulebookFunction.is_dis_advantage(referred_feature):
    referred_feature_value = referred_feature.arguments[0].string
    referred_feature_using = referred_feature.arguments[1].string
    referred_feature_level = referred_feature.arguments[2].number
    return HeroValidationError(
      type=HeroValidationError.Type.MISSING,
      message=f"Heros '{caused_feature.name}' of '{caused_feature_value}' is missing '{referred_feature.name}'"
              f" of '{referred_feature_value}'"
              f" using '{referred_feature_using}'"
              f" on level '{referred_feature_level}'.",
      parameter={
        HeroValidationParam.C_F: caused_feature.name,
        HeroValidationParam.C_F_VALUE: caused_feature_value,
        HeroValidationParam.R_F: referred_feature.name,
        HeroValidationParam.R_F_VALUE: referred_feature_value,
        HeroValidationParam.R_F_USING: referred_feature_using,
        HeroValidationParam.R_F_LEVEL: referred_feature_level,
      },
    )
  else:
    raise NotImplementedError("Using 'missing' referring non (dis)advantages is yet to be implemented.")


def _missing_level_error(error: Symbol):
  caused_feature = error.arguments[0]
  caused_feature_value = caused_feature.arguments[0].string
  referred_feature = error.arguments[1]
  referred_feature_sym = referred_feature.arguments[0]
  required_level = referred_feature.arguments[1]
  if _matches(_ErrorAtomAddon.ANY_OF, referred_feature_sym):
    # referred_feature_sym is the addon
    choices = referred_feature_sym.arguments[0]
    selection = [a.string for a in referred_feature_sym.arguments[1].arguments]
    return HeroValidationError(
      type=HeroValidationError.Type.MISSING_LEVEL,
      addon=HeroValidationError.Addon.ANY_OF,
```

```
      message=f"Heros '{caused_feature.name}' is missing minimum level '{required_level}'"
              f" for '{choices}' '{referred_feature.name}'"
              f" of '{selection}'.",
      parameter={
        HeroValidationParam.C_F: caused_feature.name,
        HeroValidationParam.C_F_VALUE: caused_feature_value,
        HeroValidationParam.R_F: referred_feature.name,
        HeroValidationParam.SELECTION: selection,
        HeroValidationParam.MIN_LEVEL: required_level.number,
        HeroValidationParam.SELECTION_MIN_CHOICES: choices.number,
      },
    )
  else:
    # referred_feature_sym is the actual referred feature value (String)
    return HeroValidationError(
      type=HeroValidationError.Type.MISSING_LEVEL,
      message=f"Heros '{caused_feature.name}' is missing minimum level '{required_level}'"
              f" for '{referred_feature.name}'"
              f" of '{referred_feature_sym.string}'.",
      parameter={
        HeroValidationParam.C_F: caused_feature.name,
        HeroValidationParam.C_F_VALUE: caused_feature_value,
        HeroValidationParam.R_F: referred_feature.name,
        HeroValidationParam.R_F_VALUE: referred_feature_sym.string,
        HeroValidationParam.MIN_LEVEL: required_level.number,
      },
    )


def _max_level_exceeded_error(error: Symbol):
  # TODO actually below is the referred feature, the caused feature is experience_level
  caused_feature = error.arguments[0]
  caused_feature_value = caused_feature.arguments[0].string
  caused_feature_level = caused_feature.arguments[1].number
  max_level = error.arguments[1].number
  return HeroValidationError(
    type=HeroValidationError.Type.MAX_LVL_EXCEEDED,
    message=f"Heros '{caused_feature.name}' of '{caused_feature_value}' exceeds maximum level '{max_level}'.",
    parameter={
      HeroValidationParam.C_F: caused_feature.name,
      HeroValidationParam.C_F_VALUE: caused_feature_value,
      HeroValidationParam.C_F_LEVEL: caused_feature_level,
      HeroValidationParam.MAX_LEVEL: max_level,
    },
  )


def _max_count_exceeded_error(error: Symbol):
  referred_feature = error.arguments[0]
  max_count = error.arguments[1].number
  return HeroValidationError(
    type=HeroValidationError.Type.MAX_COUNT_EXCEEDED,
    message=f"Hero has too many '{referred_feature.name}'. Maximum is '{max_count}'.",
    parameter={
      HeroValidationParam.C_F: referred_feature.name,
      HeroValidationParam.MAX_COUNT: max_count
    },
  )


def _missing_min_lvl_error(error: Symbol):
  referred_feature = error.arguments[0]
  referred_feature_value = referred_feature.arguments[0].string
  referred_feature_level = referred_feature.arguments[1].number
  min_level = error.arguments[1].number
```

```
234   return HeroValidationError(
235     type=HeroValidationError.Type.MISSING_MIN_LVL,
236     message=f"Heros '{referred_feature.name}' of
      ↪  '{referred_feature_value}' is below minimum '{min_level}'.",
237     parameter={
238       HeroValidationParam.C_F: referred_feature.name,
239       HeroValidationParam.C_F_VALUE: referred_feature_value,
240       HeroValidationParam.C_F_LEVEL: referred_feature_level,
241       HeroValidationParam.MIN_LEVEL: min_level
242     },
243   )
244
245
246 def _atypical_warning(warning: Symbol):
247   caused_feature = warning.arguments[0]
248   caused_feature_value = caused_feature.arguments[0].string
249   referred_feature = warning.arguments[1]
250   referred_feature_value = referred_feature.arguments[0].string
251   if RulebookFunction.is_dis_advantage(referred_feature):
252     referred_feature_using = referred_feature.arguments[1].string
253     return HeroValidationWarning(
254       type=HeroValidationWarning.Type.ATYPICAL,
255       message=f"For heros '{caused_feature.name}' is atypical:
        ↪  '{referred_feature.name}'"
256               f" of '{referred_feature_value}'"
257               f" using '{referred_feature_using}'.",
258       parameter={
259         HeroValidationParam.C_F: caused_feature.name,
260         HeroValidationParam.C_F_VALUE: caused_feature_value,
261         HeroValidationParam.R_F: referred_feature.name,
262         HeroValidationParam.R_F_VALUE: referred_feature_value,
263         HeroValidationParam.R_F_USING: referred_feature_using,
264       },
265     )
266   else:
267     raise NotImplementedError("Using 'atypical' referring non
      ↪  (dis)advantages is yet to be implemented.")
268
269
270 def _missing_typical_warning(warning: Symbol):
271   caused_feature = warning.arguments[0]
272   caused_feature_value = caused_feature.arguments[0].string
273   referred_feature = warning.arguments[1]
274   referred_feature_value = referred_feature.arguments[0].string
275   # TODO may add (dis)advantages as addon (new field; like on error)
      ↪  for clarification
276   if RulebookFunction.is_dis_advantage(referred_feature):
277     referred_feature_using = referred_feature.arguments[1].string
278     referred_feature_level = referred_feature.arguments[2].number
279     # TODO only add 'referred_feature_using' to message and parameter
      ↪  when not empty
280     return HeroValidationWarning(
281       type=HeroValidationWarning.Type.MISSING_TYPICAL,
282       message=f"Heros '{caused_feature.name}' is missing typical
        ↪  '{referred_feature.name}'"
283               f" of '{referred_feature_value}'"
284               f" using '{referred_feature_using}'"
285               f" at level '{referred_feature_level}'.",
286       parameter={
287         HeroValidationParam.C_F: caused_feature.name,
288         HeroValidationParam.C_F_VALUE: caused_feature_value,
289         HeroValidationParam.R_F: referred_feature.name,
290         HeroValidationParam.R_F_VALUE: referred_feature_value,
291         HeroValidationParam.R_F_LEVEL: referred_feature_level,
292         HeroValidationParam.R_F_USING: referred_feature_using,
293       },
294     )
295   else:
296     # TODO handle referred features having a level
297     return HeroValidationWarning(
298       type=HeroValidationWarning.Type.MISSING_TYPICAL,
```

