FH Aachen
Fachbereich Elektrotechnik und Informationstechnik

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science

Interaktive Visualisierung und Layout-Persistenz von UML-Klassendiagrammen in einem CIM-RDF-Editor

Eingereicht von: Haider Abbas
Matrikelnummer: 3567272
Studiengang: Informatik

Datum: 28. März 2026

Betreuer: Prof. Dr. Andreas Hannig
Zweitprüfer: Michael Lomb (B.Sc.), SOPTIM AG

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.
Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Nachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Aachen, 28. März 2026

Haider Abbas

Gendergerechte Sprache

In dieser Arbeit wird eine geschlechtergerechte Sprache verwendet, um alle Geschlechter einzubeziehen. Dies erfolgt durch das Binnen-I (zum Beispiel „StudentInnen“) oder den Genderstern (zum Beispiel „Teilnehmer*innen“).

Zusammenfassung

Im europäischen Energiesektor tauschen Übertragungsnetzbetreiber Netzmodelle auf Basis des Common Information Model (CIM) aus, das auf dem Resource Description Framework (RDF) basiert und durch die Common Grid Model Exchange Specification (CGMES) für den standardisierten Datenaustausch spezifiziert wird. Zur Modellierung dieser Datenmodelle haben sich Klassendiagramme der Unified Modeling Language (UML) etabliert. Der von der SOPTIM AG entwickelte RDF-Architect stellt hierfür einen modernen, webbasierten Editor bereit, dessen bisherige Diagrammvisualisierung auf Mermaid.js basierte und ausschließlich statische, nicht interaktiv manipulierbare Diagramme erzeugte. Die vorliegende Arbeit adressiert diese Einschränkungen durch die Erweiterung des RDF-Architects um eine interaktive Diagrammvisualisierung mit automatischem Layouting und dauerhafter Layout-Persistenz. Zur Auswahl geeigneter Technologien wurden strukturierte Evaluationen durchgeführt, in denen Kandidaten anhand definierter Kriterien bewertet und durch Prototyping erprobt wurden. Die Evaluation der Rendering-Bibliotheken führte zur Ablösung von Mermaid.js durch SvelteFlow, das interaktive Knotenmanipulation sowie die Integration externer Layoutalgorithmen unterstützt. Als Layoutalgorithmus wurde Layered vom Eclipse Layout Kernel (ELK) ausgewählt und durch empirisch-systematisches Fine-Tuning auf UML-Klassendiagramme im CIM-Kontext abgestimmt. Die Layout-Persistenz wurde auf Basis des CGMES 3.0 DiagramLayout-Profils realisiert, das einen standardisierten Mechanismus zur Speicherung von Diagrammlayouts als RDF-Tripel bereitstellt. Die entwickelte Lösung ermöglicht es Nutzer*innen, Diagrammelemente interaktiv anzuordnen, automatisch erzeugte Ausgangslayouts als Basis für die weitere Bearbeitung zu nutzen und manuell erstellte Layouts dauerhaft zu speichern sowie wiederherzustellen. Die definierten Akzeptanzkriterien wurden erfüllt, wobei das fehlende Edge Routing als wesentliche verbleibende Einschränkung identifiziert wurde. Die dabei gewonnenen Erkenntnisse und aufgezeigten Verbesserungspotenziale bieten eine solide Grundlage für weiterführende Arbeiten im Bereich der domänenspezifischen Diagrammvisualisierung.

Abkürzungsverzeichnis

CGMES	Common Grid Model Exchange Specification	III
CIM	Common Information Model	III
DTO	Data Transfer Object	40
ELK	Eclipse Layout Kernel	III
ENTSO-E	European Network of Transmission System Operators for Electricity	2
IEC	International Electrotechnical Commission	2
IRI	Internationalized Resource Identifier	8
JSON	JavaScript Object Notation	20
JSON-LD	JavaScript Object Notation for Linked Data	8
mRID	Master Resource Identifier	16
RDF	Resource Description Framework	III
REST	Representational State Transfer	23
SPARQL	SPARQL Protocol and Query Language	8
SVG	Scalable Vector Graphics	2
UML	Unified Modeling Language	III
URI	Uniform Resource Identifier	41
UUID	Universally Unique Identifier	16
W3C	World Wide Web Consortium	7

Inhaltsverzeichnis

Abkürzungsverzeichnis IV

1 Einleitung 2
1.1 Motivation 2
1.2 Zielsetzung 3
1.3 Aufbau der Arbeit 4

2 Stand der Technik 5

3 Theorie 7
3.1 Resource Description Framework 7
3.1.1 Serialisierung 8
3.1.2 SPARQL 9
3.2 Common Information Model 10
3.2.1 ENTSO-E 10
3.2.2 CGMES 11
3.2.3 CIM Ressourcen 11
3.3 CIM-RDF-Editoren 12
3.3.1 Enterprise Architect 13
3.3.2 RDF-Architect 13
3.4 CGMES DiagramLayout-Profil 15
3.5 Rendering-Bibliotheken 18
3.5.1 Mermaid.js 19
3.5.2 Svelvet 19
3.5.3 SvelteFlow 20
3.6 Automatisches Layouting 21
3.6.1 Dagre 21
3.6.2 Eclipse Layout Kernel 21
3.6.3 elkjs 22
3.7 Technologien 22
3.7.1 Backend 23
3.7.2 Frontend 24
3.7.3 Unified Modeling Language 25
3.7.4 REST 26

4 Methodik 27
4.1 Rendering . . . 27
4.1.1 Anforderungsanalyse und Limitationen von Mermaid.js . . . 28
4.1.2 Kriterienbasierte Evaluation von Rendering-Bibliotheken . . 29
4.1.3 Prototyping und finale Technologieentscheidung . . . 30
4.2 Layouting . . . 31
4.2.1 Anforderungsanalyse und Kriterienaufstellung . . . 32
4.2.2 Evaluation und Prototyping von Layoutalgorithmen . . . 33
4.2.3 Fine-Tuning des ELK Layered Algorithmus . . . 35
4.3 Layout-Persistenz . . . 36
4.3.1 Studie des CGMES 3.0 DiagramLayout-Profils . . . 36

5 Ergebnisse 38
5.1 Implementierung . . . 38
5.1.1 Systemarchitektur im Überblick . . . 38
5.1.2 Rendering . . . 40
5.1.3 Layouting . . . 44
5.1.4 Layout-Persistenz . . . 45
5.2 Funktionsumfang und Qualität der Visualisierung . . . 50
5.2.1 Nutzerfunktionen und Erweiterbarkeit . . . 50
5.2.2 Layoutqualität . . . 51
5.2.3 Rendering- und Layouting-Performance . . . 53

6 Evaluation 55

7 Diskussion 57

8 Ausblick 58

9 Fazit 60

Abbildungsverzeichnis 61

Tabellenverzeichnis 62

Listingverzeichnis 63

Literatur 64

A Anhang 69

1 Einleitung

Diese Arbeit dokumentiert die Erweiterung des RDF-Architects um interaktive Diagrammvisualisierung, automatisches Layouting und Layout-Persistenz. Der erste Abschnitt 1.1 beschreibt den fachlichen Kontext der Arbeit und leitet daraus die zugrundeliegende Problemstellung ab. Abschnitt 1.2 formuliert auf dieser Basis die Zielsetzung der Arbeit, eine daraus abgeleitete Forschungsfrage sowie die Akzeptanzkriterien, anhand derer die Zielerreichung bewertet wird. Abschnitt 1.3 beschreibt abschließend die Struktur der Arbeit.

1.1 Motivation

Die European Network of Transmission System Operators for Electricity (ENTSO-E) vereint als Dachverband 40 Übertragungsnetzbetreiber aus 36 Ländern und koordiniert den sicheren, grenzüberschreitenden Betrieb des europäischen Stromverbundnetzes [1]. Der Betrieb dieses Netzes setzt einen standardisierten, maschinenlesbaren Datenaustausch zwischen den beteiligten Akteuren voraus.

Für diesen Zweck wird das CIM eingesetzt, das seit 1996 vom Technical Committee 57 der International Electrotechnical Commission (IEC) betreut wird. CIM-Modelle werden dabei als RDF-Graphen strukturiert und zwischen Systemen ausgetauscht. Aufbauend auf CIM entwickelte die ENTSO-E die CGMES, welche die spezifischen Anforderungen europäischer Übertragungsnetzbetreiber adressiert und seit ihrer ersten Veröffentlichung im Dezember 2013 in mehreren Versionen parallel im Einsatz ist [2].

Als zentrales Werkzeug zur Modellierung dieser Datenmodelle haben sich Klassendiagramme der UML etabliert. Ende 2008 adoptierte das IEC TC57 den Enterprise Architect von Sparx Systems offiziell als UML-Tool zur Pflege des CIM-Standards [3]. In der Praxis ist dessen Einsatz jedoch aufwendig, da die Abhängigkeit von spezifischen Plugins zu einem fragilen Workflow mit eingeschränkter Wartbarkeit führt. Um diese Limitierungen zu überwinden sowie neue Funktionalitäten zu ermöglichen, entwickelt die SOPTIM AG seit Ende 2024 den RDF-Architect, einen modernen, webbasierten Editor für CIM-konforme Datenmodelle.

Die Visualisierung der UML-Klassendiagramme im RDF-Architect erfolgt über die JavaScript-Bibliothek Mermaid.js. Diese arbeitet auf Basis einer textuellen Diagrammbeschreibung, die als statisches Scalable Vector Graphics (SVG)-Bild

gerendert wird. Aus diesem Ansatz ergeben sich mehrere Einschränkungen. Eine manuelle Anpassung des Layouts ist aufgrund der SVG-basierten Ausgabe nicht möglich, da einzelne Diagrammelemente nicht interaktiv manipulierbar sind. Darüber hinaus ist das Rendering größerer Modelle zeitintensiv und die Erweiterbarkeit der Bibliothek begrenzt. Für die Arbeit mit komplexen CIM-Modellen stellen diese Einschränkungen ein Hindernis dar.

Die vorliegende Arbeit entsteht im Auftrag der SOPTIM AG und adressiert diese Problematik durch die Ablösung von Mermaid.js zugunsten einer interaktiven Rendering-Bibliothek. Ergänzend wurde ein automatischer Layoutalgorithmus integriert sowie die Möglichkeit geschaffen, manuell vorgenommene Layoutanpassungen persistent zu speichern.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, die Diagrammvisualisierung des RDF-Architects zu erweitern, sodass Interaktivität unterstützt und das Rendering performanter gestaltet wird. Aufbauend auf dieser Visualisierung wird ein automatischer Layoutalgorithmus integriert, der Nutzer*innen qualitativ hochwertige Ausgangslayouts als Basis für die weitere Bearbeitung bereitstellt. Darüber hinaus wird die Möglichkeit geschaffen, manuelle Layouts zu erstellen, welche persistent gespeichert und nachgeladen werden können. Diese drei Aspekte bilden zusammen ein kohärentes Konzept zur Verbesserung der Diagrammbearbeitung im RDF-Architect.

Aus der Zielsetzung ergibt sich folgende Forschungsfrage: „**Wie kann die Visualisierung von UML-Klassendiagrammen in einem CIM-RDF-Editor so erweitert werden, dass interaktive Bearbeitung, automatisches Layouting und manuell erstellte, persistente Diagrammlayouts möglich sind, ohne dabei die Qualität der Diagrammdarstellung zu beeinträchtigen?**“

Zur Beantwortung dieser Frage werden folgende Akzeptanzkriterien herangezogen, die sich aus den in der Forschungsfrage identifizierten Teilbereichen ableiten.

A1 Diagrammelemente können von Nutzer*innen interaktiv verschoben und angeordnet werden

A2 Manuell erstellte Layouts bleiben nach dem Speichern und erneutem Laden erhalten

A3 Der automatische Layoutalgorithmus erzeugt für CIM-Modelle übersichtliche, nachvollziehbare Ausgangslayouts

A4 Das Rendering von Diagrammen ist im Vergleich zur Mermaid.js- Implementierung performanter

Diese Kriterien werden abschließend in Kapitel 6 evaluiert, wobei sowohl die technische Umsetzung als auch die praktische Anwendbarkeit der entwickelten Lösung bewertet werden.

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in neun Kapitel. Kapitel 2 beleuchtet verwandte wissenschaftliche Arbeiten, die sich mit vergleichbaren Problemstellungen befassen, und ordnet die vorliegende Arbeit in diesen wissenschaftlichen Kontext ein. Kapitel 3 legt die theoretischen und technischen Grundlagen dar, die zum Verständnis der nachfolgenden Kapitel erforderlich sind. Kapitel 4 stellt das strukturierte methodische Vorgehen dar, das der Erweiterung des RDF-Architects zugrunde liegt, und beschreibt, wie technische Anforderungen analysiert, geeignete Technologien evaluiert und Implementierungsentscheidungen auf Basis dieser Erkenntnisse getroffen wurden. Kapitel 5 stellt die entstandene Implementierung vor und beleuchtet die daraus resultierenden Ergebnisse aus technischer sowie funktionaler Perspektive. Kapitel 6 bewertet die Ergebnisse anhand der in der Zielsetzung definierten Akzeptanzkriterien und prüft, inwieweit die gesetzten Ziele erreicht wurden. Kapitel 7 reflektiert das methodische Vorgehen kritisch und untersucht, ob die gewählten Methoden geeignet waren, die Forschungsfrage adäquat zu beantworten, und wo alternative Ansätze in Betracht gezogen werden könnten. Kapitel 8 identifiziert auf Basis der gewonnenen Erkenntnisse Potenziale für weiterführende Entwicklungen und offene Fragestellungen. Kapitel 9 schließt die Arbeit mit einem zusammenfassenden Fazit ab.

2 Stand der Technik

Die Erweiterung eines bestehenden Werkzeugs um interaktive Diagrammvisualisierung, automatisches Layouting und persistente Speicherung von Diagrammlayouts berührt mehrere Teilbereiche der Forschung. Konkret werden im Folgenden drei wissenschaftliche Arbeiten betrachtet: zwei Arbeiten zur interaktiven UML-Modellierung sowie eine Arbeit zur webbasierten Visualisierung von Ontologien, die dem in dieser Arbeit betrachteten CIM-RDF-Datenmodell konzeptionell nahesteht.

BIGUML von Metin und Bork [4] sowie *HyLiMo* von Krieger et al. [5] realisieren beide interaktive UML-Klassendiagramme als IDE-Extension, verfolgen dabei jedoch unterschiedliche Architekturkonzepte. BIGUML setzt das Graphical Language Server Platform (GLSP) der Eclipse Foundation ein, eine Erweiterung des Language Server Protocol (LSP) für grafische Editoren. GLSP folgt einem Client-Server-Prinzip, bei dem ein sprachspezifischer Server die Modellverarbeitung übernimmt und ein sprachagnostischer Client das Rendering realisiert. HyLiMo verfolgt einen DSL-basierten Ansatz, bei dem Diagramme in einer domänenspezifischen Sprache textuell beschrieben und in Echtzeit mit einem grafischen Editor synchronisiert werden. Dabei werden Layout- und Stilinformationen direkt in der textuellen Repräsentation gespeichert.

Beide Arbeiten sind für die Einordnung der vorliegenden Arbeit relevant, da sie zeigen, wie interaktive UML-Editoren mit expliziter Berücksichtigung von Layoutanforderungen konzipiert werden können. Für den Kontext eines CIM-RDF-Editors erweisen sich beide Ansätze jedoch als nicht übertragbar. Der GLSP-Ansatz setzt einen sprachspezifischen Server voraus, der in einem Editor mit RDF-basiertem Persistenzmodell eine konzeptionell fremde Schicht ohne Entsprechung im bestehenden System einführen würde. Der DSL-basierte Ansatz von HyLiMo widerspricht der Anforderung, Layoutdaten als integralen Bestandteil des RDF-Modells gemäß CGMES 3.0 DiagramLayout-Profil zu persistieren, da eine parallele Datenhaltung zu Inkonsistenzen führen und die Rolle des RDF-Modells als zentrale Datenquelle unterlaufen würde.

Im Bereich der Visualisierung von Ontologien stellt *WebVOWL* von Lohmann et al. einen weiteren relevanten Ansatz dar [6]. Die Web Ontology Language (OWL) ist ein auf RDF aufbauender Standard zur formalen Beschreibung von Ontologien und steht damit dem CIM-RDF-Datenmodell konzeptionell nahe. WebVOWL arbeitet vollständig client-seitig und überführt Ontologien zur Laufzeit im Browser in eine interaktive SVG-Visualisierung. Das Layout basiert dabei auf einem physikalischen Force-Directed-Algorithmus, den die JavaScript-Bibliothek D3.js realisiert. Für die

vorliegende Arbeit ist WebVOWL als dokumentierter Ansatz zur Visualisierung RDF-verwandter Strukturen relevant. Der Force-Directed-Algorithmus ist jedoch für UML-Klassendiagramme nicht geeignet, da er organische, bei jedem Ladevorgang neu berechnete Layouts erzeugt, während Klassendiagramme ein hierarchisch strukturiertes und reproduzierbares Layout erfordern.

Die betrachteten Arbeiten belegen, dass interaktive Diagrammbearbeitung, automatisches Layouting und standardbasierte Layout-Persistenz im RDF-Modell bislang als voneinander getrennte Problemfelder behandelt wurden. Keine der untersuchten Arbeiten vereint alle drei Aspekte in einem Werkzeug. Zur spezifischen Visualisierung von CIM-UML-Schemata in RDF-basierten Editoren konnte darüber hinaus keine einschlägige wissenschaftliche Publikation identifiziert werden. Die vorliegende Arbeit schließt damit eine Lücke, die sich aus der Kombination dieser drei Anforderungen im Kontext der Energiedomäne und der dort eingesetzten CGMES-Normen ergibt.

3 Theorie

Das Verständnis der in dieser Arbeit entwickelten Erweiterungen setzt Kenntnisse verschiedener technischer Grundlagen und eingesetzter Technologien voraus. Dieses Kapitel legt diese Grundlagen dar und gibt einen Überblick über die relevanten Standards, Werkzeuge und Bibliotheken.

Abschnitt 3.1 führt RDF als Standard zur Darstellung und Verarbeitung graphbasierter Daten ein und legt damit die Grundlage für die weiteren behandelten Konzepte. Abschnitt 3.2 beschreibt das CIM als etablierten Standard zur Modellierung von Netzdaten in der Energiewirtschaft. Abschnitt 3.3 gibt einen Überblick über die im CIM-Kontext eingesetzten Werkzeuge zur UML-Modellierung. Da das CGMES DiagramLayout-Profil den standardisierten Mechanismus zur Beschreibung von Diagrammlayouts in RDF darstellt, wird es in Abschnitt 3.4 gesondert behandelt. Abschnitt 3.5 stellt die relevanten Bibliotheken für das Rendering von Diagrammen vor. Abschnitt 3.6 behandelt die Bibliotheken und Algorithmen für das automatische Layouting. Abschnitt 3.7 beschreibt abschließend die eingesetzten Backend- und Frontend-Technologien.

3.1 Resource Description Framework

Das Resource Description Framework (RDF) ist ein vom World Wide Web Consortium (W3C) spezifizierter Standard zum Austausch und zur Repräsentation von Daten im Web. Es stellt heute einen zentralen Baustein des Semantic Webs dar [7]. RDF erlaubt es, beliebige Sachverhalte über Ressourcen aus unterschiedlichen Domänen formal zu beschreiben, weshalb es als Grundlage für standardisierte Datenformate wie das CIM dient [8].

Die Einheit zur Darstellung von Informationen über Ressourcen in RDF ist das Tripel, das aus drei Komponenten besteht: einem Subjekt, einem Prädikat und einem Objekt. Das Subjekt bezeichnet die Ressource, über die im Statement eine Aussage getroffen wird. Das Prädikat benennt die Eigenschaft dieser Ressource, und das Objekt weist dieser Eigenschaft einen Wert zu [9]. Eine Menge solcher Tripel bildet einen gerichteten Graphen, in dem Subjekte und Objekte als Knoten und Prädikate als gerichtete Kanten zwischen diesen fungieren. Das folgende Beispiel veranschaulicht diese Prinzipien: Das Tripel `cim:ACLineSegment rdfs:label "Leitungssegment"` beschreibt, dass die Ressource cim:ACLineSegment die Bezeichnung „Leitungssegment“ trägt.

Subjekte und Prädikate werden in RDF stets als Internationalized Resource Identifier (IRI) angegeben, die Ressourcen global eindeutig identifizieren. Eine IRI setzt sich dabei aus einem Präfix, genannt Namespace, der den Geltungsbereich der Ressource festlegt, und einem Namen, der die Ressource innerhalb dieses Namensraums eindeutig benennt. Die vollständige IRI für die CIM-Klasse DiagramObject lautet beispielsweise http://iec.ch/TC57/CIM100#DiagramObject, wobei http://iec.ch/TC57/CIM100# den Namespace und *DiagramObject* den Namen darstellen [10]. Da solche vollständigen IRIs in der Praxis lang ausfallen und deren wiederholtes Ausschreiben aufwendig ist, werden Abkürzungen für Präfixe eingesetzt, um häufig verwendete Namespaces abzukürzen. Nach der Deklaration @prefix cim: <http://iec.ch/TC57/CIM100#> lässt sich die obige IRI kompakt als cim:DiagramObject schreiben [11].

Objekte eines Tripels unterliegen im Vergleich zu Subjekten und Prädikaten weniger Einschränkungen hinsichtlich ihrer Form. Sie können entweder als IRI vorliegen, womit das Objekt auf eine weitere Ressource verweist und so eine Verbindung zwischen zwei Ressourcen herstellt, oder als Literal, das einen konkreten Datenwert wie etwa eine Zeichenkette, eine Zahl oder einen booleschen Wert repräsentiert. So verweist das Objekt im Tripel ex:PKW cims:belongsToCategory ex:Package_Fahrzeuge als IRI auf eine andere Ressource, während das Objekt in ex:PKW rdfs:label "PKW"@de als Literal eine Zeichenkette mit Sprach-Tag trägt [12].

