Entwicklung einer Softwarearchitektur zur Strukturierung und Verwaltung von FAIRen und einfach erkundbaren (FAIREr) Wissensgraphen unter der Verwendung von Semantic Units und Knowlegde Graph Building Blocks

Fakultät für Elektrotechnik und Informatik
Institut für Verteilte Systeme – Data Science und Digital Libraries
Gottfried Wilhelm Leibniz Universität Hannover

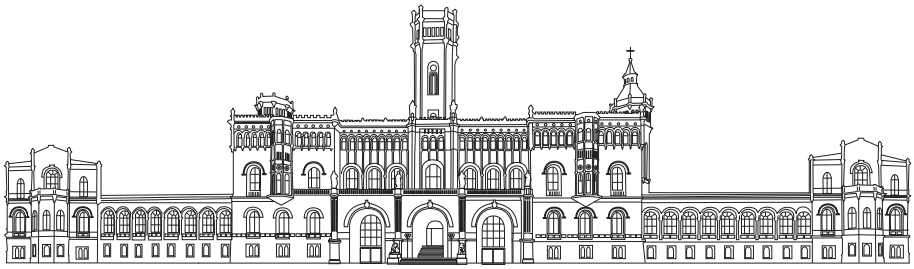
Masterarbeit

im Studiengang Informatik

von

Marcel Konrad

Matrikelnummer : 10004719



Erstprüfer: Prof. Dr. Sören Auer

Zweitprüfer: Dr. Lars Vogt

Betreuer: Dr. Lars Vogt

18. Juli 2022

Erklärung der Selbständigkeit

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden, alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind, und die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt habe.

_____ Hannover, 18. Juli 2022

Marcel Konrad

Danksagung

Zuerst möchte ich meinem Betreuer und Zweitprüfer Dr. Lars Vogt danken, für seine kontinuierliche Unterstützung, Geduld und Wegweisung. Ohne ihn wäre diese Arbeit nicht möglich gewesen. Außerdem möchte ich mich bei Manuel Prinz bedanken, der stets bei technischen Fragen zur Seite stand. Des Weiteren möchte ich mich bei Dr. Markus Stocker bedanken, für die anfängliche Themenfindung und seine kontinuierliche Unterstützung über die Dauer dieser Arbeit. Zudem bedanke ich mich bei Dr. Sören Auer, für seine Bereitschaft, meine Arbeit zu prüfen und mich an seinem Institut zu betreuen. Abschließend möchte ich mich bei meiner Familie bedanken für die bedingungslose Unterstützung und kontinuierliche Ermutigung während meines Studiums.

Abstract

The FAIR Guiding Principles (Findable, Accessible, Interoperable and Reusable) [1] are becoming an increasingly important factor in science and industry, leading to a new standard in the development of Knowledge Graph applications. Current implementations of Knowledge Graph applications rely on complex ontologies in combination with abstract graph patterns, binding their data and metadata to a specific data model to be FAIR, which in turn, makes it harder to compare to data and metadata modelled in another Knowledge Graph that uses different ontologies and different graph patterns [2]. It also ties the data of the Knowledge Graph application to a specific database technology and requires users to semantically model their data, which represents an entry barrier for non-expert users. Additionaly, the querying of the Knowledge Graph in order to retrieve data and metadata often requires users to learn a specific graph query language such as SPARQL or Cypher to interact with the Knowledge Graph, hindering the broader usage of Knowledge Graph applications. Furthermore, making statements about statements in a Knowledge Graph is a tedious process with existing technologies like RDF reification [3] or RDF-star [4, 5]. In this thesis, we explore the capabilities of Semantic Units [6] and Knowledge Graph Building Blocks [7]; and we argue that they address all of the above problems with current Knowledge Graph applications and provide new and intuitive ways for users to define their own Knowledge Graph applications that are truly FAIR. First, we define the architecture used to build the so-called KGBB-Engine that serves as a highly modular and expandable framework for tools and applications driven by Knowledge Graph Building Blocks. We then implement that KGBB-Engine for a specific subset of Knowledge Graph Building Blocks, along with two distinct implementations for persistent database storage. These include an implementation for an RDF Triple store, accessed via a SPARQL-endpoint, and an implementation for the labeled-property-graph database Neo4j, which is accessed via Cypher-queries. Then, we use that generic KGBB-Engine implementation to create a KGBB-driven Knowledge Graph application for a specific use-case, and implement a webbased user interface for it, that allows users to manage and create new Material Entity resources in a knowledge graph, and further describe their parthood relations and weight measurements in an intuitive way. Finally, we compare our approach of a Knowledge Graph application, using Semantic Units and Knowledge Graph Building Blocks, to other applications that try to solve similar aspects of the problems we face with Knowledge Graph Management Systems.

Keywords: Knowledge Graph, Semantic Unit, Knowledge Graph Building Block, FAIR

Zusammenfassung

Die FAIR-Leitprinzipien (Findable, Accessible, Interoperable and Reusable) [1] gewinnen in Wissenschaft und Industrie zunehmend an Bedeutung und tragen zu einem neuen Standard in der Entwicklung von Wissensgraphapplikationen bei. Derzeitige Implementierungen von Wissensgraphapplikationen stützen sich auf komplexe Ontologien in Kombination mit abstrakten Graphenmustern und binden ihre Daten und Metadaten an ein bestimmtes Datenmodell, um FAIR zu sein, was wiederum den Vergleich mit Daten und Metadaten erschwert, die in einem anderen Wissensgraph modelliert sind, der andere Ontologien und andere Graphenmuster verwendet [2]. Außerdem werden die Daten der Wissensgraphapplikation an eine bestimmte Datenbanktechnologie gebunden, und die Benutzer müssen ihre Daten semantisch modellieren, was eine Einstiegshürde für nicht erfahrene Benutzer darstellt. Darüber hinaus erfordert eine Datenabfrage des Wissensgraphen häufig das Erlernen einer speziellen Graphenabfragesprache wie SPARQL oder Cypher, was einer breiten Nutzung von Wissengsgraphen entgegensteht. Des Weiteren ist die Erstellung von Aussagen über Aussagen in einem Wissensgraphen ein umständlicher Prozess, mit bestehenden Technologien wie RDF Reification [3] oder RDF-star [4, 5]. In dieser Arbeit untersuchen wir die Anwendungsmöglichkeiten von Semantic Units [6] und Knowledge Graph Building Blocks [7]; und wir argumentieren, dass sie sämtliche der oben genannten Probleme aktueller Wissensgraphapplikationen adressieren, und intuitive Wege für Nutzer bieten, ihre eigenen Wissensgraphapplikationen zu definieren, die wirklich FAIR sind. Zunächst definieren wir die Architektur, die zum Aufbau der sogenannten KGBB-Engine verwendet wird, und eine hoch-modulare und erweiterbare Plattform für Knowledge Graph Building Blocks getriebene Applikationen und Werkzeuge bereitstellt. Anschließend implementieren wir diese KGBB-Engine für eine bestimmte Auswahl von Knowledge Graph Building Blocks, zusammen mit zwei verschiedenen Implementierungen für eine persistente Datenbankspeicherung. Diese beinhalten eine Implementierung für einen RDF Triple Store, auf den über einen SPARQL-Endpunkt zugegriffen wird, und eine Implementierung für die Labeled-Property-Graph-Datenbank Neo4j, auf die über Cypher-Abfragen zugegriffen wird. Danach verwenden wir die generische Implementierung der KGBB-Engine, um eine KGBB-gesteuerte Wissensgraphapplikation für einen spezifischen Use-Case zu erstellen, und entwickeln eine webbasierte Benutzerschnittstelle, die ermöglicht es den Benutzern ermöglicht, neue Material-Entity-Ressourcen in einem Wissensgraphen zu verwalten und zu erstellen, und ihre Parthood-Beziehungen und Gewichtsmessungen auf intuitive Weise zu beschreiben. Schließlich vergleichen wir unseren Ansatz

einer Wissensgraphapplikation mit anderen Wissensgraphapplikationen, die versuchen, vergleichbare Probleme mit aktuellen Wissensgraph Management Applikationen zu lösen.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Akronyme

CSV Comma Separated Values. 11

FAIR Findable, Accessible, Interoperable, Reusable. VII, IX, 1, 2, 9, 16, 17, 57, 61

FAIREr Findable, Accessible, Interoperable, Reusable, Explorability raised. 16, 17, 61

KGBB Knowledge Graph Building Block. VII, IX, XI, XII, XVII, 15, 16, 21–25, 29–43, 46–52, 54–61

NIIU Named Individual Identification Unit. 13, 22–24, 33–38, 52, 53, 58

OBDA Ontology Based Data Access. 3

ORKG Open Research Knowledge Graph [8]. 29

R2RML RDB to RDF Mapping Language [9]. 3, 5, 58, 59

RDF Resource Description Framework [10]. VII, IX, 1–3, 5, 9, 11, 43–45, 47, 61

REST Representational State Transfer. 5

SPARQL SPARQL Protocol And RDF Query Language. VII, IX, 2–5, 7, 8, 43, 44, 57, 59, 61

TSV Tab Separated Values. 11

UPRI Uniform Persistent Resource Identifier. 1, 11–13, 30, 32, 43, 44, 46, 48, 51, 54, 55, 61

URI Uniform Resource Identifier. 9, 22, 24–28, 33–38, 44, 46, 47, 51–55

URL Uniform Resource Locator. 50

YAML YAML Ain't Markup Language. 9, 11

Glossar

Cypher Abfragesprache für Labeled Property Graphen [11, 12]. VII, IX, 46–50, 57, 61

Gradle Java basierendes Open-Source Build-Management-Automatisierungs-Tool. 29, 56

GraphQL Open-Source-Datenabfrage- und Manipulationssprache. 11

HTML Textbasierte Auszeichnungssprache für Webdukumente. 5–7, 50, 59

Java Objektorientierte, kompilierte, statisch typisierte Programmiersprache. 29

Java Virtual Machine Virtuelle Laufzeitumgebung. 29, 39

JSON Kompaktes Datenaustauschformat. 11

JSON-LD JavaScript Object Notation für Linked Data. 11

JSON-schema Schemasprache für JSON [13]. 11

Kotlin Objektorientierte, kompilierte, statisch typisierte Programmiersprache. 29

Markdown Leicht lesbare Auszeichnungssprache. 11

OWL Formale Beschreibungssprache für Ontologien [14]. 3, 9

Python Objektorientierte, interpretierte, imperative, dynamisch typisierte Programmiersprache. 11, 39

SHACL Beschreibungssprache für RDF Graphen [15]. 6

ShEx Strukturelle Schemasprache für RDF-Graphen [16]. 11

SQL DDL Datenbearbeitungssprache für relationale Datenbanken. 11

TriG RDF Serialisierungssprache. Erweiterung zu Turtle. 45

Turtle RDF Serialisierungssprache [17]. XXI, 11

1 Einleitung

Die Prinzipien von FAIR haben in den letzten Jahren zunehmend an Aufmerksamkeit gewonnen und definieren eine Reihe von Leitlinien für wissenschaftliches Datenmanagement, wodurch die Findbarkeit (Findability), Zugänglichkeit (Accessibility), Interoperabilität (Interoperability) und Wiederverwendbarkeit (Reusability) von Daten erhöht werden soll [1]. In 2018 schätzte eine Kosten-Nutzen-Analyse im Auftrag der Europäischen Union (EU) für FAIRe Forschungsdaten, dass jährlich ein Schaden von ca. 10,2 Milliarden Euro für die europäische Wirtschaft entsteht, wenn Forschungsdaten und Metadaten nicht den Prinzipien von FAIR folgen. Gleichzeitig könnten FAIRe Forschungsdaten einen positiven Einfluss auf die Datenqualität und die Lesbarkeit für Maschinen haben, sodass ein zusätzlicher Vorteil von FAIRen Forschungsdaten für die europäische Wirtschaft in Höhe von 16 Milliarden Euro geschätzt wird [18]. Aufgrund dessen empfiehlt die Expertengruppe der European Open Science Cloud den Ausbau eines Internets aus FAIRen Daten und Diensten (Internet of FAIR Data and Services) [19].

Wissensgraphen spielen eine immer größer werdende Rolle für die Verarbeitung von Daten und Metadaten in Forschungseinrichtungen und Unternehmen [20–22], und stellen in Kombination mit Ontologien eine vielversprechende technische Lösung für die Umsetzung von FAIRen Wissens-Management-Applikation dar [23, 24]. Für die technische Umsetzung von Wissensgraphen gibt es bereits mehrere Ansätze. Einer dieser Ansätze sind Labeled-Property Graphen [25, 26], welche sich aus einer Menge von Knoten und gerichteten Kanten zusammensetzen, wobei zusätzlich für jeden Knoten und jede Kante eine beliebige Anzahl von Key-Value-Paaren definiert werden kann. Ein weiterer Ansatz ist das Resource Description Framework [10] (RDF). In RDF setzt sich ein Wissensgraph aus einer Menge von Klassen, Instanzen und Relationen zusammen, die jeweils durch eine eigene Uniform Persistent Resource Identifier (UPRI) repräsentiert werden, wobei jede Relation in Form eines Tripels (Subjekt, Prädikat, Objekt) dokumentiert wird. Dadurch, dass Wissensgraphen nicht an ein bestimmtes Datenmodell gebunden sind, erzielen sie im Vergleich zu relationalen Datenbanken eine bessere Performance bei Abfragen auf stark vernetzten Daten, was bei Forschungsdaten und Metadaten häufig der Fall ist. Jedoch reicht die Verwendung eines Wissensgraphen zum Dokumentieren und Speichern von Daten und Metadaten allein nicht aus, um ihre FAIRness zu garantieren. Zusätzlich muss gewährleistet sein, dass Daten und Metadaten vom gleichen Typus im Wissensgraphen vergleichbar modelliert werden. D.h., dass das gleiche Graphenmuster bei der gleichen Art von Daten und Metadaten verwendet

werden muss. Außerdem sollten Daten mit umfangreichen Metadaten versehen werden und in sogenannte FAIR Digital Objects [27, 28] organisiert sein. Eine weitere Herausforderung im Umgang mit Wissensgraphen ist die intuitive Erkundung der Daten. Je weiter die Daten eines Wissensgraphen für die Verarbeitung von Maschinen optimiert werden, umso mehr Tripel müssen dem Graphen zusätzlich hinzugefügt werden und umso schwieriger wird es für einen menschlichen Benutzer, den Graphen zu erkunden [7]. Abhilfe würden hier nur intuitiv nutzbare Werkzeuge leisten können, die den Benutzer in die Lage versetzten, die Menge an Daten auf diejenigen Informationen zu reduzieren, die ihn gerade interessieren. Zudem kann es zu Schwierigkeiten kommen, wenn ein Benutzer Daten aus verschiedenen Wissensgraphen miteinander vergleichen möchte, da diese nicht zwingend mit demselben Datenschema modelliert wurden, und somit die Vergleichbarkeit, sowie die Interoperabilität der Daten eingeschränkt ist.

RDF wird bereits von einer großen Anzahl an Werkzeugen und Applikationen unterstützt. Jedoch bringt die Verwendung von RDF auch eine Reihe von Limitationen mit sich, die den Umgang mit Wissensgraphen für unerfahrene Benutzer erschweren. So ist etwa das Erstellen von Aussagen über Aussagen mit bisherigen Lösungsansätzen, wie RDF Reification [3] oder RDF-star [4, 5], zwar technisch möglich, aber kompliziert in der Anwendung, wenn größere Teilgraphen referenziert werden müssen, da die Anzahl an benötigter Tripel für eine Modellierung sehr schnell steigt. Des Weiteren erfolgt die Abfrage von Daten in einem RDF basierten Wissensgraphen über sogenannte SPARQL-Abfragen [29], die einen zusätzlichen Lernaufwand für unerfahrene Benutzer bedeuten, und somit eine Eintrittsbarriere in der Interaktion mit dem Wissengraphen darstellen [30].

Die Konzepte der Semantic Units und Knowledge Graph Building Blocks versuchen, den Problemen der bisherigen Umsetzungen von Wissensgraphapplikationen entgegenzuwirken und gleichzeitig ihre Erkundbarkeit zu steigern, in dem der Wissensgraph in semantisch sinnvolle Teilgraphen strukturiert wird. Außerdem ermöglichen sie eine einfache Datenmodellierung, sowie die Entkopplung des unterliegenden Datenmodells von der verwendeten Speichertechnologie, wodurch die Verwendung unterschiedlicher Datenbanktechnologien ermöglicht wird. In dieser Arbeit stellen wir die Implementierung einer modularen Applikation vor, die eine Auswahl von Semantic Units mithilfe von Knowledge Graph Building Blocks verwalten kann, und ein Framework für weitere Werkzeuge und Applikationen bietet.

2 Verwandte Arbeiten

2.1 Ontology Based Data Access

Ontology Based Data Access (OBDA) [31, 32] ist ein Paradigma für den Datenzugriff auf relationale Datenbanken über eine konzeptionelle Ebene. Die konzeptionelle Ebene ist in RDF [10] oder OWL [14] dokumentiert und ermöglicht es, mithilfe von Mappings Begriffe der konzeptionellen Ebene auf die Datenebene abzubilden. Für jedes Element der konzeptionellen Ebene wird eine Abfrage für die Datenebene generiert. Aus der Datenebene wird so mithilfe der konzeptionellen Ebene ein virtueller Graph erstellt, welcher mit der Abfrage-Sprache SPARQL [29] erkundet werden kann.

2.1.1 Ontop

Ontop [32–34] ist ein virtuelles Wissensgraph-System, welches mehrere relationale Datenbanken zu einem virtuellen Wissensgraphen verbindet. Dazu wird das OBDA Paradigma in Kombination mit R2RML Mappings aus dem R2RML W3C Standard[1] verwendet, wodurch eine zusätzliche Abstraktionsschicht geschaffen wird und die Daten in ihren ursprünglichen Datenquellen verbleiben. Zudem werden die Inhalte durch eine Domänenontologie angereichert.

2.1.2 Stardog

Eine weitere Umsetzung des OBDA Paradigmas ist die Enterprise Wissensgraph-Plattform Stardog [35]. Analog zu Ontop verbindet Stardog mehrere relationale Datenbanken zu einem virtuellen Wissensgraphen. Zusätzlich können auch bereits existierende Triple Stores in den virtuellen Wissensgraphen eingebunden werden. Durch maschinelles Lernen und traditionelle Inferenzmethoden wird der Wissensgraph automatisch mit neuen Kontextinformationen angereichert, die ihm Bedeutung verleihen und eine konsistente und einheitliche Datendarstellung bringen. Es ermöglicht auch die Anwendung alternativer Schemata, um die besonderen Bedürfnisse und Anforderungen verschiedener Interessengruppen zu unterstützen.

[1] https://www.w3.org/TR/r2rml/

2.2 Metaphactory

Metaphactory [36, 37] ist eine Plattform für Wissensgraph Management Applikationen und unterstützt verschiedene Kategorien von Benutzern von Wissensgraphen, indem relevante Dienste für das Management von Wissensgraphen angeboten werden. Zudem stellt Metaphactory eine umfassende und benutzerdefinierbare Oberfläche bereit, welche eine schnelle Umsetzung verschiedener anwendungsspezifischer Applikationen, von Datenextraktion, Integration, Speichern und Abfragen, bis hin zu Visualisierung und Datenerstellung, ermöglicht. Die Datenverwaltung funktioniert mithilfe von SPARQL-Anfragen über eine spezielle Benutzeroberfläche, die den Nutzer einen umfangreichen Anfragen-Katalog bietet, und durch automatische Vorschläge bei der Erstellung von Wissensgraphen unterstützt. Endbenutzer können den Wissensgraphen auf verschiedene Arten navigieren, erkunden und visualisieren. Des Weiteren bietet Metaphactory eine einfache Möglichkeit, anwendungsspezifische Applikationen zu entwickeln, wodurch sich das Unternehmen hauptsächlich an große Unternehmen und Organisationen richtet.

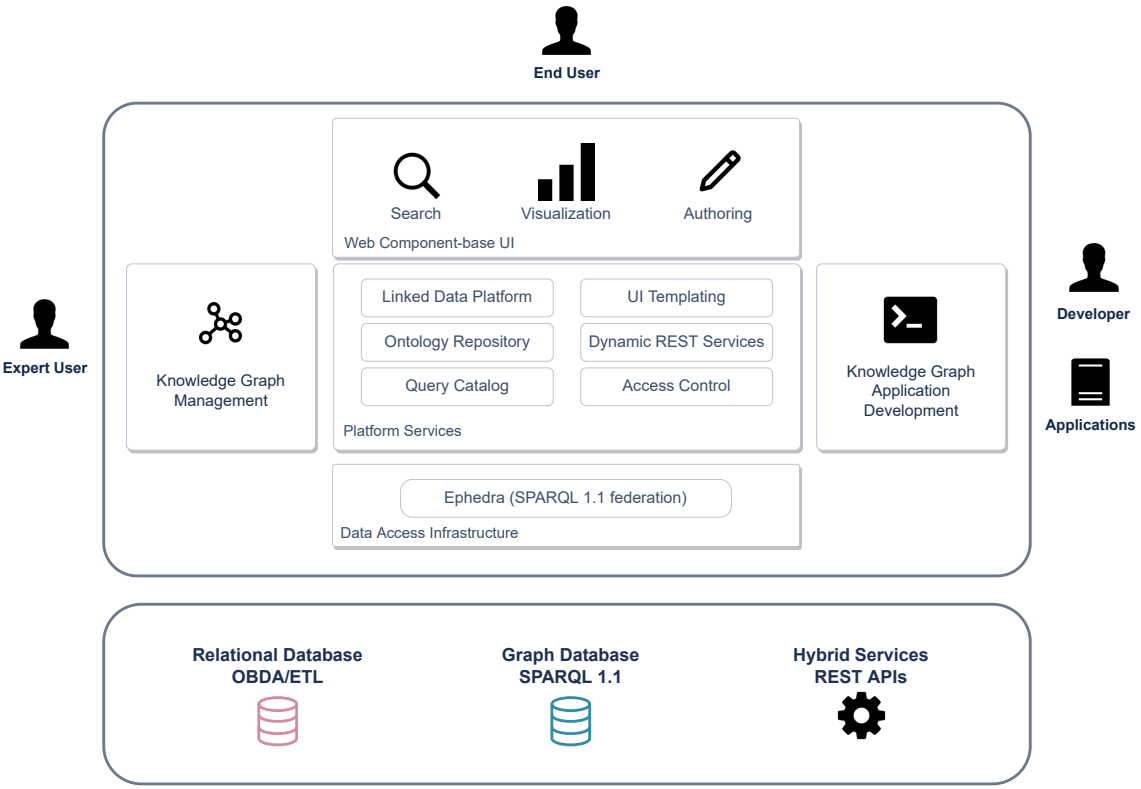


Abbildung 2.1: Metaphactory Architektur [36]

Die Architektur von Metaphactory ist in Abbildung 2.1 zu dargestellt, wobei die Plattform von den drei Benutzergruppen Endnutzer, Expertennutzer und Entwickler verwendet wird. Die Plattform arbeitet auf einer Graphdatenbank, in der der Wissensgraph gespeichert ist.