```
      message=f"Heros '{caused_feature.name}' is missing typical '{referred_feature.name}'"
              f" of '{referred_feature_value}'.",
      parameter={
        HeroValidationParam.C_F: caused_feature.name,
        HeroValidationParam.C_F_VALUE: caused_feature_value,
        HeroValidationParam.R_F: referred_feature.name,
        HeroValidationParam.R_F_VALUE: referred_feature_value,
      },
    )


def _missing_usual_warning(warning: Symbol):
  caused_feature = warning.arguments[0]
  caused_feature_value = caused_feature.arguments[0].string
  referred_feature = warning.arguments[1]
  referred_feature_value = referred_feature.arguments[0].string
  return HeroValidationWarning(
    type=HeroValidationWarning.Type.MISSING_USUAL,
    message=f"Heros '{caused_feature.name}' is missing usual '{referred_feature.name}'"
            f" of '{referred_feature_value}'.",
    parameter={
      HeroValidationParam.C_F: caused_feature.name,
      HeroValidationParam.C_F_VALUE: caused_feature_value,
      HeroValidationParam.R_F: referred_feature.name,
      HeroValidationParam.R_F_VALUE: referred_feature_value,
    },
  )


def _matches(function_name: str, symbol: Symbol) -> bool:
  """
  :returns: True whenever the symbol is a clingo function and the name matches
  """
  return symbol.type == SymbolType.Function and symbol.name == function_name
```

A.12 <repo>/app/services/rulebook_validator.py

Prüft die Regelbücher-*LPs* gegen bestimmte Anforderungen ab. Die `check` Methode wird von einem entsprechenden Test für alle Regelbücher ausgeführt.

```
from typing import List
from typing import Optional

from dsaheldenbogen.app.engine.rulebook_function import RulebookFunction
from dsaheldenbogen.app.engine.rulebook_program import RulebookProgram
from dsaheldenbogen.app.logger import getLogger
from dsaheldenbogen.app.models.rulebook import Rulebook
from dsaheldenbogen.app.resource import Resource
from dsaheldenbogen.app.services.rulebook_executor import RulebookExecutor

logger = getLogger(__name__)
```

```

class RulebookValidator:
    """
    Validates that within a rulebook resource folder all required files are present.
    TODO It does not check if any of these files are loaded (LP: #include).
    """
    REQUIRED_PROGRAMS = [
        RulebookProgram.RULEBOOK_USABLE,
    ]

    required_files: set[str] = {Rulebook.entrypoint_file_name(), 'meta.lp', 'rules.lp'}

    @classmethod
    def filter(cls, rulebooks: List[Rulebook]) -> List[Rulebook]:
        valid_books = []
        for rulebook in rulebooks:
            errors = cls.check(rulebook)
            if errors:
                logger.warning(f"Rulebook '{rulebook}' is not valid and will be ignored.")
                logger.debug(errors)
            else:
                valid_books.append(rulebook)
        return valid_books

    @classmethod
    def check(cls, rulebook: Rulebook) -> List[str]:
        """
        :return: list of errors, rulebook is valid when empty
        """
        try:
            errors = [cls._file_structure_valid(rulebook),
                      cls._has_required_programs(rulebook)] \
                     + cls._only_declares_itself(rulebook)
            return [err for err in errors if err is not None]
        except Exception as ex:
            logger.exception(f"Could not validate rulebook '{rulebook}'.")
            raise ex

    @classmethod
    def _file_structure_valid(cls, rulebook: Rulebook) -> Optional[str]:
        found_files = set(Resource.list_files(rulebook.folder()))
        if not cls.required_files.issubset(found_files):
            return f"Rulebook '{rulebook}' missing required file(s): {cls.required_files - found_files}"

    @classmethod
    def _has_required_programs(cls, rulebook: Rulebook) -> Optional[str]:
        book = RulebookExecutor(rulebook)
        missing_programs = book.has_programs(cls.REQUIRED_PROGRAMS)
        if missing_programs:
            return f"Rulebook '{rulebook}' missing required program(s): {missing_programs}"

    @staticmethod
    def _only_declares_itself(rulebook: Rulebook) -> List[str]:
        errors = []
        book = RulebookExecutor(rulebook)
        found, others = book.has_function_with_value(
            [RulebookProgram.RULEBOOK_USABLE], RulebookFunction.RULEBOOK, rulebook.name
        )
        if not found:
            errors.append(f"Rulebook '{rulebook}' does not declares itself as fact.")
```

```
        if others:
            errors.append(f"Rulebook '{rulebook}' declares to be other rulebook(s): {others}")
    return errors
```

A.13 <repo>/tests/resources/test_resources_rulebooks.py

Prüft alle „echten“ Regelbücher mit dem `RulebookValidator` (Anhang A.12 auf S. 85).

```
from dsaheldenbogen.app.models.rulebook import Rulebook
from dsaheldenbogen.app.services.rulebook_validator import RulebookValidator


class TestResourcesRulebooks:

    def test_rulebooks_valid(self):
        # given:
        known_rulebooks = Rulebook.list_known()
        # when:
        errors = []
        for rulebook in known_rulebooks:
            errors = errors + RulebookValidator.check(rulebook)
        # then:
        if errors:
            self.fail('\n'.join(errors))
```

A.14 <repo>/tests/app/services/test_meta_service.py

Komponententest das die Funktionalitäten des `MetaService` (Anhang A.15 auf S. 88) prüft. Es nutzt dazu die `TestingEngine` (Anhang A.16 auf S. 89).