Abschnitt 3.1.1 stellt die gängigen Serialisierungsformate für RDF-Daten vor, mit besonderem Fokus auf das Turtle-Format. Abschnitt 3.1.2 führt die SPARQL Protocol and Query Language (SPARQL) als Anfragesprache für RDF-Graphen ein.

3.1.1 Serialisierung

RDF-Daten lassen sich in verschiedenen Formaten speichern und übertragen. Verbreitet ist dabei RDF/XML, das insbesondere in der Energiewirtschaft für den Austausch von CIM-Daten eingesetzt wird. Darüber hinaus existieren weitere Formate wie N-Triples oder JavaScript Object Notation for Linked Data (JSON-LD). Im Fokus steht stattdessen Turtle (Terse RDF Triple Language), das unter den gängigen Formaten die kompakteste und für Menschen lesbarste Darstellung bietet und daher in dieser Arbeit durchgehend verwendet wird [13].

Im Turtle-Format werden RDF-Daten als eine Folge von Tripeln strukturiert, wobei jedes Tripel mit einem Punkt abgeschlossen wird. Teilen mehrere Tripel dasselbe Subjekt, kann dieses Subjekt nur einmal genannt werden und die weiteren Tripel lassen sich durch Semikolons teilen, sodass das Subjekt nicht wiederholt werden

muss. Literale lassen sich in Anführungsstrichen angeben.

Listing 3.1 zeigt ein einfaches Turtle-Beispiel, das diese Konventionen veranschaulicht.

```
ex:PKW
    rdf:type rdfs:Class ;
    rdfs:label "PKW"@de ;
    rdfs:subClassOf ex:Fahrzeug ;
    rdfs:comment "Repraesentiert einen Personenkraftwagen."@de ;
    cims:belongsToCategory ex:Package_Fahrzeuge .
```

Listing 3.1: Beispiel einer RDF-Ressource in Turtle-Syntax

3.1.2 SPARQL

SPARQL Protocol and Query Language (SPARQL) ist eine vom W3C standardisierte deklarative Anfragesprache für RDF-Graphen [14]. Ähnlich wie die Structured Query Language (SQL) für relationale Datenbanken ermöglicht sie das gezielte Abfragen und Manipulieren von RDF-Daten.

Das zentrale Konzept von SPARQL ist das Graph Pattern Matching [15]. Anstatt exakte Werte abzufragen, werden Anfragen als Muster formuliert, die Variablen als Platzhalter enthalten. Dadurch wird der RDF-Graph nach allen Teilgraphen durchsucht, die dem angegebenen Muster entsprechen, und die Variablen werden mit den gefundenen Werten belegt. SPARQL unterscheidet verschiedene Anfragetypen, darunter SELECT zur Rückgabe von Variablenbelegungen, CONSTRUCT zur Erzeugung neuer Tripel sowie INSERT und DELETE zur Manipulation von RDF-Daten. Eine SELECT-Anfrage besteht dabei aus einer SELECT-Klausel, die die zurückzugebenden Variablen benennt, und einer WHERE-Klausel, die das zu matchende Muster definiert [16].

Listing 3.2 zeigt ein entsprechendes Beispiel einer SELECT-Anfrage.

```
PREFIX ex:<http://example.org/>

SELECT ?pkw ?modell ?land
WHERE {
  ?pkw ex:hatModell ?modell ;
    ex:hergestelltVon ?hersteller .
  ?hersteller ex:stammtAus ?land .
}
```

Listing 3.2: Beispiel einer SPARQL SELECT-Anfrage

Die Anfrage gibt alle PKW samt Modellbezeichnung und Herkunftsland des jeweiligen Herstellers zurück. Die Variable `?hersteller` fungiert dabei als Zwischenwert, der im ersten Muster gebunden und im zweiten weiterverwendet wird, um das Herkunftsland zu ermitteln. Auf diese Weise lassen sich auch komplexe Zusammenhänge über mehrere Ressourcen hinweg in einer einzigen Anfrage ausdrücken.

3.2 Common Information Model

Das Common Information Model (CIM) ist ein von der IEC entwickelter und gepflegter Standard zur einheitlichen Modellierung von Stromnetzen und deren Betriebsmitteln [17]. Zweck des Standards ist es, einen herstellerunabhängigen Datenaustausch zwischen verschiedenen Systemen der Energiewirtschaft zu ermöglichen.

Als technische Grundlage setzt CIM auf RDF und definiert darauf aufbauend eine Ontologie, also ein domänenspezifisches Vokabular, für die Energiewirtschaft. Dieses legt fest, wie Klassen, Packages, Attribute, Assoziationen und Enumerationen innerhalb des Modells strukturiert und miteinander verknüpft werden.

Abschnitt 3.2.1 stellt die ENTSO-E als treibende Organisation hinter dem europäischen Datenaustauschstandard vor. Abschnitt 3.2.2 beschreibt das auf CIM aufbauende CGMES-Austauschformat. Abschnitt 3.2.3 erläutert die für die Modellierung relevanten CIM-Ressourcentypen.

3.2.1 ENTSO-E

Das European Network of Transmission System Operators for Electricity (ENTSO-E) wurde 2009 gegründet und bildet, wie in Abschnitt 1.1 dargestellt, mit 40 Mitgliedsorganisationen aus 36 Ländern den zentralen Koordinationsrahmen für den europäischen Stromübertragungssektor [1]. Der Verband verfolgt das Ziel, durch enge Kooperation der nationalen Netzbetreiber sowohl die Versorgungssicherheit als auch den Ausbau erneuerbarer Energiequellen im europäischen Verbundnetz zu fördern.

Ein Bestandteil dieser Kooperation ist der maschinenlesbare Austausch von Netzmodellen zwischen den beteiligten Organisationen. Hierfür hat ENTSO-E auf Grundlage des CIM die CGMES spezifiziert, die im folgenden Abschnitt 3.2.2 vorgestellt wird.

3.2.2 CGMES

Die Common Grid Model Exchange Specification (CGMES) ist ein Austauschformat für elektrische Netzmodelle, das von der ENTSO-E entwickelt und gepflegt wird. Es basiert auf dem CIM und wurde geschaffen, um den spezifischen Anforderungen des europäischen Energiemarkts an einen standardisierten Datenaustausch zwischen den Übertragungsnetzbetreibern gerecht zu werden [2].

Während CIM die grundlegenden Modellierungskonzepte wie Klassen, Attribute und Assoziationen definiert, gruppiert CGMES darauf aufbauend konkrete Austauschprofile. Jedes Profil deckt dabei einen bestimmten fachlichen Aspekt eines Netzmodells ab und legt fest, welche CIM-Klassen und -Eigenschaften für den jeweiligen Anwendungsfall relevant sind. So enthält beispielsweise das Profil „Equipment“ (EQ) die Klassen zur Beschreibung physischer Netzkomponenten, während das Profil „DiagramLayout“ (DL) die Positionierung von Elementen in einer Diagrammdarstellung spezifiziert. CGMES existiert in mehreren Versionen, wobei in der Praxis vorwiegend die bewährte Version CGMES 2.4.15 sowie die neuere Version CGMES 3.0 zum Einsatz kommen.

3.2.3 CIM Ressourcen

CIM beschreibt Komponenten von Energieversorgungsnetzen als strukturierte RDF-Ressourcen. Für die einzelnen Ressourcentypen legt der Standard jeweils eine spezifische Syntax und Semantik fest. Zu diesen Typen zählen unter anderem Packages, Klassen, Attribute, Assoziationen sowie Enum-Entries. Da UML-Klassendiagramme die strukturelle Organisation von Klassen innerhalb von Packages abbilden, sind für die Visualisierung CIM-konformer Datenmodelle vorrangig diese beiden Typen relevant.

Zur Erweiterung des RDF-Standardvokabulars um CIM-spezifische Prädikate wird der Namespace „CIM Schema“ verwendet, der mit dem Präfix `cims:` abgekürzt wird. Dieser Namespace stellt Prädikate wie `cims:stereotype`, `cims:belongsToCategory` oder `cims:multiplicity` bereit, die über das Standardvokabular hinausgehen und die präzise Modellierung energiewirtschaftlicher Konzepte ermöglichen.

CIM-Klassen

Klassen bilden die zentrale Modellierungseinheit in CIM. Sie repräsentieren konkrete oder abstrakte Konzepte und werden in einem RDF-Graphen durch das Tripel `rdf:type rdfs:Class` als solche ausgezeichnet. Einem Package werden

Klassen über das Prädikat `cims:belongsToCategory` zugeordnet, während Vererbungsbeziehungen zwischen Klassen durch `rdfs:subClassOf` ausgedrückt werden. Außerdem lassen sich bei dieser und allen anderen Ressourcen Kommentare über das Prädikat `rdfs:comment` verfassen.

Ein wesentliches Konzept bei CIM-Klassen ist der Stereotyp. Über das Prädikat `cims:stereotype` können einer Klasse semantische Typinformationen zugewiesen werden, die unter anderem bestimmen, ob eine Klasse instanziierbar ist.

Das folgende Listing 3.3 zeigt die vollständige Definition einer CIM-Klasse im Turtle-Format.

```
ex:PKW
    rdf:type rdfs:Class ;
    rdfs:label "PKW"@de ;
    rdfs:subClassOf ex:Fahrzeug ;
    rdfs:comment "Repraesentiert einen Personenkraftwagen."@de ;
    cims:belongsToCategory ex:Package_Fahrzeuge ;
    cims:stereotype http://iec.ch/TC57/NonStandard/UML#concrete .
```

Listing 3.3: Beispiel einer CIM-Klasse

CIM-Packages

Packages dienen der thematischen Gruppierung von CIM-Klassen innerhalb eines CIM-Modells. Sie werden durch das Tripel `rdf:type cims:ClassCategory` identifiziert und tragen ein `rdfs:label`.

Das folgende Listing 3.4 zeigt die Definition eines beispielhaften Packages.

```
ex:Package_Fahrzeuge
    rdf:type cims:ClassCategory ;
    rdfs:label "Fahrzeuge"@de .
```

Listing 3.4: Beispiel eines CIM-Packages

3.3 CIM-RDF-Editoren

Die Modellierung CIM-konformer RDF-Schemata, also der strukturellen Beschreibung eines Datenmodells, als UML-Klassendiagramme setzt spezialisierte Anwendungen voraus, die sowohl die Struktur des CIM-Standards als auch die Anforderungen der Energiewirtschaft abbilden können. Als etabliertes Werkzeug hat sich dabei

der Enterprise Architect von Sparx Systems durchgesetzt. Mit dem Ziel, dessen Limitierungen zu überwinden und neue Funktionalitäten zu ermöglichen, entwickelt die SOPTIM AG seit Ende 2024 den RDF-Architect als modernen, webbasierten Editor für CIM-konforme RDF-Schemata.

Abschnitt 3.3.1 stellt den Enterprise Architect vor. Abschnitt 3.3.2 beschreibt den RDF-Architect als Ausgangssystem dieser Arbeit.

3.3.1 Enterprise Architect

Der Enterprise Architect ist ein kommerzielles UML-Modellierungswerkzeug von Sparx Systems [18]. Als allgemeines Modellierungswerkzeug deckt er eine Vielzahl von Modellierungssprachen und Anwendungsdomänen ab. Für domänenspezifische Anwendungsfälle lässt er sich über Plugin-ähnliche Erweiterungspakete um benutzerdefinierte UML-Profile und Werkzeugpaletten ergänzen. Für die CIM-Modellierung wird der Enterprise Architect auf diese Weise um die Verwaltung CIM-konformer RDF-Schemata erweitert und stellt diese als UML-Klassendiagramme dar.

Die Kernfunktionalitäten des Werkzeugs für die UML-Modellierung umfassen eine paketbasierte Navigation durch das Modell, die grafische Darstellung von Klassen mit Attributen und Assoziationen sowie die Visualisierung von Vererbungsbeziehungen als Pfeile zwischen den beteiligten Klassen. Klassen und deren Eigenschaften lassen sich direkt im Diagramm bearbeiten, und manuell angepasste Layouts werden persistent im Modell gespeichert.

3.3.2 RDF-Architect

Der RDF-Architect ist eine von der SOPTIM AG entwickelte Web-Anwendung zur Modellierung und Bearbeitung CIM-konformer RDF-Schemata. Die Anwendung stellt die grundlegenden Funktionalitäten eines UML-Editors bereit und ist dabei gezielt auf den Einsatz im Kontext von CGMES-konformen Schemata ausgelegt.

Wie in Abschnitt 1.1 dargestellt, entstand der RDF-Architect Ende 2024 als Reaktion auf die Limitierungen des bislang eingesetzten Enterprise Architect von Sparx Systems und mit dem Ziel, darüber hinaus neue Funktionalitäten zu ermöglichen, die mit dem Enterprise Architect nicht realisierbar waren.

Abbildung 3.1 zeigt die Hauptansicht des RDF-Architects mit einem geöffneten Diagramm.

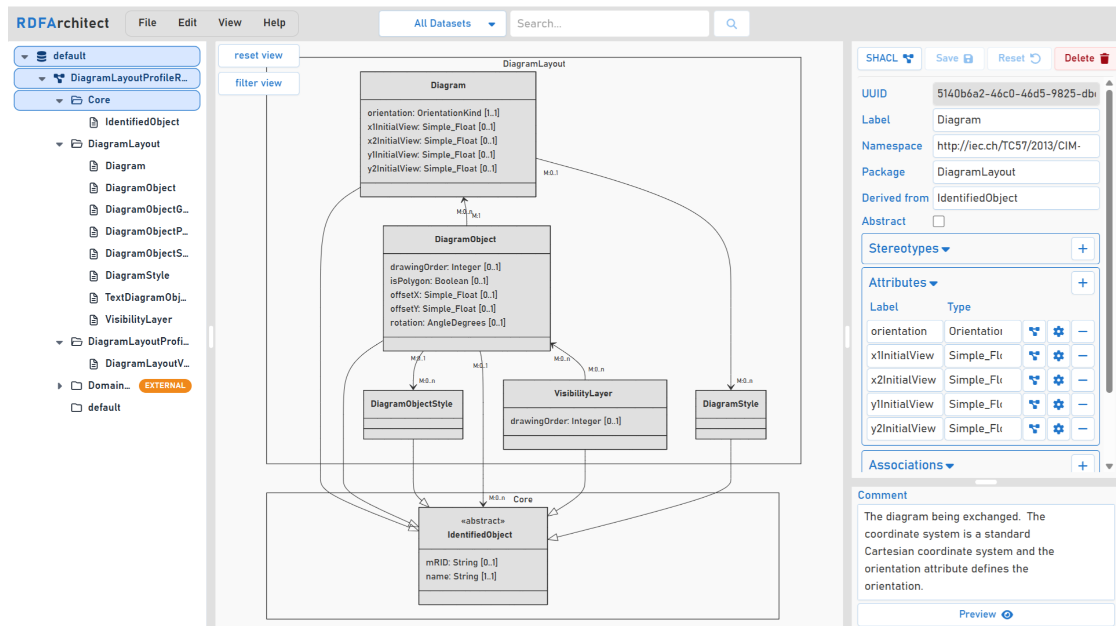

Abbildung 3.1: Hauptansicht des RDF-Architects

Die Oberfläche gliedert sich in drei Bereiche: links einen Navigationsbereich zur hierarchischen Durchsicht der importierten Schemata, eine zentrale Diagrammansicht, die das ausgewählte Package als UML-Klassendiagramm darstellt, sowie rechts den eingeblendeten Class Editor zur Bearbeitung von Klassen und deren Eigenschaften. Über ein Menü in der oberen Leiste lassen sich RDF-Graphen aus Dateien verschiedener Formate, wie CIM/XML und Turtle, importieren sowie in denselben Formaten exportieren, wobei ein Graph jeweils eine einzelne RDF-Datei repräsentiert und ein Dataset eine Menge solcher Graphen zusammenfasst. Darüber hinaus bietet die Anwendung weitere Funktionalitäten, darunter die Verwaltung von Namespaces pro Datensatz und ein Change-Log-Management pro Graph.

Das Backend ist in Java mit Spring Boot und Apache Jena implementiert, das Frontend basiert auf SvelteKit mit Svelte 5 und Tailwind CSS für das Styling. Die Kommunikation zwischen beiden Schichten erfolgt über eine RESTful API. Das Backend folgt dabei dem Prinzip der hexagonalen Architektur. Dieses Prinzip isoliert die Kernlogik der Anwendung von konkreten technischen Implementierungen, indem abstrakte Schnittstellen, sogenannte Ports, von ihren konkreten Implementierungen, den Adaptern, getrennt werden. Über Konfigurationen lässt sich dabei festlegen, welche Adapter aktiv sind, wodurch Implementierungen austauschbar werden.

3.4 CGMES DiagramLayout-Profil

Wie in Abschnitt 3.2.2 beschrieben, gliedert CGMES den Datenaustausch zwischen Übertragungsnetzbetreibern in thematisch abgegrenzte Profile. Das DiagramLayout-Profil definiert dabei, auf welche Weise die grafische Darstellung von Netzdiagrammen, also die Positionierung und Anordnung von Elementen innerhalb eines Diagramms, in einem standardisierten Format gespeichert und zwischen Systemen ausgetauscht werden kann. Das Profil ist als Teil der Norm IEC 61970-600-2 spezifiziert und in der CGMES 3.0 Application Profiles Library der ENTSO-E veröffentlicht [19]. Die nachfolgende Beschreibung stützt sich dabei maßgeblich auf Würgler und McMoran [20], die das DiagramLayout-Profil in seinem konzeptionellen Ursprung dokumentieren, und erläutert zunächst das Profil auf konzeptioneller Ebene, bevor die zentralen Klassen und deren Attribute im Anschluss im Detail beschrieben werden.

Das DiagramLayout-Profil beschreibt Netzdiagramme über eine Hierarchie miteinander verknüpfter Klassen. Ein `Diagram` bildet den übergeordneten Container aller grafischen Elemente, während jedes dargestellte Element durch ein `DiagramObject` repräsentiert wird, das optional auf eine CIM-Ressource verweist. Die Position eines `DiagramObject` wird durch einen oder mehrere `DiagramObjectPoints` im Koordinatensystem festgelegt. Darüber hinaus sieht das Profil weitere Klassen vor, etwa `DiagramObjectStyle` zur Beschreibung von Darstellungsstilen, `VisibilityLayer` zur Gruppierung von Elementen in Sichtbarkeitsebenen sowie `TextDiagramObject` für textuelle Beschriftungen.

Abbildung 3.2 zeigt das vollständige Klassendiagramm des DiagramLayout-Profils, dargestellt über die im Rahmen dieser Arbeit entwickelte Visualisierungsimplementierung.

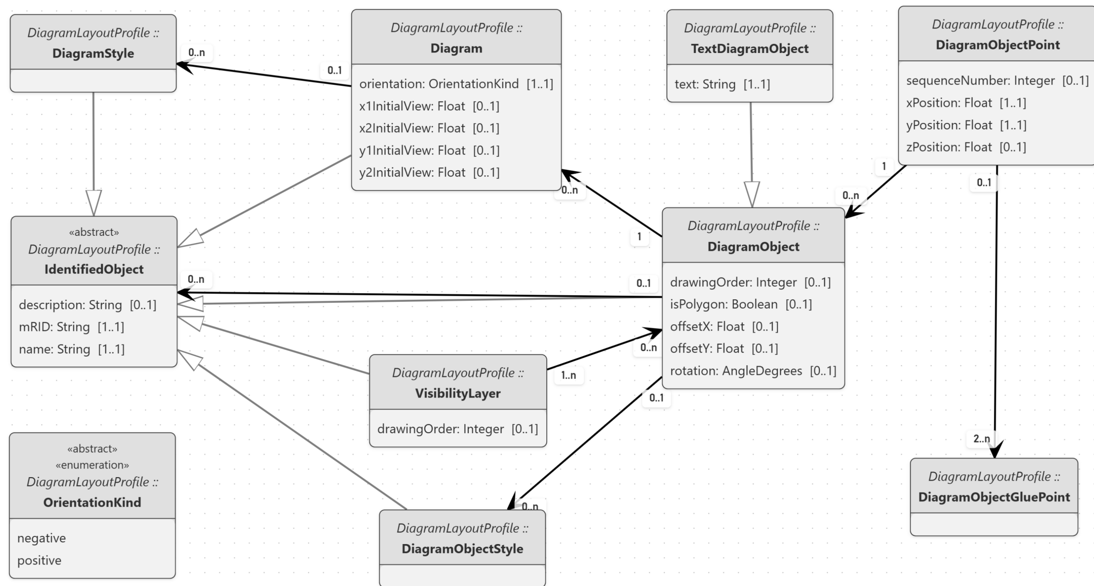

Abbildung 3.2: Klassendiagramm des CGMES DiagramLayout-Profils

Alle zentralen Klassen des Profils erben von `IdentifiedObject` aus dem Core-Paket des CIM, das eine einheitliche Identifikation jeder Ressource sicherstellt, wobei `DiagramObject` die zentrale Rolle einnimmt. Es verknüpft das übergeordnete `Diagram`, die zugehörigen Positionspunkte und das referenzierte Domänenobjekt miteinander. Da die Kernfunktionalität des Profils in der Beschreibung von Diagrammen, deren grafischen Elementen und deren Positionierung im Koordinatensystem liegt, beschränken sich die folgenden Unterabschnitte auf die Klassen `IdentifiedObject`, `Diagram`, `DiagramObject` und `DiagramObjectPoint`. Weitere Klassen wie `VisibilityLayer` oder `TextDiagramObject` werden nicht weiter behandelt. Die in den nachfolgenden Listings verwendeten IRIs entsprechen gekürzten UUIDs, wobei jeweils nur die ersten acht Zeichen dargestellt sind.