Die Kommunikation mit der Datenbank erfolgt über eine Data-Access Infrastruktur mittels SPARQL-Anfragen. Der Zugriff auf einen virtuellen Wissensgraphen erfolgt analog zu Kapitel 2.1. Dadurch ist die Plattform unabhängig von einem speziellen Datenbankanbieter und erlaubt, zusätzlich relationale Datenbanken über R2RML Mappings zu benutzen. Plattform Services implementieren generische Funktionalitäten für die Interaktion mit dem Wissensgraphen und erweitern den SPARQL-Zugriff um spezifische Funktionen für die jeweilige Benutzergruppe oder das jeweilige Analysetool. Diese werden über REST APIs bereitgestellt und ermöglichen eine detaillierte Zugriffskontrolle für verschiedene Benutzergruppen. Die webbasierte Benutzeroberfläche verwendet diese APIs, um mithilfe von Templates die Ressourcen aus dem Wissengraphen in Form von HTML-Seiten darzustellen. Für jede Ressource kann eine eigene HTML-Seite spezifiziert werden. Dies ist in den meisten Fällen jedoch nicht nötig, da Templates für Ressourcen desselben Typs definiert und angewendet werden können.

Die Data Access Infrastructure arbeitet auf einem RDF Repository, welches als Speicher für den Wissensgraphen dient und mit bestehenden Triple-Store Datenbanken verwendet werden kann, sowie mit traditionellen relationalen Datenbanken über R2RML Mappings. Der Repository Manager ermöglicht die gleichzeitige Verwendung von mehreren Repositorien, wobei verschiedene Datenquellen des Repository Managers in einer separaten RDF Ontologie gespeichert werden. Dadurch werden Systemdaten und Domänendaten getrennt. Die gesamte Data Access Infrastructure wird auf der Metaphactory Plattform von der SPARQL Föderations-Engine Ephedra übernommen. Sie wurde speziell für den Zweck entwickelt, verschiedene Datenbanktechnologien und deren Features in einer Applikation zu kondensieren.

Die Plattform Services realisieren eine Reihe von Services, die zusätzliche Funktionalitäten basierend auf den Standard-Interfaces der Triple Stores implementieren, welche von bestimmten Zielgruppen benötigt werden. Convenience-Services realisieren häufig benötigte Aufgaben, die normalerweise nicht direkt von Triple Stores bereitgestellt werden:

- Linked Data Plattform (LDP) Service für ressourcenbasiertes Management von Linked Data nach W3C LDP Spezifikationen[2]
- Data Quality Service für die Validierung von Daten
- Query Catalog Service für das Speichern und Verwalten von wiederverwendbaren SPARQL-Abfrage-Templates
- Keyword Search Service für die Datensuche

Connector-Services machen durch Vorverarbeitung die Daten aus dem Wissensgraphen für externe Applikationen einfacher nutzbar und sind primär an Entwickler gerichtet:

- Tableau Connector Service für die Datenanalyse

[2] https://www.w3.org/TR/ldp/

- Alexa[3] Skill Service für eine Sprachinteraktion mit dem Wissensgraphen

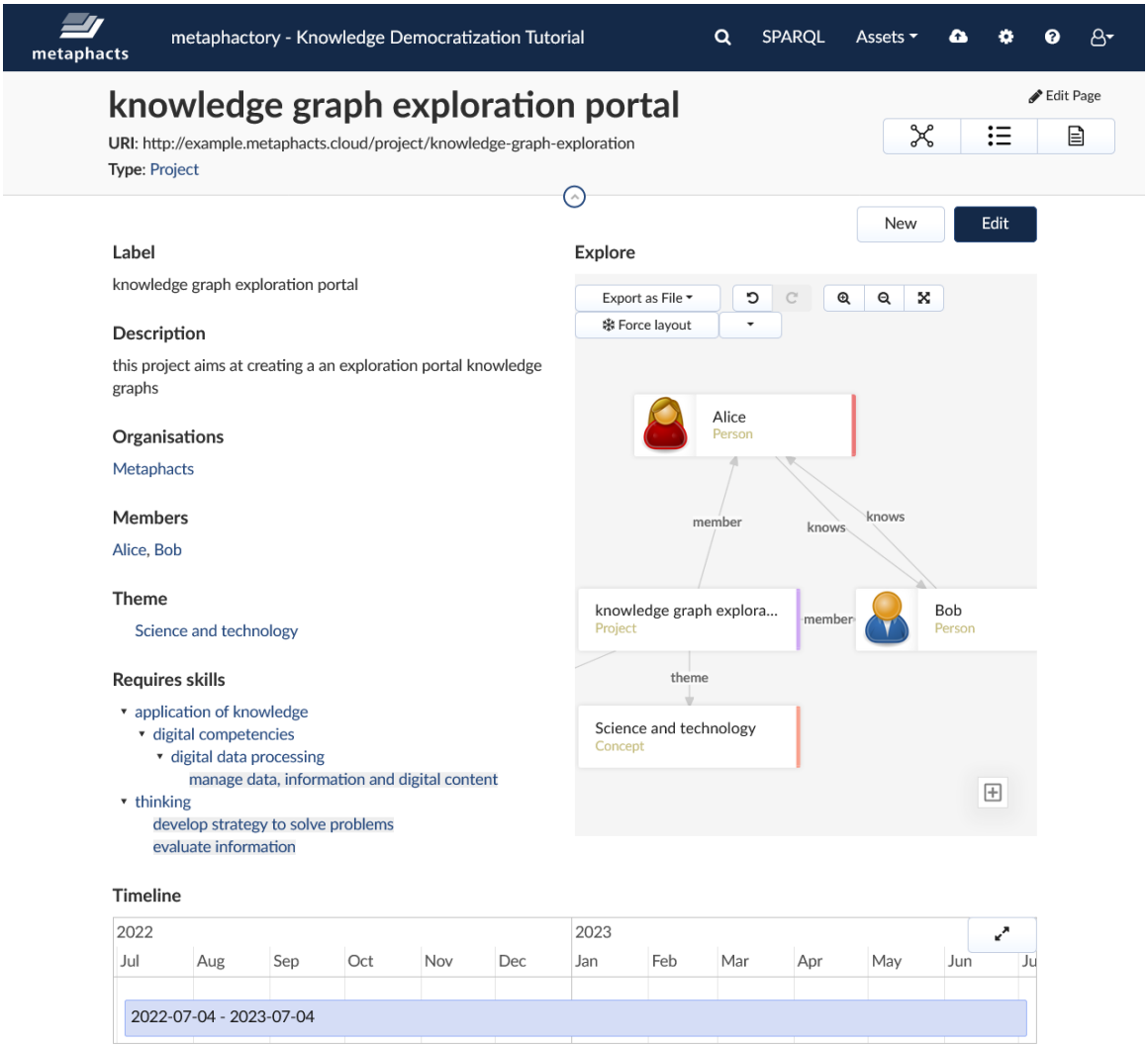


Abbildung 2.2: Beispiel-Ressourcenansicht für das Projekt „knowledge graph exploration portal“ [37]

Mithilfe dieser Services können z.B. Qualitätsreports für die Daten des Wissensgraphen erstellt werden, welche anhand von manuell erstellten oder automatisch aus OWL Ontologien generierten SHACL Constraints erstellt werden.

Der Fokus der Benutzeroberfläche liegt auf Wiederverwendbarkeit, Kompatibilität, Erweiterbarkeit und Anpassung und ist deshalb generisch implementiert, weshalb sie nicht an eine bestimmte Technologie gebunden ist. Für die Interaktion mit dem Wissensgraphen kann z.B. der Sprachassistent Alexa verwendet werden oder die mitgelieferte webbasierte Benutzeroberfläche (vgl. Abbildung 2.2). Letztere basiert auf HTML-Templates und bietet unterschiedliche Ebenen an Abstraktion. Für jede Ressource im Wissensgraphen kann ein

[3] https://developer.amazon.com/de-DE/alexa

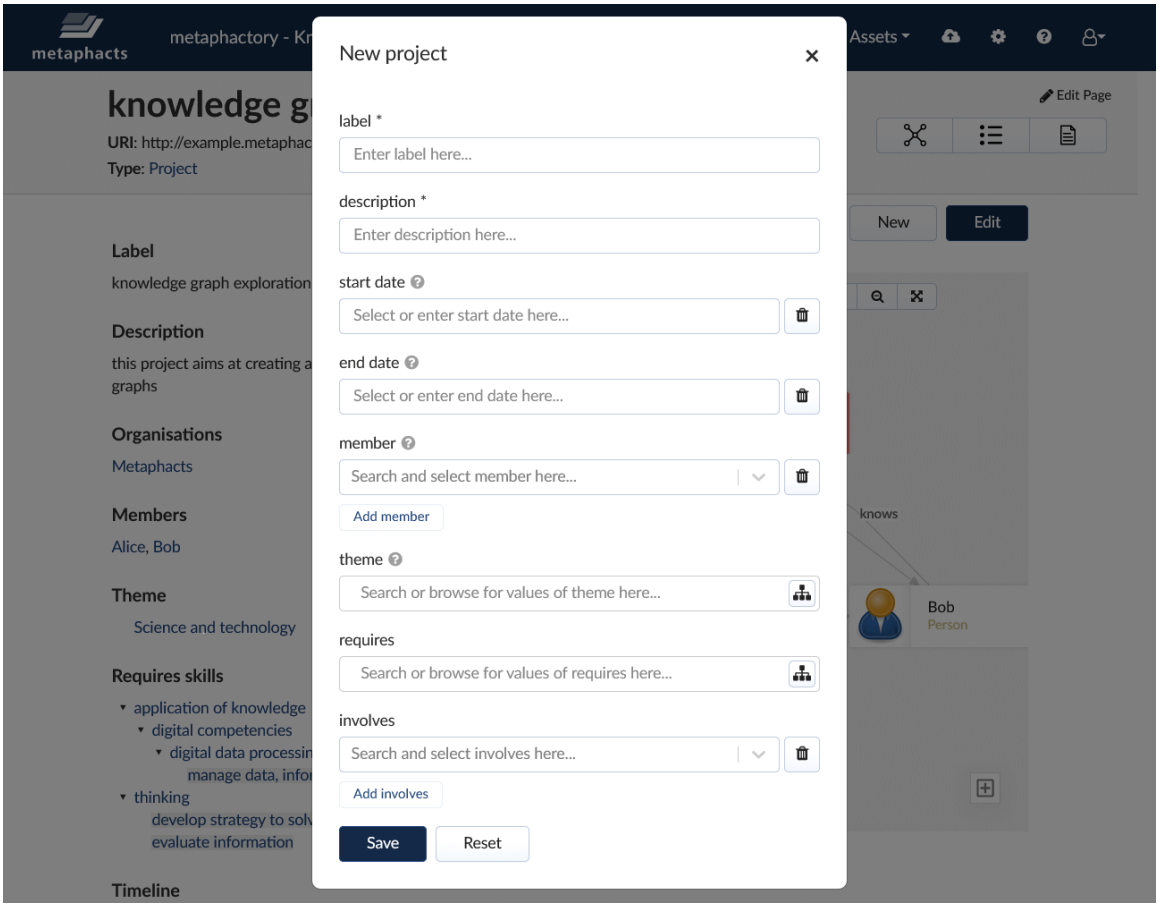


Abbildung 2.3: Eingabeformular einer neuen Projekt-Ressource basierend auf Wissensgraphmustern [36, 37]

spezifisches Template hinterlegt werden oder ein generisches Template für einen bestimmten Ressourcen-Typ erstellt werden. Dieses ist an den Wert der Eigenschaft rdf:type der Ressource gebunden, kann aber auch durch eine Template-Selection SPARQL-Abfrage auf den Wert einer anderen Eigenschaft der Ressource geändert werden. Templates können direkt in einem mitgelieferten HTML-Editor bearbeitet werden und verfügen neben einfachen HTML-Tags auch spezielle vorgefertigte Komponenten für die Benutzeroberfläche, welche untereinander kombinierbar sind und zusätzliche Funktionalitäten ermöglichen. Zudem können Teile aus anderen Templates wiederverwendet werden. Die Erkundung des Wissensgraphen bietet mehrere gleichzeitige Möglichkeiten für die Dateneingabe:

- Textfeld für die Stichwortsuche
- Parametereingabe in vordefinierten Formularen
- Relevante Eigenschaften und Constraints iterativ festlegen

Die resultierenden Daten können dann in einer Facettensuche weiter verfeinert werden und unterschiedlich dargestellt werden, wie z.B. in einer Tabelle oder über ein Diagramm.

Das Hinzufügen von Daten erfolgt über ein spezielles Authoring User Interface (vgl. Abbildung 2.3), das auf Wissensgraphmustern basiert. Diese Muster beinhalten ein SPARQL Abfrage-Pattern zusammen mit zusätzlichen Metadaten, welche die Komplexität des Wissensgraphen verbergen und gleichzeitig eine Validierung der Eingaben ermöglichen.

3 Grundlagen

In diesem Kapitel werden die konzeptionellen Grundlagen erläutert. Dazu gehört eine Einführung zur Modellierungssprache LinkML, den Semantic Units und Knowledge Graph Building Blocks, sowie das Softwaredesignpattern *Ports und Adapter*.

3.1 LinkML

Die Linked data Modeling Language (LinkML) ist ein objektorientiertes Framework zum Modellieren von Daten [38, 39]. Das Ziel von LinkML ist es, das Erstellen von FAIRen [1] und Ontologie-bereiten Daten zu vereinfachen und gleichzeitig Semantic Web Standards zu etablieren. Es können sowohl einfache Modelle, wie z.B. lineare Checklisten, modelliert werden, als auch komplexe Modelle mit vernetzten Datenstrukturen, Vererbung und Polymorphie. LinkML basiert auf dem Kernkonzept von Klassen und Slots, wobei jede Klasse ein Domänenobjekt repräsentiert und jeder Slot ein Attribut des jeweiligen Objekts modelliert. Ein vollständiges LinkML-Modell besteht stets aus einem Modellschema inklusive der zugehörigen Metadaten und wird in der Auszeichnungssprache YAML [40] dokumentiert.

In Listing 1 ist ein beispielhaftes Modell einer Person in LinkML Syntax dargestellt. Das Modell lässt sich in zwei Abschnitte unterteilen. Der erste Teil des Modells, Zeile 1-17, zeigt eine Auswahl an Metadaten, die für ein Modell angegeben werden können. Dazu zählen grundlegende Metadaten, wie die ID, Titel und Name, sowie die Lizenz und Version des Modells (Zeile 2-6). Zudem können Abkürzungen für Präfixe definiert werden, die später im Modell verwendet werden (Zeile 8-12). Des Weiteren ist es möglich, Abhängigkeiten zu anderen LinkML Modellen zu definieren, um diese wiederzuverwenden (Zeile 14-15). Der zweite Teil des LinkML Modells in Listing 1, Zeile 18-39, beschreibt das eigentliche Datenmodell mit den jeweiligen Klassen- und Slotdefinitionen. Es modelliert eine Person mit den vier Attributen *id*, *first_name*, *last_name* und *knows*. Attribute können direkt an einer Klasse spezifiziert werden (Zeile 23-32) oder separat als Slot (Zeile 33-39). Wird ein Slot separat definiert, so kann er von mehreren Klassen verwendet werden, ohne neu definiert werden zu müssen. Des Weiteren können für jede Klasse Metadaten spezifiziert werden, wie z.B. eine kurze Beschreibung der Klasse. Alle Klassen und Slots eines LinkML Modells werden mit RDF URIs hinterlegt (*class_uri*, *slot_uri*), sodass jedes Modell semantisch untermauert ist. Die Klassendefinitionen folgen dabei der objektorientierten Semantik und nicht der OWL-Semantik.

```yaml
#Metadaten
id: https://example.org/linkml/hello-world
title: Really basic LinkML model
name: hello-world
license: https://creativecommons.org/publicdomain/zero/1.0/
version: 0.0.1

prefixes:
  linkml: https://w3id.org/linkml/
  sdo: https://schema.org/
  ex: https://example.org/linkml/hello-world
default_prefix: ex

imports:
  - linkml:types

#Modell
classes:
  Person:
    description: Minimal information about a person
    class_uri: sdo:Person
    attributes:
      id:
        identifier: true
        slot_uri: sdo:taxID
      first_name:
        required: true
        slot_uri: sdo:givenName
        multivalued: true
      last_name:
        required: true
        slot_uri: sdo:familyName
    slots:
      - knows
slots:
  knows:
    range: Person
    multivalued: true
    slot_uri: foaf:knows
```

Listing 1: Beispielhaftes Modell einer Person in LinkML Syntax.

Ein wichtiger Aspekt von LinkML ist die hohe Interoperabilität mit vielen bereits existierenden Frameworks und ist sowohl mit semantischen RDF-basierten Frameworks kompatibel, als auch mit Frameworks, die für Entwickler relevant sind, wie z.B. JSON. Erstellte LinkML-Modelle lassen sich direkt in andere Formate übersetzen, wie z.B. JSON-schema, JSON-LD/RDF, SQL DDL, ShEx, GraphQL, Python Daten-Klassen, Markdown und UML-Diagramme. Zudem können Daten automatisch anhand des LinkML-Modells validiert und in verschiedene Ausgabe-Formate übertragen werden, wie z.B. YAML, JSON, RDF, Turtle, JSON-LD, CSV und TSV. Dies erlaubt es LinkML, kompatibel mit RDF Werkzeugen zu sein, ohne dass eine spezielle Implementierung gewählt werden muss. LinkML wird bereits in mehreren verschiedenen Bereichen eingesetzt, wie z.B. bei der Harmonisierung von Krebsdaten[1,2], Umweltgenomik[3,4] oder bei der Integration von Wissensgraphen[5,6].

3.2 Semantic Units

Semantic Units sind ein Konzept zur Strukturierung von Wissensgraphen in sich teilweise überlappende Teilgraphen, um die kognitive Interoperabilität von Daten und Metadaten für den Menschen zu steigern [6]. Eine Semantic Unit entspricht dabei genau einem Teilgraph des übergeordneten Wissensgraphen und repräsentiert eine für einen Menschen semantisch sinnvolle Informationseinheit. Im Wissensgraphen erhält jede Semantic Unit ihre eigene Ressource zusammen mit einer eigenen UPRI und ist einem bestimmten Semantic Unit Typ zugeordnet. Die unterschiedlichen Semantic Unit Typen liegen in Form einer Klassenontologie vor und enthalten eine für einen Menschen gut lesbare Beschreibung über die Art der Information, die von der jeweiligen Semantic Unit repräsentiert wird. Somit stellen Semantic Units, neben Klassen und Instanzen, eine dritte Art von repräsentativer Einheit in einem Wissensgraphen dar, wobei die Klassen und Instanzen die Daten-Graphen-Ebene des Wissensgraphen bilden und die Semantic Units mit ihren Relationen die Semantic-Unit-Graphen-Ebene definieren. Dabei wird jeder Teilgraph aus der Daten-Graphen-Ebene durch eine eigene Semantic Unit Ressource in der Semantic-Unit-Graphen-Ebene repräsentiert. Eine UPRI steht dabei sowohl für eine bestimmte Semantic Unit als auch für ihren dazugehörigen Teilgraphen. Somit ist eine Referenz auf eine Semantic Unit gleichbedeutend mit einer Referenz auf den Inhalt des Daten-Graphen, welcher von der Semantic Unit repräsentiert wird. Grundsätzlich gibt es vier verschiedene Arten von Semantic Units, die unterschieden werden können: Statement Units, Compound Units, List Units und Question Units. Im Folgenden werden die Statement Units und Compound Units im Detail erklärt.

[1] https://datacommons.cancer.gov/
[2] https://cancerdhc.github.io/ccdhmodel/v1.1/
[3] https://microbiomedata.org/
[4] https://github.com/microbiomedata/nmdc-schema
[5] https://ncats.nih.gov/translator
[6] https://github.com/biolink/biolink-model

3.2.1 Statement Unit

Eine Statement Unit repräsentiert eine für einen menschlichen Leser kleinste, unabhängige und semantisch sinnvolle Aussage. Sie wird über das Prädikat der Aussage definiert, indem sie eine Subjekt-Ressource in Relation zu einem oder mehreren Objekten setzt, die im Graphen entweder als Ressourcen und/oder als Literale hinterlegt werden. Die Daten-Graphen-Ebene eines Wissensgraphen wird durch die Verwendung von Statement Units mathematisch partitioniert, sodass jedes Tripel des Wissensgraphen immer genau einer Statement Unit zugeordnet ist. Wie jede Semantic Unit besitzen auch Statement Units ihre eigene Ressource mit eigener UPRI, die einer bestimmten Ontologieklasse zugeordnet ist und diese instanziiert, und mit der sie im Wissensgraphen repräsentiert wird.

In Abbildung 3.1 ist eine Statement Unit für eine „part of“ Beziehung beispielhaft dargestellt. Die part-of Statement Unit setzt die Instanz „antenna X“ in Relation mit der Instanz „head Y“. Das Subjekt in der Beziehung ist dabei die Instanz „antenna X“ und das Objekt ist die Instanz „head Y“. Bei dem Beispiel handelt es sich um eine Statement Unit mit nur einem Subjekt und einem Objekt. Jedoch sind nicht alle Statement Units auf nur eine Objekt-Position limitiert. Eine Gewichtsmessung kann z.B. mit zwei Objekt-Positionen modelliert werden, von denen die eine den Wert der Messung als Literal repräsentiert und die andere die Maßeinheit der Gewichtsmessung als Ressource. Grundsätzlich wird zwischen Literal-Objekten und Ressourcen-Objekten unterschieden.

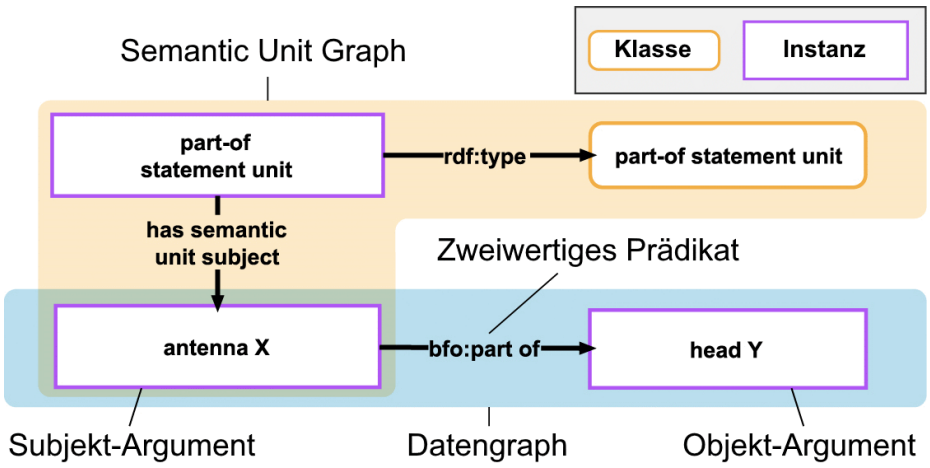


Abbildung 3.1: Beispiel einer Statement Unit [6]

Je nach Subjekt lassen sich Statement Units in die folgenden Kategorien unterteilen: Assertional Statement Units, Contingent Statement Units und Universal Statement Units. Für jede dieser Kategorien existieren zusätzlich entsprechende Identification Units, welche die Subjekt-Ressource der Statement Unit näher beschreiben. In dieser Arbeit sind insbesondere die Assertional Statement Units in Kombination mit den Named Individual Identification Units relevant.