```
from parameterized import parameterized

from dsaheldenbogen.app.models.feature import Feature
from dsaheldenbogen.app.services.meta_service import MetaService
from tests.app.engine.testing_engine import TestingEngine
from tests.app.models.testing_rulebook import TestingRulebook


class TestMetaService:
    service = MetaService(TestingEngine)
```

```
11
12      @parameterized.expand([
13          (1, Feature.EXPERIENCE_LEVEL),
14          (2, Feature.RACE),
15          (3, Feature.CULTURE),
16          (1, Feature.PROFESSION),
17          (4, Feature.ADVANTAGE),
18          (3, Feature.DISADVANTAGE),
19          (2, Feature.TALENT),
20          (1, Feature.COMBAT_TECHNIQUE),
21      ])
22      def test_feature_listing(self, expected_count: int, feature:
        ↪  Feature):
23          # given:
24          rulebooks = ['meta_service']
25          # when:
26          found = self.service.list_known_feature_values(feature,
            ↪  TestingRulebook.map(rulebooks))
27          # then:
28          assert len(found) == expected_count
```

A.15 <repo>/app/services/meta_service.py

App-Service für den `/meta` Endpunkt.

```
1  from typing import List
2
3  from dsaheldenbogen.app.engine.engine import Engine
4  from dsaheldenbogen.app.logger import getLogger
5  from dsaheldenbogen.app.models.feature import Feature
6  from dsaheldenbogen.app.models.rulebook import Rulebook
7  from dsaheldenbogen.app.services.rulebook_validator import
   ↪  RulebookValidator
8
9  logger = getLogger(__name__)
10
11
12 class MetaService:
13
14     def __init__(self, engine_clz: type = Engine) -> None:
15         self.engine_clz = engine_clz
16
17     def list_usable_rulebooks(self) -> List[Rulebook]:
18         """
19         List all rulebooks which are ready to use
20         """
21         return RulebookValidator.filter(Rulebook.list_known())
22
23     def list_known_feature_values(self, feature: Feature, rulebooks:
       ↪  List[Rulebook]) -> List[tuple[str, str, int]] | List[str]:
24         """
25         List all known feature values of given feature considering
       ↪  given rulebooks
26         :return: List[tuple[str, str, int]] in case of DisAdvantages
       ↪  else List[str]
27         """
28         engine = self.engine_clz(rulebooks)
29         known_values = engine.list_known_for(feature)
30         logger.debug(f"Value list of '{feature}' with rulebooks
           ↪  {[str(r) for r in rulebooks]}: {known_values}")
31         return known_values
```

A.16 <repo>/tests/app/engine/testing_engine.py

Ist eine einfachere Variante der echten Engine, welches es ermöglicht einfache Test-Regelbücher-*LPs* aus zu führen.

```python
from __future__ import annotations  # required till PEP 563

from typing import List

from dsaheldenbogen.app.engine.engine import Engine
from dsaheldenbogen.app.engine.rulebook_program import RulebookProgram
from dsaheldenbogen.app.logger import getLogger
from dsaheldenbogen.app.models.rulebook import Rulebook
from dsaheldenbogen.infrastructure.clingo_executor import ClingoExecutor

logger = getLogger(__name__)


class TestingEngine(Engine):
    """
    This is the actual engine for test implementations of rulebooks and hence having some modifications:
     - given rulebooks can be real ones or test implementations (TestRulebook), so mixing is possible
     - it does NOT validate given rulebooks
     - it does NOT check the rulebooks usability
    """

    def __init__(self, rulebooks: List[Rulebook]) -> None:
        self.rulebooks = rulebooks
        self.clingo_executor = ClingoExecutor(
            [r.entrypoint_file() for r in self.rulebooks],
            [RulebookProgram.BASE]
        )

        self._find_extra_hero_validation_steps()
        self.hero_validation_steps = dict(sorted(self.hero_validation_steps.items()))  # sort by keys
```

A.17 <repo>/tests/app/engine/test_hero_validation.py

Testet einzelne Spielwelt-Regeln ab, welches jedoch nur unter Einhaltung der Charakter-Validigierungsschritte möglich ist.

```python
from contextlib import nullcontext

import pytest
from parameterized import parameterized

from dsaheldenbogen.app.engine.engine import Engine
from dsaheldenbogen.app.engine.exceptions import HeroInvalidError
from dsaheldenbogen.app.models.hero import Hero
from dsaheldenbogen.app.models.rulebook import Rulebook


class TestHeroValidation:

    @parameterized.expand([
        (0, 'Mensch', 'Aranier'),
        (1, 'Mensch', ''),
        (1, 'Mensch', '_invalid_'),
        (1, 'Mensch', 'Erzzwerge'),
    ])
    def test_culture_usable(self, error_count: int, race: str, culture: str):
        # given:
        engine = Engine(Rulebook.map(['dsa5']))
        hero = Hero(name="name",
                    experience_level='Durchschnittlich',
                    race=race,
                    culture=culture,
                    profession='Händler',
                    talents=[],
                    combat_techniques=[],
                    advantages=[],
                    disadvantages=[],
                    )
        # when:
        with pytest.raises(HeroInvalidError) if error_count > 0 else nullcontext() as ctx:
            engine.validate(hero)
        # then:
        if ctx and ctx.value:
            assert error_count == len(ctx.value.errors), ctx.value.errors

    @parameterized.expand([
        (0, 'Mensch', 'Menschlichekultur', 'Händler'),
        (1, 'Mensch', 'Menschlichekultur', ''),
        (1, 'Mensch', 'Menschlichekultur', '_invalid_'),
        (1, 'Mensch', 'Menschlichekultur', 'Zauberweber'),
    ])
    def test_profession_usable(self, error_count: int, race: str, culture: str, profession: str):
        # given:
        engine = Engine(Rulebook.map(['dsa5']))
        hero = Hero(name="name",
                    experience_level='Durchschnittlich',
                    race=race,
                    culture=culture,
                    profession=profession,
                    talents=[],
                    combat_techniques=[],
                    advantages=[],
                    disadvantages=[],
                    )
        # when:
        with pytest.raises(HeroInvalidError) if error_count > 0 else nullcontext() as ctx:
            engine.validate(hero)
        # then:
        if ctx and ctx.value:
            assert error_count == len(ctx.value.errors), ctx.value.errors
```

A.18 <repo>/tests/e2e/test_hero_api.py

Ende-zu-Ende Test, welches den `/hero` Endpunkt überprüft.