IdentifiedObject

`IdentifiedObject` ist eine abstrakte Basisklasse aus dem Core-Paket des CIM und bildet die gemeinsame Grundlage für alle identifizierbaren Ressourcen im DiagramLayout-Profil. Die Klasse definiert vier Attribute: `mRID` (Master Resource Identifier) dient als eindeutige Kennung der Ressource, wobei die Verwendung von Universally Unique Identifier (UUID) in der Praxis empfohlen wird. Das Attribut `name` stellt eine menschenlesbare Bezeichnung bereit und `description` bietet Raum für eine optionale Freitextbeschreibung.

Diagram

Die Klasse `Diagram` repräsentiert ein einzelnes Netzdiagramm und fungiert als Container für alle zugehörigen grafischen Elemente. Sie erbt von `IdentifiedObject` und wird über die Assoziation `DiagramElements` mit beliebig vielen `DiagramObject`-Instanzen verknüpft. Zur Positionierung arbeitet `Diagram` mit einem kartesischen Koordinatensystem, dessen Orientierung über das Attribut `orientation` vom Typ `OrientationKind` festgelegt wird. Bei negativer Orientierung nehmen Y-Werte von oben nach unten zu, bei positiver von unten nach oben. Über die optionalen Attribute `x1InitialView`, `x2InitialView`, `y1InitialView` und `y2InitialView` lässt sich der beim Öffnen initial sichtbare Ausschnitt des Koordinatensystems festlegen.

Listing 3.5 zeigt eine minimale Definition eines `Diagram` im Turtle-Format, die nur die Pflichtattribute sowie die Orientierung enthält.

```
<_3e4587c4-...>
    rdf:type cim:Diagram ;
    cim:Diagram.orientation cim:OrientationKind.negative ;
    cim:IdentifiedObject.name "Core" .
```

Listing 3.5: Beispiel eines Diagram im CGMES DiagramLayout-Profil

Das Beispiel definiert ein `Diagram` mit dem Namen `Core`, das ein negatives Koordinatensystem verwendet.

DiagramObject

`DiagramObject` ist die zentrale Klasse des Profils und repräsentiert ein einzelnes grafisches Element innerhalb eines `Diagram`, unabhängig davon, ob es sich um einen Knoten oder eine Kante handelt. Jedes `DiagramObject` ist genau einem `Diagram` zugeordnet und kann optional über `IdentifiedObject` auf eine CIM-Domänenressource verweisen. Die Positionierung wird durch mindestens einen verknüpften `DiagramObjectPoint` definiert. Die weiteren Attribute `drawingOrder`, `offsetX`, `offsetY`, `rotation` und `isPolygon` steuern Darstellungsaspekte wie Zeichenreihenfolge, Versatz, Rotation und Polygoninterpretation.

Listing 3.6 zeigt eine minimale Definition eines `DiagramObject` im Turtle-Format.

```
<_d6655524-...>
    rdf:type cim:DiagramObject ;
    cim:DiagramObject.Diagram <_3e4587c4-...> ;
    cim:DiagramObject.IdentifiedObject <_41e71047-...> ;
    cim:IdentifiedObject.name "IdentifiedObject" .
```

Listing 3.6: Beispiel eines DiagramObject im CGMES DiagramLayout-Profil

Das Beispiel zeigt ein `DiagramObject` mit dem Namen `IdentifiedObject`, das dem `Diagram` aus Listing 3.5 zugeordnet ist und die gleichnamige CIM-Klasse aus dem geladenen Modell referenziert.

DiagramObjectPoint

`DiagramObjectPoint` repräsentiert einen einzelnen Punkt im Koordinatensystem und ist einem `DiagramObject` zugeordnet. Die Position wird durch `xPosition`, `yPosition` und `zPosition` als Gleitkommazahlen angegeben. Besitzt ein `DiagramObject` mehrere Punkte, legt `sequenceNumber` die Verbindungsreihenfolge fest. Optional kann ein Punkt einem `DiagramObjectGluePoint` zugeordnet werden, der grafische Verbindungen zwischen Punkten verschiedener `DiagramObject`-Instanzen explizit ausdrückt.

Listing 3.7 zeigt eine minimale Definition eines `DiagramObjectPoint` im Turtle-Format.

```
<_310f78a1-...>
    rdf:type cim:DiagramObjectPoint ;
    cim:DiagramObjectPoint.DiagramObject <_d6655524-...> ;
    cim:DiagramObjectPoint.xPosition "25.0" ;
    cim:DiagramObjectPoint.yPosition "-80.0" .
```

Listing 3.7: Beispiel eines DiagramObjectPoint im CGMES DiagramLayout-Profil

Das Beispiel definiert einen Punkt an Position (25.0, -80.0), der dem `DiagramObject` aus Listing 3.6 zugeordnet ist und damit die Position der `IdentifiedObject`-Klasse im Diagramm festlegt.

3.5 Rendering-Bibliotheken

Im RDF-Architect erfolgt die Darstellung von UML-Klassendiagrammen im Frontend über JavaScript-Bibliotheken, die das Rendering von Diagrammen überneh-

men.

Der vorliegende Abschnitt stellt drei solcher Bibliotheken vor, die im Kontext dieser Arbeit von Relevanz sind: Mermaid.js in Abschnitt 3.5.1, Svelvet in Abschnitt 3.5.2 sowie SvelteFlow in Abschnitt 3.5.3.

3.5.1 Mermaid.js

Mermaid.js ist eine 2014 von Knut Sveidqvist entwickelte Open-Source-JavaScript-Bibliothek, die eine eigene textbasierte Diagrammsprache definiert und rendert [21]. Die Syntax dieser Sprache orientiert sich an Markdown und ermöglicht die Definition verschiedener Diagrammtypen, wobei jeder Typ über ein eigenes Schlüsselwort eingeleitet wird. Für UML-Klassendiagramme etwa wird die Syntax mit `classDiagram` eingeleitet, worauf die Definition von Klassen, Attributen und Relationen folgt. Mermaid.js findet breite Anwendung in der Softwaredokumentation und wird von Plattformen wie GitHub und GitLab nativ in Markdown-Dateien unterstützt [21].

Die Bibliothek nimmt einen solchen Mermaid-String entgegen, berechnet auf dessen Basis automatisch das Layout und rendert das fertige Diagramm als eine statische Abbildung in SVG-Format. Für die Layoutberechnung unterstützt Mermaid.js mehrere Layoutmethoden, wobei standardmäßig `dagre-d3-es` eingesetzt wird, eine Implementierung des in Abschnitt 3.6.1 beschriebenen Dagre-Algorithmus in moderner ES6-Modulsyntax. Dagre übernimmt dabei die eigentliche Positionsberechnung und liefert Knotenpositionen sowie Kantenverlaufspunkte zurück. Anschließend durchläuft das Ergebnis mehrere Nachverarbeitungsschritte, die das Diagramm visuell verfeinern, ohne die berechneten Layoutdaten zu verändern. So werden etwa Kanten an den Knotenrändern abgeschnitten, da Dagre Kanten standardmäßig vom Mittelpunkt zum Mittelpunkt eines Knotens berechnet. Darüber hinaus werden Kantenkurven anhand der gelieferten Verlaufspunkte geglättet. Da Layout und Positionierung vollständig von der Bibliothek übernommen werden, haben Nutzer*innen keinen direkten Einfluss auf die Anordnung der Elemente im gerenderten Diagramm.

3.5.2 Svelvet

Svelvet ist eine von Open Source Labs entwickelte JavaScript-Bibliothek speziell für das Svelte-Framework, die die Erstellung interaktiver, knotenbasierter Diagramme auf einem Canvas ermöglicht [22]. Die Bibliothek stellt eine Reihe von Svelte-Komponenten bereit, über die Diagramme deklarativ aufgebaut werden.

Die zentrale Komponente ist der `Svelvet`-Wrapper, der das interaktive Canvas darstellt und grundlegende Verhaltensweisen wie Zoom und Panning bereitstellt. Innerhalb dieses Wrappers werden `Node`-Komponenten platziert, die einzelne Elemente im Diagramm repräsentieren und frei positioniert sowie verschoben werden können. Verbindungen zwischen Nodes werden über `Edge`-Komponenten realisiert. Sowohl Nodes als auch Edges sind dabei vollständig anpassbar, da eigene Svelte-Komponenten als Inhalt übergeben werden können.

3.5.3 SvelteFlow

SvelteFlow ist eine von XYFlow entwickelte Open-Source-JavaScript-Bibliothek für das Svelte-Framework, die die Erstellung interaktiver, knotenbasierter Diagramme ermöglicht [23]. Die Bibliothek nimmt Graphen in Form von Nodes und Edges entgegen und stellt diese in einem interaktiven, zoombaren Canvas dar.

Das Datenmodell der Bibliothek basiert auf den zwei zentralen Konzepten Nodes und Edges, die als JavaScript Object Notation (JSON) übergeben werden. Eine Node erfordert mindestens eine eindeutige `id`, eine `position` mit X- und Y-Koordinaten sowie ein `data`-Objekt, über das beliebige Daten an die zugehörige Darstellungskomponente weitergereicht werden. Eine Edge erfordert eine `id`, eine `source`- sowie eine `target`-ID, die jeweils auf eine Node verweisen, und nimmt ebenfalls ein optionales `data`-Objekt entgegen. Über das optionale `type`-Attribut wird festgelegt, welche Komponente zur Darstellung genutzt wird. SvelteFlow stellt hierfür eingebaute Standardtypen bereit, darüber hinaus lassen sich vollständig eigene Svelte-Komponenten als Custom-Typen registrieren und über dieses Attribut referenzieren. Die Verbindungspunkte, an denen Edges an eine Node ansetzen, werden als Handles bezeichnet. Handles lassen sich einer Node direkt oder über eine Custom-Komponente zuweisen und als Quell-, Ziel- oder kombinierter Quell- und Zielpunkt einer Edge konfigurieren. Neben diesen Kernkonzepten bietet die Bibliothek eine Vielzahl weiterer Komponenten und Konfigurationsmöglichkeiten.

Die zentrale Komponente der Bibliothek ist `SvelteFlowWrapper`, die das interaktive Canvas darstellt sowie Nodes, Edges und deren Custom-Typen entgegennimmt. Über Props lassen sich Event-Handler wie `on:nodeclick` registrieren und das Verhalten des Canvas konfigurieren. Darüber hinaus stellt die Bibliothek Hooks wie `useSvelteFlow()` bereit, über die programmatisch auf den internen Zustand des Canvas zugegriffen werden kann.

3.6 Automatisches Layouting

Das automatische Layouting bezeichnet die algorithmische Berechnung von Layoutinformationen der Elemente eines Graphen. Dabei werden Graphen in Form von Nodes und Edges als Input entgegengenommen, woraufhin der jeweilige Algorithmus geometrische Eigenschaften wie Positionen, Dimensionen und Verläufe der Elemente berechnet und zurückgibt. Ein wesentliches Merkmal dieser Ansätze ist die Unabhängigkeit vom Rendering, da die Layoutberechnung losgelöst von der konkreten Darstellungstechnologie erfolgt und die berechneten Informationen anschließend von beliebigen Rendering-Bibliotheken genutzt werden können.

Die folgenden Abschnitte stellen die für das automatische Layouting von UML-Klassendiagrammen relevanten Werkzeuge vor. Abschnitt 3.6.1 beschreibt Dagre als JavaScript-Bibliothek für hierarchisches Layouting. Abschnitt 3.6.2 stellt den ELK sowie den zentralen Algorithmus ELK Layered vor. Abschnitt 3.6.3 führt elkjs als JavaScript-Portierung von ELK ein.

3.6.1 Dagre

Dagre ist eine JavaScript-Bibliothek für das automatische Layouting gerichteter Graphen [24] und findet unter anderem in Mermaid.js Einsatz. Der zugrundeliegende Algorithmus arbeitet nach einem hierarchischen Prinzip, bei dem Nodes in Schichten angeordnet werden und Kanten möglichst in eine einheitliche Richtung verlaufen. Dagre ist dabei auf Geschwindigkeit ausgelegt und eignet sich daher insbesondere für Anwendungsfälle, in denen Layouts zur Laufzeit berechnet werden.

Das Verhalten des Algorithmus lässt sich über eine Reihe von Parametern konfigurieren. Beispielsweise lässt sich über `rankdir` die Richtung festlegen, in der Schichten angeordnet werden, etwa von oben nach unten oder von links nach rechts. Die Parameter `nodesep` und `ranksep` steuern den horizontalen Abstand zwischen Nodes innerhalb einer Schicht beziehungsweise den vertikalen Abstand zwischen den Schichten selbst.

3.6.2 Eclipse Layout Kernel

Der ELK ist ein Open-Source-Framework der Eclipse Foundation für das automatische Layouting von Graphen [25]. Das Framework stellt eine Sammlung von Layoutalgorithmen sowie eine Infrastruktur bereit, die diese Algorithmen mit Diagrammeditoren und -viewern verbindet.

ELK organisiert seine Konfiguration um drei zentrale Konzepte: Layoutalgorithmen, Layout-Optionen und Option-Gruppen. Layoutalgorithmen bilden den Kern des Frameworks und decken unterschiedliche Layoutansätze ab, darunter schichtbasierte Layouts, kraft- und stressbasierte Layouts, baumartige Layouts sowie Layouts für unverbundene Teilgraphen. Jeder Algorithmus lässt sich über eine Menge von Layout-Optionen konfigurieren, wobei nicht jede Option für jeden Algorithmus verfügbar ist. Option-Gruppen fassen thematisch zusammengehörige Optionen zusammen und dienen der besseren Übersichtlichkeit der Konfiguration.

ELK Layered

ELK Layered ist der zentrale Algorithmus des ELK und basiert auf dem Sugiyama-Framework, einem etablierten Ansatz für das hierarchische Layouting gerichteter Graphen [26]. Der Algorithmus verfügt über mehr als 150 Layout-Optionen, über die sein Verhalten detailliert konfiguriert werden kann.

Die Verarbeitung eines Graphen erfolgt in fünf aufeinanderfolgenden Phasen: Cycle Breaking, Layer Assignment, Crossing Minimization, Node Placement und Edge Routing [27]. Das Grundprinzip besteht darin, Nodes in horizontalen Schichten anzuordnen, wobei Kanten möglichst einheitlich in eine Richtung verlaufen und Kreuzungen zwischen ihnen minimiert werden. Jede Phase lässt sich über eigene Layout-Optionen konfigurieren.

3.6.3 elkjs

elkjs ist eine vom KIELER Research Project der Universität zu Kiel entwickelte Bibliothek, die die Layoutalgorithmen von ELK für JavaScript-Umgebungen zugänglich macht [28]. Dabei bildet elkjs nicht den vollständigen Funktionsumfang von ELK ab, sodass nicht alle Algorithmen und Optionen unterstützt werden.

3.7 Technologien

Dieser Abschnitt gibt einen Überblick über die technologischen Grundlagen, auf denen der RDF-Architect aufbaut und die im Rahmen dieser Arbeit maßgeblich zum Einsatz kommen. Das Verständnis der eingesetzten Technologien bildet dabei die Voraussetzung für die in späteren Kapiteln beschriebenen Erweiterungen.

Abschnitt 3.7.1 beschreibt die serverseitigen Technologien des RDF-Architects. Abschnitt 3.7.2 behandelt die clientseitigen Technologien. Abschnitt 3.7.3 führt die für die Diagrammdarstellung relevanten Grundlagen der UML ein. Abschnitt 3.7.4 beschreibt den Architekturstil Representational State Transfer (REST) als Grundlage der Kommunikation zwischen Frontend und Backend.

3.7.1 Backend

Das Backend des RDF-Architects bildet die serverseitige Grundlage der Anwendung und ist für die Verarbeitung sowie Bereitstellung der RDF-Daten verantwortlich. Die folgenden Abschnitte beschreiben die dabei eingesetzten Technologien Spring Boot und Apache Jena, eine Java-Bibliothek zur Arbeit mit RDF-Daten über eine dedizierte API.

Spring Boot

Spring Boot ist ein von VMware entwickeltes Java-Framework für die Erstellung von Enterprise-Anwendungen [29]. Es baut auf dem Spring Framework auf [30] und erweitert dieses um automatische Konfiguration, sodass Spring Boot zur Startzeit die vorhandenen Abhängigkeiten erkennt und eigenständig konfiguriert, anstelle manueller XML-Konfiguration. Abhängigkeiten zwischen Komponenten werden dabei über das Prinzip der Dependency Injection verwaltet. Die beteiligten Objekte, im Spring-Kontext als Beans bezeichnet, werden nicht manuell instanziiert, sondern vom Framework automatisch erstellt und injiziert.

Die Struktur einer Spring-Boot-Anwendung wird durch Annotationen definiert. `@RestController` kennzeichnet eine Klasse als Controller, der HTTP-Endpunkte definiert, und `@RequestMapping` legt die zugehörige URL-Route fest. `@Service` kennzeichnet eine Klasse als Service, der die Geschäftslogik kapselt. Über `@Configuration` lassen sich Konfigurationsklassen definieren, die Spring Boot bei der Initialisierung auswertet. Eingehende Anfragen sowie ausgehende Antworten werden vom Framework hinsichtlich Fehlerbehandlung automatisch verarbeitet.

Die JSON-Serialisierung erfolgt über die Jackson-Bibliothek, die Spring Boot standardmäßig einbindet. Von Endpunkten zurückgegebene Java-Objekte werden dadurch automatisch in JSON umgewandelt, ohne dass manueller Konvertierungscode erforderlich ist.

Apache Jena RDF API

Apache Jena ist eine von der Apache Software Foundation entwickelte Open-Source-Java-Bibliothek zur Verarbeitung von RDF-Daten [31]. Die Bibliothek stellt Schnittstellen zum Lesen, Schreiben und Manipulieren von RDF-Graphen bereit und unterstützt gängige Serialisierungsformate wie Turtle und RDF/XML.

Die `Model`-Klasse bildet in Apache Jena die zentrale Schnittstelle für den Zugriff auf RDF-Graphen und stellt eine API zum Hinzufügen, Entfernen und Traversieren von Tripeln bereit. Darüber hinaus ermöglicht Apache Jena das Ausführen von SPARQL-Queries direkt auf einem `Model`, wodurch RDF-Daten strukturiert abgefragt werden können.

3.7.2 Frontend

Das Frontend des RDF-Architects bildet die clientseitige Grundlage der Anwendung und ist für die Darstellung sowie die Interaktion mit den RDF-Daten verantwortlich. Als zugrunde liegendes Framework kommt Svelte zum Einsatz, das durch SvelteKit um die für vollständige Webanwendungen notwendigen Funktionalitäten erweitert wird.

Svelte

Svelte ist ein Open-Source-JavaScript-Framework zur Erstellung reaktiver Benutzeroberflächen [32]. Im Gegensatz zu Frameworks wie React oder Vue verfolgt Svelte einen compilergestützten Ansatz. Anstatt zur Laufzeit ein virtuelles DOM zu verwalten, übersetzt der Compiler Svelte-Komponenten bereits während des Build-Prozesses in optimiertes JavaScript, das das DOM direkt manipuliert. Im RDF-Architect wird die im Oktober 2024 veröffentlichte Version 5 eingesetzt, die das Reaktivitätsmodell des Frameworks grundlegend überarbeitete. Während in früheren Versionen reaktive Variablen über die `$:`-Syntax deklariert wurden, führte Svelte 5 funktionsartige Makros, genannt „Runes“, ein, die dem Compiler explizite Informationen über reaktive Zustände liefern.

Kern des Reaktivitätsmodells in Svelte ist die `$state`-Rune, über die reaktive Zustandsvariablen deklariert werden. Ändert sich der Wert einer solchen Variable, rendert Svelte die betroffenen UI-Elemente automatisch neu. Für Variablen, deren Wert sich aus anderen reaktiven Variablen ableitet, steht die `$derived`-Rune bereit. Die Weitergabe von Daten zwischen Komponenten erfolgt über Props, die mittels `$props` deklariert werden. Props repräsentieren Werte, die von einer

Elternkomponente an eine Kindkomponente übergeben werden und dort reaktiv zur Verfügung stehen.

SvelteKit ist das offizielle Anwendungsframework für Svelte [33] und erweitert es um dateibasiertes Routing, Server-Side Rendering sowie API-Endpunkte. In Kombination mit Svelte ermöglicht SvelteKit die Entwicklung moderner Single-Page Applications.

3.7.3 Unified Modeling Language

Die Unified Modeling Language (UML) ist eine standardisierte Modellierungssprache zur grafischen Darstellung von Software- und Systemstrukturen. Sie umfasst verschiedene Diagrammtypen für unterschiedliche Modellierungszwecke [34]. Für die Modellierung von CIM-Schemata haben sich dabei UML-Klassendiagramme als geeignetes Darstellungsmittel etabliert, da sie die statische Struktur eines Systems abbilden und Klassen, ihre Eigenschaften sowie die Beziehungen zwischen ihnen darstellen [35].

Abbildung 3.3 zeigt ein beispielhaftes UML-Klassendiagramm, das die für Klassendiagramme zentralen Konzepte veranschaulicht.