Named Individual Identification Unit Eine Named Individual Identification Unit (NIIU) ist eine Statement Unit, die eine Named Individual Ressource im Wissensgraphen identifiziert. Mittels zweier Objekt-Positionen wird die Klassenzugehörigkeit (rdf:type), sowie ein Label (rdfs:label) für die Subjekt-Ressource modelliert.

Assertional Statement Unit Assertional Statement Units sind Statement Units, die eine Named Individual Ressource als Subjekt besitzen. Besitzt eine Assertional Statement Unit eine Objekt-Position, bei der es sich nicht um ein Literal handelt, muss diese Ressource auch ein Named Individual sein. Folglich repräsentiert eine Assertional Statement Unit eine assertorische Aussage, d.h. eine Aussage über ein bestimmtes Individuum, und ist somit keine allgemeine Aussage. Semantisch ist die Aussage entweder wahr oder falsch und kann eine beliebige Anzahl an Objekt-Positionen besitzen.

3.2.2 Compound Unit

Compound Units organisieren die Statement Units eines Wissensgraphen in größere, aber dennoch semantisch sinnvolle Teilgraphen. Sie können als eine Art Container von Semantic Units verstanden werden. Jede Compound Unit besitzt eine eigene Ressource im Wissensgraphen mit einer eigenen UPRI und instanziiert eine entsprechende Ontologieklasse. Der Daten-Graph einer Compound Unit setzt sich aus den Daten-Graphen der ihr assoziierten Semantic Units zusammen. Je nachdem, welche Semantic Units im Daten-Graphen einer Compound Unit assoziiert werden können, lassen sich die Compound Units in mehrere Kategorien unterteilen: Typed Statement Units, Quality Measurement Units, Item Units, Item Group Units, Dataset Units, Granularity Tree Units und Granularity Item Group Units. Im Folgenden werden die Typed Statement Units, die Quality Measurement Units und die Item Units kurz beschrieben.

Typed Statement Unit Als Typed Statement Unit werden Compound Units bezeichnet, die zwei verschiedene Arten von Statement Units assoziieren. Zum einen, eine Statement Unit, die keine Identification Unit ist und als Referenz-Statement-Unit funktioniert, zum anderen all jene Identification Units, die Informationen über die Klassenzugehörigkeit derjenigen Ressourcen enthalten, die in der Referenz-Statement-Unit verwendet worden sind. Die Subjekt-Ressource der Referenz-Statement-Unit ist auch die Subjekt-Ressource der Typed Statement Unit. In Abbildung 3.2 ist eine Typed part-of Statement Unit dargestellt.

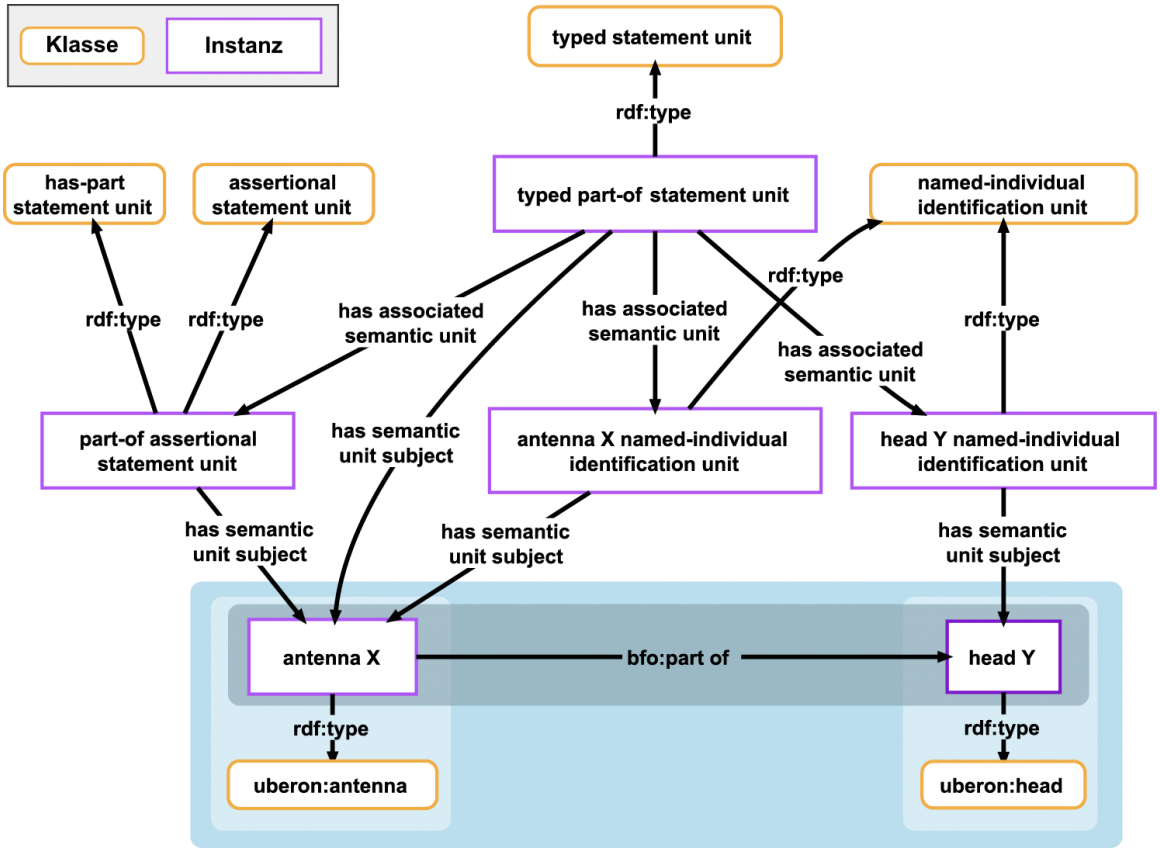


Abbildung 3.2: Beispiel einer Typed Statement Unit [6]

Quality Measurement Unit Als Quality Measurement Unit werden Compound Units bezeichnet, die eine qualitative Eigenschaft beschreibende Typed Statement Unit assoziieren und eine oder mehrere quantitative Messwerte beschreibende Typed Statement Units assoziieren. Die Referenz-Statement-Unit der qualitativen Typed Statement Unit spezifiziert dabei eine Qualität eines Objekts. Die Statement Unit jeder quantitativen Typed Statement Unit spezifiziert eine Messung dieser Qualität. Jede Messung enthält den jeweiligen Messwert und die dazugehörige Maßeinheit. Eine Quality Measurement Unit ist somit eine Sammlung von Typed Statement Units, welche eine Qualität eines bestimmten Objekts zusammen mit allen quantitativen Messungen der Qualität beschreiben.

Item Unit Eine Item Unit ist eine Compound Unit, die alle Statement Units, Typed Statement Units und Quality Measurement Units beinhaltet, welche dieselbe Subjekt-Ressource teilen. Diese Subjekt-Ressource ist folglich auch die Subjekt-Ressource der Item Unit. Damit ist eine Item Unit eine Sammlung an semantisch verwandten Statement Units, Typed Statement Units und Quality Measurement Units über eine bestimmte Subjekt-Ressource.

3.3 Knowledge Graph Building Blocks

Ein Knowledge Graph Building Block (KGBB) [2, 7] ist ein kleiner Baustein, ein Informationsmodul für das Wissensmanagement, das einer bestimmten Art von Semantic Unit zugeordnet ist. Jeder KGBB beinhaltet Informationen für eine Wissensgraph-Applikation, die zum Verarbeiten eines bestimmten Semantic Unit Typen erforderlich sind. Analog zu den Semantic Units können KGBBs in vier Kategorien unterteilt werden: Statement KGBBs, Compound KGBBs, List KGBBs und Question KGBBs. Jede Kategorie enthält zusätzliche Informationen, die spezifisch für die jeweilige Art des KGBB sind. Zusammen können mehrere KGBBs die Interaktion aller verschiedenen Arten von Semantic Units in einem Wissensgraphen modellieren und definieren somit den möglichen Daten- und Propositionsraum einer Wissensgraphapplikation. Eine KGBB-getriebene Applikation kann anhand eines solchen Modells neue Semantic Units instanziieren, organisieren und verwalten. Dazu muss für jede Semantic Unit der entsprechende KGBB vorhanden sein.

Ein KGBB wird in Form einer Klasse dokumentiert, wobei alle Eigenschaften von der Elternklasse vererbt werden. Die Gesamtheit aller KGBBs bildet folglich eine Taxonomie. An jeder Klasse sind ein oder mehrere (partielle) Storage-Modelle in Form von LinkML-Templates hinterlegt. Ein Storage-Modell für den Semantic-Unit-Graphen und ein weiteres für den Daten-Graphen der Semantic Unit, wobei das Storage-Modell für den Daten-Graphen nur für Statement Units benötigt wird, da nur diese über einen Daten-Graphen verfügen. Diese Storage-Modelle spezifizieren die Graphenstruktur, in der die jeweiligen Semantic Units gespeichert werden sollen, sowie deren Metadaten und Input Constraints. Des Weiteren enthält ein KGBB ein oder mehrere Display-Templates. Sie definieren, welche Informationen der Semantic Unit in einem User-Interface angezeigt werden und wie diese formatiert werden sollen. Zudem kann ein KGBB über Access Templates und Import Templates verfügen. Mithilfe dieser Templates lassen sich die Informationen aus dem Daten-Graphen einer Semantic Unit auf beliebige Graphenstrukturen für einen Datenexport übertragen oder existierende Daten-Graphen in Semantic Unit Instanzen überführen.

Die KGBB-Klassen lassen sich dann nutzen, um eine KGBB-getrieben Wissensgraphapplikation in Form eines semantischen Graphen zu beschreiben, indem Instanzen bestimmter KGBBs über KGBB-Association-Knoten und KGBB-Link-Knoten Instanzen in Beziehung gesetzt werden. Eine KGBB-Association ermöglicht es einem KGBB andere KGBBs (und somit Semantic Units) zu instanziieren, sofern diese dieselbe Subjekt-Ressource besitzen. Es handelt sich dabei immer um eine direktionale Relation mit einem Ausgangs-KGBB und einem Ziel-KGBB. Zusätzlich kann definiert werden, ob eine Semantic Unit der Ziel-KGBB benötigt wird, um die Semantic Unit der Ausgangs-KGBB neu anlegen zu können, wie oft der Ziel-KGBB instanziiert werden darf und ob die Subjekt-Ressource der Semantic Unit des Ziel-KGBBs bestimmte Vorgaben erfüllen muss. Ein KGBB-Link ist analog zu einer KGBB-Association, jedoch wird hier die Objekt-Ressource der Semantic Unit des

Ausgangs-KGBB als Subjekt-Ressource für die Semantic Unit des Ziel-KGBB verwendet. Als Ausgangs-KGBB sind nur KGBBs für Statement Units zugelassen, da Compound Units keine Objekt-Ressourcen besitzen.

3.4 FAIREr

FAIREr ist eine Erweiterung zu den „FAIR Guiding Principles for scientific data management and stewardship“ [1], welche 2016 in der „Scientific Data“ veröffentlicht wurden. Die Prinzipien von FAIR definieren einen Standard für wissenschaftliches Datenmanagement, wodurch die Findbarkeit (Findability), Zugänglichkeit (Accessibility), Interoperabilität (Interoperability) und Wiederverwendbarkeit (Reusability) von (digitalen) Daten erhöht werden soll. Hauptziel von FAIR ist es, die Fähigkeiten von Maschinen zu verbessern, da Menschen durch die steigende Komplexität der Daten immer mehr an Maschinen gebunden sind. FAIREr erweitert diese Prinzipien um eine bessere Erkundbarkeit (Explorability raised) der Daten für menschliche Benutzer [6]. Dies soll die kognitive Interoperabilität von Wissensgraphen durch einen verbesserten Datenzugriff und gesteigerte Datendurchsuchbarkeit erhöhen. Mit kognitiver Interoperabilität ist hier die Eigenschaft einer Informationstechnologie gemeint, ihre Informationen auf möglichst effiziente Weise mit einem menschlichen Nutzer kommunizieren zu können, indem sie dem Nutzer intuitive Werkzeuge zu Verfügung stellt, die Daten und Metadaten zu finden, zu filtern und zu erkunden, wobei die allgemeinen kognitiven Bedingungen des Menschen berücksichtigt werden. Außerdem sollte sie sowohl von Entwicklern leicht implementiert und eingesetzt und von Betreibern leicht verwaltet werden können.

Findbarkeit Das erste Prinzip ist die Findbarkeit von Daten und Metadaten. Diese sollen sowohl von Menschen als auch von Maschinen einfach zu finden sein. Maschinen-lesbare Metadaten sind grundlegend für die automatische Verarbeitung von Daten und sollten deshalb eindeutig und einheitlich identifizierbar sein.

Zugänglichkeit Das zweite Prinzip von FAIR ist die Zugänglichkeit. Dies beinhaltet gegebenenfalls ein Authentifizierungs- und Autorisierungsverfahren und sollte über ein standardisiertes Kommunikationsprotokoll erfolgen. Zudem sollen Metadaten zugänglich sein, auch wenn die eigentlichen Daten nicht mehr verfügbar sind.

Interoperabilität Das dritte Prinzip ist die Interoperabilität von Metadaten und Daten. Um die Integration von Daten und Metadaten zu verbessern, sollen kontrollierte Vokabulare und Ontologien zur Wissensrepräsentation verwendet werden. Des Weiteren sollen Benutzer die Möglichkeit haben, die Daten in verschiedenen Applikationen oder Workflows zur Analyse, Speicherung oder Weiterverarbeitung zu verwenden.

Wiederverwendbarkeit Das vierte Prinzip ist eines der Hauptziele von FAIR und besteht darin, die Wiederverwendbarkeit von Daten und Metadaten zu optimieren. Um dies zu erreichen, sollen Daten mit einer Vielzahl relevanter Metadaten beschrieben werden, sodass die Daten in den unterschiedlichsten Situationen repliziert oder kombiniert werden können. Metadaten und Daten sollen zudem mit einer klaren und zugänglichen Lizenz veröffentlicht werden, sowie mit detaillierten Herkunftsdaten versehen sein und domänenrelevante Community-Standards erfüllen.

Erkundbarkeit Das fünfte Prinzip ist die Erkundbarkeit der Daten, welches eine Erweiterung von FAIR darstellt (FAIREr). Durch eine Strukturierung der Daten in mehre Abstraktionsebenen mit unterschiedlicher repräsentativer Granularität sollen Daten für einen Menschen erkundbarer und zugänglicher gemacht werden. Des Weiteren soll dadurch die kognitive Interoperabilität des Wissensgraphen für die Benutzer erhöht werden, und somit das Interoperabilitäts-Framework der European Open Science Cloud (EOSC) um eine zusätzliche Ebene erweitert werden.

3.5 Ports und Adapter (Hexagonale Architektur)

Ports und Adapter oder auch *Hexagonale Architektur* ist ein Design Pattern der Softwareentwicklung. Die Bezeichnung „Hexagonale Architektur“ geht zurück auf eine Darstellung einer Applikation in Form eines Hexagons von Alistair Cockburn [41]. Das Ziel von *Ports und Adapter* ist es, die Applikation so zu definieren, dass sie von verschiedenen Clients gesteuert werden kann. Dies ermöglicht ein einfaches Testen in einer von externen Geräten isolierten Umgebung. Im Hexagon befindet sich das Business-Problem bzw. die Applikationslogik (vgl. Abbildung 3.3). Die innere Struktur der Applikation ist dabei nicht vorgegeben und es werden keine Referenzen zu einer speziellen Technologie gemacht. Somit bleibt die Applikation technologie-agnostisch. Außerhalb des Hexagons befinden sich die Akteure, wie z.B. Menschen, Programme, Hardware- oder Software-Geräte. Sie bilden die Umgebung der Applikation. Die Anordnung in der Darstellung der Akteure ergibt sich aus der Interaktionsart mit der Applikation. Wird die Interaktion vom Akteur ausgelöst, so handelt es sich um einen primären Akteur. Primäre Akteure treiben die Applikation und werden oben links außerhalb des Hexagons dargestellt (vgl. Abbildung 3.3). Primäre Akteure interagieren mit der Applikation, um ein bestimmtes Ziel zu erreichen und stellen gleichzeitig die Benutzer der Applikation dar. Sie werden auch als Treiber der Applikation bezeichnet und können entweder Menschen oder andere Geräte sein. Wird die Interaktion hingegen von der Applikation ausgelöst, so handelt es sich um einen sekundären Akteur. Sekundäre Akteure werden von der Applikation getrieben und werden unten rechts außerhalb des Hexagons dargestellt (vgl. Abbildung 3.3). Sie stellen Funktionen für die Applikation bereit, die von ihr benötigt

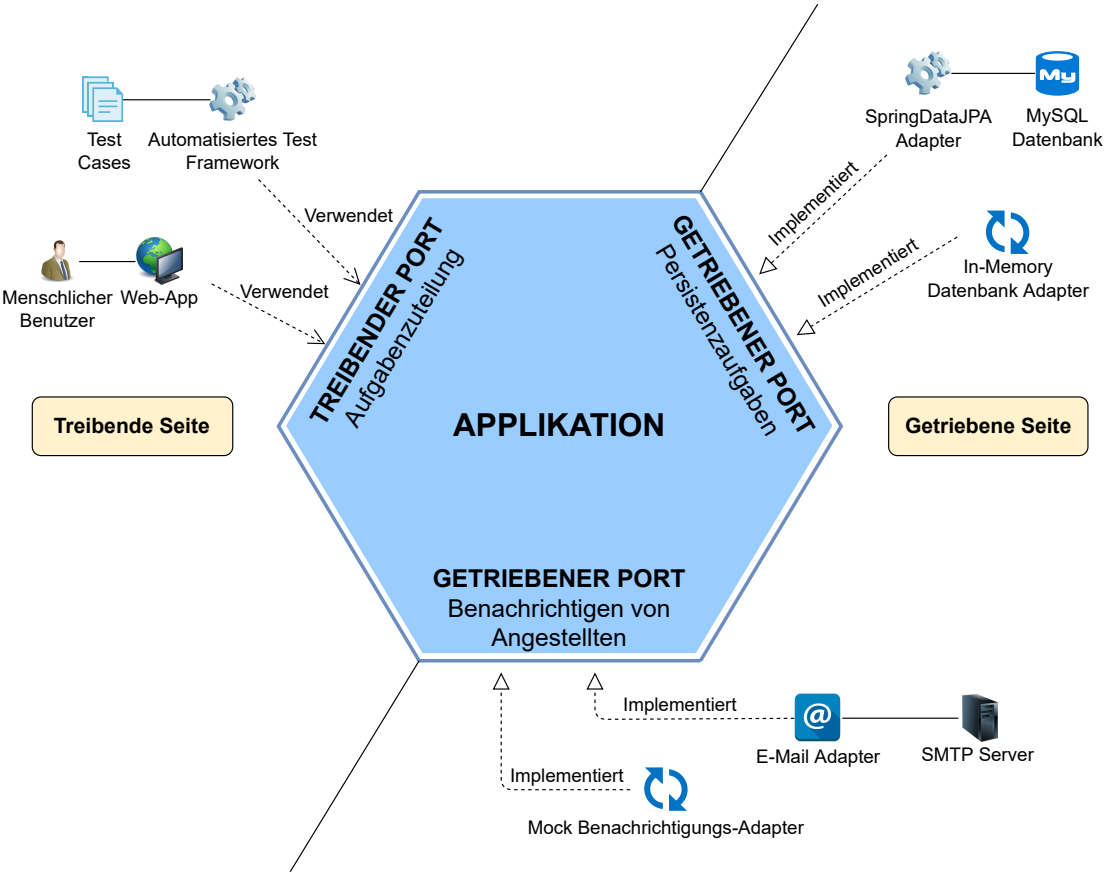


Abbildung 3.3: Diagramm der Hexagonalen Architektur für eine Applikation zur Aufgabenzuteilung in einem Unternehmen mit einer webbasierten Benutzeroberfläche [42]

werden, um die Applikationslogik zu implementieren, wie z.B. Datenbanken oder externe Webservices.

Ports Als Port wird die Kante des Hexagons zwischen einem Akteur und dem inneren des Hexagons bezeichnet. Die Interaktion erfolgt über ein von der Applikation definiertes Interface, welches die Applikation für die äußere Welt bereitstellt. Durch die Abstraktion über das Interface wird Applikation agnostisch zur jeweiligen Implementierung, sowie der verwendeten Technologie. Ein Akteur interagiert ausschließlich mit dem Port der Applikation und nicht mit dem inneren der Applikation. Ports, die von primären Akteuren gesteuert werden (treibende Ports), spiegeln somit die Funktionalität der Applikation wider. Sie definieren das Application Programming Interface (API) der Applikation. Ports von sekundären Akteuren (getriebene Ports) hingegeben spiegeln die von der Applikation benötigten Funktionen wider, um die Applikationslogik zu implementieren. Sie definieren das Service Provider Interface (SPI) der Applikation.

Adapter Adapter sind Softwarekomponenten und befinden sich außerhalb des Hexagons, und ermöglichen es einer spezifischen Technologie, mit einem Port der Applikation zu interagieren. Somit kann es mehrere Implementierungen für verschiedene Technologien für einen Port geben. Adapter, die von primären Akteuren gesteuert werden (treibende Adapter) überführen Anfragen aus einer Technologie in eine Technologie-agnostische Anfrage für die Applikation. Diese können dann über das Treiber Port Interface an die Applikation gestellt werden. Pro Treiber Port gibt es oftmals zwei Adapter-Implementierungen. Eine Implementierung für die Akteure und eine Implementierung zum Testen der Applikation. Adapter von sekundären Akteuren (getriebene Adapter) hingegeben überführen Technologie-agnostische Anfragen der Applikation in Technologie-spezifische Funktionen. Pro getriebenem Port gibt es ebenfalls zwei Adapter-Implementierungen. Eine Implementierung für die jeweilige Technologie und eine Test-Implementierung, welche die echten Funktionen für Testzwecke imitiert.

4 Anforderungserfassung

Im Rahmen dieser Arbeit soll eine Softwareapplikation entwickelt werden, welche Semantic Units unter der Verwendung von KGBBs verwalten kann. In den folgenden Kapiteln werden die Anforderungen für die Applikation ermittelt und analysiert, sowie die entsprechenden Use-Cases definiert.

4.1 Anforderungen

Die Softwareapplikation soll es den Benutzern ermöglichen, in einem Wissensgraphen neue Ressourcen vom Typ Material Entity (oder einer Unterklasse) anzulegen und weiter zu beschreiben. Eine jeweilige Material Entity Instanzenressource soll neben der jeweiligen Klasse auch mit einem Label versehen werden, welches frei vom Benutzer gewählt werden kann. Wird vom Benutzer kein Label festgelegt, so soll das Label der jeweiligen Klasse verwendet werden. Des Weiteren sollen Material Entity Instanzen, mittels einer Parthood-Beziehung, in hierarchische Relationen gesetzt werden können, um eine Partonomie materieller Entitäten beschreiben zu können. Zudem sollen Benutzer, für jede Material Entity Instanz, eine beliebige Anzahl an Gewichtsmessungen, inklusive eines Konfidenzintervalls anlegen können.