```python
from http import HTTPStatus

from httpx import Response
from parameterized import parameterized
from starlette.testclient import TestClient

from dsaheldenbogen.api.root import app
from dsaheldenbogen.api.schemas.hero_validation_result import HeroValidationResult
from tests.e2e.invalid_heros import InvalidHeroTestcase
from tests.e2e.invalid_heros import InvalidHeroTestcases
from tests.e2e.valid_heros import ValidHeroTestcase
from tests.e2e.valid_heros import ValidHeroTestcases


def _is_subset_of(subset: dict, superset: dict) -> bool:
    return all(item in superset.items() for item in subset.items())


class TestHeroApi:
    client = TestClient(app)

    @parameterized.expand(InvalidHeroTestcases.all())
    def test_invalid_heros(self, testcase: InvalidHeroTestcase):
        # when:
        response: Response = self.client.post(
            "/api/hero/validate",
            json=testcase.hero.dict(),
            params={'rulebooks': testcase.rulebooks}
        )
        # then:
        assert response.status_code == HTTPStatus.OK
        result = HeroValidationResult.parse_obj(response.json())
        # and:
        found_type = False
        found_params = False
        for error in result.errors:
            if error.type == testcase.error_type:
                found_type = True
                if _is_subset_of(testcase.error_params, error.parameter):
                    found_params = True
                    break
        assert found_type, \
            f"Did not find error of expected type." \
            f"\nexpected type: {testcase.error_type}" \
            f"\nfound errors: {result.errors}"
        params_of_correct_error_type = [str(e.parameter) for e in result.errors if e.type == testcase.error_type]
        assert found_params, \
            f"Found error type but does not have expected params." \
            f"\nexpected params: {testcase.error_params}" \
            f"\nfound params:" \
            f"\n{chr(10).join(params_of_correct_error_type)}"

    @parameterized.expand(ValidHeroTestcases.all())
```

```
    def test_valid_hero(self, testcase: ValidHeroTestcase):
        # when:
        response: Response = self.client.post(
            "/api/hero/validate",
            json=testcase.hero.dict(),
            params={'rulebooks': testcase.rulebooks}
        )
        # then:
        assert response.status_code == HTTPStatus.OK
        result = HeroValidationResult.parse_obj(response.json())
        # and:
        assert result.valid, "A valid Hero has validation errors."
```

A.19 <repo>/tests/e2e/valid-heros.py

Dies ist, aufgrund der Länge, nur ein Auszug und nicht die vollständige Datei. Es representiert Testfälle mit gültigen Charakteren und wird von `TestHeroApi` (Anhang A.18 auf S. 91) genutzt.

```
from enum import Enum
from typing import List
from typing import NamedTuple

from dsaheldenbogen.api.schemas.hero import Hero


# pylint: disable=duplicate-code

class ValidHeroTestcase(NamedTuple):
    rulebooks: List[str]
    hero: Hero


class ValidHeroTestcases(ValidHeroTestcase, Enum):

    @classmethod
    def all(cls) -> List[tuple[ValidHeroTestcase]]:
        """List all values of the enum"""
        return list(map(lambda c: (c.value,), cls))

    SOELDNER = ValidHeroTestcase(['dsa5'], Hero(
        name='valid_söldner',
        experience_level='Durchschnittlich',
        race='Zwerg',
        culture='Ambosszwerge',
        profession='Söldner',
        talents={'Körperbeherrschung': 3, 'Kraftakt': 3,
        ↪  'Selbstbeherrschung': 4, 'Zechen': 5, 'Menschenkenntnis':
        ↪  3, 'Überreden': 3, 'Orientierung': 4, 'Wildnisleben': 3,
        ↪  'Götter & Kulte': 3, 'Kriegskunst': 6, 'Sagen &
        ↪  Legenden': 5, 'Handel': 3, 'Heilkunde Wunden': 4},
        combat_techniques={'Armbrüste': 10, 'Raufen': 10,
        ↪  'Hiebwaffen': 10},
        advantages=[('Dunkelsicht', '', 1), ('Immunität',
        ↪  'Tulmadron', 1)],
        disadvantages=[('Unfähig', 'Schwimmen', 1)],
    ))

```

```
    SKULDRUN = ValidHeroTestcase(['dsa5',
    ↪ 'dsa5_aventurisches_götterwirken_2'], Hero(
        name='valid_skuldrun',
        experience_level='Erfahren',
        race='Mensch',
        culture='Fjarninger',
        profession='Skuldrun',
        talents={"Körperbeherrschung": 4, "Kraftakt": 6,
        ↪ "Selbstbeherrschung": 4, "Sinnesschärfe": 4, "Bekehren &
        ↪ Überzeugen": 2, "Einschüchtern": 2, "Etikette": 4,
        ↪ "Menschenkenntnis": 4, "Willenskraft": 4, "Orientierung":
        ↪ 4, "Pflanzenkunde": 2, "Tierkunde": 4, "Wildnisleben": 4,
        ↪ "Geschichtswissen": 4, "Götter & Kulte": 5, "Rechnen": 2,
        ↪ "Rechtskunde": 5, "Sagen & Legenden": 6, "Heilkunde
        ↪ Krankheiten": 4, "Heilkunde Seele": 4, "Heilkunde
        ↪ Wunden": 3},
        combat_techniques={'Hiebwaffen': 11, 'Raufen': 10,
        ↪ 'Stangenwaffen': 10, 'Wurfwaffen': 8},
        advantages=[('Geweihter', '', 1)],
        disadvantages=[('Prinzipientreue', '', 1),
        ↪ ('Verpflichtungen', '', 2)],
    ))
```

A.20 <repo>/tests/e2e/invalid-heros.py

Dies ist, aufgrund der Länge, nur ein Auszug und nicht die vollständige Datei. Es representiert Testfälle mit ungültigen Charakteren und wird von `TestHeroApi` (Anhang A.18 auf S. 91) genutzt.

```python
from enum import Enum
from typing import List
from typing import NamedTuple
from typing import Optional

from dsaheldenbogen.api.schemas.hero import Hero
from dsaheldenbogen.app.models.hero_validation_error import
↪ HeroValidationError
from dsaheldenbogen.app.models.hero_validation_param import
↪ HeroValidationParam


# pylint: disable=duplicate-code

class InvalidHeroTestcase(NamedTuple):
    # expected:
    error_type: HeroValidationError.Type
    error_params: dict[HeroValidationParam, str, Optional[List[str]]]
    # given:
    rulebooks: List[str]
    hero: Hero

```