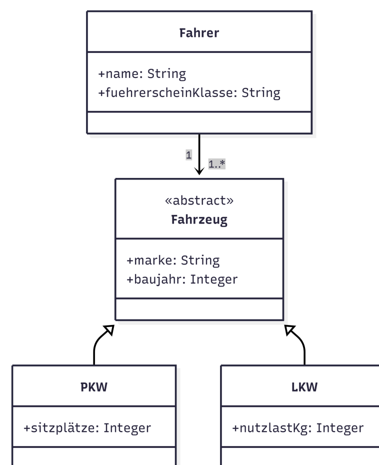

Abbildung 3.3: Beispielhaftes UML-Klassendiagramm

Eine Klasse wird als benannter Kasten dargestellt und kann eine Menge von Attributen besitzen, denen jeweils ein Datentyp zugewiesen ist. So verfügt die Klasse *Fahrzeug* in Abbildung 3.3 über die Attribute *marke* vom Typ *String* sowie

baujahr vom Typ *Integer*. Darüber hinaus können Klassen über Stereotypen verfügen, die ihre semantische Bedeutung kennzeichnen. *Fahrzeug* trägt den Stereotyp <<abstract>> und kann daher nicht direkt instanziiert werden. Instanziierbar sind ausschließlich die konkreten Unterklassen *PKW* und *LKW*. Beziehungen zwischen Klassen werden durch zwei Pfeiltypen ausgedrückt: Vererbungspfeile kennzeichnen eine Generalisierungsbeziehung, bei der eine Unterklasse die Eigenschaften ihrer Oberklasse übernimmt. In Abbildung 3.3 erben *PKW* und *LKW* folglich die Attribute der Klasse *Fahrzeug*. Assoziationspfeile hingegen beschreiben eine Beziehung zwischen zwei eigenständigen Klassen und werden mit Multiplizitätsangaben versehen, die die erlaubte Anzahl beteiligter Instanzen definieren. Die Assoziation zwischen *Fahrer* und *Fahrzeug* gibt an, dass ein Fahrer mindestens ein und beliebig viele Fahrzeuge führen kann.

3.7.4 REST

Representational State Transfer (REST) ist ein Architekturstil für die Gestaltung von Web-APIs, der auf dem HTTP-Protokoll basiert [36]. REST-Schnittstellen verwenden die standardisierten HTTP-Methoden GET, POST, PUT und DELETE und strukturieren den Zugriff auf Ressourcen über eindeutige URLs. Die Kommunikation erfolgt zustandslos, sodass jede Anfrage alle notwendigen Informationen enthält und der Server keinen Sitzungszustand speichert. Als Datenformat wird dabei üblicherweise JSON eingesetzt. Im RDF-Architect bildet eine REST-Schnittstelle die Kommunikationsgrundlage zwischen dem Svelte-Frontend und dem Spring-Boot-Backend.

4 Methodik

Die geplanten Erweiterungen des RDF-Architects erforderten ein strukturiertes methodisches Vorgehen, das technische Entscheidungen nachvollziehbar begründet und an der Forschungsfrage ausrichtet. Dieses Kapitel legt dieses Vorgehen dar und schafft damit die konzeptionelle Grundlage für die in Kapitel 5 beschriebene Implementierung. Die getroffenen Entscheidungen, sowie die dabei aufgetretenen Schwierigkeiten werden in Kapitel 7 kritisch reflektiert.

Die Entwicklung gliederte sich in drei aufeinander aufbauende Phasen: die Anpassung des Renderings an die Anforderungen interaktiver UML-Klassendiagramme, die Einführung eines automatischen Layoutalgorithmus, sowie die Realisierung der Layout-Persistenz, also die dauerhafte Speicherung und Wiederherstellung von Diagrammlayouts. In den ersten beiden Phasen kamen jeweils Anforderungsanalyse, kriterienbasierte Evaluation und Prototyping als Methoden zum Einsatz. Die dritte Phase erforderte ein abweichendes Vorgehen, da mit dem CGMES 3.0 DiagramLayout-Profil ein etablierter Standard existiert, dessen gezielte Analyse den konzeptionellen Ausgangspunkt bildete. Die Frage der Performance begleitete die Entwicklung dabei phasenübergreifend. Konkrete Messergebnisse werden in Abschnitt 5.2.3 vorgestellt.

Abschnitt 4.1 beschreibt das Vorgehen bei der Auswahl und Einführung einer geeigneten Rendering-Lösung. Abschnitt 4.2 behandelt die Auswahl und Konfiguration eines Layoutalgorithmus. Abschnitt 4.3 widmet sich dem methodischen Vorgehen bei der Realisierung der Layout-Persistenz.

4.1 Rendering

Die Anpassung des Renderings bildete die erste Entwicklungsphase. Ausgangspunkt war die bestehende Mermaid.js-Implementierung, die hinsichtlich Interaktivität und Erweiterbarkeit Einschränkungen aufwies. Ziel dieser Phase war die Auswahl und Einführung einer geeigneteren Rendering-Lösung.

Abschnitt 4.1.1 analysiert die Anforderungen an das Rendering und zeigt auf, warum die bestehende Lösung diese nicht erfüllt. Abschnitt 4.1.2 beschreibt die darauf aufbauende Evaluation von Kandidaten. Abschnitt 4.1.3 dokumentiert das Prototyping, die finale Technologieentscheidung, sowie das konzeptionelle Vorgehen bei der Implementierung.

4.1.1 Anforderungsanalyse und Limitationen von Mermaid.js

Die Erweiterung des RDF-Architects um eine interaktive Diagrammdarstellung setzt voraus, dass die eingesetzte Rendering-Lösung bestimmte Anforderungen und Auswahlkriterien erfüllt. Um diese zu konkretisieren, wurde zunächst Mermaid.js, dessen Grundlagen in Abschnitt 3.5.1 beschrieben sind, hinsichtlich seiner Eignung für die Zielsetzung dieser Arbeit analysiert.

Die Analyse von Mermaid.js ergab, dass die Bibliothek die gestellten Anforderungen in mehreren zentralen Punkten nicht erfüllt, die im Folgenden aufgeführt sind.

- **Statisches Rendering** Mermaid.js rendert Diagramme als statisches SVG-Bild, wodurch eine interaktive Manipulation einzelner Diagrammelemente grundsätzlich ausgeschlossen ist.
- **Fehlende Knoteninteraktion** Für Klassendiagramme sind keine Interaktionsmöglichkeiten über einfache Klick-Ereignisse hinaus vorgesehen, sodass das Verschieben von Knoten nicht unterstützt wird.
- **Keine manuelle Positionierung** Die Positionierung von Elementen wird ausschließlich durch interne Algorithmen bestimmt, ohne dass eine manuelle Einflussnahme möglich ist.
- **Performance** Bei der Darstellung größerer CIM-Modelle traten messbare Verzögerungen im Rendering auf, die die praktische Nutzbarkeit beeinträchtigen. Eine systematische Quantifizierung dieser Dauern findet sich in Abschnitt 5.2.3.

Eine Erweiterung von Mermaid.js wurde zwar in Betracht gezogen, erwies sich jedoch als architektonisch nicht zielführend, da die Bibliothek grundlegend auf die Erzeugung statischer Diagramme ausgelegt ist und die erforderliche Interaktivität nicht nachträglich sinnvoll integriert werden kann. Daher war ein Wechsel der eingesetzten Rendering-Lösung erforderlich.

Aus der Analyse der Limitationen sowie den in Abschnitt 1.2 formulierten Zielen wurden Anforderungen an eine alternative Rendering-Bibliothek abgeleitet und in Bewertungskriterien überführt. Diese orientieren sich an den Interaktions- und Integrationsanforderungen des RDF-Architects sowie an den Bedürfnissen der Nutzer*innen, die Diagramme flexibel bearbeiten und anordnen können sollen. Da der RDF-Architect als Open-Source-Projekt entwickelt wird, wurden zudem ausschließlich Bibliotheken mit kompatiblen Open-Source-Lizenzen berücksichtigt.

B1 **Manuelle Positionierung** Die Bibliothek muss das interaktive Verschieben und Anordnen von Knoten durch Nutzer*innen unterstützen, sodass

benutzerdefinierte Diagrammlayouts erstellt und gespeichert werden können.

B2 **Automatisches Layouting** Die Bibliothek muss die Integration von Layoutalgorithmen unterstützen, die Knoten- und Kantenpositionen automatisch berechnen und ein kollisionsfreies Ausgangslayout erzeugen.

B3 **Interaktivität und Anpassbarkeit** Benutzerinteraktionen wie Zoomen, Verschieben des Ansichtsbereichs und Selektieren von Elementen müssen konfigurierbar sein, sodass das Verhalten der Diagrammkomponenten gezielt an die Anforderungen des RDF-Architects angepasst werden kann.

B4 **Darstellung von UML-Klassendiagrammen** Die Bibliothek muss entweder native Unterstützung für UML-Klassendiagramme bieten oder die Erstellung benutzerdefinierter Knoten- und Kantenkomponenten ermöglichen, die UML-Klassendiagramme abbilden können.

B5 **Performance** Das Rendering muss bei größeren CIM-Modellen geringere Dauern aufweisen als die bestehende Mermaid.js-Implementierung.

B6 **Dokumentationsqualität** Die Bibliothek muss über eine hinreichende Dokumentation verfügen, die eine effiziente Einarbeitung und nachhaltige Wartbarkeit gewährleistet.

B7 **Open-Source-Lizenz** Die Lizenz der Bibliothek muss mit dem Open-Source-Charakter des RDF-Architects kompatibel sein.

Die definierten Kriterien bildeten die Grundlage für die im folgenden Abschnitt durchgeführte Evaluation.

4.1.2 Kriterienbasierte Evaluation von Rendering-Bibliotheken

Auf Basis der in Abschnitt 4.1.1 definierten Kriterien wurden verfügbare Rendering-Bibliotheken systematisch evaluiert. Zur Identifikation geeigneter Kandidaten wurden mehrere community-gepflegte Übersichten über JavaScript-basierte Diagrammbibliotheken herangezogen, darunter ein zusammengestellter Vergleich verfügbarer Lösungen sowie eine kuratierte Liste node-basierter UI-Bibliotheken [37] [38]. Die dort aufgeführten Kandidaten wurden anhand der in Abschnitt 4.1.1 definierten Kriterien vorbewertet.

Ein Teil der Bibliotheken schied dabei frühzeitig aus. Kommerzielle Lösungen wie GoJS oder yFiles wurden aufgrund ihrer Lizenzmodelle nicht weiter berücksichtigt. Bibliotheken wie nomnoml verfolgen denselben textbasierten Ansatz wie Mermaid.js und erfüllten damit das Kriterium der manuellen Positionierung nicht.

Die Bibliothek cytoscape.js ist auf Graphlayouts ausgelegt und für die Darstellung von UML-Klassendiagrammen nicht konzipiert.

Für eine eingehendere Betrachtung verblieben JointJS, MaxGraph, D3.js sowie Svelvet. JointJS [39] steht in einer kostenlosen Community-Version unter der Mozilla Public License Version 2.0 zur Verfügung und bietet Interaktivität sowie die Möglichkeit, benutzerdefinierte Shapes zu erstellen. Unklar blieb jedoch, welche Funktionalitäten ausschließlich der kommerziellen Version vorbehalten sind, was eine verlässliche Einschätzung der tatsächlich verfügbaren Möglichkeiten erschwerte. maxGraph [40], der aktiv weiterentwickelte Nachfolger der in draw.io eingesetzten mxGraph-Bibliothek, bringt integrierte Layoutalgorithmen und umfangreiche Interaktionsmöglichkeiten mit, ist jedoch framework-agnostisch und hätte eine zusätzliche Integrationsschicht in die Svelte-Architektur des RDF-Architects erfordert. D3.js [41] bietet mächtige Werkzeuge zur Datenvisualisierung und unterstützt Interaktivität, ist jedoch primär auf allgemeine Visualisierungsaufgaben ausgerichtet, weshalb für die Darstellung von UML-Klassendiagrammen ein erhöhter Eigenentwicklungsaufwand zu erwarten gewesen wäre.

Svelvet [22] sticht in diesem Vergleich durch seine native Svelte-Kompatibilität hervor. Da der RDF-Architect auf dem Svelte-Framework aufbaut, lässt sich Svelvet direkt in die bestehende Architektur integrieren, ohne zusätzlichen Adaptierungsaufwand. Die Bibliothek erfüllt die Kriterien der manuellen Positionierung, der Interaktivität sowie der Darstellung benutzerdefinierter UML-Klassendiagramme, da Knoten frei verschiebbar sind und eigene Svelte-Komponenten als Knoten- und Kanteninhalte eingesetzt werden können. Auf Basis dieser Vorbetrachtung erschien Svelvet als vielversprechendster Kandidat und wurde für ein erstes Prototyping ausgewählt.

4.1.3 Prototyping und finale Technologieentscheidung

Aus der kriterienbasierten Evaluation in Abschnitt 4.1.2 ging Svelvet als vielversprechendster Kandidat hervor und bildete daher den Ausgangspunkt für ein erstes Prototyping. Dabei zeigte sich, dass die Bibliothek die grundlegenden Anforderungen an Interaktivität und manuelle Positionierung (B1, B3) prinzipiell erfüllt. Im Verlauf des Prototypings traten jedoch Einschränkungen zutage, die eine vollständige Umsetzung der Anforderungen nur über nicht-triviale Workarounds ermöglicht hätten. So stand etwa kein nativer Callback-Mechanismus für Knotenbewegungen zur Verfügung, und bestimmte Standardinteraktionen ließen sich nicht ohne weiteres über die Bibliotheks-API deaktivieren, was insbesondere Kriterium B3 betraf. Diese Beobachtungen deuteten darauf hin, dass die Umsetzung mit Svelvet

einen erhöhten Entwicklungsaufwand erfordert hätte, was die Suche nach einer geeigneteren Lösung motivierte.

Parallel dazu wurde SvelteFlow, das in Abschnitt 3.5.3 näher beschrieben wird, als alternative Bibliothek in Betracht gezogen. Da SvelteFlow ebenfalls als Svelte-Bibliothek konzipiert ist, fügt es sich wie Svelvet direkt in die bestehende Architektur des RDF-Architects ein. Im Vergleich zu Svelvet handelt es sich um eine in der Community deutlich breiter etablierte Lösung mit einem größeren Entwicklerkreis und aktivem Wartungszyklus. Ein erster Blick auf den Funktionsumfang zeigte zudem nativen Support für die Integration von Layoutalgorithmen (Kriterium B2) sowie eine feingranularere Konfigurierbarkeit von Interaktionen (Kriterium B3). Da SvelteFlow in mehreren relevanten Punkten vielversprechender erschien, wurde ein zweiter Prototyp entwickelt, um die Eignung der Bibliothek in der Praxis zu erproben.

Die Bibliothek erfüllt die definierten Kriterien ohne aufwändige Workarounds, wird aktiv weiterentwickelt und bietet eine umfangreiche sowie stabil dokumentierte API. Svelvet hingegen hätte für mehrere der definierten Anforderungen, insbesondere Kriterien B1 und B3, individuelle Lösungen außerhalb der vorgesehenen API erfordert, was den Entwicklungsaufwand erhöht und die langfristige Wartbarkeit der Implementierung erschwert hätte. Der direkte Vergleich beider Prototypen führte daher zur finalen Entscheidung für SvelteFlow als Rendering-Bibliothek des RDF-Architects.

Die anschließende Implementierung orientierte sich konzeptionell an etablierten Werkzeugen der UML-Modellierung. Als Referenz diente dabei insbesondere der Enterprise Architect, der, wie in Abschnitt 1.1 beschrieben, als etabliertes Werkzeug zur Pflege des CIM-Standards eingesetzt wird und damit vergleichbare Anforderungen an die Darstellung und Interaktion mit UML-Klassendiagrammen adressiert. Darüber hinaus wurden die im RDF-Architect vor der Einführung von Mermaid.js vorhandenen Diagrammfunktionalitäten berücksichtigt, um eine inhaltliche Kontinuität zu gewährleisten. Auf dieser Grundlage wurden die Struktur der UML-Klassenknoten, das Verhalten von Kanten sowie die Interaktionsmöglichkeiten für Nutzer*innen konzipiert. Die konkrete Umsetzung wird in Abschnitt 5.1.2 detailliert beschrieben.

4.2 Layouting

Nachdem SvelteFlow als Rendering-Bibliothek eingeführt und die Grundlage für eine interaktive Diagrammdarstellung geschaffen worden war, widmete sich die zweite

Entwicklungsphase der Frage, wie Nutzer*innen ein sinnvolles Ausgangslayout für ihre Diagramme erhalten, ohne Knoten manuell positionieren zu müssen. Dazu wurde ein automatischer Layoutalgorithmus gesucht, der für UML-Klassendiagramme geeignete Ausgangslayouts erzeugt und dabei mit den Strukturen großer CIM-Modelle umgehen kann.

Abschnitt 4.2.1 analysiert die Anforderungen an einen solchen Algorithmus und leitet daraus Bewertungskriterien ab. Abschnitt 4.2.2 beschreibt die Evaluation infrage kommender Algorithmen und die dabei gewonnenen Erkenntnisse. Abschnitt 4.2.3 dokumentiert schließlich das methodische Vorgehen bei der Konfiguration des gewählten Algorithmus für die spezifischen Anforderungen des RDF-Architects.

4.2.1 Anforderungsanalyse und Kriterienaufstellung

Mit der Einführung von SvelteFlow entfiel das in Mermaid.js integrierte Layouting, das, wie in Abschnitt 3.5.1 beschrieben, auf dem Dagre-Algorithmus basiert. Für größere CIM-Modelle hatte sich dabei gezeigt, dass das erzeugte Layout die Übersichtlichkeit beeinträchtigte, da sich Diagramme stark in einer Dimension streckten und dabei große Leerräume entstanden. Diese Beobachtung bildete den Ausgangspunkt für die Anforderungsanalyse an einen eigenständig zu integrierenden Layoutalgorithmus, aus der im weiteren Verlauf konkrete Bewertungskriterien abgeleitet wurden.

Bevor geeignete Algorithmen evaluiert werden konnten, war zunächst zu klären, ob das Layouting im Frontend oder im Backend ausgeführt werden sollte. Ein Backend-seitiger Ansatz hätte eine vollständige Entkopplung vom Rendering bedeutet, wodurch der Zugriff auf Daten der Darstellungsschicht, wie etwa Knotendimensionen, die für viele Algorithmen zur korrekten Positionsberechnung benötigt werden, nicht ohne weiteres möglich gewesen wäre. Ein Frontend-seitiger Ansatz ermöglicht hingegen die direkte Anbindung an SvelteFlow, dessen API nativen Support für gängige Layoutalgorithmen bietet. Da das Layouting lediglich beim erstmaligen Laden eines Diagramms ohne bestehendes Layout sowie auf explizite Anfrage angestoßen werden sollte, wurde der damit verbundene zusätzliche Backend-Call zur Persistierung der berechneten Positionen als akzeptabel bewertet und das Layouting im Frontend angesiedelt.

Aus den Anforderungen an das Layouting, den Eigenschaften des Anwendungsfalls sowie den beobachteten Schwächen des Dagre-basierten Mermaid.js-Layoutings wurden folgende Bewertungskriterien abgeleitet.

- **Übersichtliche Layouts für UML-Klassendiagramme** Der Algorithmus soll Knoten so anordnen, dass das resultierende Diagramm gut lesbar ist,

Klassen gleichmäßig im Raum verteilt werden und keine unnötigen Leerräume entstehen.

- **Keine Knotenüberschneidungen und minimale Kantenkreuzungen** Knoten dürfen sich im erzeugten Layout nicht überlagern, und Kantenkreuzungen sollen so weit wie möglich minimiert werden.
- **Deterministisches Verhalten** Für denselben Eingabegraphen muss stets dasselbe Layout erzeugt werden.
- **Performance** Das Layouting soll selbst für die größten in der Praxis vorkommenden CGMES 3.0 Profile in akzeptabler Zeit abgeschlossen sein.
- **Unterstützung von Subflows** Der Algorithmus soll die Möglichkeit bieten, Knoten in Gruppen zusammenzufassen, sodass der Layouter diese Gruppen als zusammenhängende Einheiten behandelt und entsprechend im Diagramm platziert.
- **Rendering-Unabhängigkeit** Der Algorithmus muss unabhängig von der eingesetzten Rendering-Schicht arbeiten können.

Das Kriterium des deterministischen Verhaltens ergibt sich unmittelbar aus der Anforderung an persistierbare Layouts. Erzeugt ein Algorithmus bei jedem Aufruf ein abweichendes Layout, lassen sich manuell angepasste Positionen nicht sinnvoll wiederverwenden, da das Ausgangslayout nicht reproduzierbar ist. Algorithmen mit zufallsbasierter Initialisierung wurden daher bewusst ausgeschlossen. Das Kriterium der Rendering-Unabhängigkeit stellt sicher, dass der Algorithmus ohne Abhängigkeit von einer konkreten Rendering-Schicht arbeitet und sich damit direkt in die SvelteFlow-basierte Frontend-Architektur einbinden lässt. Anhand dieser Kriterien wurden im folgenden Abschnitt infrage kommende Algorithmen evaluiert und erprobt.

4.2.2 Evaluation und Prototyping von Layoutalgorithmen

Nachdem die Entscheidung für eine Frontend-seitige Layouting-Lösung gefallen war, wurden geeignete JavaScript-basierte Layout-Bibliotheken auf Basis der in Abschnitt 4.2.1 formulierten Bewertungskriterien evaluiert. Als Quellen zur Identifikation geeigneter Kandidaten dienten die offizielle SvelteFlow-Dokumentation, die einen dedizierten Abschnitt zu kompatiblen Layouting-Bibliotheken bereitstellt, sowie eine kuratierte Sammlung node-basierter UI-Bibliotheken, die bereits im Rahmen der Rendering-Evaluation herangezogen worden war [42] [38].

Aus diesen Quellen wurden dagre sowie elkjs, das in Abschnitt 3.6.3 beschrieben wird, als aussichtsreiche Kandidaten identifiziert. Physiksimulationsbasierte Ansätze wie d3-force schieden wegen nicht-deterministischen Verhaltens und mangelnder Kontrolle über Knotenüberschneidungen unmittelbar aus, was sich mit den Beobachtungen aus Kapitel 2 am Beispiel von WebVOWL deckt. Hierarchiebasierte Bibliotheken wie d3-hierarchy schieden aufgrund der zyklischen Struktur von CIM-Modellen aus.