Für jede Eingabe eines Benutzers soll die Applikation automatisch entsprechende Metadaten erstellen und dem Wissensgraphen hinzufügen, sodass eindeutig dokumentiert ist, welcher Benutzer die entsprechende Information dem Wissensgraphen hinzugefügt hat und zu welchem Zeitpunkt. Außerdem sollen alle entstehenden Ressourcen, nachdem sie erstellt worden sind, nachträglich vom Benutzer bearbeitbar sein. Jedoch sollen keine Informationen aus dem Wissensgraphen verloren gehen. Dies bedeutet, dass die Informationen aus dem Wissensgraphen nicht wirklich gelöscht werden dürfen, sondern stattdessen nur auf *gelöscht* gesetzt werden (Soft-Delete).

Die gesamte Applikation soll dabei von KGBBs verwaltet werden, d.h. die Applikation soll mithilfe von KGBB-Instanzen als eigener Graph beschrieben werden, der auch die Beziehungen und somit die Interaktionsmöglichkeiten zwischen den verschiedenen KGBB-Instanzen definiert, um automatisch neue Semantic Units im Wissensgraphen instanziieren und verwalten zu können. Die für die KGBB-Instanzen benötigten KGBB-Klassen sollen ebenfalls in der Applikation hinterlegt werden und frei konfigurierbar sein.

Durch einen modularen Aufbau soll die Speichertechnologie der Applikation austauschbar sein und es Benutzern ermöglichen, zwischen mindestens zwei unterschiedlichen Speichertechnologien zu wählen. Zur Reduktion der Komplexität der Applikation soll eine zustandslose Verarbeitung der Benutzereingaben erfolgen. Außerdem soll eine einfache Authentisierung von Benutzerinformationen bei der Umsetzung berücksichtigt werden.

4.2 Analyse

Aus den Anforderungen aus Kapitel 4.1 lassen sich die benötigten Semantic Units ableiten. Diese werden im Folgenden kurz vorgestellt.

Named Individual Identification Unit Die NIIU ist eine Statement Unit mit zwei Objekt-Positionen. Sie dokumentiert die Klassen-URI und das Label ihrer Subjekt-Ressource. Folglich handelt es sich bei der Subjekt-Ressource immer um eine Named-Individual-Ressource.

Has-Part Statement Unit Die Has-Part Statement Unit ist eine Statement Unit mit nur einer Objekt-Position. Sie modelliert eine Parthood-Beziehung zwischen zwei Named-Individual-Ressourcen in Form einer „X has-part Y“ Aussage.

Quality Statement Unit Die Quality Statement Unit ist ebenfalls eine Statement Unit mit nur einer Objekt-Position. Sie modelliert eine beliebige Qualität ihrer Subjekt-Ressource.

Weight Measurement Statement Unit Die Weight Measurement Statement Unit besitzt vier Objekt-Positionen und modelliert eine Gewichtsmessung. Zu den Objekt-Positionen zählen der Mittelwert, die Unter- und die Obergrenze, jeweils als Literals gespeichert, und die Ontologieklasse der Maßeinheit der Messwerte, welche als Klassenressource gespeichert wird.

Weight Measurement Compound Unit Die Weight Measurement Compound Unit kombiniert die Quality Statement Unit und die Weight Measurement Statement Unit zu einer Compound Unit.

Material Entity Item Unit Die Material Entity Item Unit ist eine Compound Unit und dient als Container für alle Semantic Units, die dieselbe Subjekt-Ressource vom Typ Material Entity (BFO:0000040) besitzen.

Für jede dieser Semantic Units muss eine eigenständige KGBB-Klasse angelegt werden,

wobei der KGBB für die NIIU ein fester Bestandteil der Applikation werden soll. Die Interaktionen zwischen den einzelnen KGBBs werden mithilfe von KGBB-Association-Knoten und KGBB-Link-Knoten zwischen Instanzen der KGBB-Klassen modelliert (vgl. Kapitel 3.3). Für die in Kapitel 4.1 angegebenen Anforderungen wird für jeden KGBB genau eine KGBB-Instanz benötigt, mit Ausnahme des NIIU-KGBBs, welcher keine eigene Instanz erhält, da er ein fester Bestandteil der Applikation sein soll und immer dann instanziiert wird, wenn eine neue Named-Individual-Ressource im Wissensgraphen angelegt wird.

4.3 Use Cases

Aus den Anforderungen aus Kapitel 4.1 und der Analyse aus Kapitel 4.2 lassen sich insgesamt sechs Use-Cases ableiten. Eine Use-Case-Beschreibung setzt sich dabei aus einer Use-Case-Nummer und einem Namen zusammen, die den Use-Case identifizieren. Zusätzlich werden die Akteure aufgelistet, die an einem Use-Case beteiligt sind. Dies können sowohl Menschen als auch Softwarekomponenten oder Hardware sein. Häufig wird auch eine Voraussetzung für einen Use-Case definiert. Sie gibt an, welche Bedingungen gelten müssen, bevor der Use-Case ausgelöst werden kann. Die wichtigste Beschreibung innerhalb eines Use-Cases liefert der Ablauf. Er beschreibt, welche Komponenten in welcher Art miteinander interagieren, um das Ziel des Use-Cases zu erreichen. Unter Umständen existieren für den Ablauf mehrere verschiedene Pfade, da z.B. die Eingaben des Benutzers zu unterschiedlichen Abläufen führen oder Fehler bei der Verarbeitung auftreten können. Diese werden als alternative Abläufe dokumentiert. Jeder Ablauf erhält zusätzlich eine eigene Identifikationsnummer. Die abgeleiteten Use-Cases sind:

- UC1: Anlegen einer Compound Unit
- UC2: Anlegen einer Statement Unit
- UC3: Finden einer Semantic Unit
- UC4: Löschen einer Compound Unit
- UC5: Löschen einer Statement Unit
- UC6: Aktualisieren einer Statement Unit

Die genauen Definitionen der Use-Cases sind in den folgenden Tabellen festgehalten. Jeder Use-Case ist dabei unabhängig von den anderen Use-Cases.

Tabelle 4.1: Use-Case: Anlegen einer Compound Unit

Use-Case 1	Anlegen einer Compound Unit
Akteur	Eingabe-Adapter
Voraussetzung	Die Compound Unit ist vom Typ Item Unit oder Quality Measurement Unit
Ablauf (M)	1: Adapter übergibt KGBB-Instanz-URI der Compound Unit, Eingaben für Objekt-Positionen und Informationen von benötigten Association-Knoten und Benutzerdaten an das System 2: System lädt die KGBB-Instanz 3: System verifiziert die Eingaben 4: System erstellt neue NIIU 5: System erstellt neue Compound Unit 6: System erstellt assoziierte Semantic Units 7: System speichert die erstellten Semantic Units 8: System gibt erstellte Semantic Units an den Eingabe-Adapter zurück
Alternativer Ablauf (A1)	1a: Adapter gibt KGBB-Instanz-URI der Compound Unit, Argumente für benötigte Assoziationen, Benutzerdaten und URI der Ausgangs-Semantic-Unit an das System 4a1: System lädt Ausgangs-Semantic-Unit 4a2: System verifiziert Constraints der Ausgangs-Semantic-Unit
Alternativer Ablauf (A2)	3a: System erkennt, dass die maximale Anzahl an assoziierten Semantic Units dieses Typs bereits erreicht ist 3b: System bricht den Vorgang ab und gibt eine Fehlermeldung aus
Alternativer Ablauf (A3)	3c: System erkennt, dass eine benötigte Eingabe fehlt 3d: System bricht den Vorgang ab und gibt eine Fehlermeldung aus

Tabelle 4.2: Use-Case: Anlegen einer Statement Unit

Use-Case 2	Anlegen einer Statement Unit
Akteur	Eingabe-Adapter
Voraussetzung	Die Ausgangs-Semantic-Unit existiert bereits
Ablauf (M)	1: Adapter übergibt KGBB-Instanz-URI der Statement Unit, Eingaben für Objekt-Positionen und Informationen von benötigten Association-Knoten und Benutzerdaten an das System 2: System lädt die KGBB-Instanz 3: System lädt die Ausgangs-Semantic-Unit 4: System verifiziert Eingaben 5: System verifiziert Constraints der Ausgangs-Semantic-Unit 6: System erstellt neue Statement Unit 7: System erstellt neue Objekt-Position Instanzen der Statement Unit 8: System erstellt verlinkte Semantic Units 9: System speichert die erstellten Semantic Units 10: System gibt erstellte Semantic Units an den Eingabe-Adapter zurück
Alternativer Ablauf (A1)	3a: System erkennt, dass die maximale Anzahl an assoziierten Semantic Units dieses Types bereits erreicht ist 3b: System bricht den Vorgang ab und gibt eine Fehlermeldung aus
Alternativer Ablauf (A2)	7a: System erkennt, dass eine benötigte Eingabe fehlt 7b: System bricht den Vorgang ab und gibt eine Fehlermeldung aus

Tabelle 4.3: Use-Case: Finden einer Semantic Unit

Use-Case 3	Finden einer Semantic Unit
Akteur	Eingabe-Adapter
Voraussetzung	Der Adapter verfügt über die URI der zu findenden Semantic Unit
Ablauf (M)	1: Adapter übergibt URI der Statement Unit und Benutzerdaten an das System 2: System lädt die Semantic Unit 3: System gibt die Semantic Unit zurück
Alternativer Ablauf (A1)	1a: Adapter übergibt URI der Subjekt Ressource, KGBB-Instanz der Semantic Unit und Benutzerdaten an das System
Alternativer Ablauf (A2)	2a: System erkennt, dass die Semantic Unit nicht existiert 3a: System bricht den Vorgang ab und gibt eine Fehlermeldung aus
Alternativer Ablauf (A3)	2b: System erkennt, dass die Semantic Unit nicht mit der gegebenen KGBB-Instanz übereinstimmt 3b: System bricht den Vorgang ab und gibt eine Fehlermeldung aus

Tabelle 4.4: Use-Case: Löschen einer Compound Unit

Use-Case 4	Löschen einer Compound Unit
Akteur	Eingabe-Adapter
Voraussetzung	Der Adapter verfügt über die URI der zu löschenden Compound Unit
Ablauf (M)	1: Adapter übergibt URI der Compound Unit und Benutzerdaten an das System 2: System lädt die Compound Unit 3: System setzt die Compound Unit auf *gelöscht* 4: System lädt alle assoziierten Semantic Units 5: System setzt alle assoziierten Semantic Units auf *gelöscht* 6: System speichert alle geänderten Semantic Units
Alternativer Ablauf (A1)	2a: System erkennt, dass die Compound Unit nicht existiert 2b: System bricht den Vorgang ab und gibt eine Fehlermeldung aus
Alternativer Ablauf (A2)	2c: System erkennt, dass die Compound Unit bereits gelöscht wurde 2d: System bricht den Vorgang ab und gibt eine Fehlermeldung aus

Tabelle 4.5: Use-Case: Löschen einer Statement Unit

Use-Case 5	Löschen einer Statement Unit
Akteur	Eingabe-Adapter
Voraussetzung	Der Adapter verfügt über die URI der zu löschenden Statement Unit und die URI der Ausgangs-Semantic-Unit
Ablauf (M)	1: Adapter übergibt URI der Statement Unit, URI der Ausgangs-Semantic-Unit und Benutzerdaten an das System 2: System lädt die Statement Unit 3: System setzt die Statement Unit auf *gelöscht* 4: System lädt die Ausgangs-Semantic-Unit 5: System aktualisiert die Ausgangs-Semantic-Unit 6: System lädt alle assoziierten Semantic Units 7: System setzt alle assoziierten Semantic Units auf *gelöscht* 8: System speichert alle geänderten Semantic Units
Alternativer Ablauf (A1)	2a: System erkennt, dass die Statement Unit nicht existiert 2b: System bricht den Vorgang ab und gibt eine Fehlermeldung aus
Alternativer Ablauf (A2)	2c: System erkennt, dass die Statement Unit bereits auf *gelöscht* gesetzt wurde 2d: System bricht den Vorgang ab und gibt eine Fehlermeldung aus
Alternativer Ablauf (A4)	4a: System erkennt, dass die Ausgangs-Semantic-Unit bereits auf *gelöscht* gesetzt wurde 4b: System bricht den Vorgang ab und gibt eine Fehlermeldung aus

Tabelle 4.6: Use-Case: Aktualisieren einer Statement Unit

Use-Case 6	Aktualisieren einer Statement Unit
Akteur	Eingabe-Adapter
Voraussetzung	Der Adapter verfügt über die URI der zu aktualisierenden Statement Unit und die URI der Ausgangs-Semantic-Unit
Ablauf (M)	1: Adapter übergibt URI der Statement Unit, URI der Ausgangs-Semantic-Unit, Eingaben für Objekt-Positionen und Informationen von benötigten Association-Knoten und Benutzerdaten an das System 2: System lädt die Statement Unit 3: System lädt die Ausgangs-Semantic-Unit 4: System verifiziert die Eingaben 5: System erstellt neue Objekt-Position Instanzen der Statement Unit 6: System setzt alte Objekt-Position Instanzen der Statement Unit auf *gelöscht* 7: System erstellt verlinkte Semantic Units 8: System aktualisiert die Ausgangs-Semantic-Unit 9: System speichert die erstellten Semantic Units 10: System gibt erstellte Semantic Units zurück
Alternativer Ablauf (A1)	2a: System erkennt, dass die Statement Unit nicht existiert 2b: System bricht den Vorgang ab und gibt eine Fehlermeldung aus
Alternativer Ablauf (A2)	2c: System erkennt, dass die Statement Unit bereits auf *gelöscht* gesetzt wurde 2d: System bricht den Vorgang ab und gibt eine Fehlermeldung aus
Alternativer Ablauf (A3)	3a: System erkennt, dass die Ausgangs-Semantic-Unit bereits auf *gelöscht* gesetzt wurde 3b: System bricht den Vorgang ab und gibt eine Fehlermeldung aus
Alternativer Ablauf (A4)	4a: System erkennt, dass eine benötigte Eingabe fehlt 4b: System bricht den Vorgang ab und gibt eine Fehlermeldung aus

5 Implementierung

Dieses Kapitel beschäftigt sich mit der Umsetzung einer Softwareapplikation zur Verwaltung von Semantic Units basierend auf KGBBs. Es wird zunächst der Projektaufbau beschrieben, sowie der konzeptionelle Aufbau der Applikation. Weitere Unterkapitel erläutern die Funktionsweise und Komponenten der KGBB-Engine im Detail.

5.1 Projektaufbau

Der Projektaufbau und die Projektstruktur orientieren sich an denen des ORKG Backend[1]. So wird ebenfalls das Design Pattern *Ports und Adapter* für die logische Struktur der Applikation verwendet, sowie das Open-Source Build-Tool Gradle[2] zum Erstellen des lauffähigen Programms. Im Gegensatz zum ORKG Backend wird für das Projekt nicht die Programmiersprache Kotlin verwendet, sondern die Programmiersprache Java. Beide Programmiersprachen sind auf Java Virtual Machine lauffähig, weshalb sie untereinander kompatibel sind und miteinander kommunizieren können. Folglich könnte das Projekt sehr einfach in ein Kotlin-Projekt integriert werden.

Eine Aufteilung des Programmcodes in einzelne Packages ergibt sich durch das Design Pattern *Ports und Adapter*, welches für die Applikation verwendet wird. Die Implementierung der Applikationslogik erfolgt im Package „domain“. Benötigte Interfaces für getriebene Ports befinden sich im Package „spi“, wohingegen benötigte Interfaces für treibende Ports im Package „api“ abgelegt werden. Implementierungen für Adapter befinden sich im Package „adapter“, wobei zusätzlich eine Unterscheidung zwischen Eingabe- und Ausgabeadaptern in den Subpackages „input“ und „output“ erfolgt. Letzteres ist eine Konvention für treibende und getriebene Ports. Durch das *Ports und Adapter* Design Pattern ist die Applikation nicht an eine spezielle Technologie gebunden und bleibt somit Technologie-unabhängig.

5.2 KGBB-Engine

In diesem Kapitel werden die Ports der KGBB-Engine vorgestellt, sowie die Implementierung der verschiedenen Funktionen der KGBB-Engine. Eine Übersicht über die Anordnung

[1] https://gitlab.com/TIBHannover/orkg/orkg-backend
[2] https://gradle.org/

der *Ports und Adapter* ist in Abbildung 5.1 dargestellt. Im Zentrum der Abbildung befindet sich das Hexagon bzw. die KGBB-Engine. Sie beinhaltet die Applikationslogik zum Verwalten der Semantic Units basierend auf KGBBs. Auf der oberen linken Seite des Hexagons befinden sich die treibenden Adapter. Sie greifen über den treibenden Port „Semantic Units verwalten“ auf die Applikation zu. Auf der unteren und rechten Seite des Hexagons befinden sich die getriebenen Adapter. Sie werden von der Applikation gesteuert und werden von ihr benötigt, um die Applikationslogik zu implementieren. Bei einem Großteil der getriebenen Adapter handelt es sich um „In-Memeory“ Adapter, welche die benötigte Funktionalität für die Applikation bereitstellen, durch ihren begrenzten Umfang aber ausschließlich dem Testen dienen.

5.2.1 Ausgabe-Ports

Im Folgenden werden zunächst die Ausgabe-Ports der KGBB-Engine genauer vorgestellt. Dabei handelt es sich durchweg um getriebene Ports, welche die Funktionen beschreiben, die von der KGBB-Engine benötigt werden, um die Applikationslogik zu implementieren. Hierbei wird zu einem Großteil das Repository-Pattern verwendet, um bestimmte Objekt-Instanzen aus dem jeweiligen Port zu laden und zu speichern [43].

5.2.1.1 KGBB Repository

Das KGBB Repository dient als Speicher für die KGBB-Klassen, auf welche die KGBB-Engine zugreifen kann. Die KGBB-Engine kann dabei selbst keine neuen KGBB-Klassen erstellen oder bestehende KGBBs modifizieren. Folglich kann das KGBB Repository lediglich gelesen werden. Jede KGBB-Klasse ist dabei eindeutig über eine UPRI auffindbar.

5.2.1.2 KGBB Application Specification Repository

Das KGBB Application Specification Repository beinhaltet einen Graphen, der die semantische Beschreibung der Applikation liefert und aus mehreren KGBB-Instanzen besteht, die durch KGBB-Association-Knoten und KGBB-Link-Knoten miteinander verknüpft sind und die Interaktionsmöglichkeiten zwischen den einzelnen KGBB-Instanzen definieren. Der Graph kann von der KGBB-Engine nicht modifiziert werden.

5.2.1.3 Storage Model Repository

Der Port für das Storage Model Repository ermöglicht es der KGBB-Engine, Storage-Modelle für die Semantic Units zu beziehen, welche für die Speicherung der Semantic Units benötigt werden. Darunter befinden sich die Storage-Modelle für die Semantic Unit Ressour-

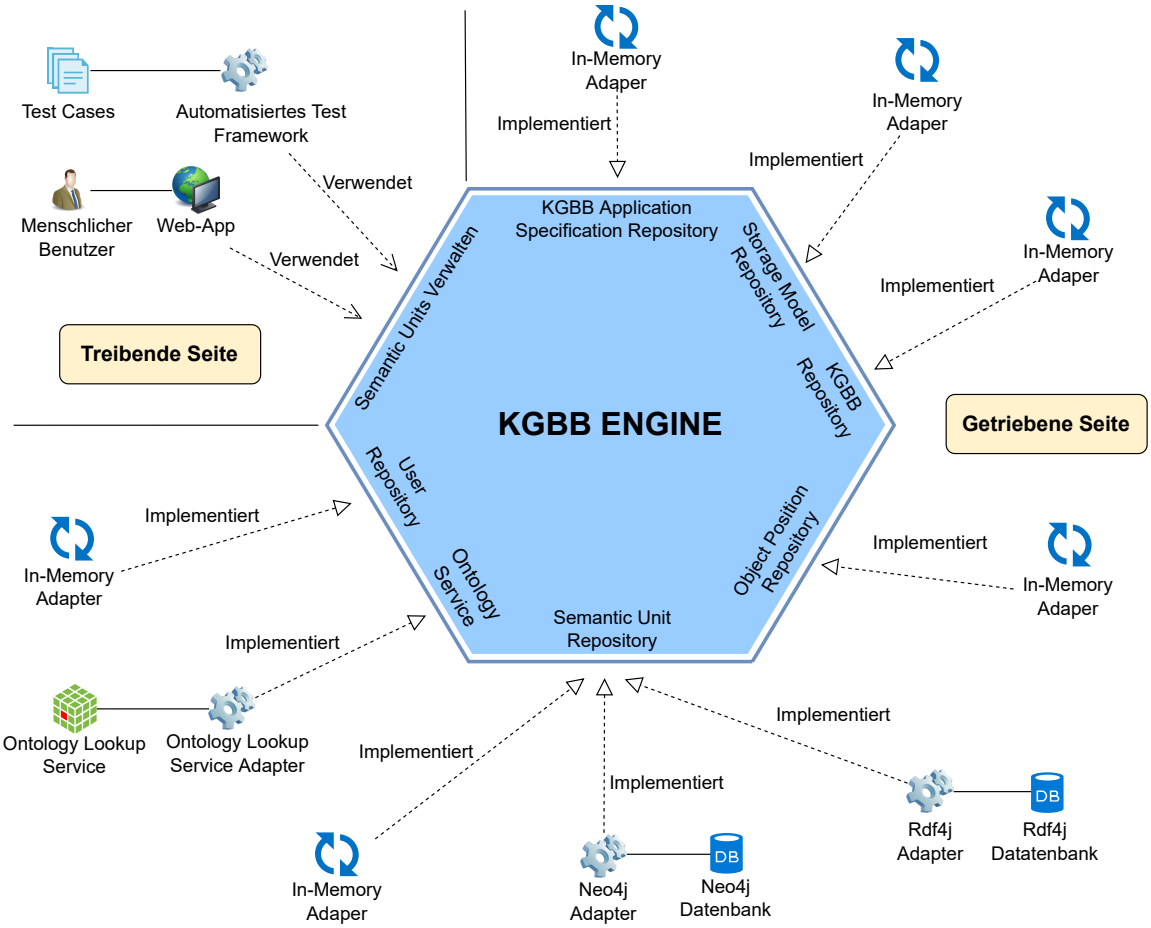


Abbildung 5.1: Hexagonale Architektur der KGBB-Engine mit einem treibenden Port (Semantic Units verwalten), zwei Adapter für den treibenden Port (Automatisiertes Test Framework, Web-App), sieben getriebenen Ports (KGBB Application Specification Repository, Storage Model Repository, KGBB Repository, Object Position Repository, Semantic Unit Repository, Ontology Service, User Repository) und neuen Adapter für die getriebenen Ports (Rdf4j Adapter, Neo4j Adapter, Ontology Lookup Service und sechs In-Memory Adapter).

cen, die Subjekt-Ressourcen und die Storage-Modelle für die einzelnen Objekt-Positionen. Jedes Storage-Modell ist über eine eindeutige UPRI auffindbar.

5.2.1.4 Object Position Repository

Das Object Position Repository dient als Speicher für die Ontologieklassen der Objekt-Positionen, die von Statement KGBBs verwendet werden. Auch das Object Position Repository kann von der KGBB-Engine nur gelesen werden. Jede Objekt-Positions-Klasse ist eindeutig über eine UPRI auffindbar.