```
21
22 class InvalidHeroTestcases(InvalidHeroTestcase, Enum):
23
24     @classmethod
25     def all(cls) -> List[tuple[InvalidHeroTestcase]]:
26         """List all values of the enum"""
27         return list(map(lambda c: (c.value,), cls))
28
29     UNKNOWN_RACE = InvalidHeroTestcase(
30         HeroValidationError.Type.UNKNOWN,
31         {
32             HeroValidationParam.C_F: 'race',
33             HeroValidationParam.C_F_VALUE: '__unknown__',
34         },
35         ['dsa5'],
36         Hero(
37             name='valid_söldner',
38             experience_level='Durchschnittlich',
39             race='__unknown__',
40             culture='Ambosszwerge',
41             profession='Söldner',
42             talents={},
43             combat_techniques={},
44             advantages=[],
45             disadvantages=[],
46         )
47     )
48
49     CULTURE_UNUSABLE_BY_RACE = InvalidHeroTestcase(
50         HeroValidationError.Type.UNUSABLE_BY,
51         {
52             HeroValidationParam.C_F: 'culture',
53             HeroValidationParam.C_F_VALUE: 'Andergaster',
54             HeroValidationParam.R_F: 'race',
55             HeroValidationParam.R_F_VALUE: 'Zwerg',
56         },
57         ['dsa5'],
58         Hero(
59             name='valid_söldner',
60             experience_level='Durchschnittlich',
61             race='Zwerg',
62             culture='Andergaster',
63             profession='Söldner',
64             talents={},
65             combat_techniques={},
66             advantages=[],
67             disadvantages=[],
68         )
69     )
70
71     PROFESSION_UNUSABLE_BY_CULTURE = InvalidHeroTestcase(
72         HeroValidationError.Type.UNUSABLE_BY,
73         {
74             HeroValidationParam.C_F: 'profession',
75             HeroValidationParam.C_F_VALUE: 'Skuldrun',
76             HeroValidationParam.R_F: 'culture',
77             HeroValidationParam.R_F_VALUE: 'Ambosszwerge',
78         },
79         ['dsa5', 'dsa5_aventurisches_götterwirken_2'],
80         Hero(
81             name='valid_söldner',
82             experience_level='Durchschnittlich',
83             race='Zwerg',
84             culture='Ambosszwerge',
85             profession='Skuldrun',
86             talents={},
87             combat_techniques={},
88             advantages=[],
89             disadvantages=[],
90         )
91     )
92
93     PROFESSION_MISSING_LEVEL_FOR_TALENT = InvalidHeroTestcase(
```

```
        HeroValidationError.Type.MISSING_LEVEL,
        {
            HeroValidationParam.C_F: 'profession',
            HeroValidationParam.C_F_VALUE: 'Söldner',
            HeroValidationParam.R_F: 'talent',
            HeroValidationParam.R_F_VALUE: 'Körperbeherrschung',
            HeroValidationParam.MIN_LEVEL: 3,
        },
        ['dsa5'],
        Hero(
            name='valid_söldner',
            experience_level='Durchschnittlich',
            race='Zwerg',
            culture='Ambosszwerge',
            profession='Söldner',
            talents={},
            combat_techniques={},
            advantages=[],
            disadvantages=[],
        )
    )

    PROFESSION_MISSING_LEVEL_FOR_COMBAT_TECHNIQUE =
    ↪ InvalidHeroTestcase(
        HeroValidationError.Type.MISSING_LEVEL,
        {
            HeroValidationParam.C_F: 'profession',
            HeroValidationParam.C_F_VALUE: 'Söldner',
            HeroValidationParam.R_F: 'combat_technique',
            HeroValidationParam.R_F_VALUE: 'Armbrüste',
            HeroValidationParam.MIN_LEVEL: 10,
        },
        ['dsa5'],
        Hero(
            name='valid_söldner',
            experience_level='Durchschnittlich',
            race='Zwerg',
            culture='Ambosszwerge',
            profession='Söldner',
            talents={},
            combat_techniques={},
            advantages=[],
            disadvantages=[],
        )
    )

    PROFESSION_MISSING_LEVEL_FOR_ANY_OF_COMBAT_TECHNIQUES =
    ↪ InvalidHeroTestcase(
        HeroValidationError.Type.MISSING_LEVEL,
        {
            HeroValidationParam.C_F: 'profession',
            HeroValidationParam.C_F_VALUE: 'Söldner',
            HeroValidationParam.R_F: 'combat_technique',
            HeroValidationParam.SELECTION: ['Hiebwaffen',
            ↪ 'Schwerter', 'Stangenwaffen', 'Zweihandschwerter',
            ↪ 'Zweihandhiebwaffen'],
            HeroValidationParam.MIN_LEVEL: 10,
            HeroValidationParam.SELECTION_MIN_CHOICES: 1,
        },
        ['dsa5'],
        Hero(
            name='valid_söldner',
            experience_level='Durchschnittlich',
            race='Zwerg',
            culture='Ambosszwerge',
            profession='Söldner',
            talents={},
            combat_techniques={},
            advantages=[],
            disadvantages=[],
        )
    )
```