Da elkjs nicht einen einzelnen Algorithmus, sondern eine Auswahl der im ELK verfügbaren Algorithmen bereitstellt, war zusätzlich zu klären, welche davon geeignet waren. Baumbasierte und kreisförmige Algorithmen wie ELK MrTree und ELK Radial sind auf die Darstellung hierarchischer Strukturen ausgerichtet und erzeugen für UML-Klassendiagramme keine übersichtlichen Layouts. Algorithmen zur Anordnung unverbundener Knoten wie ELK Box und ELK Rect Packing erfüllten das Kriterium übersichtlicher Gesamtlayouts nicht. ELK Force schied als physiksimulationsbasierter Ansatz aus denselben Gründen wie d3-force aus, ELK Randomizer widersprach dem Kriterium deterministischen Verhaltens. Damit verblieben ELK Stress und ELK Layered als relevante Kandidaten.

Da ELK Stress in seiner Konzeption zunächst überschaubarer wirkte als ELK Layered, wurde er zuerst prototypisch erprobt. ELK Stress ist, ähnlich wie ELK Force, ein kraftbasierter Algorithmus, der Knotenabstände iterativ an vorgegebene Zielwerte annähert [43]. Im Prototyping zeigte sich, dass die Layoutqualität im Wesentlichen nur über die gewünschte Kantenlänge `elk.stress.desiredEdgeLength` steuerbar war und Kantenkreuzungen nur unzureichend minimiert wurden. Dieses Ergebnis entsprach der algorithmischen Grundlage, da kraftbasierte Algorithmen Kantenkreuzungen nicht als primäres Optimierungsziel behandeln. Die praktische Evaluation war dennoch notwendig, um auszuschließen, dass eine gezielte Konfiguration die strukturellen Grenzen ausgleichen kann.

Darüber hinaus wurde dagre, das in Abschnitt 3.6.1 näher beschrieben wird, prototypisch erprobt, da es bereits im Kontext von Mermaid.js eingesetzt worden war und damit als erprobte Referenz galt. Dabei offenbarte sich jedoch ein grundlegenderes Problem. Der Algorithmus legt Knoten hierarchisch in Schichten an, was sich durch Parameter wie `ranksep` nicht prinzipiell ändern lässt. Als Folge wurden große Graphen stark in einer Dimension gestreckt, und kantenlose Graphen in einer einzigen Linie angeordnet.

Weder dagre noch ELK Stress erfüllten die Kriterien hinreichend, sodass die Wahl auf ELK Layered fiel. Wie in Abschnitt 3.6.2 beschrieben, bietet der Algorithmus mit mehr als 150 Optionen den größten Anpassungsspielraum aller verfügbaren Kandidaten. Entscheidend war, dass ELK Layered als einziger Algorithmus alle Be-

wertungskriterien prinzipiell erfüllen konnte, darunter deterministisches Verhalten, Unterstützung von Subflows sowie nachgewiesene Eignung für komplexe Graphen mit vielen Kanten. Die konkrete Konfiguration wird in Abschnitt 4.2.3 behandelt.

4.2.3 Fine-Tuning des ELK Layered Algorithmus

Das Fine-Tuning des ELK Layered Algorithmus stellte eine methodische Herausforderung dar, die sich unmittelbar aus seiner Komplexität ergab. Wie in Abschnitt 3.6.2 beschrieben, bietet ELK Layered über 150 konfigurierbare Optionen, von denen viele voneinander abhängen und deren Wechselwirkungen sich nicht ohne weiteres aus der Dokumentation ableiten lassen. Da etablierte Best Practices zum Fine-Tuning des Algorithmus speziell für UML-Klassendiagramme nicht verfügbar waren, wurde von Beginn an ein empirisch-systematisches Vorgehen gewählt, das auf ein vollständiges algorithmisches Verständnis aller Wechselwirkungen bewusst verzichtete.

Dazu wurden alle verfügbaren Optionen anhand der offiziellen ELK-Dokumentation nach ihrer Funktion im Algorithmus gruppiert und hinsichtlich ihrer Relevanz bewertet. Optionen rund um TopDown-Hierarchien, Edge Labels sowie Wrapping- und Cutting-Mechanismen wurden ausgeschlossen. Die verbleibenden Optionen wurden anhand der in Abschnitt 4.2.1 definierten Kriterien priorisiert. Als besonders ausschlaggebend erwiesen sich dabei die Gruppen Node Layering, Node Placement und Crossing Minimization, da diese den größten Einfluss auf Übersichtlichkeit und Kantenkreuzungen hatten. Ergänzend wurden Optionen zur Behandlung von High-Degree-Nodes sowie Spacing-Parameter gezielt konfiguriert, um das Layout für die in CGMES 3.0 Profilen typischerweise vorkommenden stark vernetzten Klassen zu verbessern.

Die priorisierten Optionen wurden iterativ mit drei Kategorien von Testdiagrammen erprobt: großen CGMES 3.0 Profilen mit vielen Knoten und Kanten, durchschnittlichen Diagrammen sowie kantenlosen Diagrammen mit reinen Datentyp-Klassen. Anzumerken ist, dass das erreichbare Layoutergebnis durch eine Einschränkung auf Rendering-Seite begrenzt wurde, da die Rendering-Implementierung kein Kanten-Routing unterstützt und die von ELK Layered berechneten Kantenpfade daher nicht angewandt werden konnten, was die Qualität des finalen Layouts beeinträchtigte. Diese Einschränkung wird in Kapitel 6 näher diskutiert. Aus dem Gesamtprozess ging eine finale Konfiguration hervor, die in Abschnitt 5.1.3 dokumentiert ist.

4.3 Layout-Persistenz

Mit der Einführung des automatischen Layoutings wurde beim erstmaligen Laden eines Diagramms automatisch ein sinnvolles Ausgangslayout erzeugt, das jedoch beim Schließen des Diagramms verloren ging. Die dritte Entwicklungsphase hatte daher zum Ziel, sowohl automatisch erzeugte als auch manuell angepasste Diagrammlayouts persistent zu speichern und beim erneuten Laden wiederherzustellen, sodass die in den vorherigen Phasen geschaffenen Rendering- und Layouting-Funktionalitäten vollständig nutzbar werden. Da mit dem CGMES 3.0 DiagramLayout-Profil ein etablierter Standard zur einheitlichen Beschreibung von Diagrammlayouts in RDF existiert, bildete dessen Studie die konzeptionelle Grundlage für die Implementierung in Kapitel 5.

Abschnitt 4.3.1 dokumentiert diese Studie und legt dar, wie das Profil für die Anforderungen des RDF-Architects nutzbar gemacht wurde.

4.3.1 Studie des CGMES 3.0 DiagramLayout-Profils

Das CGMES 3.0 DiagramLayout-Profil stellt, wie in Abschnitt 3.4 beschrieben, ein standardisiertes Vokabular zur Beschreibung von Diagrammlayouts in RDF bereit. Gegenstand dieser Studie war die Frage, welche seiner Klassen für die Persistierung von Layoutdaten im RDF-Architect geeignet sind und wie die zugehörigen Tripel konkret zu setzen sind.

Als geeignet eingestuft wurden die drei Klassen Diagram, DiagramObject und DiagramObjectPoint. Die dabei gewählte Konvention sieht vor, dass für jedes CIM-Package ein Diagram angelegt wird, das als Container für die enthaltenen Elemente dient und dessen IRI der Package-UUID entspricht. Für jede CIM-Klasse wird ein DiagramObject erstellt, das über seine UUID mit dem zugehörigen IdentifiedObject verknüpft ist, sowie ein DiagramObjectPoint, der die x- und y-Position der Klasse im Diagramm speichert. DiagramObject und DiagramObjectPoint erhalten dabei jeweils eine zufällig generierte UUID als IRI. Das Diagram erhält den Namen des zugehörigen Packages sowie die Orientierung `OrientationKind.negative`. Das DiagramObject erhält den Namen der zugehörigen Klasse, eine Verknüpfung zum Diagram über die Package-UUID sowie eine Verknüpfung zum IdentifiedObject über die Klassen-UUID. Der DiagramObjectPoint erhält die x- und y-Koordinaten der Knotenposition sowie eine Verknüpfung zum zugehörigen DiagramObject.

Listing 4.1 zeigt beispielsweise die gesetzten Tripel eines DiagramObjects, wobei die IRIs gekürzten UUIDs entsprechen und jeweils nur die ersten acht Zeichen dargestellt sind.

```
<_b6539372-...>
   rdf:type cim:DiagramObject ;
   cim:IdentifiedObject.name "ACLineSegment" ;
   cim:DiagramObject.Diagram <_fcc8c2da-...> ;
   cim:DiagramObject.IdentifiedObject <_a1480398-...> .
```

Listing 4.1: Beispiel eines DiagramObjects für eine CIM-Klasse im CGMES DiagramLayout-Profil

Das Subjekt der Tripel ist eine dem DiagramObject zugewiesene eindeutige IRI. Über `cim:DiagramObject.Diagram` wird das DiagramObject dem zugehörigen Diagram zugeordnet. Die Verknüpfung über `cim:DiagramObject.IdentifiedObject` referenziert die IRI der zugehörigen CIM-Klasse, die damit vom Subjekt des DiagramObjects abweicht.

Die übrigen Klassen des Profils wurden für den aktuellen Implementierungsumfang nicht berücksichtigt. DiagramStyle und DiagramObjectStyle dienen der Beschreibung von Darstellungsstilen, wobei das Styling im RDF-Architect vollständig im Frontend erfolgt. Für einen zukünftigen Export wäre DiagramObjectStyle jedoch relevant, um verschiedene Kantentypen wie Assoziationen oder Vererbungsbeziehungen für externe Anwendungen zu kennzeichnen. VisibilityLayer ermöglicht die Zuweisung von DiagramObjects zu benannten Ebenen, was sich etwa mit den bestehenden Filter-Ansichten des RDF-Architects kombinieren ließe, jedoch keinen unmittelbaren Bedarf darstellte. TextDiagramObject ermöglicht freie Textplatzierung im Diagramm und wäre etwa für die Darstellung von Multiplizitätslabels an Kanten denkbar, wurde jedoch aufgrund des Implementierungsaufwands zurückgestellt.

5 Ergebnisse

Die in Kapitel 4 erarbeitete konzeptionelle Grundlage bildete die Basis für die Erweiterung des RDF-Architects. Im Zuge der Entwicklung wurden die in der Zielsetzung formulierten Anforderungen an interaktive Diagrammbearbeitung, automatisches Layouting und Layout-Persistenz im RDF-Architect realisiert. Identifizierte Einschränkungen und offene Aspekte werden dabei jeweils benannt, ihre Bewertung erfolgt jedoch gesammelt in Kapitel 6.

Abschnitt 5.1 beschreibt die Implementierung dieser Erweiterungen, beginnend mit einem Überblick über den Rendering-Ablauf und die Systemarchitektur, gefolgt von einer detaillierten Ausführung der drei Implementierungsbereiche Rendering, Layouting und Layout-Persistenz. Abschnitt 5.2 betrachtet die Ergebnisse aus Perspektive der Nutzer*innen und schließt mit einer Darstellung der Layoutqualität sowie einer Messung der Rendering- und Layouting-Performance.

5.1 Implementierung

Die Implementierung der in Kapitel 4 erarbeiteten Erweiterungen gliedert sich in die drei Bereiche Rendering, Layouting und Layout-Persistenz. Zur Absicherung der Backend-Logik wurden für das Rendering und die Layout-Persistenz darüber hinaus Unit-Tests implementiert, auf deren detaillierte Darstellung jedoch verzichtet wird.

Abschnitt 5.1.1 vermittelt zunächst einen Überblick über den Rendering-Ablauf und die resultierende Systemarchitektur. Abschnitt 5.1.2 beschreibt die Implementierung des Renderings im Backend und Frontend. Abschnitt 5.1.3 behandelt die Integration und Konfiguration des ELK Layered Algorithmus. Abschnitt 5.1.4 beschreibt die Persistierung von Diagrammlayouts im RDF-Architect.

5.1.1 Systemarchitektur im Überblick

Die Erweiterung des RDF-Architects betrifft mehrere Schichten des Systems und reicht von der Backend-seitigen Aufbereitung der RDF-Daten bis zur interaktiven Darstellung im Frontend. Zuvor basierte die Diagrammdarstellung ausschließlich auf dem `RenderCIMCollectionMermaidService`, der einen statischen Mermaid-Diagramm-String erzeugte und diesen direkt an das Frontend übergab. Im Zuge der Erweiterung wurde dieser Ablauf an mehreren Stellen ergänzt und umstrukturiert.

Anhand des zentralen Rendering-Ablaufs lassen sich die wesentlichen Änderungen und neu eingeführten Komponenten überblicken.

Abbildung 5.1 stellt diesen Ablauf von der Nutzeranfrage bis zur Diagrammanzeige dar. Aspekte, die über den dargestellten Ablauf hinausgehen, werden in den nachfolgenden Abschnitten vertieft.

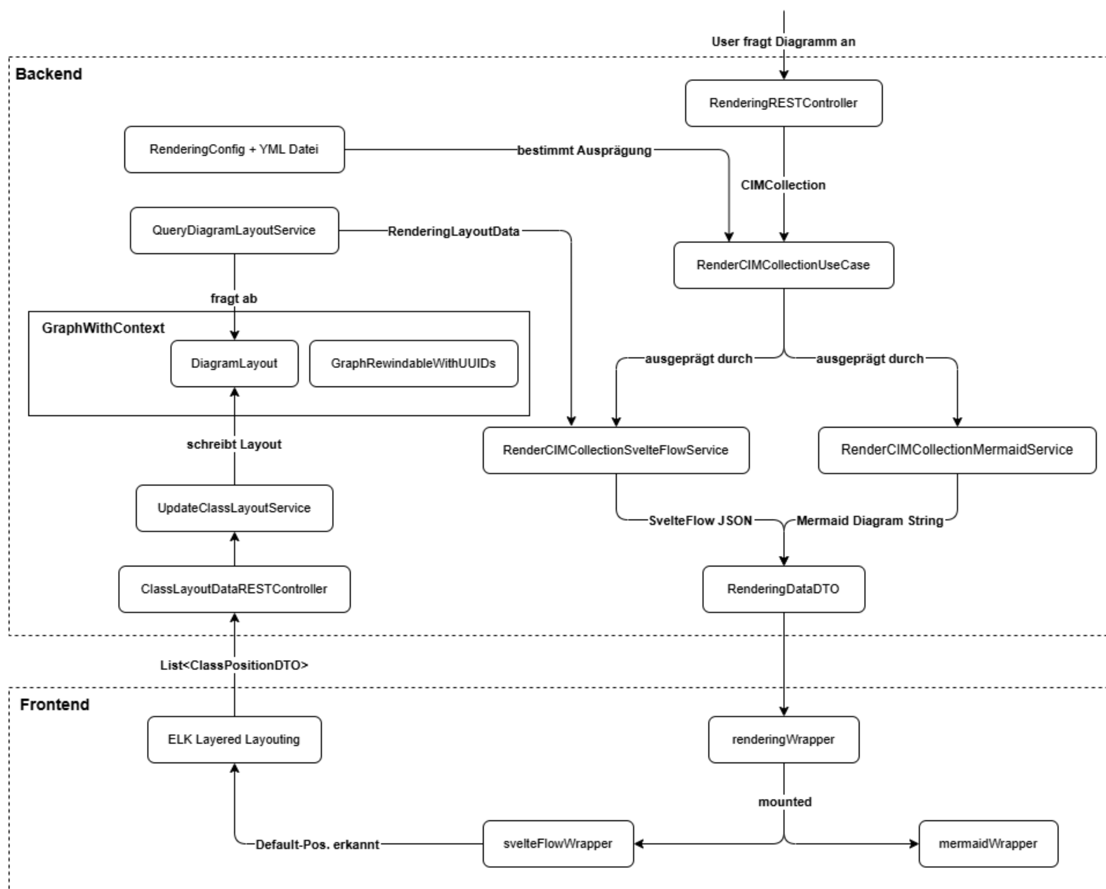

Abbildung 5.1: Rendering-Ablauf und Systemarchitektur des RDF-Architect

Wird ein Diagramm im RDF-Architect geöffnet, erreicht die Anfrage zunächst den `RenderingRESTController`. Dieser übergibt eine `CIMCollection`, eine Sammlung der darzustellenden CIM-Objekte wie Klassen, Attribute und Assoziationen, an den `RenderCIMCollectionUseCase`. Welcher Rendering-Service daraufhin zum Einsatz kommt, wird durch die `RenderingConfig` in Verbindung mit einer YAML-Konfigurationsdatei bestimmt. Je nach Konfiguration übernimmt entweder der neu eingeführte `RenderCIMCollectionSvelteFlowService` oder der bestehende `RenderCIMCollectionMermaidService` die weitere Verarbeitung. Der `RenderCIMCollectionSvelteFlowService` greift dabei auf den `GraphWithContext`

zu, der neben dem DiagramLayout auch einen GraphRewindableWithUUIDs enthält. Letzterer ist ein projektinterner Wrapper für GraphRewindable, das als projekteigene Implementierung den Umgang mit RDF-Graphen kapselt. Der QueryDiagramLayoutService fragt die im DiagramLayout gespeicherten Layoutinformationen ab und stellt sie dem Service als RenderingLayoutData zur Verfügung. Der RenderCIMCollectionSvelteFlowService befüllt daraufhin ein RenderingDataDTO, ein Data Transfer Object (DTO), mit den generierten Nodes und Edges als SvelteFlowDTO. Der RenderCIMCollectionMermaidService legt hingegen den erzeugten Mermaid-Diagramm-String als MermaidDTO in dasselbe DTO.

Im Frontend empfängt der neu eingeführte renderingWrapper das DTO, wertet das enthaltene Renderingformat aus und mounted entsprechend den svelteFlowWrapper oder den mermaidWrapper. Der mermaidWrapper übernahm zuvor zusätzlich die Aufgaben des heutigen renderingWrapper und wurde im Zuge der Erweiterung auf die reine Mermaid-Darstellung reduziert. Beide Wrapper sind für die Anzeige des Diagramms zuständig und verarbeiten dafür die ihnen übergebenen Daten. Liegen im svelteFlowWrapper ausschließlich Default-Positionen vor, wird vor der Darstellung das ELK Layered Layouting angestoßen, dessen berechnete Positionen als List<ClassPositionDTO> an den ClassLayoutDataRESTController übertragen werden. Der UpdateClassLayout-Service aktualisiert daraufhin die entsprechenden Positionstripel im DiagramLayout.

Der dargestellte Ablauf erfasst die zentralen Komponenten des Rendering-Prozesses, lässt jedoch verschiedene Aspekte aus. Dazu zählen unter anderem die Synchronisation zwischen CIM- und DiagramLayout-Daten, die nähere Backend-Struktur rund um die Layout-Persistenz einschließlich der SPARQL-Queries und Apache-Jena-Model-API-Operationen, sowie Details zur SvelteFlow-Frontend-Implementierung. Diese Aspekte werden in den folgenden Abschnitten ausgeführt.

5.1.2 Rendering

Die Implementierung des Renderings bildet den zentralen Bestandteil der entwickelten Erweiterungen. Aufbauend auf dem in Abschnitt 5.1.1 beschriebenen Überblick über den Rendering-Ablauf und die wesentlichen Komponenten führt dieser Abschnitt die Umsetzung auf Detailebene aus. Dazu wird zunächst das DTO-basierte Datenformat betrachtet, das das Backend an das Frontend überträgt. Darauf folgt eine Betrachtung der Backend-Logik rund um Controller und Rendering-Service. Anschließend wird die Diagrammdarstellung im Frontend beschrieben, bevor abschließend die austauschbare Rendering-Architektur erläutert wird.

Rendering DTOs

Wie in Abschnitt 3.5.3 beschrieben, erwartet SvelteFlow Nodes und Edges in einem definierten JSON-Format. Dieses Format wird im Backend durch DTO-Klassen garantiert, die Spring Boot, wie zuvor erläutert, automatisch zu JSON serialisiert. Das resultierende JSON-Objekt findet sich im Anhang unter Listing A.1, wobei UUIDs auf die ersten acht Zeichen gekürzt wurden und von jedem Ressourcentyp jeweils ein exemplarischer Eintrag enthalten ist. Die folgende Beschreibung bezieht sich auf dieses Format.

Neben den von SvelteFlow vorgeschriebenen Pflichtfeldern enthält das `data`-Objekt einer Node die darstellungsrelevanten Informationen der CIM-Klasse, die von der zugehörigen custom Svelte-Komponente direkt zur Anzeige ausgewertet werden. `label` entspricht dem Klassennamen, `stereotypes` enthält die zugehörigen CIM-Stereotypen und `belongsToCategory` das CIM-Tripel der zugehörigen Kategorie, wie in Abschnitt 3.2.3 beschrieben. Die Liste `attributes` enthält die Attribute der Klasse, wobei jeder Eintrag aus `label`, `type` und `multiplicity` besteht. Handelt es sich bei der Klasse um ein Enum, enthält `enumEntries` die entsprechenden Einträge als Strings, andernfalls bleibt die Liste leer. Analog dazu werden auch Edge-DTOs mit einem `data`-Objekt versehen, sofern die Kante eine Assoziation darstellt. Dieses enthält `toMultiplicity` und `fromMultiplicity` als UML-Kantenbeschriftungen sowie `useToAssociation` und `useFromAssociation`, die angeben, ob die jeweilige Richtung der CIM-Assoziation genutzt wird und damit die Darstellung der Pfeilspitzen steuern. Vererbungskanten hingegen erfordern keine dieser Informationen und enthalten daher kein `data`-Objekt.