5.2.1.5 Ontology Service

Der Port für den Ontology Service ermöglicht der KGBB-Engine, Ontologieinformationen über Klassen-Ressourcen von einem externen Service zu beziehen. Diese beinhalten das Auflösen von Hierarchieinformationen, sowie das Auflösen von Klassenlabels. Mit diesen Informationen kann die KGBB-Engine Benutzereingaben validieren und ergänzen.

5.2.1.6 Semantic Unit Repository

Das Semantic Unit Repository dient als Speicher für Semantic Unit Instanzen, welche die KGBB-Engine lesen, modifizieren und speichern kann. Zusätzlich stellt das Semantic Unit Repository einige spezifische Funktionen bereit, um die Verwaltung der Semantic Units zu optimieren. Jede Semantic Unit ist dabei über eine eindeutige UPRI auffindbar.

5.2.1.7 User Repository

Das User Repository ermöglicht der KGBB-Engine, Benutzerinformationen zu validieren bzw. zu autorisieren.

5.2.2 Eingabe-Ports

Die KGBB-Engine verfügt nur über einen einzigen treibenden Port, der es Akteuren ermöglicht, Eingaben an die KGBB-Engine zu übergeben, um Semantic Units organisieren und verwalten zu können. Er spiegelt gleichzeitig die API der Applikation wider, welche eine Beschreibung der Use-Cases ist.

5.2.3 Domain

Basierend auf den Use-Cases aus Kapitel 4.3 muss die Applikation sechs Funktionen zur Verwaltung von Semantic Units bereitstellen, welche die API der Applikation widerspiegeln. In den folgenden Unterkapiteln wird die Implementierung dieser Funktionen am Beispiel

von Anna erklärt. Anna ist eine Biologin und möchte in ihrer Forschungsarbeit verschiedene Arten von Insekten beschreiben und katalogisieren. Dabei möchte sie jedes einzelne Individuum nach seinen äußerlichen Merkmalen beschreiben und in einem Wissensgraphen dokumentieren, damit auch andere Biologen von ihren Erkenntnissen profitieren können.

5.2.3.1 Anlegen einer Compound Unit

Anna möchte über das Interface der KGBB-Engine eine neue Beschreibung einer Ameise beginnen. Dazu muss sie eine neue Material Entity Item Unit (vgl. Kapitel 4.1) mithilfe der KGBB-Engine anlegen.

Das Anlegen einer Compound Unit erfolgt nach dem Ablaufschema des Use-Cases aus Tabelle 4.1. Dazu muss Anna der KGBB-Engine eine KGBB-Instanz URI der anzulegenden Compound Unit, ihre Benutzerdaten, benötigte Eingaben zum Erstellen weiterer assoziierter Semantic Units und optional eine URI für eine bereits existierende Semantic Unit, die als Ausgangs-Semantic-Unit der Compound Unit dienen soll, übergeben.

Im ersten Schritt werden Annas Benutzerdaten verifiziert. Dafür werden die übergebenen Benutzerdaten an das User Repository weitergegeben. Als Antwort erhält die KGBB-Engine, ob die Benutzerdaten gültig sind. Falls nicht, wird der Vorgang abgebrochen und Anna erhält eine Fehlermeldung.

Wurden Annas Benutzerdaten hingegen erfolgreich verifiziert, wird im zweiten Schritt die Ausgangs-Semantic-Unit überprüft, sofern vorhanden. Als Ausgangs-Semantic-Unit sind für Compound Units nur Semantic Units vom Typ NIIU zugelassen. Diese wird aus dem Semantic Unit Repository anhand der übergebenen Ausgangs-Semantic-Unit URI geladen.

Im dritten Schritt wird die KGBB-Instanz der Ausgangs-Semantic-Unit aus dem KGBB Application Specification Repository geladen. Sie enthält Informationen über die gültigen Assoziationen, die von der Ausgangs-Semantic-Unit ausgehen dürfen und ihrer maximal erlaubten Anzahl pro KGBB-Instanz. Existiert keine Assoziation zwischen der KGBB-Instanz der Ausgangs-Semantic-Unit und der von Anna angegebenen KGBB-Instanz oder die maximale Anzahl an Assoziationen ist bereits erreicht, wird eine Fehlermeldung ausgegeben und der Vorgang wird abgebrochen. Sind jedoch alle Bedingungen erfüllt oder keine Ausgangs-Semantic-Unit spezifiziert, wird die KGBB-Instanz der Compound Unit aus dem KGBB Application Specification Repository geladen. Sie enthält die nötigen Informationen zum Verarbeiten der Eingaben im vierten Schritt.

Wenn es sich bei der Eingabe um die URI einer Klasse handelt, werden zunächst mithilfe des Ontology Services die Constraints verifiziert. Kommt es zu einem Fehler, wird der gesamte Vorgang abgebrochen und Anna erhält eine Fehlermeldung. Wurden alle Constraints erfolgreich verifiziert, so wird eine neue NIIU erstellt. Handelt es sich hingegen bei der Eingabe um die URI einer Instanz, wird die entsprechende Semantic Unit, die die Instanz als ihre Subjekt-Ressource führt, im Semantic Unit Repository gesucht. Dabei darf es sich aus-

schließlich um Semantic Units vom Typ NIIU handeln. Wenn die angegebene NIIU nicht im Semantic Unit Repository existiert, kommt es zum Fehlerfall und der gesamte Vorgang wird abgebrochen. Andernfalls werden die Constraints mithilfe des Ontology Services verifiziert. Werden nicht alle Constraints erfüllt, kommt es zu einem Fehlerfall. Ungültige oder unvollständige Eingaben führen ebenfalls zu einem Fehlerfall.

Sofern alle Eingaben erfolgreich verarbeitet werden konnten, wird nun eine neue Compound Unit erstellt. Diese wird allerdings noch nicht im Semantic Unit Repository gespeichert, da zunächst im fünften Schritt noch weitere assoziierte Semantic Units erstellt werden. Die Informationen, welche Semantic Units zusätzlich erstellt werden müssen, ist in der bereits geladenen KGBB-Instanz der Compound Unit hinterlegt. Für jede Assoziation wird eine neue Semantic Unit erstellt und verlinkt. Kommt es beim Erstellen der assoziierten Semantic Units zu Fehlern, da z.B. benötigte Argumente fehlen, wird der gesamte Vorgang abgebrochen.

Erst wenn alle Semantic Units erfolgreich erstellt wurden, werden sie im sechsten Schritt an das Semantic Unit Repository zum Speichern übergeben. Die KGBB-Engine gibt dann alle im Prozess erstellten Semantic Units zurück. Der gesamte Prozess ist als Sequenzdiagramm im Anhang A in der Abbildung A.1 dargestellt.

Anna hat nun eine neue Material Entity Item Unit mithilfe der KGBB-Engine angelegt. Im Hintergrund hat die KGBB-Engine zusätzlich eine NIIU für die Ameisen-Ressource angelegt. Im Folgenden kann Anna die angelegte Ressource für die Ameise weiter beschreiben.

5.2.3.2 Anlegen einer Statement Unit

Nachdem Anna eine Ressource für die Ameise angelegt hat, zusammen mit einer Ameisen Material Entity Item Unit und einer Ameisen NIIU, möchte sie die Anatomie der Ameise näher beschreiben. Als Erstes möchte Anna mittels einer Parthood-Beziehung beschreiben, dass die Ameise einen Kopf besitzt. Dazu muss Anna eine neue Has-Part Statement Unit (vgl. Kapitel 4.1) mithilfe der KGBB-Engine anlegen.

Das Anlegen einer Statement Unit erfolgt nach dem Ablaufschema des Use-Cases zum Anlegen einer Statement Unit (siehe Tabelle 4.2). Dazu muss Anna der KGBB-Engine eine KGBB-Instanz URI der anzulegenden Statement Unit, ihre Benutzerdaten, benötigte Eingaben zum Erstellen weiterer assoziierter Semantic Units und eine URI für eine bereits existierende Semantic Unit, dessen Subjekt für die Statement Unit wiederverwendet werden soll, übergeben.

Im ersten Schritt verifiziert die KGBB-Engine Annas Benutzerdaten. Dieser Prozess ist analog zur Benutzerverifikation aus Kapitel 5.2.3.1.

Im zweiten Schritt wird die Ausgangs-Semantic-Unit geladen. Diese wird aus dem Semantic Unit Repository anhand der übergebenen Ausgangs-Semantic-Unit URI geladen. Als Ausgangs-Semantic-Unit sind für Statement Units alle Semantic Unit Typen zugelassen.

Im dritten Schritt wird die KGBB-Instanz der Ausgangs-Semantic-Unit aus dem KGBB Application Specification Repository geladen. Im Fall von Anna wäre die Ausgangs-Semantic-Unit die zuvor angelegte Ameisen Material Entity Item Unit. Eine Verifikation der Assoziation zwischen Statement Unit und Ausgangs-Semantic-Unit erfolgt analog zur Verifikation der Assoziation zwischen Compound Unit und Ausgangs-Semantic-Unit aus Kapitel 5.2.3.1. Sofern die Verifikation erfolgreich war, wird eine neue Statement Unit erstellt. Auch hier wird die neu erstellte Statement Unit noch nicht im Semantic Unit Repository gespeichert, da zunächst die Eingaben verarbeitet werden und ggf. noch weitere assoziierte Semantic Units erstellt werden müssen.

Im vierten Schritt werden die Eingaben für die einzelnen Objekt-Positionen verarbeitet. Handelt es sich bei einer Eingabe um ein Literal, wird eine neue Instanz der Literal Objekt-Position Klasse mit den entsprechenden Metadaten und einem mit dieser Instanz verbundenem Literal-Object-Node erstellt und der Statement Unit hinzugefügt. Kommt es beim Erstellen dieser Instanz zu einem Fehler bei der Verifikation der Constraints, wird der Vorgang abgebrochen und Anna erhält eine Fehlermeldung. Handelt es sich bei einer Eingabe hingegen um eine Ressource, so werden zunächst die Constraints mithilfe des Ontology Services überprüft, bevor eine Instanz der entsprechenden Ressource Objekt-Position Klasse angelegt wird. Im Fehlerfall wird der Vorgang abgebrochen und Anna erhält eine Fehlermeldung. Eine Ressource kann entweder eine Klasse oder eine Instanz einer Klasse sein.

Im Fall einer Klasse muss geprüft werden, ob in der KGBB-Instanz für die entsprechende Objekt-Position der Eingabe eine Link-Assoziation existiert. Ist dies der Fall, wird eine neue Semantic Unit des entsprechenden Typs erstellt. Anschließend wird eine Instanz der entsprechenden Ressource Objekt-Position Klasse zusammen mit einem mit dieser Instanz verbundenem Resource-Object-Nodes für die Eingabe angelegt und der Statement Unit hinzugefügt. Je nachdem, ob eine verlinkte Semantic Unit erstellt wurde, wird für die Resource-URI des Resource-Object-Nodes die Klassen URI oder die URI der Subjekt Ressource der erstellten Semantic Unit verwendet.

Im Fall einer Instanz muss die entsprechende NIIU aus dem Semantic Unit Repository geladen werden. Wenn die angegebene NIIU nicht im Semantic Unit Repository existiert, kommt es zum Fehlerfall und der gesamte Vorgang wird abgebrochen. Andernfalls werden die Constraints mithilfe des Ontology Services verifiziert. Werden nicht alle Constraints erfüllt, kommt es zu einem Fehlerfall. Ansonsten wird ein neues Resource-Object-Node erstellt, mit der URI der Subjekt Ressource der NIIU als Resource-URI.

Der vierte Schritt wird für jede Objekt-Position der Statement Unit wiederholt. Erst wenn alle Semantic Units erfolgreich erstellt wurden, werden sie im fünften Schritt an das Semantic Unit Repository zum Speichern übergeben. Die KGBB-Engine gibt dann alle im Prozess erstellten Semantic Units zurück. Der gesamte Prozess ist als Sequenzdiagramm im Anhang A in der Abbildung A.2 dargestellt.

Anna hat nun eine neue Parthood-Beziehung mithilfe der KGBB-Engine erstellt. Neben der Has-Part-Statement Unit hat die KGBB-Engine zusätzlich eine neue Ressource für den Kopf der Ameise angelegt und gleichzeitig eine Kopf Material Entity Item Unit, sowie eine Kopf NIIU, sodass die Kopf Ressource ebenfalls weiter beschrieben werden kann.

5.2.3.3 Aktualisieren einer Statement Unit

Während der Beschreibung der Ameise unterläuft Anna ein Fehler. Sie hat sich beim Klassen-Label der Ameise verschrieben. Da sie mit ihrer Modellierung nicht noch einmal von vorne beginnen möchte, will sie das Label korrigieren. Dazu muss sie die NIIU der Ameisen-Ressourcen mithilfe der KGBB-Engine aktualisieren.

Das Aktualisieren einer Statement Unit erfolgt nach dem Ablaufschema des Use-Cases zum Aktualisieren einer Statement Unit (siehe Tabelle 4.6). Dazu muss Anna der KGBB-Engine die URI der zu aktualisierenden Statement Unit, ihre Benutzerdaten, Eingaben für zu aktualisierende Objekt-Positionen, sowie benötigte Eingaben zum Erstellen weiterer assoziierter Semantic Units und eine optionale URI für eine bereits existierende Semantic Unit, die das Subjekt der Statement Unit angibt, übergeben.

Im ersten Schritt verifiziert die KGBB-Engine Annas Benutzerdaten. Dieser Prozess ist analog zur Benutzerverifikation aus Kapitel 5.2.3.1.

Im zweiten Schritt wird die Statement Unit anhand der angegebenen URI aus dem Semantic Unit Repository geladen. Wird die Statement Unit nicht gefunden oder wurde die Statement Unit bereits als *gelöscht* markiert, wird der Vorgang abgebrochen und Anna erhält eine Fehlermeldung. Ansonsten wird darauf die KGBB-Instanz der Statement Unit aus dem KGBB Application Specification Repository geladen. Sie enthält die benötigten Informationen zum Verarbeiten der Eingaben.

Im dritten Schritt werden die Eingaben verarbeitet. Für Eingaben, bei denen es sich um ein Literal handelt, wird die aktuelle Objekt-Position Instanz auf *gelöscht* gesetzt und eine neue Instanz derselben Objekt-Position-Klasse mit den entsprechenden Metadaten und dem neuen Literal-Object-Node erstellt. Kommt es bei der Verifikation der Contraints zu Fehlern, wird der gesamte Vorgang abgebrochen und Anna erhält eine Fehlermeldung. Ähnlich sieht der Prozess für Ressourcen-Eingaben aus, die die URI einer Klasse enthalten. Hier werden die Constraints mithilfe des Ontology Services verifiziert. Die aktuelle Objekt-Position Instanz wird auf *gelöscht* gesetzt und eine neue Instanz derselben Objekt-Positions Klasse mit den entsprechenden Metadaten und dem neuen Resource-Object-Node erstellt. Ein Fehler

bei der Verifikation führt zum Abbruch des gesamten Vorgangs. Für Instanzen-Eingaben muss analog zu Kapitel 5.2.3.2 die entsprechende NIIU aus dem Semantic Unit Repository geladen werden. Wenn die angegebene NIIU nicht im Semantic Unit Repository existiert, kommt es zu einem Fehlerfall. Sofern eine Ausgangs-Semantic-Unit URI übergeben wurde, muss diese ebenfalls aus dem Semantic Unit Repository geladen werden, damit diese entsprechend aktualisiert werden kann. Anschließend werden die Constraints mittels des Ontology Services verifiziert. Die aktuelle Objekt-Position Instanz wird auf *gelöscht* gesetzt und eine neue Instanz derselben Objekt-Position Klasse mit den entsprechenden Metadaten und dem neuen Resource-Object-Node erstellt. Ungültige Eingaben oder ein Fehler bei der Verifikation führen zum Abbruch des gesamten Vorgangs. Der dritte Schritt wird für jede Objekt-Position der Statement Unit wiederholt.

Im vierten Schritt werden die geänderten Semantic Units an das Semantic Unit Repository zum Speichern übergeben. Am Ende des Prozesses gibt die KGBB-Engine die geänderte Statement Unit zurück. Der gesamte Prozess ist als Sequenzdiagramm im Anhang A in der Abbildung A.3 dargestellt.

Anna konnte mithilfe der KGBB-Engine das Klassen-Label der Ameisen-Ressource aktualisieren.

5.2.3.4 Löschen einer Compound Unit

Als Anna in der darauf folgenden Woche ein weiteres Insekt beschreiben will, unterläuft ihr erneut ein Fehler. Denn sie hat bemerkt, dass sie genau diese Ameise schon in der vorherigen Woche beschrieben hat. Sie möchte die neu erstellte Beschreibung wieder löschen. Dazu muss sie die neu erstellte Ameisen Material Entity Item Unit mithilfe der KGBB-Engine löschen.

Das Löschen einer Compound Unit erfolgt nach dem Ablaufschema des Use-Cases zum Löschen einer Compound Unit (siehe Tabelle 4.4). Dazu muss Anna der KGBB-Engine die URI der zu löschenden Compound Unit, sowie ihre Benutzerdaten übergeben.

Im ersten Schritt verifiziert die KGBB-Engine Annas Benutzerdaten. Dieser Prozess ist analog zur Benutzerverifikation aus Kapitel 5.2.3.1.

Im zweiten Schritt wird die Compound Unit anhand der angegebenen URI aus dem Semantic Unit Repository geladen. Wenn die Compound Unit nicht existiert oder die Compound Unit bereits gelöscht wurde, erhält Anna eine Fehlermeldung und der Vorgang kann nicht fortgesetzt werden.

Ansonsten wird im dritten Schritt die Compound Unit auf *gelöscht* gesetzt. Dabei wird die Compound Unit nicht aus dem Semantic Unit Repository entfernt, sondern nur der Status *currentVersion* auf den Wert *false* gesetzt und eine Referenz auf Annas Benutzerkonto hinterlegt. Zusätzlich werden alle assoziierten Semantic Units aus dem Semantic Unit Repository geladen und ebenfalls und in der gleichen Art und Weise auf *gelöscht* gesetzt. Je nach

Semantic Unit Typen kaskadiert dieser Vorgang über weitere assoziierte Semantic Units. Eine Ausnahme stellt die NIIU an der Subjekt Ressource der Compound Unit dar. Sie wird nur dann gelöscht, wenn es keine anderen Semantic Units im gesamten Wissensgraphen gibt, welche die Subjekt Ressource der NIIU referenzieren.

Im vierten Schritt werden die auf *gelöscht* gesetzten Semantic Units an das Semantic Unit Repository zum Speichern übergeben. Die KGBB-Engine gibt Anna keine Rückgabewerte. Der gesamte Prozess ist als Sequenzdiagramm im Anhang A in der Abbildung A.4 dargestellt.

Anna konnte mithilfe der KGBB-Engine die doppelte Beschreibung der Ameise mit nur einer Aktion wieder löschen.

5.2.3.5 Löschen einer Statement Unit

Als Anna ein paar Wochen später noch einmal die Beschreibung der Ameise durchgeht, bemerkt sie, dass sie aus Versehen die Ressource des Kopfes in Relation zu einer Bein-Ressource der Ameise gesetzt hat. Sie möchte diese Parthood-Beziehung nun wieder löschen. Dazu muss Anna die zugehörige Has-Part-Statement Unit mithilfe der KGBB-Engine löschen.

Das Löschen einer Statement Unit erfolgt nach dem Ablaufschema des Use-Cases zum Löschen einer Statement Unit (siehe Tabelle 4.5). Dazu muss Anna der KGBB-Engine die URI der zu löschenden Statement Unit, ihre Benutzerdaten, sowie eine URI für eine bereits existierende Semantic Unit, die beim erstellen der Statement Unit als Ausgangs-Semantic-Unit gedient hat, übergeben.

Im ersten Schritt verifiziert die KGBB-Engine Annas Benutzerdaten. Dieser Prozess ist analog zur Benutzerverifikation aus Kapitel 5.2.3.1.

Im zweiten Schritt wird die Statement Unit anhand der angegebenen URI aus dem Semantic Unit Repository geladen. Wenn die Statement Unit nicht existiert oder die Statement Unit bereits gelöscht wurde, erhält Anna eine Fehlermeldung und der Vorgang kann nicht fortgesetzt werden.

Im dritten Schritt wird die Ausgangs-Semantic-Unit aus dem Semantic Unit Repository geladen. Auch hier kommt es zu einem Fehler, wenn die entsprechende Semantic Unit nicht im Semantic Unit Repository gefunden werden kann. Als Nächstes wird die KGBB-Instanz der Ausgangs-Semantic-Unit aus dem KGBB Application Specification Repository geladen. Sie enthält Informationen über die gültigen Assoziationen, die von der Ausgangs-Semantic-Unit ausgehen dürfen und ihrer maximal erlauben Anzahl an Assoziationen pro KGBB-Instanz. Existiert keine Assoziation zwischen der KGBB-Instanz der Ausgangs-Semantic-Unit und der Statement Unit oder die maximale Anzahl an Assoziationen wurde bereits erreicht, wird eine Fehlermeldung ausgegeben und der Vorgang wird abgebrochen. Ansonsten wird eine Referenz auf Annas Benutzerkonto an der Statement Unit hinterlegt und die Statement Unit auf *gelöscht* gesetzt. Zudem werden alle Literal und Ressourcen Objekt-Position

Instanzen der Statement Unit auf *gelöscht* gesetzt. Eine Referenz auf Annas Benutzerkonto wird ebenfalls hinterlegt.

Im vierten Schritt wird die auf *gelöscht* gesetzte Statement Unit an das Semantic Unit Repository zum Speichern übergeben. Die KGBB-Engine gibt Anna keine Rückgabewerte. Der gesamte Prozess ist als Sequenzdiagramm im Anhang A in der Abbildung A.5 dargestellt.

Durch das Löschen der Has-Part-Statement Unit konnte Anna ihre Beschreibung der Ameise wieder korrigieren und musste die Modellierung nicht von vorne beginnen.