```
163     TALENT_EXCEEDS_MAX_LEVEL_BY_EXPERIENCE = InvalidHeroTestcase(
164         HeroValidationError.Type.MAX_LVL_EXCEEDED,
165         {
166             HeroValidationParam.C_F: 'talent',
167             HeroValidationParam.C_F_VALUE: 'Körperbeherrschung',
168             HeroValidationParam.C_F_LEVEL: 17,
169             HeroValidationParam.MAX_LEVEL: 16,
170         },
171         ['dsa5'],
172         Hero(
173             name='valid_söldner',
174             experience_level='Meisterlich',
175             race='Zwerg',
176             culture='Ambosszwerge',
177             profession='Söldner',
178             talents={'Körperbeherrschung': 17, 'Kraftakt': 3,
                    ↪  'Selbstbeherrschung': 4, 'Zechen': 5,
                    ↪  'Menschenkenntnis': 3, 'Überreden': 3,
                    ↪  'Orientierung': 4, 'Wildnisleben': 3, 'Götter &
                    ↪  Kulte': 3, 'Kriegskunst': 6, 'Sagen & Legenden': 5,
                    ↪  'Handel': 3, 'Heilkunde Wunden': 4},
179             combat_techniques={'Armbrüste': 10, 'Raufen': 10,
                    ↪  'Hiebwaffen': 10},
180             advantages=[('Dunkelsicht', '', 1), ('Immunität',
                    ↪  'Tulmadron', 1)],
181             disadvantages=[('Unfähig', 'Schwimmen', 1)],
182         )
183     )
```

Tabellenverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

Literaturverzeichnis

Alle Online-Referenzen (jene mit URLs) wurden am 03.05.2023 abgerufen.

[1] Ulrich Kiesow, Ina Kramer: „DSA Regelwerk“. 1. Edition (2. Auflage). Schmidt Spiele GmbH und Droemer Knaur, 1. März 1984. ISBN: 3-426-30000-1

[2] Alex Spohr, Jens Ullrich, Tobias Rafael Junge: „DSA Regelwerk“. 5. Edition. Ulisses Medien und Spiel Distribution GmbH, 2015. ISBN: 978-3-95752-103-3

[3] Simon Maria Glasmacher: „Rollenspiele in Bibliotheken“. Bachelorarbeit. Technische Hochschule Köln, 3. Jan. 2022, S. 12–18. URL: https://publiscologne.th-koeln.de/frontdoor/deliver/index/docId/1848/file/BA_Glasmacher_Simon.pdf

[4] Marcel Mertz, Jan Schürmann: „Wissen und Wissen-lassen – Wissenstypen und Wissensverteilung im Pen-and-Paper-Rollenspiel. Eine wissenssoziologisch informierte empirische und konzeptuelle Studie mit wissenstypologischem Schwerpunkt“. München: Grin Verlag, 2008, S. 41–46. ISBN: 978-3-6406-400-65

[5] Alex Spohr: „DSA5 Heldendokumente“. 17. Jan. 2023. URL: https://www.ulisses-ebooks.de/product/159699/DSA5-Heldendokumente-PDF-als-Download-kaufen

[6] Thorsten Most: „Selbstrechnende Dokumente“. 16. Jan. 2023. URL: https://www.ulisses-ebooks.de/product/214532/Selbstrechnende-Dokumente

[7] Bernhard Jung: „The Dark Aid“. 4. Sep. 2022. URL: https://www.ulisses-ebooks.de/product/212543/The-Dark-Aid-alpha?cPath=10353_26558

[8] Wiki Aventurica: „DSA-Tools“. 20. Sep. 2022. URL: https://de.wiki-aventurica.de/wiki/DSA-Tools?oldid=2536196

[9] Wiki Aventurica: „DSA-Tools Deluxe“. 20. Sep. 2022. URL: https://de.wiki-aventurica.de/wiki/DSA-Tools_Deluxe?oldid=2536113

[10] Wiki Aventurica: „Programm“. 11. Sep. 2022. URL: https://de.wiki-aventurica.de/wiki/Programm?oldid=2521608

[11] Lukas Obermann: „GitHub - Optolith Heldengenerator“. 12. Dez. 2019. URL: https://github.com/elyukai/optolith-client

[12] helden-software.de: „Großes Q&A - Entwicklungsumgebung“. 25. Apr. 2007. URL: https://wiki.helden-software.de/wiki/Antworten?oldid=860#Entwicklungsumgebung

[13] helden-software.de: „Team“. 23. Juni 2010. URL: https://wiki.helden-software.de/wiki/Team?oldid=2575

[14] helden-software.de: „Steigern/Menü/Internet“. 28. Feb. 2007. URL: http://wiki.helden-software.de/wiki/Steigern/Men%C3%BC/Internet?oldid=527

[15] helden-software.de: „Plugins“. 23. Juni 2010. URL: http://wiki.helden-software.de/wiki/Plugins?oldid=2928

[16] Yantur Patrik Rüegge: „Excel Heldenblatt - Heldenblatt.ch“. 10. Mai 2021. URL: https://www.orkenspalter.de/filebase/index.php?file/1890-excel-heldenblatt-heldenblatt-ch/#overview

[17] Yantur Patrik Rüegge: „Heldenblatt.ch - Startseite“. 15. Mai 2022. URL: https://www.heldenblatt.ch

[18] MeisterGeister: „Was ist MeisterGeister?“ 2016. URL: https://meistergeister.org/meistergeister

[19] MeisterGeister: „BugFix Version 2.6.0.7 erschienen“. 6. Juni 2022. URL: https://meistergeister.org/2022/06/06/bugfix-version-2-6-0-7-erschienen

[20] MeisterGeister: „Team“. 2014. URL: http://meistergeister.org/team/

[21] MeisterGeister: „Readme.md“. 15. Mai 2022. URL: https://github.com/Constructor0987/MeisterGeister

[22] helden-software.de: „Helden-Software und DSA 5.0“. 25. Aug. 2015. URL: https://www.helden-software.de/index.php/2015/08/25/helden-software-und-dsa-5-0-2

[23] MeisterGeister: „MeisterGeister Projekt – Wir leben noch!“ 17. Okt. 2018. URL: https://meistergeister.org/2018/10/17/meistergeister-projekt-wir-leben-noch

[24] Wiki Aventurica: „Regelsystem - Regelsystem“. 31. Jan. 2022. URL: https://de.wiki-aventurica.de/wiki/Regelsystem?oldid=2339553

[25] Wiki Aventurica: „Grundregel“. 31. Okt. 2022. URL: https://de.wiki-aventurica.de/wiki/Grundregel?oldid=2566326

[26] Wiki Aventurica: „Optionale Regel“. 31. Okt. 2022. URL: https://de.wiki-aventurica.de/wiki/Optionale_Regel?oldid=2566306

[27] Wiki Aventurica: „Fokusregel“. 30. Okt. 2022. URL: https://de.wiki-aventurica.de/wiki/Fokusregel?oldid=2275125

[28] Ulisses Spiele GmbH: „DSA Regel Wiki - Jagd“. 2023. URL: https://www.ulisses-regelwiki.de/Fokus_Jagd.html

[29] Wiki Aventurica: „Hausregel“. 21. Apr. 2021. URL: https://de.wiki-aventurica.de/wiki/Hausregel?oldid=2178798

[30] Ulisses Spiele GmbH: „DSA Regel Wiki - Heldenerschaffung“. 2023. URL: https://www.ulisses-regelwiki.de/Heldenerschaffung.html

[31] Ulisses Spiele GmbH: „DSA Regel Wiki“. Aus Unterseiten. 2023. URL: https://www.ulisses-regelwiki.de/start.html

[32] Ulisses Spiele GmbH: „DSA Regel Wiki - Spezies Elfen“. 2023. URL: https://www.ulisses-regelwiki.de/Spez_Elfen.html

[33] Zoe Adamietz, David Schmidt, Alex Spohr: „Aventurisches Götterwirken 2“. Seite 139. Ulisses Medien und Spiel Distribution GmbH, 28. Nov. 2019. ISBN: 978-3-96331-264-9. URL: https://www.ulisses-regelwiki.de/Gew_bruder_des_feuers.html