Backend-Logik

Wie in Abschnitt 5.1.1 beschrieben, erreicht eine Diagrammanfrage zunächst den `RenderingRESTController`, der über einen dedizierten REST-Endpunkt erreichbar ist. Dieser konvertiert den zugehörigen RDF-Graphen in eine `CIMCollection` und übergibt sie an `renderUML`. Das resultierende `RenderingDataDTO` wird ans Frontend übertragen.

Der `RenderCIMCollectionSvelteFlowService` arbeitet dabei ähnlich wie der bisherige `RenderCIMCollectionMermaidService` und baut die DTO-Objekte durch verschachtelte Methodenaufrufe zusammen. Zu Beginn werden eine `uriToUUIDMap` zur gegenseitigen Abbildung von Uniform Resource Identifier (URI) und UUID aufgebaut sowie die Layoutdaten des angefragten Diagramms über einen dedizierten Use Case abgerufen.

Listing 5.1 zeigt `assembleNodeDTO`, eine der zentralen Methoden des Services, die exemplarisch für die weiteren Assemblier-Methoden steht.

```
private NodeDTO assembleNodeDTO(RenderContext renderContext, CIMClass cimClass)
     {
    var dop = renderContext.layoutingData().getClassLayoutingData()
        .get(cimClass.getUuid());
    var nodeDTO = NodeDTO.builder().id(cimClass.getUuid())
        .type(CLASS_NODE_TYPE);
    var stereotypes = getClassStereotypes(cimClass);
    var attributes = getClassAttributes(renderContext, cimClass);
    var enumEntries = getClassEnumEntries(renderContext, cimClass);
    var positionDTO = PositionDTO.builder()
        .x(dop.getPosition().getX())
        .y(dop.getPosition().getY())
        .build();
    nodeDTO.position(positionDTO);
    var nodeDataDTO = NodeDataDTO.builder()
        .label(cimClass.getLabel().getValue())
        .belongsToCategory(cimClass.getBelongsToCategory() != null
           ? cimClass.getBelongsToCategory().getLabel().getValue() : null)
        .stereotypes(stereotypes)
        .attributes(attributes)
        .enumEntries(enumEntries)
        .build();
    nodeDTO.data(nodeDataDTO);
    return nodeDTO.build();
}
```

Listing 5.1: Methode `assembleNodeDTO` zur Assemblierung eines `NodeDTO` im `RenderCIMCollectionSvelteFlowService`

Die Methode erhält den `RenderContext` sowie eine `CIMClass` und baut daraus ein vollständiges `NodeDTO` zusammen. Zunächst wird der zugehörige `DiagramObjectPoint` aus den Layoutdaten des `RenderContext` anhand der UUID der Klasse geholt und daraus die Position des Knotens bestimmt. Die weiteren Felder des `NodeDataDTO` werden über dedizierte Hilfsmethoden befüllt, die jeweils die relevanten Informationen aus der `CIMClass` extrahieren. Analoge Methoden existieren für Edges sowie für die jeweiligen Listen-Assemblierungen, die die `CIMCollection` durchlaufen und `assembleNodeDTO` für jede Klasse aufrufen.

Diagrammdarstellung im Frontend

Mit der neuen Backend-Logik waren im Frontend entsprechende Komponenten erforderlich, die die übertragenen Daten entgegennehmen und als interaktives

Diagramm darstellen. Zentrale Datei dafür ist der `svelteFlowWrapper`. Dieser erhält Nodes und Edges per `$props` vom `renderingWrapper` und bindet, wie in Abschnitt 3.5.3 beschrieben, die `<SvelteFlow>`-Komponente ein. Dabei werden über `nodeTypes` und `edgeTypes` die Custom Svelte-Komponenten registriert.

Wie in Abschnitt 3.5.3 beschrieben, erhalten Custom-Komponenten das `data`-Objekt aus dem JSON und werten es zur Darstellung aus. Als einzige Node-Komponente kommt `ClassNode` zum Einsatz, die eine UML-Klasse mit Stereotypen, Attributen und Enum-Einträgen darstellt. Für Kanten existieren `AssociationEdge` und `InheritanceEdge`, entsprechend den zwei verwendeten UML-Kantentypen.

Listing 5.2 zeigt den Template-Teil der `AssociationEdge`.

```
<BaseEdge {id} {path} {markerStart} {markerEnd} {style} />
<EdgeLabel>
    {#if data.fromMultiplicity}
        <div
            style:transform={`translate(-50%, -50%) translate(${
                edgeParams.sx + edgeParams.startX}px, ${
                edgeParams.sy + edgeParams.startY}px)`}
            class="..."
        >
            {data.fromMultiplicity}
        </div>
    {/if}
    {#if data.toMultiplicity}
        <div
            style:transform={`translate(-50%, -50%) translate(${
                edgeParams.tx + edgeParams.endX}px, ${edgeParams.ty
                 + edgeParams.endY}px)`}
            class="..."
        >
            {data.toMultiplicity}
        </div>
    {/if}
</EdgeLabel>
```

Listing 5.2: Template der Komponente `AssociationEdge`

Die `BaseEdge`-Komponente übernimmt die Darstellung des SVG-Kantenpfads, wobei `markerStart` und `markerEnd` die Pfeilspitzen abhängig von `useFromAssociation` und `useToAssociation` setzen. Die Komponente `InheritanceEdge` zur Darstellung von UML-Vererbungspfeilen ist analog aufgebaut, verzichtet jedoch auf das `data`-Objekt und verwendet einen festen Vererbungspfeil.

Im Gegensatz zum typischen Handle-basierten Ansatz, wie in Abschnitt 3.5.3 beschrieben, bei dem Kanten zwischen dedizierten Quell- und Ziel-Handles ver-

bunden werden, setzt die Implementierung auf eine vereinfachte Variante: Jeder Knoten besitzt einen einzelnen, unsichtbaren Handle im Knotenmittelpunkt, der sowohl als Quell- als auch als Zielpunkt aller Kanten dient. Die Kanten selbst werden als direkte Verbindungslinien gezeichnet, wobei die Pfeilspitzen nicht am Handle, sondern am Knotenrand platziert werden. Die dafür notwendigen Berechnungen, darunter der Schnittpunkt zwischen Verbindungslinie und Knotenrand sowie die Labelversätze entlang und senkrecht zur Kante, liefert eine Hilfsfunktion als `edgeParams`-Objekt. Wie in Listing 5.2 zu sehen, fließen diese Werte direkt in die Positionierung der Multiplizitätslabels ein. Dieser Ansatz orientiert sich am Beispiel „Easy Connect“ der offiziellen SvelteFlow-Dokumentation [44].

Austauschbare Rendering-Architektur

Um die Mermaid-Implementierung als Fallback-Option zu erhalten, wurde das Rendering von Beginn an so konzipiert, dass der aktive Rendering-Service per Konfiguration umschaltbar ist. Die technische Grundlage dafür wurde bereits in Abschnitt 5.1.1 beschrieben. Die `RenderingConfig` schaltet abhängig vom Wert in der YAML-Konfigurationsdatei den aktiven Service per Dependency Injection um, beide Services befüllen einheitlich dasselbe `RenderingDataDTO`, und der `renderingWrapper` im Frontend mounted abhängig vom enthaltenen Format den `svelteFlowWrapper` oder den `mermaidWrapper`. Das Rendering lässt sich damit durch eine einzige Konfigurationsänderung umschalten, ohne dass weitere Anpassungen am System notwendig sind.

5.1.3 Layouting

Das Layouting ergänzt das entwickelte Rendering um die automatische Positionierung der Diagrammelemente. Für die Einbindung des ELK Layered Algorithmus in das Frontend wurde `elkjs` verwendet, die offizielle JavaScript-Implementierung von ELK, beschrieben in Abschnitt 3.6.3. Dessen Konfiguration wurde im Zuge des Fine-Tunings in Abschnitt 4.2.3 erarbeitet. Dieser Abschnitt stellt die finale Konfiguration sowie deren technische Integration in die SvelteFlow-Umgebung vor. Die vollständige Konfiguration ist im Anhang unter Listing A.2 einsehbar, wohingegen nachfolgend ausgewählte Optionen erläutert werden.

Für die Konfiguration wurden mehrere Optionsgruppen festgelegt. Einige davon sind im Hinblick auf das Fine-Tuning besonders hervorzuheben. Die Option `elk.aspectRatio` mit dem Wert `1.78` richtet die Ausdehnung des Diagramms am 16:9-Seitenverhältnis aus, das bei Computermonitoren verbreitet ist. Der Wert `2.0`

für `elk.edge.thickness` stimmt mit der in SvelteFlow verwendeten Kantenbreite überein. Der Parameter `elk.layered.thoroughness` wurde auf `150` gesetzt, ein Wert, der im Fine-Tuning als geeignet für die auftretenden Diagrammgrößen identifiziert wurde. Niedrigere Werte beeinträchtigen die Layoutqualität, höhere Werte erhöhen die Berechnungsdauer, ohne die Qualität weiter zu verbessern. Weitere Optionsgruppen steuern die Knotenplatzierungsstrategie, die Kantenkreuzungsminimierung, die Knotenpromotion, die Behandlung von Knoten mit hohem Knotengrad sowie das Abstandsverhalten zwischen Knoten und Kanten.

Der ELK Layered Algorithmus berechnet geroutete Kanten, also Verbindungen mit Zwischenpunkten, und berücksichtigt diese bei der Bestimmung der Knotenpositionen. Die Rendering-Implementierung stellt Kanten hingegen als direkte Verbindungen dar. Durch diese Diskrepanz werden die berechneten Knotenpositionen übernommen, die gerouteten Kantenverläufe jedoch nicht abgebildet. Das vom Algorithmus vorgesehene Layout kann dadurch nicht in vollem Umfang realisiert werden, was die Layoutqualität einschränkt.

Die technische Einbindung des Layoutalgorithmus erfolgt über die Funktion `getLayoutedNodes`, die aus den aktuellen Nodes und Edges einen `elkGraph` aufbaut und das Layouting asynchron über einen Web Worker ausführt, also einen separaten Browser-Thread, der die Benutzeroberfläche während der Berechnung nicht blockiert. Die berechneten Positionen werden anschließend in die SvelteFlow-States übernommen und unmittelbar an das Backend übermittelt, womit die Layout-Persistenz direkt nach dem initialen Layouting sichergestellt wird. Liegen beim Laden eines Diagramms ausschließlich Default-Positionen vor, wird das Layouting automatisch vor der ersten Anzeige ausgeführt. Andernfalls wird das gespeicherte Layout wiederhergestellt. Das resultierende Layoutergebnis wird in Abschnitt 5.2.2 visuell dargestellt.

5.1.4 Layout-Persistenz

Die Studie des CGMES 3.0 DiagramLayout-Profils in Abschnitt 4.3.1 legte fest, welche Klassen des Profils für die Persistierung von Layoutdaten im RDF-Architect geeignet sind. Auf dieser Grundlage wurde die Layout-Persistenz implementiert, sodass Layouts gespeichert und beim erneuten Laden eines Diagramms wiederhergestellt werden. Dieser Abschnitt stellt diese Implementierung vor.

Zunächst werden die Datenstrukturen im Backend eingeführt. Darauf aufbauend wird erläutert, wie auf DiagramLayout-Daten zugegriffen und wie diese manipuliert werden. Anschließend wird beschrieben, wie Positionsdaten aus dem Frontend

persistiert werden, wie der Rendering-Prozess diese konsumiert und wie die Konsistenzhaltung zwischen CIM- und DiagramLayout-Daten gewährleistet wird.

Datenmodell und Backend-Struktur

Wie in Abschnitt 5.1.1 vorweggenommen, bildet der `GraphWithContext` eine zentrale Datenstruktur des Backends. Dieser Wrapper entstand im Rahmen der Implementierung der Layout-Persistenz, denn zuvor wurden Graphen in Form von `GraphRewindableWithUUIDs`-Objekten direkt in einer `GraphCollection` gehalten. Der `GraphWithContext` fasst nun den `GraphRewindableWithUUIDs` und die neu entstandene `DiagramLayout`-Klasse zusammen, wobei Letztere ein Apache Jena Model kapselt und darauf aufbauend Funktionalitäten bereitstellt. Wie in Abschnitt 3.3.2 beschrieben, repräsentiert ein Graph im RDF-Architect eine einzelne RDF-Datei, während ein Dataset eine Menge solcher Graphen zusammenfasst. Die bereits bestehende `GraphCollection` wurde entsprechend zur `GraphWithContextCollection` weiterentwickelt und repräsentiert weiterhin das Dataset.

Da der Renderer für jedes Diagramm auf DiagramLayout-Informationen angewiesen ist, muss eine initiale Struktur vorliegen, bevor ein Diagramm erstmals geöffnet wird. Aus diesem Grund wird beim Importieren von Graphen nach dem eigentlichen Import die initiale DiagramLayout-Struktur angelegt. Die dabei gesetzten Positionen sind zunächst ausschließlich Default-Positionen, woraus sich der in Abschnitt 5.1.3 beschriebene Mechanismus ergibt, der beim ersten Öffnen eines Diagramms das automatische Layouting anstößt. Darüber hinaus wurden für die code-interne Repräsentation der DiagramLayout-Klassen eigene DTO-Objekte angelegt, die etwa beim Erstellen von Tripeln oder beim Auslesen aus dem Model einheitlich eingesetzt werden.

Interaktion mit DiagramLayout-Daten

Für die Implementierung der Layout-Persistenz wurden dedizierte Komponenten zur Interaktion mit den DiagramLayout-Tripeln entwickelt. Diese lassen sich in lesende und schreibende Operationen unterteilen, die jeweils über eigene Klassen abgewickelt werden.

Das Auslesen von DiagramLayout-Objekten aus dem Apache Jena Model erfolgt über den `DLObjectFetcher`. Dieser nimmt einen Diagramm-Identifikator sowie das `DiagramLayout`-Model entgegen, führt darauf eine SPARQL-Query aus und gibt die Ergebnisse als interne DTO-Objekte zurück. Die zurückgegebenen Abfrageergebnisse werden dabei an die `DLObjectFactory` und `DLQuerySolutionParser`

weitergereicht. Erstere konstruiert aus den Abfrageergebnissen die internen DTO-Objekte, während Letzterer das kontrollierte Auslesen der einzelnen Felder aus den `QuerySolution`-Objekten übernimmt. Der `DLObjectFetcher` stellt damit die einzige Schnittstelle dar, über die DiagramLayout-Objekte im Code abgerufen werden.

Listing 5.3 zeigt exemplarisch eine der verwendeten Queries.

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX cim:<http://iec.ch/TC57/CIM100#>
SELECT ?ioMRID ?dopMRID ?doMRID ?xPosition ?yPosition
WHERE {
    ?diagramMRID rdf:type cim:Diagram .
    ?doMRID rdf:type cim:DiagramObject ;
                cim:DiagramObject.IdentifiedObject ?ioMRID ;
                cim:DiagramObject.Diagram ?diagramMRID .
    ?dopMRID rdf:type cim:DiagramObjectPoint ;
                 cim:DiagramObjectPoint.DiagramObject ?doMRID ;
                 cim:DiagramObjectPoint.xPosition ?xPosition ;
                 cim:DiagramObjectPoint.yPosition ?yPosition .
    FILTER(STR(?diagramMRID) = "DIAGRAM_MRID")
}
```

Listing 5.3: SPARQL-Query zum Abrufen von DiagramObjectPoints und Class UUIDs eines Diagramms

Wie in Abschnitt 3.1.2 beschrieben, arbeitet SPARQL mit Pattern Matching über Variablen. Diese Query wird für ein gegebenes Package aufgerufen, dessen Diagramm-mRID über den `FILTER`-Ausdruck als Einstiegspunkt dient. Über ein zweistufiges Pattern Matching werden zunächst alle zu diesem Diagram gehörenden DiagramObjects mit ihren Klassen-mRIDs ermittelt und darüber wiederum die zugehörigen DiagramObjectPoints mit ihren Positionsdaten. Das Ergebnis ist eine Zuordnung von Klassen-mRID zu DiagramObjectPoint-Daten, die für die weitere Verarbeitung im Backend zur Verfügung steht.

Alle schreibenden Operationen, also das Erstellen, Ersetzen und Löschen von Tripeln, werden über `DLUpdates` abgewickelt. Diese Klasse arbeitet ausschließlich mit der Apache Jena Model API und verwendet kein SPARQL.

Listing 5.4 zeigt die Methode `insertDiagram` als repräsentatives Beispiel.

```
public void insertDiagram(Model model, Diagram diagram) {
    var newDiagram = model.createResource(diagram.getMRID().getFullMRID());

    newDiagram.addProperty(RDF.type, DL.diagramType);
    newDiagram.addProperty(CIM.ioName, ResourceFactory.createPlainLiteral(
        diagram.getName()));
    newDiagram.addProperty(DL.orientation, DL.negativeOrientation);