5.2.4 Storage-Modelle

Die Storage-Modelle der KGBB-Engine basieren auf LinkML-Templates. Mittels der Definition von Importen können Templates andere Templates integrieren und erweitern. Dies erhöht die Wiederverwendbarkeit der Templates, bedeutet aber auch gleichzeitig, dass die Templates vor der Verwendung zusammengeführt werden müssen. Für die Weiterverarbeitung von Templates bietet LinkML bereits ein stetig wachsendes Ökosystem von Tools an. Jedoch basieren diese Tools auf der Programmiersprache Python und sind deshalb schwierig in ein auf der Java Virtual Machine basierendes Projekt einzubinden. Zudem bieten die Tools nicht den benötigten Low-Level Zugriff auf die verarbeiteten Templates, weshalb die benötigten Funktionalitäten von LinkML für die Java Virtual Machine reimplementiert werden müssen. Da eine vollumfängliche Implementierung der LinkML Funktionalitäten den Rahmen dieser Arbeit übersteigen würde, wird von der KGBB-Engine nur eine begrenzte Anzahl an Funktionalitäten der LinkML-Templates unterstützt:

- Definition von Präfixes
- Definition von Imports (limitiert auf den Präfix „http://orkg.org“)
- Definition von Klassen mit den Attributen *is_a*, *subclass_of* und *slots*
- Definition von Slots mit den Attributen *slot_uri*, *range*, *required* und *multivalued*
- Verwendung der Slot-Typen *boolean*, *date*, *decimal* (mit *min*, *max*), *integer* (mit *min*, *max*), *string* (mit *pattern* und *ifabsent*), *uri* und *class*

Wie bereits in Kapitel 5.2.1.3 erwähnt, besitzt eine Semantic Unit ein Storage-Modell für die Semantic Unit-Ressource, ein Storage-Modell für die Subjekt-Ressource und jeweils ein Storage-Modell für jede Objekt-Position. Für jedes dieser Storage-Modelle wird ein „lineares“ Storage-Modell erstellt, welches die spätere Verarbeitung für die KGBB-Engine erleichtert. Dazu wird die Hierarchie der importierten Storage-Modelle aufgelöst und zu einem einzigen Storage-Modell zusammengefasst. Dabei dürfen Präfixe, Klassen und Slots nicht neu definiert werden. Eine Ausnahme bilden Slots, bei denen der Typ übereinstimmt. Eine Validierung der Constraints erfolgt jedoch nicht. Des Weiteren dürfen keine Zyklen in den Klassendefinitionen vorkommen. Nach dem Auflösen der importierten Storage-Modelle wird die Klassenhierarchie reduziert, d.h. alle Slots aus Elternklassen werden den jeweiligen Klas-

sen angefügt und die Elternklassen entfernt. Die KGBB-Engine verwendet am Ende nur die entsprechende Klasse, die am jeweiligen KGBB bzw. der jeweiligen Objekt-Position hinterlegt ist. Der Prozess ist in Listing 5 schematisch dargestellt. Die resultierenden (linearen) Storage-Modelle werden zur Laufzeit nur einmalig aufgelöst und anschließend für bessere Performance zwischengespeichert.

```yaml
classes:
  ParentClass:
    slots:
      - slot1
slots:
  slot1:
    range: string
```

Listing 2: Elternmodell

```yaml
imports:
  - ParentModel
classes:
  ChildClass:
    is_a: ParentClass
    slots:
      - slot2
slots:
  slot2:
    range: string
```

Listing 3: Kindmodell

```yaml
classes:
  ChildClass:
    slots:
      - slot1
      - slot2
slots:
  slot1:
    range: string
  slot2:
    range: string
```

Listing 4: Abgeleitetes lineares Modell für „ChildClass“

Listing 5: Ableitung des linearen Storage-Modells „ChildClass“ aus dem Eltern- und Kindmodell

5.3 Output-Adapter

Output-Adapter sind Adapter, die einen getriebenen Port der Applikation für eine spezifische Technologie implementieren. Sie werden von der Applikation benötigt, um die Applikationslogik zu implementieren. In den folgenden Kapiteln werden die einzelnen Output-Adapter, die für diese Arbeit umgesetzt wurden, vorgestellt und deren Funktionsweise erläutert.

5.3.1 In-Memory KGBB Repository

Das In-Memory KGBB Repository implementiert einen Adapter für den Port des KGBB Repositories zum Testen der Applikation. Die KGBBs-Klassen sind dafür fest im KGBB Repository hinterlegt und können zur Laufzeit nicht geändert werden. Das In-Memory KGBB Repository beinhaltet zehn KGBBs-Klassen, welche in einer in Abbildung 5.2 dargestellten Klassen-Unterklassen-Hierarchie angeordnet sind. Einzelne KGBB-Instanzen sind direkt über ein assoziatives Array (Java Map) verfügbar.

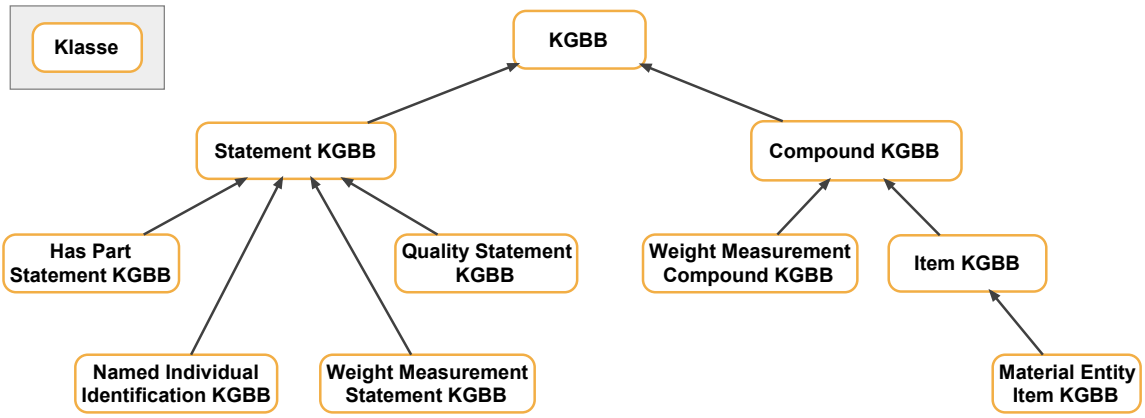


Abbildung 5.2: Hierarchie der KGBBs des In-Memory KGBB Repository

5.3.2 In-Memory Object Position Repository

Das In-Memory Object Position Repository implementiert einen Adapter für den Port des Object Position Repositories zum Testen der Applikation. Die Objekt-Position Klassen sind dafür fest im Object Position Repository hinterlegt und können zur Laufzeit nicht geändert werden. Das In-Memory Object Position Repository beinhaltet zwölf Objekt-Position Klassen, welche in einer in Abbildung 5.3 dargestellten Klassen-Unterklassen-Hierarchie angeordnet sind. Einzelne Objekt-Position Klassen sind direkt über ein assoziatives Array (Java Map) verfügbar.

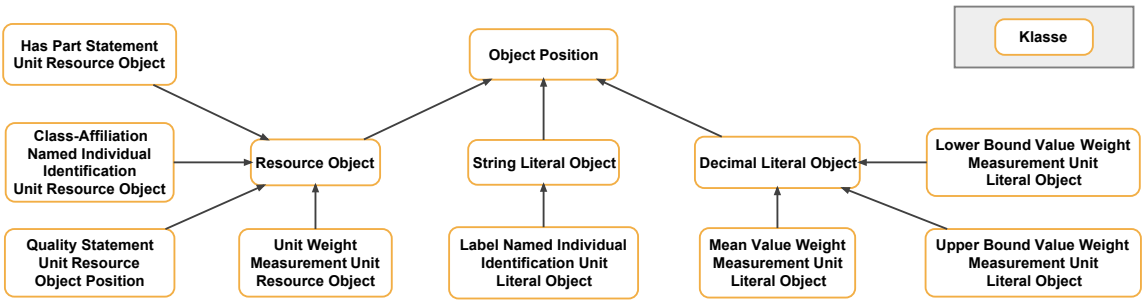


Abbildung 5.3: Hierarchie der Objekt-Positionen des In-Memory Object Position Repository

5.3.3 In-Memory KGBB Application Specification Repository

Das In-Memory KGBB Application Specification Repository implementiert einen Adapter für den Port des KGBB Application Specification Repository zum Testen der Applikation. Der im KGBB Application Specification Repository hinterlegte Graph ist in der In-Memory-Implementierung fest hinterlegt und kann zur Laufzeit nicht geändert werden. Er besteht aus

fünf KGBB-Instanzen, die mit KGBB-Association-Knoten und KGBB-Link-Knoten miteinander verknüpft sind, welche die Interaktionen zwischen den KGBBs modellieren. Der Graph ist in Abbildung 5.4 dargestellt.

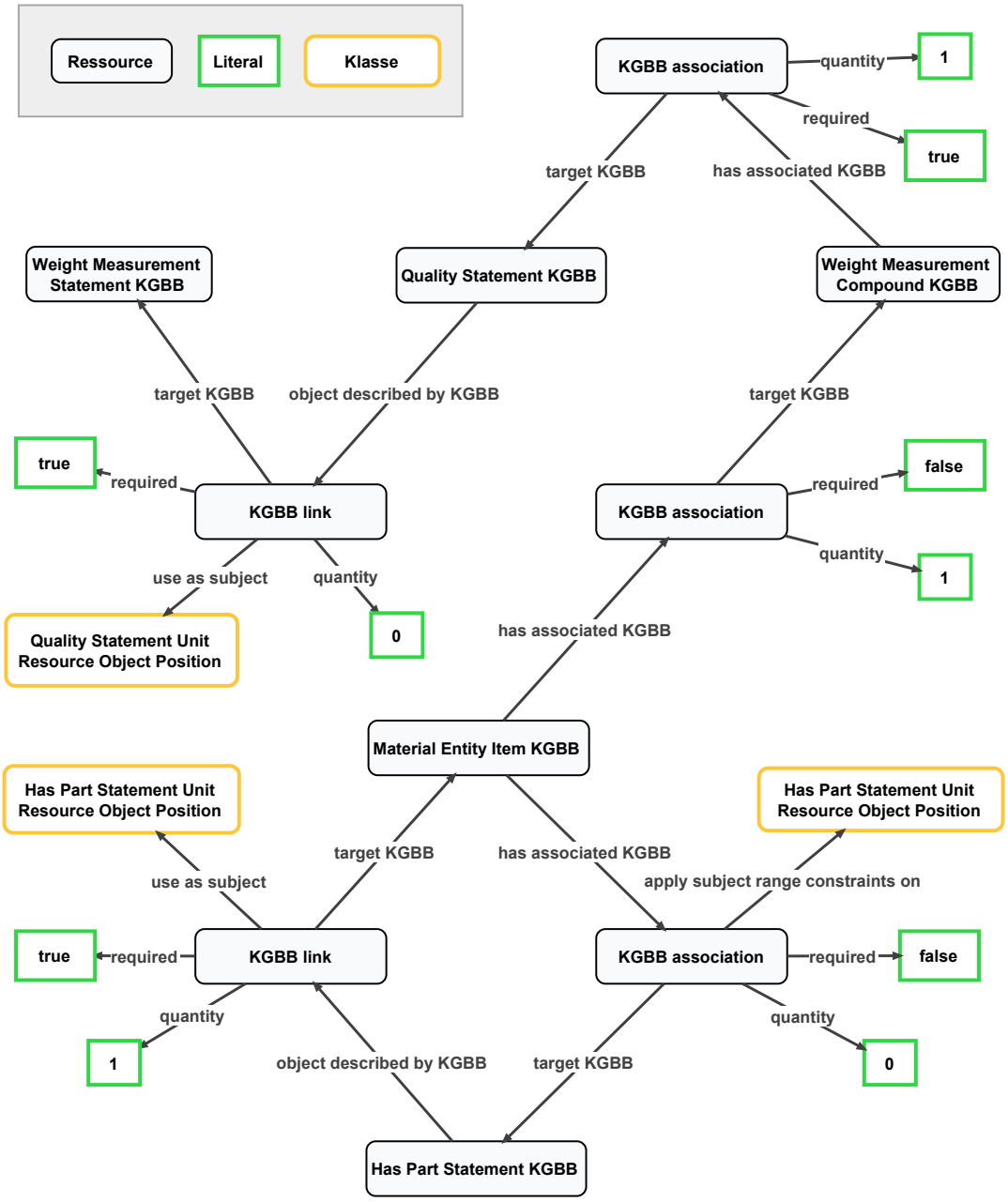


Abbildung 5.4: Modell der KGBB-Instanzen

5.3.4 In-Memory Storage Model Repository

Das In-Memory Storage Model Repository implementiert einen Adapter für den Port des Storage Model Repositories zum Testen der Applikation. Die Storage-Modelle sind im Ressour-

cenordner des Projekts gespeichert. Beim Start der Applikation werden die entsprechenden LinkML-Templates eingelesen und sind mittels eines assoziativen Arrays (Java Map) über ihre jeweilige UPRI verfügbar.

5.3.5 Ontology Lookup Service

Für den Port des Ontology Lookups wird ein Adapter basierend auf dem Ontology Lookup Service [44] implementiert. Der Ontology Lookup Service stellt alle benötigten Ontologien für die verwendeten Beispiele (UBERON[3], BFO[4], NCIT[5]) bereit, schränkt die KGBB-Engine aber auch auf den verwendeten Service ein. Mittels der „hierarchicalAncestors“ API werden Klassenhierarchien über eine Webanfrage aufgelöst. Ein lokales Caching der Ergebnisse sorgt dabei für eine verbesserte Performance bei wiederholten Anfragen. Zudem werden Klassenlabels über die „terms“ API bezogen.

5.3.6 In-Memory Semantic Unit Repository

Die erste Implementierung eines Adapters für das Semantic Unit Repository ist das In-Memory Semantic Unit Repository. Es speichert die In-Memory-Repräsentation der Semantic Units direkt im Arbeitsspeicher in Form eines assoziativen Arrays (Java Map). Folglich bietet diese Implementierung keine persistente Speicherung der Semantic Units über einen Neustart der Applikation hinaus. Das In-Memory Semantic Unit Repository dient hauptsächlich dem Testen der Applikation.

5.3.7 Rdf4j Semantic Unit Repository

Eine weitere Implementierung eines Adapters des Semantic Unit Repositories erfolgt für eine Speicherung in RDF [10]. Dafür wird das Open-Source Framework RDF4J von der Eclipse Foundation verwendet [45]. Es bietet eine Vielzahl von verschiedenen Speichertechnologien für Graphen in RDF und wird bereits in vielen Projekten verwendet. Der Datenaustausch mit RDF4J erfolgt über die Abfragesprache SPARQL [29], welche speziell für Anfragen auf RDF-Wissensgraphen entwickelt wurde. Das RDF4J Semantic Unit Repository muss folglich die Semantic Unit Instanzen anhand des Storage-Modells in SPARQL-Abfragen übersetzen und umgekehrt. Eine Semantic Unit setzt sich dabei aus zwei Teilgraphen zusammen (vgl. Abbildung 3.2). Den Daten-Graphen und den Semantic-Unit-Graphen. Im Daten-Graphen werden die Subjekt-Ressource und die Objekt-Position Instanzen samt Metadaten und Object Nodes (sofern vorhanden) gespeichert. Im Semantic-Unit-Graphen wird nur die Ressource für die Semantic Unit gespeichert, sowie, insofern die Semantic Unit eine Compound Unit

[3] https://obophenotype.github.io/uberon/
[4] https://ifomis.org/bfo/
[5] https://github.com/NCI-Thesaurus/thesaurus-obo-edition

ist, die URI jeder mit ihr assoziierten Semantic Units. Um diese beiden Teilgraphen effizient unterscheiden zu können, werden Named Graphs [46] verwendet. D.h. zu jedem Tripel wird zusätzlich eine Graphen URI gespeichert, sodass das Tripel zu einem Quad erweitert wird. Jeder Semantic Unit Teilgraph erhält eine eigene UPRI. Die UPRI des Daten-Graphen ist dabei gleich der UPRI der Semantic Unit Ressource aus dem Semantic-Unit-Graphen. Als Standardpräfix wird „http://orkg.org/“ (orkg) verwendet, sofern es nicht anders im Storage-Modell spezifiziert ist.

5.3.7.1 Speichern einer Semantic Unit

Für das Speichern einer Semantic Unit Instanz müssen aus der In-Memory-Repräsentation RDF Tripel bzw. Quads generiert werden.

Im ersten Schritt wird der Semantic-Unit-Graph erstellt, welcher die Semantic Unit Ressource enthält und eventuell noch Assoziationsbeziehungen zu assoziierten Semantic Units für den Fall einer Compound Unit. Die Werte der Prädikate und Objekte der Semantic Unit Ressource werden anhand des Storage-Modells der Semantic Unit aus der In-Memory-Repräsentation in die Tripel überführt. Dabei werden feste Slotnamen für die Zuordnung der Slots aus dem Storage-Modell und der In-Memory-Repräsentation verwendet. Zudem erhält jede Semantic Unit Ressource den Typen (rdf:type) „orkg:SemanticUnit“. Dies erleichtert eine spätere Abfrage. Im zweiten Schritt wird der Daten-Graph der Semantic Unit erstellt. Er enthält die Subjekt-Ressource und die Objekt-Position Instanzen samt Metadaten und Object Nodes der Semantic Unit, sofern die Semantic Unit eine Statement Unit ist. Das Verfahren zur Generierung der Prädikate und Objekte für die Subjekt-Ressource und die Object Nodes ist analog zur Semantic Unit Ressource. Die Subjekt-Ressource wird mit dem Typen (rdf:type) „orkg:Subject“ versehen. Resource-Object-Nodes werden mit dem Typen (rdf:type) „orkg:ResourceObject“ gekennzeichnet und Literal-Object-Nodes mit dem Typen (rdf:type) „orkg:LiteralObject“. Anschließend werden die zwei Teilgraphen in einer Transaktion gespeichert. Dabei werden zunächst alle Tripel des Semantic-Unit-Graphen und des Daten-Graphen der Semantic Unit aus dem Wissensgraphen gelöscht und neu gespeichert. Da es sich hierbei um eine Transaktion handelt, kann es nicht zu einem kurzzeitigen Datenverlust kommen. Das vollständige Löschen und Speichern verringert die Anzahl der benötigten Anfragen, da im Vorfeld nicht geprüft werden muss, welche Änderungen durchgeführt werden müssen. Zudem wird die Integrität der Daten erhöht, da es nicht dazu kommen kann, dass Tripel mehrfach im Graphen gespeichert werden.

5.3.7.2 Finden einer Semantic Unit

Für das Finden einer Semantic Unit muss das Storage-Modell der Semantic Unit in eine SPARQL-Abfrage übersetzt werden. Konkret müssen alle Tripel gefunden werden, in der die UPRI der Semantic Unit in der Subjekt-Position vorkommt. Für Tripel, die als Objekt

```
@base <http://orkg.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
<8> {
    <7> a <HasPartStatementKGBB>, <SemanticUnit>;
        rdfs:label "Has-Part Statement Unit";
        <creator> <KGBBEngineUser>;
        <hasSemanticUnitsGraph> <8>;
        dc:description "Has-Part Statement Unit Description";
        <objectDescribedBySemanticUnit> <14>;
        <creationDate> "2022-06-03T16:17:22.6894178"^^xsd:dateTime ;
        <KGBB_URI> <HasPartStatementKGBBInstance>;
        <createdWithApplication> <KGBBEngine>;
        <currentVersion> true;
        <hasSemanticUnitSubject> <0> .
}
<7> {
    <16> a <ResourceObject>,<HasPartStatementUnitResourceObjectPosition>;
        <currentVersion> true;
        <creator> <KGBBEngineUser>;
        <hasConstraint> "http://purl.obolibrary.org/obo/BFO_0000040";
        <createdWithApplication> <KGBBEngine>;
        <inputTypeLabel> "has-part-resource";
        <resourceURI> <9>;
        <creationDate> "2022-06-03T16:17:23.400656"^^xsd:dateTime .
    <0> a <Subject>;
        <resourceObjectInput> <16> .
}
```

Listing 6: RDF Graph einer Has-Part Statement Unit mit einem Resource Object in TriG Notation.

eine Semantic Unit UPRI besitzen, muss zudem abgefragt werden, ob die entsprechende Semantic Unit bereits gelöscht wurde (*currentVersion* besitzt den Wert *false*). Eine referenzierte Semantic Unit gilt genau dann als nicht gelöscht, wenn der Wert von *currentVersion* auf *true* ist und sich die beiden Semantic Units dieselbe Subjekt-Ressource teilen oder die Objekt-Ressource der einen die Subjekt-Ressource der anderen ist, wobei die zur Objekt-Ressource gehörende Objekt-Position Instanz das Prädikat *currentVersion* mit dem Wert *true* besitzen muss. Für eine einfachere Zuordnung wird zusätzlich zu Subjekt, Prädikat, Objekt und *currentVersion* noch der feste Typ (rdf:type) der Semantic Unit abgefragt. Zudem müssen alle Tripel aus dem Daten-Graphen der Semantic Unit abgefragt werden. Dieser besitzt dieselbe UPRI wie die Semantic Unit Ressource, d.h. alle Quads mit der UPRI in der Named-Graph-Position gehören zum Daten-Graphen der Semantic Unit. Die resultierenden Tupel werden anschließend anhand der entsprechenden Storage-Modelle in die In-Memory-Repräsentation der KGBB-Engine überführt. Eine Zuordnung erfolgt anhand einer Gruppierung nach der Subjekt-URI der Tripel. Alle Tripel, bei denen die Subjekt-URI gleich der gesuchten Semantic Unit UPRI sind, gehören folglich zur Semantic Unit Ressource. Die restliche Zuordnung erfolgt anhand der Typen der Resource-Object-Nodes (orkg:ResourceObject) und der Literal-Object-Nodes (orkg:LiteralObject). Die Subjekt-Ressource ist für die In-Memory-Repräsentation nicht relevant, da sie keine zusätzlichen Informationen liefert, und kann deshalb vernachlässigt werden. Mittels des Storage-Modells der Semantic Unit und der Storage-Modelle der Objekt-Positionen werden anschließend die Prädikate und Objekte den jeweiligen Slots der In-Memory-Repräsentation zugeordnet. Durch Unions der Teilabfragen wird nur eine Abfrage für das Finden einer Semantic Unit benötigt.

5.3.7.3 Löschen einer Semantic Unit

Beim Löschen einer Semantic Unit Instanz wird diese lediglich auf *gelöscht* gesetzt und gespeichert. Der Speichervorgang beim Löschen einer Semantic Unit Instanz ist analog zum Speichern einer Semantic Unit Instanz.

5.3.8 Neo4j Semantic Unit Repository

Eine dritte Implementierung eines Adapters für den Port des Semantic Unit Repositories erfolgt für die Open-Source-Graphdatenbank Neo4j [47]. Sie speichert die Semantic Units in Form eines Labeled-Property Graphs. Zum Speichern oder Lesen von Teilgraphen wird die von Neo4j mitgelieferte Abfrage-Sprache Cypher [11, 12] verwendet. Es muss also eine Übersetzung der Semantic Unit Instanzen anhand des Storage-Modells in Cypher-Abfragen erfolgen und umgekehrt. Eine Semantic Unit wird mit mehreren Knoten im Labeled-Property Graph modelliert. Eine Unterteilung zwischen dem Semantic-Unit-Graphen und dem Daten-Graphen der Semantic Unit gibt es nicht. Es wird jeweils ein Knoten für die Semantic Unit Ressource angelegt, für die Subjekt-Ressource und ein Knoten für jede Objekt-Position

Instanz. Die Anzahl der Objekt-Positionen ist dabei abhängig vom jeweiligen KGBB der Semantic Unit. Jeder Knoten wird mit dem Label *URI_Holder* versehen, um eine spätere Performance-Optimierung durch einen Index zu ermöglichen [48].

5.3.8.1 Speichern einer Semantic Unit

Für das Speichern einer Semantic Unit Instanz muss aus der In-Memory-Repräsentation eine Cypher-Abfrage generiert werden. Dazu wird eine einzige große Cypher-Abfrage erstellt, welche sich aus drei Abschnitten zusammensetzt. Als Erstes wird die Teil-Abfrage für den Semantic Unit Knoten erstellt. Es folgt dem grundlegenden Abfrage-Schema aus Listing 7.

```
1  MERGE (identifier:LABEL1:LABEL2:... {URI: $URI})
2      SET identifier.property1 = value1, identifier.property2 = value2, ...
```

Listing 7: Generische Cypher-Teil-Abfrage für einen Knoten.