[34] Nikolai Hoch, Peter Horstmann, Rafael Knop, Andreas Landkammer, Jeanette Marsteller, David Schmidt, Alex Spohr, Nina Wendelken: „Die Gestade des Gottwals – Thorwal & das Gjalskerland“. Seite 153. Ulisses Medien und Spiel Distribution GmbH, 29. Apr. 2021. ISBN: 978-3-96331-482-7. URL: https://ulisses-regelwiki.de/allgemeine_magische_sonderfertigkeit.html?sonderfertigkeit=Blutrunen

[35] Ulisses Spiele GmbH: „DSA Regel Wiki - Berufsgeheimnis“. 2023. URL: https://www.ulisses-regelwiki.de/Berugsgeheimnis.html

[36] Ulisses Spiele GmbH: „DSA Regel Wiki - Vor- und Nachteile“. 2023. URL: https://www.ulisses-regelwiki.de/vor-und-nachteile.html

[37] Alex Spohr, Jens Ullrich, Tobias Rafael Junge: „DSA Regelwerk“. zweite 3. Edition. Seite 350 ff. Fantasy Productions Verlags- und Medienvertriebsgesellschaft mbH, 1997. ISBN: 978-3-81185-499-4. URL: https://www.ulisses-regelwiki.de/Erfahrung.html

[38] Benno Luthiger: „Alles aus Spaß? Zur Motivation von Open-Source-Entwicklern“. In: „Open Source Jahrbuch. Zwischen Softwareentwicklung und Gesellschaftsmodell“. Hrsg. von Robert A. Gehring, Bernd Lutterbeck. Kap. 2. Berlin: Lehmanns Media - LOB.de, 2004, S. 93–106. ISBN: 3-936427-78-X

[39] Gregorio Robles: „A Software Engineering approach to Libre Software“. Englisch. In: „Open Source Jahrbuch. Zwischen Softwareentwicklung und Gesellschaftsmodell“. Hrsg. von Robert A. Gehring, Bernd Lutterbeck. Berlin: Lehmanns Media - LOB.de, 2004, S. 193–208. ISBN: 3-936427-78-X

[40] Niels Henrik Jørgensen: „Putting it all in the trunk: incremental software development in the FreeBSD open source project“. Englisch. In: „Information Systems Journal“ 11.4 (2001), S. 321–336. ISSN: 1350-1917

[41] A. Hars, Shaosong Ou: „Working for free? Motivations of participating in open source projects“. Englisch. In: „Proceedings of the 34th Annual Hawaii International Conference on System Sciences“. 2001, 9 pp. DOI: 10.1109/HICSS.2001.927045

[42] Bert J. Dempsey, Debra Weiss, Paul Jones, Jane Greenberg: „Who is an Open Source Software Developer?“ Englisch. In: „Commun. of the ACM“ 45.2 (Feb. 2002). Association for Computing Machinery, NY, USA, S. 67–72. DOI: 10.1145/503124.503125

[43] R. Ghosh, R. Glott, B. Krieger, Gregorio Robles: „The free/libre and open source software survey and study—FLOSS final report“. Englisch. International Institute of Infonomics, University of Maastricht und Berlecon Research GmbH, Jan. 2002

[44] Sandeep Krishnamurthy: „Cave or Community?: An Empirical Examination of 100 Mature Open Source Projects“. Englisch. Bothell: University of Washington, 7. Jan. 2002, S. 8–10

[45] Andrea Bonaccorsi, Cristina Rossi-Lamastra: „Altruistic Individuals, Selfish Firms? The Structure of Motivation in Open Source Software“. Englisch. In: „First Monday, Peer Reviewed Journal on the Internet“ 9 (Dez. 2003). DOI: 10.2139/ssrn.433620

[46] Guido Hertel, Sven Niedner, Stefanie Herrmann: „Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel“. Englisch. In: „Research Policy“ 32.7 (2003), S. 1159–1177. DOI: 10.1016/S0048-7333(03)00047-7

[47] Karim Lakhani, Robert Wolf: „Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects“. Englisch. In: „Perspectives on Free and Open Source Software“ (Sep. 2003). DOI: 10.2139/ssrn.443040

[48] Krzysztof R. Apt: „From logic programming to Prolog“. Englisch. Prentice-Hall international series in computer science. 1997. ISBN: 013230368X

[49] Ivan Bratko: „Prolog programming for artificial intelligence“. Englisch. Fourth edition. Harlow, England: Pearson Addison-Wesley, 2012. ISBN: 9780321417466

[50] Lintao Zhang, Sharad Malik: „The Quest for Efficient Boolean Satisfiability Solvers“. In: „Computer Aided Verification“. Hrsg. von Kim Guldstrand Brinksma Edand Larsen. Berlin, Heidelberg: Springer, 2002, S. 17–36. ISBN: 978-3-540-45657-5

[51] Joao Marques-Silva: „Practical applications of Boolean Satisfiability“. In: „2008 9th International Workshop on Discrete Event Systems“. IEEE. 2008, S. 74–80. DOI: 10.1109/WODES.2008.4605925

[52] Yakir Vizel, Georg Weissenbacher, Sharad Malik: „Boolean Satisfiability Solvers and Their Applications in Model Checking“. In: „Proceedings of the IEEE“ 103.11 (2015), S. 2021–2035. DOI: 10.1109/JPROC.2015.2455034

[53] Thomas Eiter, Giovambattista Ianni, Thomas Krennwallner: „Answer Set Programming: A Primer“. Englisch. In: „Reasoning Web. Semantic Technologies for Information Systems: 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4“. Hrsg. von Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, Renate A. Schmidt. Springer Berlin Heidelberg, 2009, S. 40–110. DOI: 10.1007/978-3-642-03754-2_2

[54] Gerhard Brewka, Thomas Eiter, Mirosław Truszczyński: „Answer Set Programming at a Glance“. Englisch. In: „Commun. of the ACM“ 54.12 (Dez. 2011). Association for Computing Machinery, NY, USA, S. 92–103. DOI: 10.1145/2043174.2043195

[55] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, Francesco Scarcello: „The DLV system for knowledge representation and reasoning“. Englisch. In: „ACM transactions on computational logic“ 7.3 (Juli 2006). Association for Computing Machinery, NY, USA, S. 499–562. DOI: 10.1145/1149114.1149117

[56] Francesco Calimeri, Francesco Ricca: „On the Application of the Answer Set Programming System DLV in Industry: a Report from the Field“. Englisch. In: 1. Jan. 2012. DOI: 20.500.11770/180815

[57] DLVSYSTEM S.R.L.: „Company History“. Englisch. 2023. URL: https://www.dlvsystem.it/dlvsite/corporation-history/

[58] Nicola Leone, Francesco Ricca: „Answer Set Programming: A Tour from the Basics to Advanced Development Tools and Industrial Applications“. Englisch. In: Juli 2015, S. 308–326. DOI: 10.1007/978-3-319-21768-0_10

[59] Onofrio Febbraro, Giovanni Grasso, Nicola Leone, Francesco Ricca: „JASP: A Framework for Integrating Answer Set Programming with Java“. Englisch. In: „Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning“. KR'12. Rome, Italy: AAAI Press, 2012, S. 541–551. ISBN: 978-1-57735-560-1

[60] Universtity of Calabria – Department of Mathematics, Computer Science: „DLV“. Englisch. 2023. URL: https://dlv.demacs.unical.it/home

[61] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, Sven Thiele, Philipp Wanko: „Potassco: User Guide“. Englisch. Version 2.2.0. University of Potsdam, 15. Jan. 2019. URL: https://github.com/potassco/guide/releases/tag/v2.2.0

[62] Roland Kaminski, Dominik Moritz: „Clingo: A grounder and solver for logic programs“. Englisch. 7. Jan. 2018. URL: https://github.com/potassco/clingo