    model.add(newDiagram.listProperties());
}
```

Listing 5.4: Methode `insertDiagram` in `DLUpdates` zum Anlegen eines Diagram

Zunächst wird über `model.createResource` eine neue Ressource mit der mRID des Diagrams angelegt. Über `addProperty` werden ihr daraufhin die zugehörigen Tripel hinzugefügt, also Typ, Name und Orientierung. Abschließend werden die gesetzten Properties über `model.add` in das Model übernommen.

Für die einheitliche Verwendung von Bezeichnern im Code wurden ergänzend unterstützende Klassen angelegt. `DLQueryVars` zentralisiert die Variablennamen für SPARQL-Queries, sodass dieselben Bezeichner sowohl in den Queries des `DLObjectFetcher` als auch in der `DLObjectFactory` beim Auslesen der Ergebnisse Verwendung finden. Die Vokabularklassen `DL` und `CIM`, aus denen etwa die Konstanten `DL.diagramType`, `DL.negativeOrientation` und `CIM.ioName` in Listing 5.4 stammen, stellen Apache Jena Properties, Resources und Strings bereit, die in `DLUpdates` und weiteren Klassen eingesetzt werden, um das wiederholte Erstellen identischer Objekte zu vermeiden.

Frontend-Abspeicherung

Die Abspeicherung von Layoutdaten wird durch Nutzerinteraktion im Frontend angestoßen. Wie in Abschnitt 5.1.1 im Kontext des initialen Layoutings bereits angerissen, werden berechnete Positionen unmittelbar nach dem Layouting ans Backend übermittelt. Dieser Abschnitt stellt die zugrunde liegende Struktur zur Abspeicherung vollständig vor.

Im `svelteFlowWrapper` existieren zwei Auslöser für die Abspeicherung. Der erste ist das manuelle Verschieben von Nodes durch Nutzer*innen, der zweite das Anstoßen nach dem automatischen Layouting. In beiden Fällen werden die betroffenen Nodes mit ihren aktuellen Positionen als `ClassPositionDTO`-Liste an den `ClassLayoutDataRESTController` übermittelt. Dieser gibt die Daten an den

`UpdateClassLayoutService` weiter, der die entsprechenden DiagramObjectPoints über die zuvor beschriebenen Operationen aktualisiert.

Rendering-Ertüchtigung

Damit gespeicherte Layoutdaten beim Rendering tatsächlich genutzt werden können, musste der Rendering-Prozess entsprechend erweitert werden. Abschnitt 5.1.1 stellte diesen Ablauf bereits überblicksartig vor, dieser Abschnitt beschreibt die dafür vorgenommenen Anpassungen.

Der `RenderingRESTController` wurde erweitert, sodass neben der `CIMCollection` nun auch die zur Identifikation des angefragten Packages notwendigen Informationen an den Rendering-Aufruf weitergegeben werden. Damit kann der `RenderCIMCollectionSvelteFlowService` über den `QueryDiagramLayoutService` die gespeicherten Layoutdaten für das angefragte Package abrufen. Die dabei verwendete SPARQL-Query entspricht der in Listing 5.3 gezeigten. Die abgerufenen Daten werden als `RenderingLayoutData` zurückgegeben und im Rendering-Service genutzt, um den Node-DTOs die gespeicherten Positionen zuzuweisen, wie in Listing 5.1 in Abschnitt 5.1.2 ersichtlich.

Synchronisation von CIM- und DiagramLayout-Daten

Änderungen an CIM-Daten müssen sich in den DiagramLayout-Daten widerspiegeln, damit beide Modelle konsistent bleiben. Für Operationen an CIM-Klassen und Packages wurden daher die bestehenden CIM-Services um analoge DiagramLayout-Operationen ergänzt, da ausschließlich solche Operationen die DiagramLayout-Daten betreffen. Die dazugehörigen Layout-Operationen wurden in `UpdateClassLayoutService` und `UpdatePackageLayoutService` ergänzt und aus den jeweiligen CIM-Services `UpdateClassService` und `UpdatePackageService` heraus aufgerufen.

Dieser Ansatz deckt jedoch nicht alle Fälle ab. Operationen wie das Verschieben einer Klasse zwischen Packages oder Undo/Redo-Aktionen lassen sich nicht ohne weiteres durch gezielte Service-Aufrufe abfangen, da sie auf einer anderen Ebene im System stattfinden und komplexere Anpassungslogik erfordern würden. Aus diesem Grund wurde ergänzend ein Failsafe-Mechanismus eingebaut. Die Methode `ensureDiagramLayoutExists` wird vor jedem Rendering-Aufruf ausgeführt und stellt sicher, dass für das angefragte Package vollständige DiagramLayout-Objekte vorliegen. Fehlende Objekte werden dabei mit Default-Positionen neu angelegt.

5.2 Funktionsumfang und Qualität der Visualisierung

Die Implementierung bildet die technische Grundlage, auf der die eigentlichen Ergebnisse dieser Arbeit aufbauen. Während Abschnitt 5.1 die Umsetzung der interaktiven Diagrammvisualisierung, des automatischen Layoutings und der Layout-Persistenz beschrieb, rückt dieser Abschnitt die daraus resultierenden Ergebnisse in den Vordergrund. Betrachtet wird dabei sowohl, welche Funktionen den Nutzer*innen des RDF-Architects nun zur Verfügung stehen, als auch, wie sich die Layoutqualität und die Rendering-Performance des neuen Ansatzes im Vergleich zum bisherigen Mermaid-Rendering verhalten.

Abschnitt 5.2.1 stellt die entstandenen Nutzerfunktionen vor und beleuchtet die Erweiterbarkeit der neuen Architektur. Abschnitt 5.2.2 vergleicht die Layoutqualität des neuen Ansatzes mit dem bisherigen Mermaid-Rendering. Abschnitt 5.2.3 stellt abschließend die Ergebnisse gezielter Messungen der Rendering- und Layouting-Dauern vor.

5.2.1 Nutzerfunktionen und Erweiterbarkeit

Durch die in Abschnitt 5.1 beschriebenen Erweiterungen stehen den Nutzer*innen des RDF-Architects neue Funktionen zur Interaktion mit Diagrammen zur Verfügung. Dieser Abschnitt stellt diese vor und beleuchtet darüber hinaus, welche Erweiterungsmöglichkeiten die entstandene Architektur bietet.

Abbildung 5.2 veranschaulicht die implementierten Nutzerfunktionen anhand eines beispielhaften Diagramms.

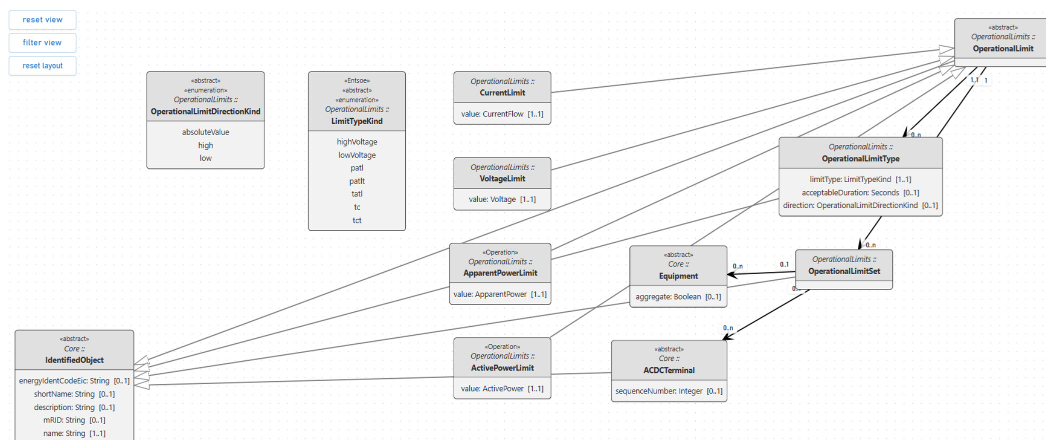

Abbildung 5.2: Darstellung der Nutzerfunktionen im RDF-Architect

Beim ersten Öffnen eines Diagramms wird automatisch ein Ausgangslayout generiert, das die enthaltenen Klassen strukturiert anordnet. Die Klassen des Diagramms sind dabei verschiebbar, wie in der Abbildung an den Klassen `IdentifiedObject` in der unteren linken Ecke und `OperationalLimit` in der oberen rechten Ecke erkennbar ist, die aus ihren ursprünglichen Positionen in der Mitte des Ausgangslayouts manuell bewegt wurden. Das resultierende Layout wird persistiert und beim erneuten Laden des Diagramms wiederhergestellt. Über den Reset-Layout-Button in der oberen linken Ecke lässt sich das Layout jederzeit neu berechnen lassen, etwa nach strukturellen Änderungen am Diagramm.

Die beschriebenen Nutzerfunktionen sind das unmittelbare Ergebnis der implementierten Erweiterungen. Durch den Wechsel zu SvelteFlow liegt nun eine komponentenbasierte Rendering-Architektur vor, in der Diagrammelemente als eigenständige Svelte-Komponenten realisiert sind. In Verbindung mit der Integration von elkjs sowie der Layout-Persistenz über das CGMES 3.0 DiagramLayout-Profil eröffnet dies eine Reihe von Erweiterungsmöglichkeiten. Das Erscheinungsbild und Verhalten aller Diagrammelemente lässt sich durch Anpassung der Komponenten gezielt steuern. Welche Elemente im Diagramm dargestellt werden, ist ebenso konfigurierbar wie Qualität, Typ und Konfiguration des Layoutings über elkjs. Darüber hinaus sind sowohl programmatische Zugriffe auf Diagrammelemente als auch weitergehende nutzergetriebene Interaktionen, etwa kontextbezogene Diagrammansichten, realisierbar.

5.2.2 Layoutqualität

Die Layoutqualität ist ein zentrales Kriterium für die Nutzbarkeit der Diagrammvisualisierung. Dieser Abschnitt stellt das Layout des Mermaid-Ansatzes dem des fine-tuned ELK Layered Algorithmus gegenüber. Wie in Abschnitt 3.5.1 beschrieben, nutzt Mermaid den Dagre-Algorithmus in Verbindung mit Post-Processing zur Layoutberechnung, während der RDF-Architect nun den ELK Layered Algorithmus einsetzt. Für den Vergleich wurde ein Diagramm mit einer repräsentativen Anzahl an Klassen gewählt, das sowohl Assoziations- als auch Vererbungsbeziehungen enthält. Die Abbildung des Diagramms im Mermaid-Rendering ist vergrößert im Anhang unter A.1 zu finden.

Abbildung 5.3 zeigt das Diagramm im Mermaid-Rendering, das über die austauschbare Rendering-Architektur aus Abschnitt 5.1.2 zu Vergleichszwecken aktiviert wurde.

Abbildung 5.3: Layoutdarstellung eines Diagramms im Mermaid-Rendering

Das durch Dagre erzeugte Layout ordnet die Klassen in horizontalen Schichten an. Dabei werden Klassen, die unterschiedlichen Packages angehören, in eigene Containerbereiche gruppiert, sodass etwa `ControlArea` oder `GeneratingUnit` innerhalb der Grenzen ihres jeweiligen Packages dargestellt werden. Durch diese Containerstruktur sowie die Anordnung der Schichten entstehen im Diagramm großflächige Leerstellen.

Abbildung 5.4 zeigt dasselbe Diagramm mit dem fine-tuned ELK Layered Algorithmus.

Abbildung 5.4: Layoutdarstellung eines Diagramms mit dem ELK Layered Algorithmus

Der ELK Layered Algorithmus ordnet die Klassen in vertikalen Schichten an, wobei die Richtung der Vererbungsbeziehungen den Lesefluss von links nach rechts bestimmt. Eine Containerstruktur ist im dargestellten Layout nicht vorhanden. Der ELK Layered Algorithmus unterstützt zwar die Darstellung von Subgraphen, was eine Package-basierte Gruppierung analog zu Mermaid ermöglichen würde, jedoch wurde dies in der aktuellen Rendering-Implementierung nicht umgesetzt. Knoten mit hohem Knotengrad wie `ConductingEquipment` und `PowerSystemResource`, also Klassen mit einer Vielzahl eingehender oder ausgehender Kanten, werden durch die High-Degree-Node-Behandlung des Algorithmus in zentralen Positionen platziert. Die Vererbungshierarchie von `ConductingEquipment` über `Equipment` und `PowerSystemResource` bis hin zu `IdentifiedObject` ist dadurch von links nach rechts nachvollziehbar.

Ein Aspekt schränkt die dargestellte Layoutqualität jedoch ein. Wie in Abschnitt 5.1.3 beschrieben, berechnet der ELK Layered Algorithmus bei der Positionierung der Knoten auch die Verläufe der Kanten mit ein, da diese den verfügbaren Raum zwischen den Schichten beeinflussen. Die Rendering-Implementierung bildet diese berechneten Kantenverläufe nicht ab, sondern stellt Kanten stets als direkte Verbindungen dar. Dadurch wird das vom Algorithmus vorgesehene Layout nur teilweise realisiert.

5.2.3 Rendering- und Layouting-Performance

Die Rendering- und Layouting-Performance war ein durchgehender Aspekt bei der Entwicklung der Erweiterungen. Wie in Abschnitt 4.1.1 beschrieben, wies das Mermaid-Rendering bei größeren Diagrammen spürbare Verzögerungen auf, was als einer der Treiber für den Wechsel zu SvelteFlow diente. Darüber hinaus stellte die Integration des Layouting-Algorithmus eine potenzielle Quelle zusätzlicher Verzögerungen dar. Um die tatsächlichen Auswirkungen der Erweiterungen auf die Rendering- und Layouting-Dauern zu erfassen, wurden gezielte Messungen durchgeführt, deren Ergebnisse dieser Abschnitt vorstellt. Darüber hinaus wurde die allgemeine Interaktionsperformance, etwa bei der Bewegung von Knoten, als qualitativer Aspekt bei der Entwicklung berücksichtigt, ohne dass hierzu systematische Messungen durchgeführt wurden.

Gemessen wurden vier Messtypen: Mermaid-Rendering, SvelteFlow-Rendering ohne Layouting, SvelteFlow-Rendering mit Layouting sowie das Layouting isoliert. Die Messung erfolgte im Frontend über die Web-API-Funktion `performance.now()`, die einen hochauflösenden Zeitstempel in Millisekunden liefert. Der Startpunkt liegt für alle Rendering-Messtypen direkt nach dem Empfang der Backend-Daten im `renderingWrapper`, der Endpunkt jeweils nach Abschluss des entsprechenden

Vorgangs. Für das isolierte Layouting wurden die Zeitstempel um den `getLayoutedNodes`-Aufruf gesetzt. Je Messtyp und Diagrammgröße wurden 11 Läufe durchgeführt, wobei der erste Lauf verworfen wurde. Aus den verbleibenden 10 Messungen wurden Median und Standardabweichung berechnet.

Gemessen wurde mit drei Diagrammgrößen: einem konstruierten mittleren Diagramm mit 50 Nodes und 100 Edges, dem CGMES 3.0 EQ-Profil mit 210 Nodes und 232 Edges sowie einem konstruierten großen Diagramm mit 350 Nodes und 525 Edges. Die Messtypen erforderten dabei unterschiedliche Auslösemechanismen. Beim Mermaid-Rendering sowie beim SvelteFlow-Rendering ohne Layouting wurde das Diagramm durch wiederholtes Wechseln neu aufgerufen. Beim SvelteFlow-Rendering mit Layouting wurde das Diagramm für jede Messung neu importiert. Für das isolierte Layouting wurde die implementierte Reset-Layout-Funktion verwendet.

Tabelle 5.1 zeigt die Ergebnisse aller Messtypen und Diagrammgrößen als Median mit Standardabweichung in Millisekunden.

Tabelle 5.1: Rendering- und Layouting-Performance nach Messtyp und Diagrammgröße (Median ± Standardabweichung in ms)

Messtyp	**Diagramm 1**	**Diagramm 2**	**Diagramm 3**
Mermaid-Rendering	552 ± 20.8	2623 ± 283.6	3782 ± 160.6
SvelteFlow ohne Layouting	19 ± 4.9	73 ± 3.0	96 ± 1.8
SvelteFlow mit Layouting	1443 ± 149.9	2332 ± 161.6	22114 ± 2501
Layouting isoliert	1178 ± 24.8	1893 ± 306.5	22004 ± 2260

Die Ergebnisse zeigen, dass das reine SvelteFlow-Rendering ohne Layouting über alle Diagrammgrößen hinweg deutlich geringere Dauern aufweist als das Mermaid-Rendering. Der Anteil des Layoutings an der Gesamtdauer des SvelteFlow-Renderings mit Layouting ist dabei erheblich, wie der Vergleich mit dem isolierten Layouting-Messtyp verdeutlicht. Eine Bewertung dieser Ergebnisse sowie deren Auswirkungen auf die Nutzbarkeit erfolgt in Kapitel 6.

6 Evaluation

Die im Rahmen dieser Arbeit entwickelten Erweiterungen des RDF-Architects adressieren mehrere zentrale Aspekte der Diagrammvisualisierung und werden im Folgenden anhand der in Abschnitt 1.2 definierten Akzeptanzkriterien bewertet. Darüber hinaus werden Limitationen der Implementierung aufgezeigt, die im Rahmen dieser Arbeit nicht vollständig adressiert werden konnten.

Die Kriterien A1 und A2 betreffen die grundlegende Interaktivität sowie die Layout-Persistenz des Editors. A1 fordert, dass Diagrammelemente von Nutzer*innen interaktiv verschoben und angeordnet werden können, während A2 fordert, dass manuell erstellte Layouts nach dem Speichern und erneutem Laden erhalten bleiben. Wie in Abschnitt 5.2.1 dargestellt, wurden beide Funktionalitäten vollständig implementiert. Damit schafft die entwickelte Lösung die Voraussetzung für eine praxistaugliche, interaktive Diagrammbearbeitung, die über die statischen Möglichkeiten der bisherigen Mermaid.js-Implementierung wesentlich hinausgeht.

Kriterium A3 bewertet, ob der automatische Layoutalgorithmus für CIM-Modelle übersichtliche und nachvollziehbare Ausgangslayouts erzeugt. Der eingesetzte Layered-Algorithmus des ELK-Frameworks erfüllt dieses Kriterium. Durch die Optimierung zentraler Layoutziele wie Kantenkreuzungsminimierung, hierarchische Strukturierung und einheitliche Kantenrichtung erzeugt er Diagramme, die die strukturellen Abhängigkeiten zwischen CIM-Klassen klar ablesbar machen. Im Vergleich zur bisherigen Mermaid.js-Implementierung, deren Rendering keine interaktive Anpassung der Diagrammdarstellung ermöglichte, stellt dies eine wesentliche Verbesserung dar. Jedoch weist die aktuelle Implementierung eine relevante Einschränkung auf, die sowohl die Layoutqualität als auch die Benutzbarkeit betrifft. Wie in Abschnitt 5.1.3 ausführlich dargestellt, können die von ELK Layered berechneten Kantenpfade nicht auf das Rendering übertragen werden, da SvelteFlow keine native Unterstützung für gebrochene, mehrsegmentige Kanten bereitstellt. Eine eigene Implementierung wäre grundsätzlich möglich und wurde konzeptionell bereits ausgearbeitet, konnte jedoch im Rahmen dieser Arbeit nicht realisiert werden. Daraus ergeben sich gerade, undifferenzierte Kanten, die bei komplexeren Diagrammen zu visuellen Überschneidungen führen können. Grafische Editoren vergleichbarer Art bieten diese Funktionalität standardmäßig an, weshalb deren Fehlen aus Nutzer*innensicht eine Einschränkung darstellt.

Kriterium A4 bewertet, ob das Rendering von Diagrammen im Vergleich zur Mermaid.js-Implementierung performanter ist. Wie in Abschnitt 5.2.3 dokumentiert, wurden hierzu Laufzeitmessungen für die zentralen Rendering- und Lay-

outingvorgänge durchgeführt. Die Messungen liefern zwei konkrete Erkenntnisse. Das reine Rendering ohne Layouting fällt deutlich schneller aus als das bisherige Mermaid.js-Rendering, was darauf zurückzuführen ist, dass Mermaid.js bei jedem Rendervorgang neben der Layoutberechnung auch umfangreiches Post-Processing durchführt, wie in Abschnitt 3.5.1 beschrieben. Die Laufzeit des initialen Layoutings durch den ELK Layered Algorithmus erreicht hingegen eine vergleichbare Größenordnung wie das ehemalige Mermaid.js-Rendering. Da dieser Schritt jedoch ausschließlich beim erstmaligen Laden eines Diagramms sowie bei einer explizit angeforderten Layoutneuberechnung ausgeführt wird, fällt er im regulären Nutzungsbetrieb weniger ins Gewicht. Kriterium A4 ist damit als erfüllt zu bewerten.

Zusammenfassend erfüllt die entwickelte Lösung alle vier Akzeptanzkriterien und adressiert damit die in der Forschungsfrage gestellten Anforderungen an Interaktivität, automatisches Layouting, Layout-Persistenz und Rendering-Performance. Insbesondere die Kombination aus schnellem Rendering und einmalig berechnetem, persistent gespeichertem Layout erweist sich als tragfähiges Konzept für die praktische Nutzung im RDF-Architect. Das fehlende Edge Routing sowie das Verbesserungspotenzial beim Layouting stellen die wesentlichen Ansatzpunkte für eine Weiterentwicklung dar, die die Qualität der Diagrammdarstellung weiter steigern könnten.

7 Diskussion

Die in den vorangegangenen Kapiteln getroffenen methodischen Entscheidungen lassen sich nun, nach Abschluss der Implementierung und Evaluation, kritisch einordnen. Das vorliegende Kapitel untersucht, inwiefern das gewählte Vorgehen geeignet war, die Forschungsfrage zu beantworten, und wo Anpassungen zu besseren Ergebnissen hätten führen können.

Ein wesentlicher Bestandteil der Methodik war der strukturierte Auswahlprozess für die eingesetzten Technologien. Sowohl für die Rendering-Bibliothek als auch für den Layouting-Algorithmus wurden zunächst Auswahlkriterien definiert, potenzielle Kandidaten identifiziert, prototypisch erprobt und anschließend kriterienbasiert evaluiert. Dieses Vorgehen erwies sich als tragfähig, da die daraus resultierenden Entscheidungen für SvelteFlow und ELK Layered eine solide Basis für die Implementierung bildeten, was sich in den in Kapitel 5 erzielten Ergebnissen widerspiegelt. Einschränkend ist jedoch anzumerken, dass die Identifikation der Kandidaten überwiegend auf praxisorientierten Quellen aus Entwicklerkreisen basierte. Wissenschaftliche Vergleichsstudien zu Diagramm-Rendering-Bibliotheken oder Layouting-Algorithmen für UML-Klassendiagramme sind kaum verfügbar, weshalb dieser Ansatz pragmatisch vertretbar war. Eine stärkere Einbeziehung formaler Quellen hätte die Auswahlentscheidungen jedoch auf eine objektivere Grundlage gestellt und potenzielle Kandidaten möglicherweise vollständiger erfasst.

Kritischer zu bewerten ist die Vorgehensweise beim Fine-Tuning des ELK Layered Algorithmus. Aufgrund seiner Komplexität wurde von Beginn an auf eine vertiefte Auseinandersetzung mit seiner internen Funktionsweise verzichtet. Die Konfiguration erfolgte stattdessen durch systematisches Testen, ohne ein Verständnis der zugrundeliegenden Konzepte des Algorithmus aufzubauen. Rückblickend hätte ein frühes Einarbeiten in die konzeptionellen Grundlagen des Algorithmus, etwa anhand der von der Eclipse Foundation bereitgestellten Einführung in den Layered-Algorithmus [27], den Konfigurationsaufwand reduziert und den Spielraum für eine gezieltere Optimierung des Layoutergebnisses erweitert. Der explorative Ansatz führte nicht nur zu einem erhöhten Aufwand beim Fine-Tuning, sondern begrenzte auch das erreichbare Optimierungspotenzial.

Jedoch erwiesen sich die methodischen Entscheidungen dieser Arbeit insgesamt als geeignet, die Forschungsfrage zu beantworten. Der strukturierte Auswahlprozess hat sich dabei als verlässliches Instrument bewährt. Das Fine-Tuning des Layouting-Algorithmus zeigt jedoch, dass eine gezieltere initiale Auseinandersetzung mit der Dokumentation den Prozess effizienter gestaltet hätte.

8 Ausblick

Im Rahmen dieser Arbeit konnte gezeigt werden, wie interaktive Diagrammvisualisierung, automatisches Layouting und Layout-Persistenz in einem CIM-RDF-Editor realisiert werden können. Die entwickelte Lösung erfüllt die gesetzten Akzeptanzkriterien und schafft damit eine Grundlage für die interaktive Bearbeitung von UML-Klassendiagrammen im RDF-Architect. Dennoch eröffnet die Arbeit Fragestellungen, die im gesetzten Rahmen nicht vollständig adressiert werden konnten und Ansatzpunkte für weiterführende Arbeiten darstellen.

Während des Fine-Tunings des ELK Layered Algorithmus in Abschnitt 4.2.3 zeigte sich, dass objektive Qualitätsmetriken für automatisch generierte Layouts fehlten. Die Beurteilung, ob ein Layout gut oder schlecht ist, erfolgte dabei ausschließlich visuell und subjektiv. Metriken wie die Anzahl von Kantenkreuzungen, das Ausmaß von Knotenüberschneidungen, die Symmetrie des Diagramms oder die gleichmäßige Raumausnutzung könnten eine solche Beurteilung objektivieren und reproduzierbar machen. Di Bartolomeo et al. haben in ihrem systematischen Review Evaluierungsmethoden für Layoutalgorithmen umfassend aufgearbeitet und dabei sowohl rechnergestützte als auch nutzerzentrierte Ansätze beschrieben [45]. Die dort beschriebenen Methoden wurden jedoch für die Entwicklung und den Vergleich ganzer Algorithmen konzipiert. Ihre Übertragung auf die gezielte Qualitätsbewertung von Layoutkonfigurationen für domänenspezifische Diagrammtypen wie UML-Klassendiagramme in CIM-basierten Editoren stellt eine weiterführende Fragestellung dar, die auf den Erkenntnissen dieser Arbeit aufbauen könnte.

Eng damit verbunden ist die Frage nach systematischen Konfigurationen von Layoutalgorithmen für konkrete Diagrammtypen. Wie in Kapitel 7 reflektiert, erfolgte das Fine-Tuning des ELK Layered Algorithmus in dieser Arbeit durch systematisches Testen, ohne ein tieferes Verständnis der Abhängigkeiten zwischen den Parametern aufzubauen. Eine Arbeit, die diesen Prozess in den Mittelpunkt stellt und methodisch fundiert untersucht, welche Konfigurationen des ELK Layered Algorithmus für UML-Klassendiagramme optimale Ergebnisse liefern, existiert nach aktuellem Kenntnisstand nicht. Dabei könnte eine solche Untersuchung über UML-Klassendiagramme hinausgehen und weitere etablierte Diagrammtypen einbeziehen, mit dem Ziel, erprobte Ausgangskonfigurationen bereitzustellen, die Entwickler*innen als Grundlage für den jeweiligen Anwendungsfall dienen. Das KIELER Research Project der Universität zu Kiel, das ELK sowie die zugehörigen Layoutalgorithmen entwickelt und pflegt, bietet dabei einen naheliegenden Forschungskontext für eine solche Untersuchung.

Eine weitere Fragestellung ergibt sich aus dem Verhältnis zwischen der implementierten Layout-Persistenz und dem Potenzial des zugrunde liegenden Standards. Das CGMES DiagramLayout-Profil stellt ein standardisiertes Vokabular bereit, das in seiner vollen Ausprägung mehr Ausdrucksmöglichkeiten bietet, als im Rahmen dieser Arbeit genutzt wurden. Ob ein umfassenderer Einsatz dieses Vokabulars den Informationsgehalt der gespeicherten Layoutdaten so erweitern könnte, dass ein standardisierter Austausch von Diagrammlayout-Daten zwischen unterschiedlichen CGMES-konformen Werkzeugen realistisch wird, stellt eine offene Forschungsfrage dar. Darüber hinaus bleibt ungeklärt, wie Layoutinformationen aus werkzeugspezifischen Formaten, etwa dem Export des Enterprise Architect, auf das CGMES DiagramLayout-Profil abgebildet werden könnten und welche Anforderungen ein portables Diagrammlayout im Kontext der Energiewirtschaft erfüllen müsste.

Die genannten Fragestellungen knüpfen unmittelbar an die in dieser Arbeit gewonnenen Erkenntnisse an und zeigen, dass die entwickelte Lösung nicht nur einen praktischen Beitrag für den RDF-Architect leistet, sondern darüber hinaus Anknüpfungspunkte für weiterführende Forschung im Bereich der domänenspezifischen Diagrammvisualisierung eröffnet.

9 Fazit

Im Rahmen dieser Arbeit wurde der RDF-Architect der SOPTIM AG um eine interaktive Diagrammvisualisierung mit automatischem Layouting und Layout-Persistenz erweitert. Ausgangspunkt war die bestehende Mermaid.js-basierte Visualisierung, die weder eine interaktive Manipulation von Diagrammelementen noch eine manuelle Layout-Anpassung ermöglichte und bei größeren CIM-Modellen messbare Einschränkungen in der Rendering-Performance aufwies. Die entwickelte Lösung ersetzt diesen Ansatz durch ein kohärentes Gesamtkonzept auf Basis von SvelteFlow, ELK Layered und dem CGMES 3.0 DiagramLayout-Profil.

Das methodische Vorgehen gliederte sich in drei aufeinanderfolgende Entwicklungsphasen, denen jeweils eine strukturierte Anforderungsanalyse sowie eine kriterienbasierte Evaluation der infrage kommenden Technologien vorausging. Für das Rendering fiel die Entscheidung auf SvelteFlow, für das Layouting auf ELK Layered, der durch iteratives Fine-Tuning auf die spezifischen Anforderungen von CIM-UML-Klassendiagrammen abgestimmt wurde. Die Layout-Persistenz wurde auf Basis des CGMES 3.0 DiagramLayout-Profils realisiert. Diese drei Komponenten bilden zusammen ein kohärentes Gesamtkonzept, das interaktive Bearbeitung, automatisches Layouting und Layout-Persistenz im RDF-Architect vereint.

Die in Abschnitt 1.2 definierten Akzeptanzkriterien wurden durch die entwickelte Lösung erfüllt, wie in Kapitel 6 dargelegt. Die Kombination aus schnellem Rendering, einmalig berechnetem Ausgangslayout und persistent gespeicherten Diagrammpositionen erweist sich dabei als tragfähiges Konzept für den praktischen Einsatz. Als wesentliche verbleibende Einschränkung ist das fehlende Edge Routing zu nennen, das dazu führt, dass die von ELK Layered berechneten Kantenpfade nicht auf die Darstellung übertragen werden.

Die in Kapitel 8 aufgezeigten Anknüpfungspunkte verdeutlichen, dass die entwickelte Lösung über ihren unmittelbaren Beitrag hinaus Fragestellungen eröffnet, die weiterführender Untersuchung bedürfen. Die vorliegende Arbeit stellt damit einen Schritt in der Weiterentwicklung des RDF-Architects dar und legt eine Grundlage, auf der zukünftige Arbeiten aufbauen können. Damit leistet sie einen Beitrag zu einem modernen, interaktiven Werkzeug für die Modellierung CIM-konformer Datenmodelle, das den Anforderungen der Energiewirtschaft an eine effiziente und standardkonforme Arbeit mit UML-Klassendiagrammen gerecht wird.

Abbildungsverzeichnis

3.1 Hauptansicht des RDF-Architects 14
3.2 Klassendiagramm des CGMES DiagramLayout-Profils 16
3.3 Beispielhaftes UML-Klassendiagramm 25

5.1 Rendering-Ablauf und Systemarchitektur des RDF-Architect 39
5.2 Darstellung der Nutzerfunktionen im RDF-Architect 50
5.3 Layoutdarstellung eines Diagramms im Mermaid-Rendering 52
5.4 Layoutdarstellung eines Diagramms mit dem ELK Layered Algorithmus 52

A.1 Layoutdarstellung eines Diagramms im Mermaid-Rendering (vergrößert) 71

Tabellenverzeichnis

5.1 Rendering- und Layouting-Performance nach Messtyp und Diagrammgröße (Median ± Standardabweichung in ms) 54

Listingverzeichnis

3.1 Beispiel einer RDF-Ressource in Turtle-Syntax 9
3.2 Beispiel einer SPARQL SELECT-Anfrage 9
3.3 Beispiel einer CIM-Klasse 12
3.4 Beispiel eines CIM-Packages 12
3.5 Beispiel eines Diagram im CGMES DiagramLayout-Profil 17
3.6 Beispiel eines DiagramObject im CGMES DiagramLayout-Profil . . 18
3.7 Beispiel eines DiagramObjectPoint im CGMES DiagramLayout-Profil 18

4.1 Beispiel eines DiagramObjects für eine CIM-Klasse im CGMES DiagramLayout-Profil 37

5.1 Methode `assembleNodeDTO` zur Assemblierung eines `NodeDTO` im `RenderCIMCollectionSvelteFlowService` 42
5.2 Template der Komponente `AssociationEdge` 43
5.3 SPARQL-Query zum Abrufen von DiagramObjectPoints und Class UUIDs eines Diagramms 47
5.4 Methode `insertDiagram` in `DLUpdates` zum Anlegen eines Diagram 48

A.1 Vom Backend übertragenes JSON-Format für das SvelteFlow-Rendering (reduziert) 69
A.2 Finale Konfiguration des ELK Layered Algorithmus 70

Literatur

[1] „ENTSO-E," besucht am 13. Feb. 2026. Adresse: https://www.entsoe.eu/about/inside-entsoe/mission-statement/

[2] „CGMES," besucht am 13. Feb. 2026. Adresse: https://www.entsoe.eu/digital/common-information-model/cim-for-grid-models-exchange/

[3] „International Electrotechnical Commission moves Common Information Model to Sparx Systems Enterprise Architect.," besucht am 13. Feb. 2026. Adresse: https://sparxsystems.com/press/articles/iec.html

[4] H. Metin und D. Bork, „Introducing BIGUML: A Flexible Open-Source GLSP-Based Web Modeling Tool for UML," in *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2023, S. 40–44. DOI: 10.1109/MODELS-C59198.2023.00016 besucht am 16. Feb. 2026. Adresse: https://ieeexplore.ieee.org/abstract/document/10350769

[5] N. Krieger, S. Speth und S. Becker, „HyLiMo: A Hybrid Live-Synchronized Modular Diagramming Editor as IDE Extension for Technical and Scientific Publications," in *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments*, Ser. IDE '24, Lisbon, Portugal: Association for Computing Machinery, 2024, S. 70–73, ISBN: 9798400705809. DOI: 10.1145/3643796.3648458 besucht am 16. Feb. 2026. Adresse: https://dl.acm.org/doi/abs/10.1145/3643796.3648458

[6] S. Lohmann, S. Negru, F. Haag und T. Ertl, „Visualizing ontologies with VOWL," *Semantic Web*, Jg. 7, Nr. 4, S. 399–419, 2016. DOI: 10.3233/SW-150200 eprint: https://journals.sagepub.com/doi/pdf/10.3233/SW-150200. besucht am 16. Feb. 2026. Adresse: https://journals.sagepub.com/doi/full/10.3233/SW-150200

[7] „RDF 1.1 Concepts and Abstract Syntax," besucht am 20. Feb. 2026. Adresse: https://www.w3.org/TR/rdf11-concepts/

[8] „RDF Resources and Statements," besucht am 20. Feb. 2026. Adresse: https://www.w3.org/TR/rdf11-concepts/#resources-and-statements

[9] „RDF Triples," besucht am 20. Feb. 2026. Adresse: https://www.w3.org/TR/rdf11-concepts/#section-triples

[10] „RDF IRIs," besucht am 20. Feb. 2026. Adresse: https://www.w3.org/TR/rdf11-concepts/#section-IRIs

[11] „RDF Vocabularies and Namespaces,“ besucht am 20. Feb. 2026. Adresse: https://www.w3.org/TR/rdf11-concepts/#vocabularies
[12] „RDF Literals,“ besucht am 20. Feb. 2026. Adresse: https://www.w3.org/TR/rdf11-concepts/#section-Graph-Literal
[13] „RDF 1.1 Turtle Format,“ besucht am 20. Feb. 2026. Adresse: https://www.w3.org/TR/turtle/
[14] „SPARQL 1.1 Query Language,“ besucht am 23. Feb. 2026. Adresse: https://www.w3.org/TR/sparql11-query/
[15] „SPARQL Graph Patterns,“ besucht am 23. Feb. 2026. Adresse: https://www.w3.org/TR/sparql11-query/#GraphPattern
[16] „SPARQL SELECT Query,“ besucht am 23. Feb. 2026. Adresse: https://www.w3.org/TR/sparql11-query/#select
[17] „IEC Common Information Model,“ besucht am 25. Feb. 2026. Adresse: https://www.iec.ch/blog/iec-common-information-model-under-spotlight-1
[18] „Enterprise Architect,“ besucht am 26. März 2026. Adresse: https://sparxsystems.com/products/ea/
[19] „CGMES 3.0 DiagramLayout Profil.“ Relevante Datei im Archiv: CGMES/CurrentRelease/RDFS/61970-600-2_Diagram-Layout-AP-Voc-RDFS2020.rdf, besucht am 27. Feb. 2026. Adresse: https://github.com/entsoe/application-profiles-library/releases/tag/v1.1.1
[20] E. Würgler und A. McMoran, „CIM Diagram Layout Profile for graphics exchange,“ in *2011 IEEE Power and Energy Society General Meeting*, 2011, S. 1–5. DOI: 10.1109/PES.2011.6039444 besucht am 27. Feb. 2026.
[21] „Mermaid.js,“ besucht am 27. Feb. 2026. Adresse: https://mermaid.js.org/
[22] „Svelvet,“ besucht am 27. Feb. 2026. Adresse: https://www.svelvet.io/
[23] „SvelteFlow,“ besucht am 27. Feb. 2026. Adresse: https://svelteflow.dev/
[24] „Dagre,“ besucht am 1. März 2026. Adresse: https://github.com/dagrejs/dagre
[25] „Eclipse Layout Kernel,“ besucht am 1. März 2026. Adresse: https://eclipse.dev/elk/
[26] „ELK Layered,“ besucht am 1. März 2026. Adresse: https://eclipse.dev/elk/reference/algorithms/org-eclipse-elk-layered.html
[27] „ELK Layered Phases overview,“ besucht am 12. März 2026. Adresse: https://eclipse.dev/elk/blog/posts/2025/25-08-21-layered.html

[28] „elkjs,“
besucht am 1. März 2026. Adresse: https://github.com/kieler/elkjs
[29] „Spring Boot,“ besucht am 1. März 2026. Adresse:
https://spring.io/projects/spring-boot
[30] „Spring Framework,“ besucht am 1. März 2026. Adresse:
https://spring.io/projects/spring-framework
[31] „Apache Jena RDF API,“ besucht am 1. März 2026. Adresse:
https://jena.apache.org/documentation/rdf/index.html
[32] „Svelte,“ besucht am 1. März 2026. Adresse: https://svelte.dev/
[33] „SvelteKit,“ besucht am 1. März 2026. Adresse:
https://svelte.dev/docs/kit/introduction
[34] „UML,“ besucht am 1. März 2026. Adresse:
https://www.omg.org/uml/what-is-uml.htm
[35] „UML Klassendiagramme,“ besucht am 1. März 2026. Adresse:
https://www.geeksforgeeks.org/system-design/unified-modeling-language-uml-class-diagrams/
[36] „Representational State Transfer,“ besucht am 1. März 2026. Adresse:
https://www.ibm.com/de-de/think/topics/rest-apis
[37] J. Cabot. „20+ JavaScript libraries to draw your own diagrams (2024 edition),“ besucht am 6. März 2026. Adresse: https://modeling-languages.com/javascript-drawing-libraries-diagrams/
[38] XYFlow. „A curated list with resources about node-based UIs,“
besucht am 6. März 2026. Adresse:
https://github.com/xyflow/awesome-node-based-uis
[39] „JointJS,“ besucht am 6. März 2026. Adresse: https://www.jointjs.com/
[40] „maxGraph,“ besucht am 6. März 2026. Adresse:
https://maxgraph.github.io/maxGraph/
[41] „D3.js,“ besucht am 6. März 2026. Adresse: https://d3js.org/
[42] „SvelteFlow layouting libraries,“ besucht am 10. März 2026. Adresse:
https://svelteflow.dev/learn/layouting/layouting-libraries
[43] „ELK Stress,“ besucht am 25. März 2026. Adresse:
https://eclipse.dev/elk/reference/algorithms/org-eclipse-elk-stress.html
[44] „SvelteFlow EasyConnect example,“ besucht am 18. März 2026. Adresse:
https://svelteflow.dev/examples/nodes/easy-connect
[45] S. Di Bartolomeo, T. Crnovrsanin, D. Saffo, E. Puerta, C. Wilson und C. Dunne, „Evaluating Graph Layout Algorithms: A Systematic Review of Methods and Best Practices,“
Computer Graphics Forum, Jg. 43, Nr. 6, e15073, 2024.
DOI: https://doi.org/10.1111/cgf.15073 eprint:

https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.15073.
Adresse:
https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.15073

Einsatz generativer KI

Für die Erstellung dieser Bachelorarbeit wurde die generative künstliche Intelligenz Claude des Herstellers Anthropic genutzt, konkret die Modelle Sonnet 4.5 sowie Opus 4.5. Der Schwerpunkt des Einsatzes lag auf der sprachlichen Ausformulierung von Textabschnitten. Die inhaltliche Konzeption der Arbeit hingegen, einschließlich Kapitelstruktur, thematischer Gliederung und fachlicher Inhalte, lag vollständig beim Verfasser. Sämtliche KI-generierten Formulierungen durchliefen eine kritische Prüfung hinsichtlich inhaltlicher Korrektheit und wurden bei Bedarf überarbeitet. Damit folgt diese Arbeit den Grundsätzen wissenschaftlicher Integrität, wie sie die Deutsche Forschungsgemeinschaft (DFG) in ihrer Stellungnahme von September 2023 formuliert hat. Transparenz über den KI-Einsatz sowie die uneingeschränkte Verantwortung des Verfassers für Inhalt und Form der Arbeit bleiben dabei gewahrt.

A Anhang