Dem Knoten wird ein zufälliger Identifier zugeordnet, um ihn in anderen Teilen der Abfrage referenzieren zu können. Der Semantic Unit Knoten wird immer mit dem Knoten-Label *SemanticUnit* versehen. Zudem erhält er den Namen der Klasse des Storage-Modells als Knoten-Label. Alle im Storage-Modell definierten Slots, bei denen es sich um Literals handelt, werden als Labeled-Properties am Knoten gespeichert. Der Name einer jeweiligen Property geht aus dem Storage-Modell für den entsprechenden Slot hervor. Die Werte der Properties werden über fest definierte Slotnamen aus der In-Memory-Repräsentation dem Storage-Modell zugeordnet. Eine Ausnahme stellt die URI eines Knotens dar. Im Gegensatz zu URIs in RDF besitzt ein Knoten in Neo4j keine persistente Kennung, weshalb für jeden Knoten die entsprechende URI als feste Property gesetzt wird. Relationen zu anderen Knoten werden zunächst zwischengespeichert.

Im zweiten Schritt werden die Teil-Abfragen für die Objekt-Position Instanzen erstellt, sofern die Semantic Unit eine Statement Unit ist. Der Prozess ist analog zur Teil-Abfrage des Semantic Unit Knoten. Jede Objekt-Position Instanz erhält dabei einen eigenen Knoten im Graphen. Knoten für Instanzen von Objekt-Positionen für Ressourcen-Objekte erhalten anstelle des *SemanticUnit* Knoten-Label das Knoten-Label *ResourceObject* und jene für Literal-Objekte das Knoten-Label *LiteralObject*.

Im dritten Schritt wird der Subjekt-Knoten erstellt. Dieser wird immer erstellt, auch wenn die Semantic Unit kein Subjekt-Storage-Modell besitzt. Grund dafür ist die Art und Weise, wie Relationen in der Anfrage-Sprache Cypher erstellt werden. Alle in dieser Arbeit verwendeten Semantic Units besitzt zwingend eine Relation zum Subjekt-Knoten. Wurde der Subjekt-Knoten jedoch noch nicht erstellt, kann die Relation nicht erstellt werden. Folglich muss der Subjekt-Knoten zusammen mit der Semantic Unit erstellt werden. Dies

geschieht mittels einer MERGE-Anfrage, sodass es nicht zu Doppelungen des Knotens im Wissensgraphen kommen kann. Da das Subjekt zudem keine weiteren Properties bestitzt, sondern nur Relationen zu Instanzen von Objekt-Position Klassen, führt dies zu keinen weiteren Problemen. Als Knoten-Label erhält der Subjekt-Knoten das Label *Subject*. Nach den Knoten werden die Teil-Abfragen für die Relationen ausgehend von diesen Knoten erstellt. Alle Relationen enthalten die UPRI der Semantic Unit als Property mit dem Namen *SemanticUnitURI*. Dies vereinfacht eine spätere Suche nach der Semantic Unit. Zudem wird an jeder Relation über die Property *currentVersion* vermerkt, ob diese noch die aktuelle Version der Semantic Unit wiederspiegelt. Die generische Teil-Abfrage für eine Relation ist in Listing 8 abgebildet.

```
1 MERGE (from)-[:relationName {SemanticUnitURI: $URI, currentVersion:
2         $currentVersion}]->(to)
```

Listing 8: Generische Cypher-Teil-Abfrage für eine Relation.

Der Zielknoten wird dabei über den zuvor definierten Identifier referenziert, sofern es sich um einen zuvor erstellten Knoten handelt. Falls der Knoten nicht zuvor erstellt wurde, wird eine weitere Teil-Abfrage erstellt (vgl. Listing 9). Wie beim Subjekt-Knoten muss auch hier ggf. ein neuer Knoten erstellt werden, damit die Relation erstellt werden kann. Das garantiert, dass beim späteren Finden der Semantic Unit keine Informationen verloren gehen.

```
1 MERGE (identifier:URI_Holder {URI: $URI})
```

Listing 9: Generische Cypher-Teil-Abfrage für einen Knoten einer Semantic Unit im Wissensgraphen.

Alle Teil-Abfragen zusammen ergeben die gesamte Abfrage, welche an eine Neo4j-Instanz zum Speichern übergeben wird. Somit ergibt sich nur eine einzige Abfrage für das Speichern einer Semantic Unit. Der resultierende Graph einer Material Entity Item Unit und einer Named Individual Identification Unit in Neo4j ist beispielhaft in Abbildung 5.5 dargestellt.

5.3.8.2 Finden einer Semantic Unit

Für das Finden einer Semantic Unit in Neo4j muss eine Cypher-Abfrage anhand des Storage-Modells der Semantic Unit erstellt werden und das Resultat in die In-Memory-Repräsentation der KGBB-Engine überführt werden. Die Abfrage ist dabei sehr einfach zu generieren. Beim Speichern wurde an allen Relationen der Semantic Unit die Semantic Unit UPRI hinterlegt.

Diese wird nun benutzt, um die Relationen mit den entsprechenden Knoten zu finden. Die generische Abfrage ist in Listing 10 abgebildet.

```
1  MATCH (a)-[r {SemanticUnitURI: $URI}]->(b) RETURN a, r, b
```

Listing 10: Generische Cypher-Teil-Abfrage für eine Relation zwischen zwei Knoten einer Semantic Unit im Wissensgraphen.

Eine Zurodnung der Knoten erfolgt dann anhand der festen Labels der Knoten (*SemanticUnit*, *Subject*, *ResourceObject*, *LiteralObject*). Die Properties der jeweiligen Knoten werden mittels des Storage-Modells dem jeweiligen Slot zugeordnet und anschließend in die In-Memory-Repräsentation der KGBB-Engine überführt. Für das Finden einer Semantic Unit in einer Neo4j Datenbank mit sämtlichen Objekt-Position Knoten ist somit ebenfalls nur eine Abfrage nötig.

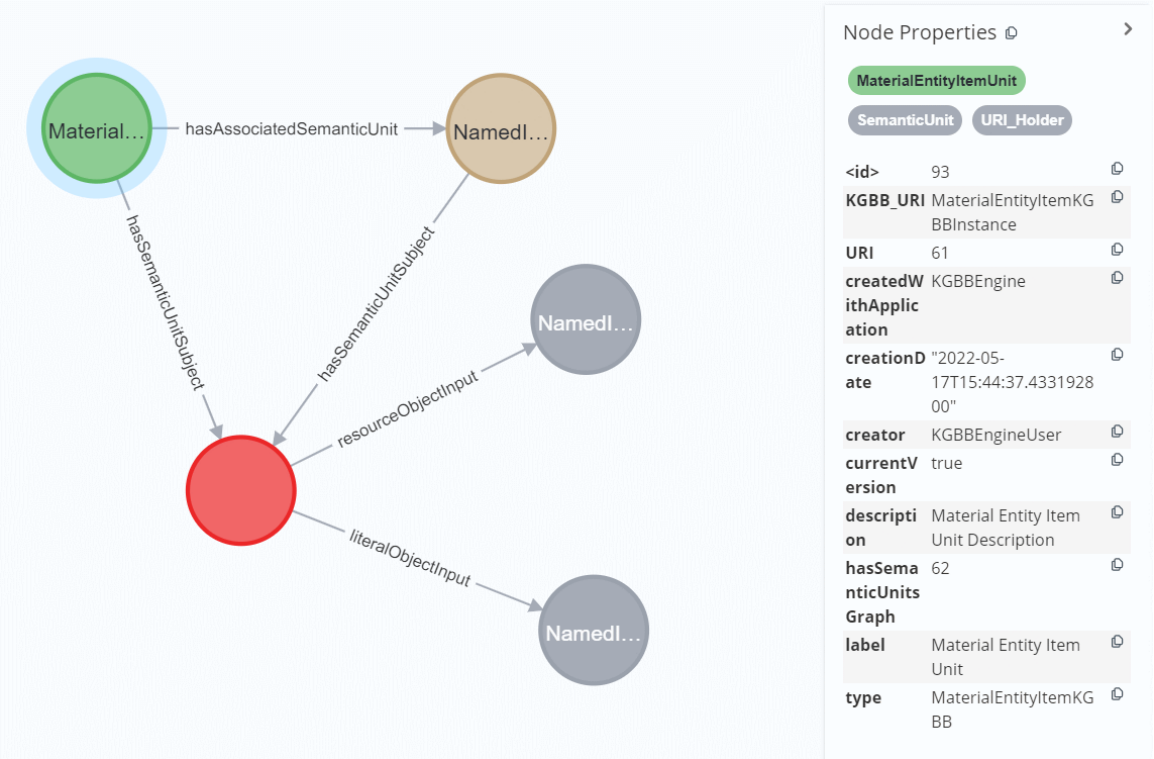


Abbildung 5.5: Labeled-Property Graph einer Material Entity Item Unit und einer Named Individual Identification Unit mit ausgewähltem Material Entity Item Unit Knoten in Neo4j

5.3.8.3 Löschen einer Semantic Unit

Das Löschen einer Semantic Unit erfolgt in zwei Abfragen. Zunächst wird der Wert von *currentVersion* für die Semantic Unit auf *false* gesetzt, um die Semantic Unit als *gelöscht* zu markieren. Der darauffolgende Speichervorgang erfolgt analog zum Anlegen einer Semantic Unit. Zudem werden alle Relationen, die auf einen Knoten der Semantic Unit zeigen auf *currentVersion false* gesetzt. Dies geschieht in einer zweiten Abfrage und ist in Listing 11 schematisch dargestellt.

```
1 OPTIONAL MATCH ()-[r]->(:URI_Holder {URI: $URI})
2 SET r.currentVersion = false
```

Listing 11: Generische Cypher-Abfrage zum Aktualisieren der Relationen.

5.4 Input-Adapter

Input-Adapter sind Adapter, die einen Input-Port der Applikation für eine spezifische Technologie implementieren. Im folgenden Kapitel wird eine Beispielimplementierung für eine Webanwendung unter der Verwendung von Thymeleaf [49] und Spring [50] vorgestellt.

5.4.1 Front-End mit Thymeleaf

Für den Eingabe-Port der KGBB-Engine (vgl. Kapitel 5.2.3) erfolgt die Implementierung eines Adapters in Form einer Webanwendung für menschliche Benutzer. Sie basiert auf der serverseitigen Template Engine Thymeleaf [49] und dem Applikations-Framework Spring [50] und stellt alle von der KGBB-Engine implementierten Use-Cases (vgl. Kapitel 4.3) bereit. Da der Schwerpunkt dieser Arbeit auf dem Speichern und Verwalten der Semantic Units liegt und nicht auf der Benutzeroberfläche, werden die Display-Templates nicht aus dem jeweiligen KGBB geladen, sondern liegen als statische HTML-Templates im Projektverzeichnis vor. Darunter befindet sich ein HTML-Template für eine Item Unit, HTML-Fragments für eine Has-Part Statement Unit und zwei verschiedene HTML-Fragments für die Weight Measurement Compound Unit. HTML-Fragments sind wiederverwendbare Abschnitte von HTML-Dokumenten, die in separaten Dateien gespeichert werden. Aufgrund der Verwendung von statischen HTML-Templates wird die von dem Adapter bereitgestellte Webseite bei jeder Anfrage neu geladen, da keine dynamische Aktualisierung möglich ist.

Die Hauptseite der Webanwendung ist unter der relativen URL „/“ oder „/index“ abrufbar und ist in Abbildung 5.6 dargestellt. Sie implementiert die Use-Cases zum Finden einer Semantic Unit und zum Anlegen einer Compound Unit. Der Use-Case zum Anlegen einer Compound Unit steht dem Benutzer über die Eingabefelder in der Mitte zu Verfügung.

Durch Angabe einer Klassen-URI und eines optionalen Labels kann eine neue Material Entity Instanz anlegt werden. Die Klassen-URI muss dabei von einer Subklasse von Material Entity (BFO:0000040) sein. Die Überprüfung der Constraints übernimmt die KGBB-Engine. Das Label wird ebenfalls von der KGBB-Engine automatisch vom Ontology Service bezogen, sofern es nicht vom Benutzer angegeben wurde. Mittels einer POST-Anfrage werden die Daten von der Benutzeroberfläche an den unterliegenden Server übermittelt. Die Anfrage erfolgt über den Endpunkt „/create“ mit den Parametern *resource* und *label*. Der Eingabe-Adapter erstellt daraufhin die Eingaben für die KGBB-Engine zum Anlegen einer Material Entity Item Unit. Folglich können mit diesem Endpunkt nur Material Entity Item Units erstellt werden.

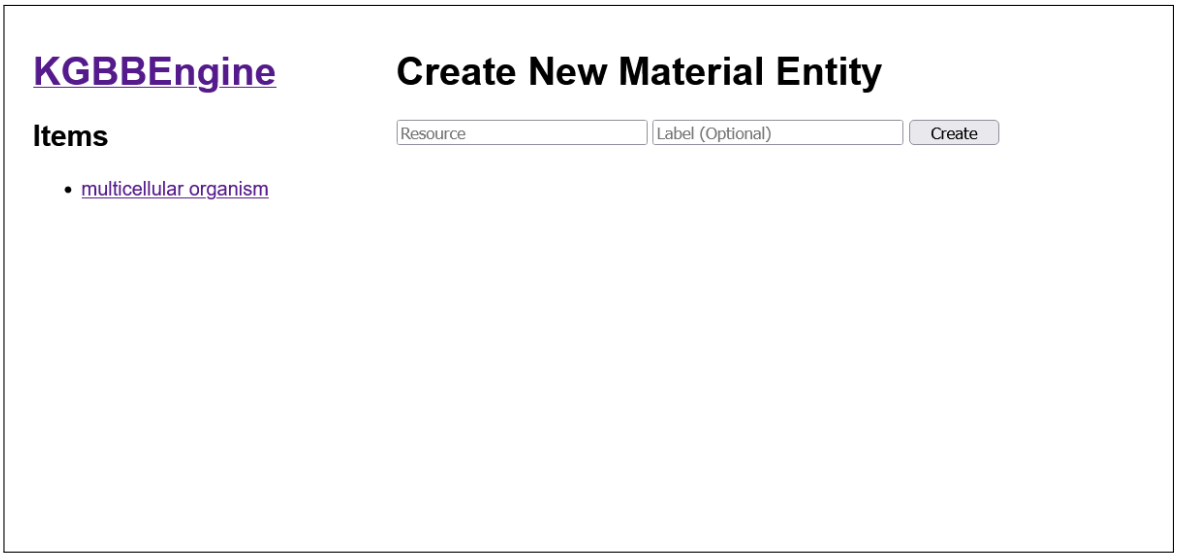


Abbildung 5.6: Hauptseite der Benutzeroberfläche

Auf der linken Seite der Benutzeroberfläche befindet sich eine Navigation für Item Units. Sie ist auf der Hauptseite auf Item Units limitiert, die den Anfang einer Parthood-Beziehungs-Kette darstellen. D.h. es werden nur Item Units angezeigt, die nicht als Objekt in einer Parthood-Beziehung von anderen Item Unit verwendet werden.

Zusätzlich zur Hauptseite können Material Entity Item Units in der Benutzeroberfläche angezeigt und verwaltet werden. Auf diese kann direkt über den Pfad „/item/{uuid}“ oder über die Hauptseite navigiert werden. Der Parameter *uuid* steht dabei für die UPRI der jeweiligen Material Entity Item Unit. Die Benutzeroberfläche einer Material Entity Item Unit ist in Abbildung 5.7 dargestellt und implementiert alle Use-Cases aus Kapitel 4.3. Analog zur Hauptseite ist sie in zwei Bereiche unterteilt. Auf der linken Seite befindet sich eine Navigation für die Material Entity Item Units, welche im Gegensatz zur Hauptseite die Hierarchie der Parthood-Beziehungen zwischen den Material Entity Item Units widerspiegelt und diese entsprechend anordnet (vgl. Box 1 in Abbildung 5.7). In der Mitte der Benutzeroberfläche befinden sich die Inhalte der gerade in der Navigation ausgewählten Material

Entity Item Unit. Dazu zählt die Klasse und das Label der entsprechenden NIIU (vgl. Box 2 in Abbildung 5.7), sowie die Parthood-Beziehungen der Has-Part Statement Units (vgl. Box 3 in Abbildung 5.7) und die Gewichtsmessungen der Weight Measurement Compound Unit (vgl. Box 4 in Abbildung 5.7), die aus einer Quality Statement Unit und dazugehörigen Weight Measurement Statement Units besteht. Das Label der NIIU wird dabei als Überschrift in der Benutzeroberfläche dargestellt und ist gleichzeitig eine Schaltfläche zum Öffnen eines Dialogfensters, in dem das Label und die Klassen-URI der in der NIIU enthaltenen Named-Individual Ressource bearbeitet werden können (vgl. Abbildung 5.8). Zudem kann in diesem Dialogfenster die Material Entity Item Unit inklusive aller zugehöriger Statement Units gelöscht werden. Die Parthood-Beziehungen werden in Form einer Tabelle dargestellt. Eine Eingabemaske ermöglicht es, weitere Parthood-Beziehungen anzulegen. Dazu ist entweder eine Klassen-URI und optional ein Label erforderlich oder eine URI für eine bereits existierende Material Entity Item Unit, dessen Subjekt-Ressource als Objekt der Parthood-Beziehung verwendet werden soll. Unter der Tabelle mit den Parthood-Beziehungen befindet sich eine Tabelle mit den Gewichtsmessungen. Für diese stehen dem Benutzer zwei Display-Templates zu Verfügung. „Template 1“ stellt die Gewichtsmessungen in einer Tabelle mit „Quality“, „Value“, „Lower Bound“, „Upper Bound“ und „Unit“ dar (vgl. Box 4 in Abbildung 5.7). In „Template 2“ werden die Werte hingegen in natürlicher Sprache angezeigt (vgl. Abbildung 5.9). Analog zu den Parthood-Beziehungen gibt es ebenfalls eine Eingabemaske zum Anlegen neuer Gewichtsmessungen. Für eine neue Gewichtsmessung muss der Benutzer einen Mittelwert angeben, eine untere Grenze und obere Grenze der Messwerte, sowie eine Ontologieklasse für die Einheit der Messwerte. Die Ontologieklasse für die Qualität der Messung ist festgelegt auf die Qualität weight (PATO:0000128). Sollte es bei der Verarbeitung der Benutzereingaben zu Fehlern kommen, werden dem Benutzer diese in Form einer Fehlermeldung angezeigt (vgl. Abbildung 5.10).

Assoziierte Material Entity Item Units werden über den Pfad „/item/{uuid}/{part}“ referenziert, wobei *uuid* unverändert bleibt und *part* die URI der assoziierten Material Entity Item Unit angibt. Dies ermöglicht es Material Entity Item Units in einem bestimmten Kontext anzuzeigen. Dem Benutzer werden die Inhalte der Material Entity Item Unit mit der URI *part* angezeigt und der Navigationsbaum wird ausgehend von der Material Entity Item Unit mit der URI *uuid* dargestellt.

5.4.1.1 Hinzufügen einer Statement Unit

Das Hinzufügen einer Statement Unit erfolgt über den Endpunkt „/item/{uuid}“ oder „/item/{uuid}/{part}“ mit *addStatement* als Parameter. Dieser gibt die URI der KGBB-Instanz der Statement Unit an, die erstellt werden soll. Der Parameter *subject* wird dazu genutzt, die Ausgangs-Semantic-Unit-URI anzugeben. Sollte er nicht gesetzt sein, wird der Wert von *part* verwendet, ansonsten *uuid*. Die Parameter für die Eingaben der Objekt-

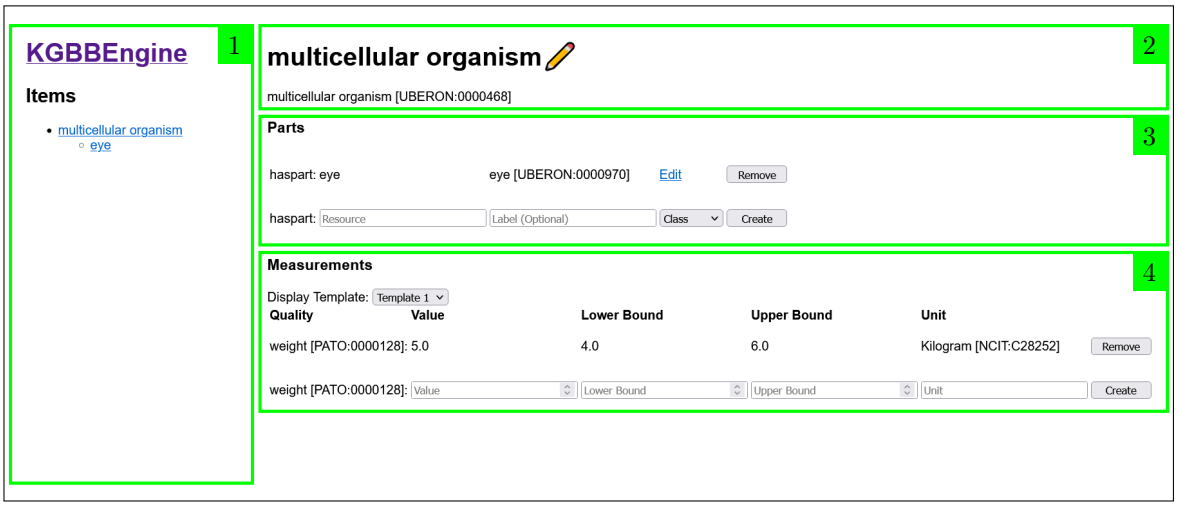


Abbildung 5.7: Benutzeroberfläche für eine Material Entity Item Unit. (1) Navigationsbereich für Material Entity Item Units, (2) Klassen-URI und Label und der assoziierten NIIU, (3) Bereich zum Verwalten von Parthood-Beziehungen, (4) Bereich zum Verwalten von Gewichtsmessungen