[63] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub: „Clingo = ASP + Control: Preliminary Report“. Englisch. 2014. DOI: 10.48550/ARXIV.1405.3694

[64] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, Philipp Wanko: „Theory Solving Made Easy with Clingo 5“. Englisch. In: „Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)“. Hrsg. von Manuel Carro, Andy King, Neda Saeedloei, Marina De Vos. Bd. 52. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 2:1–2:15. DOI: 10.4230/OASIcs.ICLP.2016.2

[65] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub: „Multi-shot ASP solving with clingo“. Englisch. 2017. DOI: 10.48550/ARXIV.1705.09811

[66] „clingcon“. Englisch. 2023. URL: https://potassco.org/clingcon/

[67] David Rajaratnam: „Clingo ORM (Clorm)“. Englisch. 8. Nov. 2021. URL: https://github.com/potassco/clorm

[68] Bjarne Stroustrup: „Evolving a Language in and for the Real World: C++ 1991-2006“. Englisch. In: „Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages“. HOPL III. San Diego, California: Association for Computing Machinery, 2007, 4–1 –4–59. DOI: 10.1145/1238844.1238848

[69] Graham M. Seed: „An Introduction to Object-Oriented Programming in C++. with Applications in Computer Graphics“. Englisch. Second Edition. Springer London, 2001. DOI: 10.1007/978-1-4471-0289-2

[70] Suraj Sharma: „Performance comparison of Java and C++ when sorting integers and writing/reading files.“ Englisch. Bachelor's thesis. Blekinge Institute of Technology, Department of Computer Science, 2019. URL: http://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Abth-18330

[71] Travis E. Oliphant: „Python for Scientific Computing“. Englisch. In: „Computing in Science & Engineering“ 9.3 (2007), S. 10–20. DOI: 10.1109/MCSE.2007.58

[72] K. Jarrod Millman, Michael Aivazis: „Python for Scientists and Engineers“. Englisch. In: „Computing in Science & Engineering“ 13.2 (2011), S. 9–12. DOI: 10.1109/MCSE.2011.36

[73] Michał Jaworski, Tarek Ziadé: „Expert Python programming : master Python by learning the best coding practices and advanced programming concepts“. Packt Publishing, 2021, S. 1–13. ISBN: 978-1-801-07110-9

[74] Muhammad Ateeq, Hina Habib, Adnan Umer, Muzammil Ul Rehman: „C++ or Python? Which One to Begin with: A Learner's Perspective“. Englisch. In: „International Conference on Teaching and Learning in Computing and Engineering“. USA: IEEE Computer Society, 2014, S. 64–69. DOI: 10.1109/LaTiCE.2014.20

[75] Brajesh De: „API Management: An Architect's Guide to Developing and Managing APIs for Your Organization“. Berkeley, CA: Apress, 2017. DOI: 10.1007/978-1-4842-1305-6_4

[76] Ulisses Spiele GmbH: „DSA Regel Wiki - Vorteil: Hass auf“. 2023. URL: https://ulisses-regelwiki.de/vorteil.html?vorteil=Hass+auf

[77] Ulisses Spiele GmbH: „DSA Regel Wiki - Vorteil: Immunität gegen (Krankheit)“. 2023. URL: https://ulisses-regelwiki.de/vorteil.html?vorteil=Immunit%C3%A4t+gegen+%28Krankheit%29

[78] Ulisses Spiele GmbH: „DSA Regel Wiki - Nachteil: Unfähig“. 2023. URL: https://ulisses-regelwiki.de/nachteil.html?nachteil=Unf%C3%A4hig

[79] Ulisses Spiele GmbH: „DSA Regel Wiki - Nachteil: Schlechte Eigenschaft“. 2023. URL: https://ulisses-regelwiki.de/nachteil.html?nachteil=Schlechte+Eigenschaft

[80] Ulisses Spiele GmbH: „DSA Regel Wiki - Kultur: Auelfen“. 2023. URL: https://ulisses-regelwiki.de/Kul_Auelfen.html

[81] Ulisses Spiele GmbH: „DSA Regel Wiki - Profession: Söldner“. 2023. URL: https://ulisses-regelwiki.de/Pro_S%C3%B6ldner.html

[82] Ulisses Spiele GmbH: „DSA Regel Wiki - Profession: Ritter“. 2023. URL: https://ulisses-regelwiki.de/Pro_Ritter.html

[83] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub: „Answer Set Solving in Practice“. Englisch. In: „Synthesis Lectures on Artificial Intelligence and Machine Learning“ 6 (Dez. 2012), S. 1–238. DOI: 10.2200/S00457ED1V01Y201211AIM019

[84] Inc. GitHub: „About READMEs“. 2023. URL: https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-readmes

[85] Mariano Anaya: „Clean Code in Python - Refactor your legacy code base“. Birmingham: Packt Publishing Ltd, 2018. ISBN: 978-1-78883-583-1

[86] Robert C. Martin: „Clean Architecture: A Craftsman's Guide to Software Structure and Design“. Robert Martin Series. Pearson Technology Group, 2017. ISBN: 978-1-78883-583-1

[87] Jeffrey Palermo: „The Onion Architecture : part 1“. Blog. 29. Juni 2008. URL: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

[88] E. Evans, E.J. Evans, M. Fowler: „Domain-driven Design: Tackling Complexity in the Heart of Software“. Addison-Wesley, 2004. ISBN: 9780321125217

[89] Martin Fowler: „Inversion of Control Containers and the Dependency Injection pattern“. In: (23. Jan. 2004). URL: https://www.martinfowler.com/articles/injection.html

[90] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides: „Design Patterns: Elements of Reusable Object-Oriented Software“. 1. Aufl. Addison-Wesley Professional, 1994. ISBN: 978-0-20163-361-0

[91] Guido van Rossum, Barry Warsaw, Nick Coghlan: „PEP 8 – Style Guide for Python Code“. 5. Juli 2001. URL: https://peps.python.org/pep-0008/

[92] „The Python Tutorial“. Version 3.11.3. Python Software Foundation, 8. Apr. 2026. URL: https://docs.python.org/3/tutorial/modules.html

[93] ISO/IEC 25010: „Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models“. März 2011

[94] ISO/IEC/IEEE 29119: „Software and systems engineering – Software testing – Part 1: General concepts“. Jan. 2022

[95] International Software Testing Qualifications Board (ISTQB): „What We Do“. 2022. URL: https://www.istqb.org/about-us/what-we-do

[96] Ham Vocke: „The Practical Test Pyramid“. 26. Feb. 2018. URL: https://martinfowler.com/articles/practical-test-pyramid.html

[97] Thomas Klein, Elena Semenova: „Tests in der Softwareentwicklung: Ein Klassifizierungsansatz“. 2021. URL: https://www.redbots.de/blog/software-tests-klassifizierung/

[98] Mike Cohn: „Succeeding with Agile: Software Development Using Scrum“. Addison-Wesley Professional, 2009. ISBN: 978-0-32157-936-2

[99] Kent C. Dodds: „Write tests. Not too many. Mostly integration.“ 13. Juni 2019. URL: https://kentcdodds.com/blog/write-tests

[100] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub: „Multi-shot ASP solving with clingo“. 2018. eprint: 1705.09811