```
{
   "edges": [
      {
         "data": null,
         "id": "a9ef74e9-...",
         "source": "fd36fcc3-..",
         "target": "126ba900-..",
         "type": "inheritance"
      },
      {
         "data": {
            "fromMultiplicity": "0..n",
            "toMultiplicity": "1",
            "useFromAssociation": true,
            "useToAssociation": false
         },
         "id": "9caf3a8d-...",
         "source": "fd36fcc3-...",
         "target": "215c69c5-...",
         "type": "association"
      }, ...
   ],
   "format": "SVELTEFLOW",
   "nodes": [
      {
         "data": {
            "attributes": [
               {
                  "label": "orientation",
                  "multiplicity": "1..1",
                  "type": "OrientationKind"
               },
               {
                  "label": "x1InitialView",
                  "multiplicity": "0..1",
                  "type": "Simple_Float"
               }, ...
            ],
            "belongsToCategory": "DiagramLayout",
            "enumEntries": [],
            "label": "Diagram",
            "stereotypes": []
         },
         "id": "fd36fcc3-...",
         "position": {
            "x": 0,
            "y": 0
         },
         "type": "class"
      }, ...
   ]}
```

Listing A.1: Vom Backend übertragenes JSON-Format für das SvelteFlow-Rendering (reduziert)

```
layoutOptions: {
    //BASE
    "elk.algorithm": "layered",
    "elk.aspectRatio": "1.78f",
    "elk.edge.thickness": "2.0",
    "elk.direction": "RIGHT",
    "elk.layered.thoroughness": "150",
    "elk.edgeRouting": "POLYLINE",
    "elk.layered.slopedEdgeZoneWidth": "0.0",
    "elk.separateConnectedComponents": "false",
    "elk.layered.mergeHierarchyEdges": "false",

    //NODE PLACEMENT
    "elk.layered.nodePlacement.strategy": "NETWORK_SIMPLEX",
    "elk.layered.nodePlacement.favorStraightEdges": "false",

    //CROSSING MINIMIZATION
    "elk.layered.crossingMinimization.greedySwitchType":
        "TWO_SIDED",
    "elk.layered.greedySwitch.activationThreshold": "40",

    //NODE PROMOTION
    "elk.layered.layering.nodePromotion.strategy":
        "NIKOLOV_IMPROVED",
    "elk.layered.layering.nodePromotion.maxIterations": "20",

    //NODE LAYERING
    "elk.layered.layering.strategy": "STRETCH_WIDTH",

    //HIGH DEGREE NODES
    "elk.layered.highDegreeNodes.treatment": "true",
    "elk.layered.highDegreeNodes.threshold": "10",
    "elk.layered.highDegreeNodes.treeHeight": "5",

    //SPACING
    "elk.layered.spacing.edgeEdgeBetweenLayers": "20",
    "elk.layered.spacing.edgeNodeBetweenLayers": "40",
    "elk.spacing.edgeNode": "30",
    "elk.spacing.edgeEdge": "15",
    "elk.layered.spacing.nodeNodeBetweenLayers": "80",
    "elk.spacing.nodeNode": "60",
}
```

Listing A.2: Finale Konfiguration des ELK Layered Algorithmus

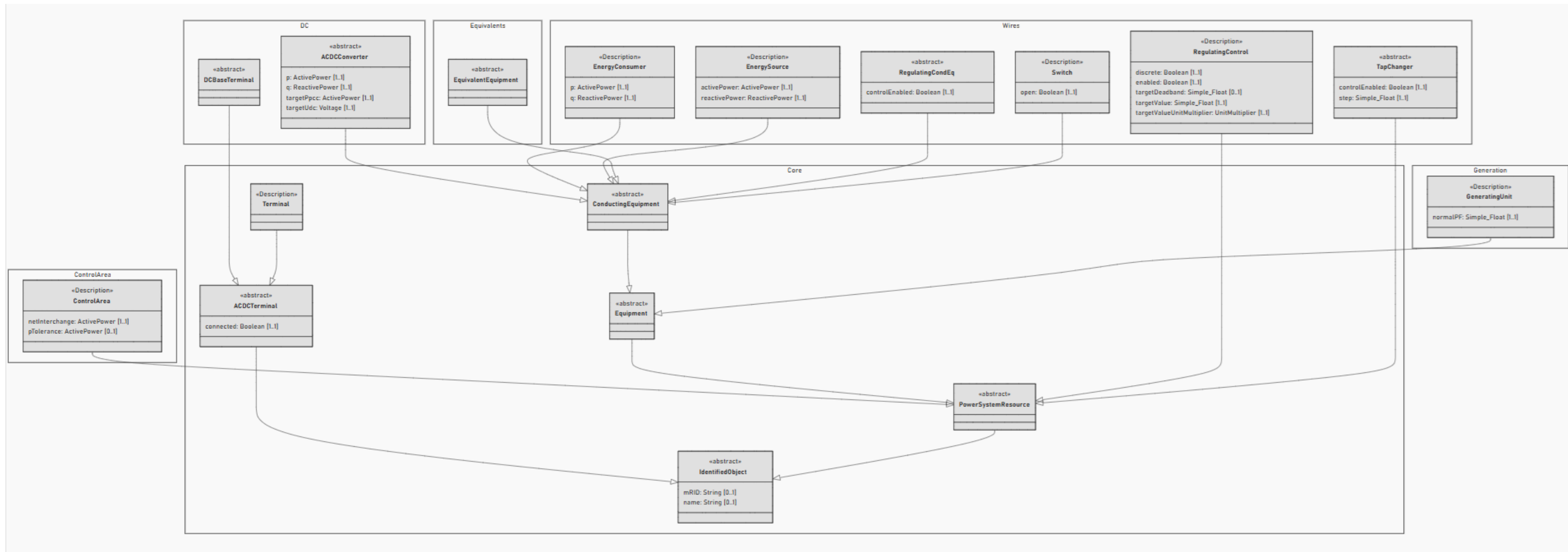

Abbildung A.1: Layoutdarstellung eines Diagramms im Mermaid-Rendering (vergrößert)