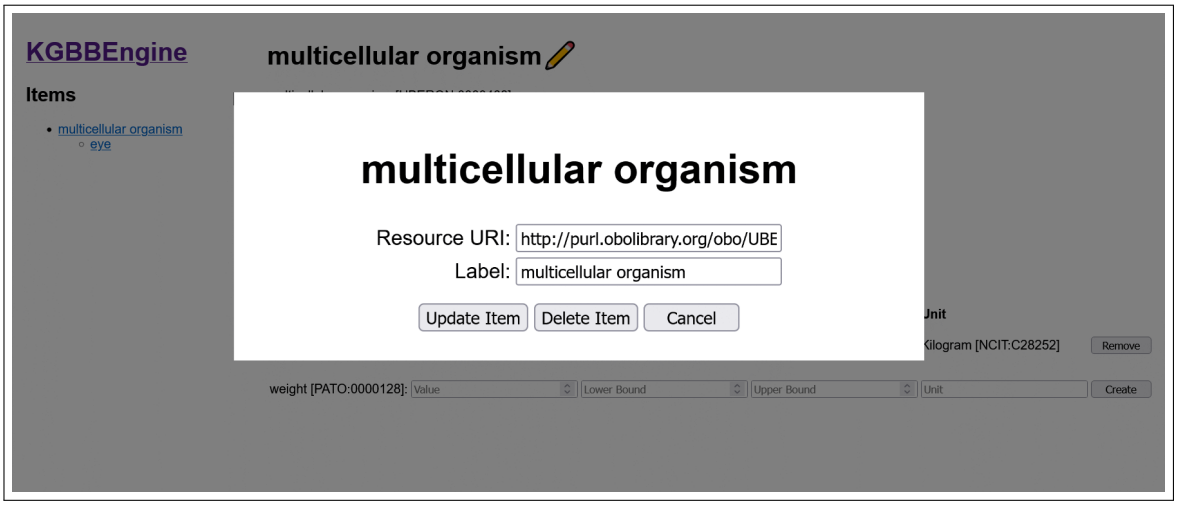


Abbildung 5.8: Dialogfenster zum Bearbeiten des Labels und der Klasse

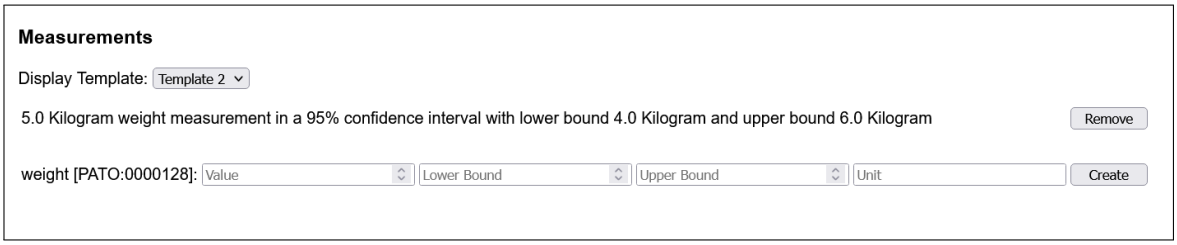


Abbildung 5.9: Alternatives Display-Template für Gewichtsmessungen in natürlicher Sprache

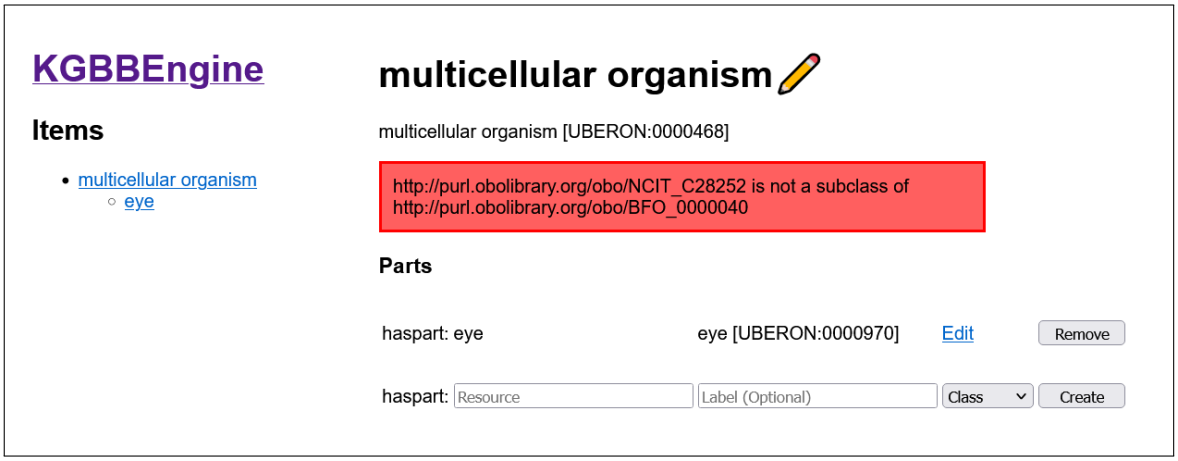


Abbildung 5.10: Fehlermeldung in der Benutzeroberfläche

Positionen der Statement Unit folgen einem bestimmten Namensschema, damit der Adapter diese in gültige Eingaben für die KGBB-Engine überführen kann. Das Schema ist folgendermaßen definiert: `@input:(label|resource|type|literal):URI`
Wobei URI die jeweilige Klassen-URI einer Objekt-Position beschreibt. Eine vollständige Eingabe für eine Ressourcen-Eingabe besteht aus den Angaben zu *resource*, *type* und optional *label*. Mit *resource* wird die entsprechende URI der Ressource angegeben. Der Parameter *type* gibt den Typen der Ressource an. Dieser kann entweder den Wert *CLASS* für eine Klasse oder *INSTANCE* für eine Instanz annehmen. Mit *label* wird das Label übermittelt. Für Eingaben zu einem Literal wird nur die Angabe zu *literal* mit dem entsprechenden Wert benötigt.

5.4.1.2 Löschen einer Statement Unit

Das Löschen einer Statement Unit erfolgt über denselben Endpunkt. Anstelle von *addStatement* wird jedoch der Parameter *removeStatement* verwendet. Er gibt die UPRI der Statement Unit an, die gelöscht werden soll. Zudem kann über den Parameter *removeSubject* die URI der Ausgangs-Semantic-Unit angegeben werden, die beim Löschvorgang aktualisiert werden soll. Ist dieser Parameter nicht vorhanden, wird der Wert von *part* verwendet, ansonsten der Wert von *uuid*.

5.4.1.3 Hinzufügen einer Compound Unit

Das Hinzufügen einer Compound Unit erfolgt ebenfalls über denselben Endpunkt. Der entsprechende Parameter dafür ist *addItem*. Er gibt die URI der KGBB-Instanz der Compound Unit an, die angelegt werden soll. Mit dem optionalen Parameter *subject* wird die UPRI der Ausgangs-Semantic-Unit angegeben. Die Eingaben für die Compound Unit werden analog zu den Eingaben zum Anlegen einer Statement Unit angegeben.

5.4.1.4 Löschen einer Compound Unit

Auch das Löschen einer Compound Unit erfolgt über denselben Endpunkt. Es wird der Parameter *removeItem* für die UPRI der zu löschenden Compound Unit verwendet.

5.4.1.5 Aktualisieren einer Statement Unit

Das Aktualisieren einer Statement Unit erfolgt ebenfalls über den Endpunkt „/item/{uuid}“ oder „/item/{uuid}/{part}“. Als Parameter wird *updateStatement* verwendet. Er gibt die UPRI der Statement Unit an, die aktualisiert werden soll. Über den Parameter *updateSubject* wird die URI der Ausgangs-Semantic-Unit angegeben, die ebenfalls aktualisiert werden soll. Falls dieser nicht definiert ist, wird der Wert von *part* verwendet, ansonsten der Wert von *uuid*. Die Eingaben werden analog zu den Eingaben zum Hinzufügen einer Statement Unit angegeben.

5.5 Testen

Das Testen einer Applikation dient zur Qualitäts- und Stabilitätsprüfung der jeweiligen Applikation und wird in der Regel von automatisierten Test-Frameworks übernommen, für die zuvor manuell oder systematisch Testfälle erstellt worden sind, und spezifische Funktionen der Applikation, insbesondere ihre Randfälle und Fehlerfälle, überprüfen sollen. Ein weiterer wichtiger Aspekt ist die Verifikation der Applikation mithilfe von Testfällen, bei der überprüft wird, ob die Applikation alle Anforderungen der Spezifikation erfüllt. Dadurch können auch spätere Änderungen an der Applikation auf Korrektheit überprüft werden. Im Folgenden wird zwischen zwei verschiedenen Arten von Testfällen unterschieden, den Unit Tests und den Integration Tests [51, 52]. Bei einem Unit Test wird die kleinste Einheit eines Programms auf Fehler überprüft, wie z.B. eine Methode oder eine Funktion. Dabei wird jede Einheit isoliert getestet. Bei einem Integration Test wird ein Verbund mehrerer Einheiten eines Programms getestet, um das Zusammenspiel zwischen den Einheiten auf Fehler zu prüfen. Die Einheiten aus einem Integration Test stammen dabei aus mindestens zwei verschiedenen Klassen des Programms.

Die kritischen Bereiche der KGBB-Engine werden mithilfe des Open-Source Test-Frameworks JUnit 5[6] getestet, damit eine ordnungsgemäße Funktionsweise der KGBB-Engine sichergestellt ist. Zu den kritischen Bereichen der KGBB-Engine zählen der Eingabe-Port (vgl. Kapitel 5.2.2) und die Adapter für die unterschiedlichen Datenbanktechnologien (vgl. Kapitel 5.3.7 und 5.3.8). Bei den Tests für den Eingabe-Port handelt es sich um einen abstrakten Integration Test für jeden Use-Case der KGBB-Engine (vgl. Kapitel 4.1). Diese werden auch „Contract Tests“ genannt und sollen überprüfen, ob sich die jeweilige Implementierung an die

[6] https://junit.org/junit5/

vorgegebene Spezifikation hält. Dazu wird jeder Integration Test nur einmalig implementiert und kann somit für alle verschiedenen Ausgabeadapter-Konfigurationen der KGBB-Engine wiederverwendet werden. So ist gleichzeitig sichergestellt, dass die verwendeten Testfälle bei jeder Konfiguration identisch sind. Für die Datenbankadapter wurden ebenfalls abstrakte Testfälle in Form von Unit Tests erstellt, um diese systematisch auf Fehler zu überprüfen und gleichzeitig die Äquivalenz der verschiedenen Implementierungen zu garantieren, obwohl sich die unterliegenden Speichertechnologien unterscheiden. Die Verwendung von abstrakten Testfällen reduziert zudem den Entwicklungsaufwand, da Testfälle nur einmalig erstellt werden müssen und anschließend wiederverwendet werden können.

Das verwendete Test-Framework JUnit 5 ist direkt im verwendeten Build-Tool Gradle integriert und wird automatisch beim Erstellen der Applikation ausgeführt.

6 Diskussion

In dieser Arbeit wurde eine funktionsfähige Applikation entwickelt, die einen Wissensgraphen unter der Verwendung von ausgewählten Semantic Units mithilfe von KGBBs verwalten kann. Im Folgenden werden die Vorteile und Limitationen der Applikation aufgelistet und ein Vergleich zu ähnlichen Applikationen gemacht. Des Weiteren wird ein Ausblick für die weitere Entwicklung gegeben.

6.1 Vorteile

Die entwickelte KGBB-Engine zeigt, dass sich die Konzepte der Semantic Units und KGBBs in einer Applikation umsetzen lassen. Gleichzeitig bringt die Verwendung von Semantic Units und KGBBs einige Vorteile gegenüber anderer Wissensgraph Management Applikationen. Einer dieser Vorteile ist die Entkopplung des Speichermodells von der Speichertechnologie, wodurch es der KGBB-Engine ermöglicht wird, mit unterschiedlichen Speichertechnologien zu kommunizieren, und diese über ein einheitliches Interface bereitzustellen, ohne dass die Verwendung von Abfragesprachen wie SPARQL oder Cypher benötigt wird. Ein weiterer Vorteil ist die Abstraktion des Datenmodells durch die Verwendung von Semantic Units. Mit ihnen können auch nicht-Semantik-Experten ihre Daten in einem Wissensgraphen modellieren, ohne dass die Interoperabilität und die Vergleichbarkeit der Daten durch ein spezifisches Datenmodell eingeschränkt wird. Zudem wird das Anlegen von Statements über Statements vereinfacht, da die Ressource einer Statement Unit als Referenz für ihr Statement verwendet werden kann. Der Einsatz von Semantic Units und KGBBs macht die KGBB-Engine zu einer FAIRen Applikation, da der entstehende Wissensgraph durch die Semantic Units in FAIR Digital Objects organisiert werden kann, und die gesamte Applikation durch einen semantischen Graphen beschrieben wird. Des Weiteren bietet die KGBB-Engine ein Framework für die Entwicklung weiterer Applikationen und Werkzeuge, mit denen Wissensgraphen besser erkundbar gemacht werden können. Weiterführende Projekte könnten die Interaktion mit dem Wissensgraphen noch weiter vereinfachen, wodurch die Eintrittsbarriere für die Verwendung von Wissensgraphen gesenkt wird und die kognitive Interoperabilität erhöht wird.

6.2 Limitationen

Die KGBB-Engine, die in dieser Arbeit entwickelt wurde, bringt noch einige Limitationen mit sich. So ist die Auswahl an den unterstützten Semantic Units auf die Namend Individual Identification Units (NIIU), Item Units, Statement Units und die Typed Statement Units beschränkt, wodurch nur eine begrenzte Datenmodellierung möglich ist. Dies spiegelt sich auch in der aktuellen Implementierung der Benutzeroberfläche wider, die nur für das in dieser Arbeit verwendete Beispiel verwendet werden kann. Zudem können keine Daten und Metadaten von bereits bestehenden Wissensgraphen verwendet werden, die nicht auf dem Konzept der Semantic Units basieren. Eine Datenintegration müsste manuell erfolgen. Des Weiteren ist der Funktionsumfang für die LinkML-Templates der KGBB-Engine auf die in Kapitel 5.2.4 beschriebenen Funktionen beschränkt, da das LinkML Ökosystem die verwendete Zielplattform zum aktuellen Zeitpunkt nicht ausreichend unterstützt. Außerdem fehlt die Implementierung eines KGBB-Editors, der Domänenexperten ohne einen Hintergrund in Semantik das Anlegen neuer KGBBs und die Spezifikation einer eigenen KGBB-getriebenen Wissensgraphapplikation ermöglicht. Die Entwicklung eines solchen Editors war aber auch nicht Gegenstand der vorliegenden Arbeit.

6.3 Vergleich

6.3.1 Vergleich mit Ontology Based Data Access Applikationen

Ähnlich zu den Ontology Based Data Access Applikationen erfolgt der Datenzugriff der KGBB-Engine über eine konzeptionelle Ebene. Im Gegensatz zu R2RML Mappings werden von der KGBB-Engine jedoch LinkML-Templates verwendet, die während der Laufzeit von einem jeweiligen Adapter in entsprechende Anfragen einer bestimmten Technologie überführt werden. Die Wissensgraphapplikationen Ontop und Stardog bieten die Möglichkeit, mehrere verschiedene Datenbanken unterschiedlicher Typen gleichzeitig zu verwenden. Dies ist mit den aktuellen Adaptern für RDF4J (Kapitel 5.3.7) und Neo4j (Kapitel 5.3.8) mit der KGBB-Engine nicht möglich, jedoch können verschiedene Datenbanktechnologien verwendet werden. Zudem bietet nur Stardog die automatische Erweiterung des Wissensgraphen mit neuen Kontextinformationen durch maschinelles Lernen und traditionellen Inferenzmethoden. Der Fokus von Ontop und Stardog liegt allerdings eher in der Kondensierung von Datenbanken und nicht in der Strukturierung und Aufbau von neuen Wissensgraphen. Beide Applikationen unterstützen das Konzept von Semantic Units und KGBBs nicht und bieten auch keine Möglichkeit für einen Menschen, den Wissensgraphen intuitiv zu erkunden.

6.3.2 Vergleich mit Metaphactory

Wohl am besten lässt sich die KGBB-Engine zur Wissensgraph Plattform Metaphactory vergleichen. Bereits die Architektur ähnelt der des *Ports und Adapter* Designpatterns, welches für die KGBB-Engine angewendet wurde, durch die Abstraktion der Benutzeroberfläche und der Datenzugriff Infrastruktur (vgl. Abbildung 2.2). Effektiv können diese als Adapter betrachtet werden, die bei Bedarf durch andere Implementierungen ausgetauscht werden könnten. Der Datenzugriff erfolgt über einen virtuellen Wissensgraphen, welcher auf ein oder mehreren Triple Stores basiert und analog zu den Ontology Based Data Access Applikationen den Zugriff auf relationale Datenbanken über R2RML Mappings ermöglicht, wohingegen die KGBB-Engine LinkML-Templates verwendet und spezifische Adapter für einen jeweiligen Datenbanktypen. Zu den Plattform Services, die von der Metaphactory Plattform angeboten werden, hat die KGBB-Engine zu diesem Zeitpunkt keine vergleichbaren Komponenten, da es lediglich möglich ist, Semantic Units im Wissensgraphen zu erstellen, zu ändern oder zu löschen. Die webbasierte Benutzeroberfläche der Metaphactory Plattform hingegen bietet eine ähnliche Funktionalität wie die KGBB-Engine. Für jeden Ressourcen-Typ oder spezifisch für eine Ressource lassen sich HTML-Templates definieren, wodurch die Daten in der Benutzeroberfläche intuitiv dargestellt werden. Dies funktioniert in der KGBB-Engine auf einer feineren Ebene, da jede Statement Unit und somit jede einzelne Aussage ein oder mehrere Display-Templates besitzen kann, die in verschiedenen Kontexten wiederverwendet werden können. Die Erkundung des Wissengraphen erfolgt für die beiden Wissensgraph Management Applikationen jedoch unterschiedlich. Metaphactory verwendet eine Kombination aus mehreren Eingabemöglichkeiten, um automatisch SPARQL Abfragen zu generieren. Dazu zählen die Eingabe über ein Textfeld zur Stichwortsuche, die Parametereingabe in vordefinierten Formularen und die iterative Auswahl relevanter Eigenschaften und Constraints. In der KGBB-Engine wird die Erkundung durch die Strukturierung des Wissengraphen durch Semantic Units für den Benutzer erleichtert, sodass Informationen immer in einem für den Benutzer relevanten Kontext angezeigt werden können. Das Hinzufügen von Daten wird auf der Metaphactory Plattform durch spezifische Templates ermöglicht, welche von Semantik-Experten erstellt werden müssen. Die Verwendung von Semantic Units und KGBBs in der KGBB-Engine ermöglichen es auch nicht-Semantik-Experten Daten zu modellieren, ohne dass sie an eine spezielle Darstellung der Daten gebunden sind. Das Konzept der Semantic Units und der KGBBs werden von der Metaphactory Plattform nicht unterstützt.

6.4 Ausblick

Als nächster Schritt für die KGBB-Engine wäre die Implementierung weiterer Semantic Unit Typen denkbar, um eine voll umfassende Datenmodellierung zu ermöglichen, sowie die Unterstützung aller LinkML Funktionen. Des Weiteren wäre die Implementierung von weite-

ren Adaptern für unterschiedliche Technologien der verschiedenen Repositorien denkbar. So könnten z.B. auch traditionelle relationale Datenbanken von der KGBB-Engine verwendet werden oder externe KGBB-Beschreibungen bezogen werden. Durch die Verwendung des *Ports und Adapter* Designpatterns ist es bereits möglich, die verwendete Speichertechnologie der KGBB-Engine zu wechseln. Über eine mögliche Import-/Export-Funktion könnten die Betreiber einer KGBB-getriebenen Wissensgraphen Applikation jederzeit die verwendete Speichertechnologie wechseln, und somit ihre Applikation unabhängig von einer bestimmten Speichertechnologie machen.

Eine weitere Möglichkeit wäre die Entwicklung eines KGBB-Editors, welcher es Domänenexperten ohne semantischen Hintergrund ermöglicht, neue KGBBs zu erstellen und zu verknüpfen. Auf diese Weise könnte das KGBB-Framework von Domänenexperten dazu genutzt werden, ihre eigenen Wissengraph-Applikationen zu erstellen, indem sie verfügbare KGBBs wiederverwenden und kombinieren und bei Bedarf neue KGBBs erstellen.

Zusätzlich könnte die Benutzeroberfläche über eine vollständige Display-Template Unterstützung erweitert werden, sowie Import- und Export-Templates, wodurch die Wiederverwendbarkeit der Semantic Units erhöht werden würde. Daten könnten mithilfe der Import-Templates in die Wissensgraphen basierend auf Semantic Units integriert werden oder mithilfe der Export-Templates in beliebige Datenmodelle überführt und von bereits existierenden Analysewerkzeuge weiterverarbeitet werden. Alternativ können auch weitere Adapter für die Benutzeroberfläche der KGBB-Engine entwickelt werden. Diese könnten z.B. auf REST- oder GraphQL-Adapter im Backend basieren und eine dynamische Benutzeroberfläche basierend auf einem JavaScript-Framework wie React, Vue oder Angular bereitstellen.

7 Fazit

Die Konzepte der Semantic Units und KGBBs bieten eine neuartige Möglichkeit, FAIRe [1] und gleichzeitig besser erkundbare Wissensgraphen (FAIREr) zu erstellen und zu verwalten.

In dieser Arbeit wurde ein Framework entwickelt, welches einen Wissensgraphen mithilfe von ausgewählten Semantic Units (vgl. Kapitel 4.1) und KGBBs systematisch aufbauen und verwalten kann, und somit eine Grundlage für die Entwicklung weiterer Werkzeuge bietet, mit denen Wissensgraphen besser erkundbar gemacht werden können. Die Applikation unterstützt dabei sechs verschiedene Arten von Semantic Units bzw. KGBBs, welche es Nutzern ermöglicht, eigene Wissensgraphapplikationen zu erstellen, ohne dass sie sich dabei Gedanken über die Datenmodellierung oder über komplexe SPARQL-Abfragen machen müssen. Aufgrund des modularen Aufbaus der Applikation, durch die Verwendung des *Ports und Adapter* Designpatterns [41, 42], ist es möglich, verschiedene Technologien für das Front- und Backend zu verwenden, wodurch die Applikation eine hohe Flexibilität, Erweiterbarkeit und Anpassbarkeit erzielt. Außerdem wurden bereits zwei verschiedene Speichertechnologieadapter für das Backend vorgestellt und implementiert. Diese ermöglichen es der Applikation, Semantic Units in RDF Triple Stores mithilfe von SPARQL-Abfragen zu speichern oder die Semantic Units in einer Neo4j Datenbank mithilfe von Cypher-Abfragen zu speichern. Für das Frontend wurde ebenfalls ein Adapter vorgestellt und implementiert, welcher auf statischen Thymeleaf Templates basiert, und Nutzern eine webbasierte Benutzeroberfläche zum Verwalten und Erkunden von Material Entity Item Units bietet.

Folglich wurde in dieser Arbeit gezeigt, dass sich die Konzepte der Semantic Units und KGBBs in einer Applikation umsetzen lassen und die entstehenden Wissensgraphen von den Vorteilen der Semantic Units und KGBBs profitieren können. Die von der KGBB-Engine erzeugten Wissensgraphen zeichnen sich durch ihre besondere Strukturierung und umfangreicher Metadaten aus, wodurch sie besser erkundbar sind und gleichzeitig die Prinzipien von FAIR einhalten. Außerdem wird das Anlegen von Statements über Statements erleichtert, da jede Aussage im Wissensgraphen durch eine eigene Semantic Unit repräsentiert wird und über eine eigene UPRI referenziert werden kann. Zugleich abstrahieren die Semantic Units das unterliegende Datenmodell und ermöglichen es auch nicht-Semantik-Experten ihre Daten im Wissensgraphen zu modellieren und gleichzeitig eine erhöhte Vergleichbarkeit der Daten zu erreichen. Die in dieser Arbeit entwickelte Applikation könnte in Zukunft weiter ausgebaut werden und in Kombination mit einem KGBB-Editor ein mächtiges Framework für FAIRe und einfach zu erkundende (FAIREr) Wissensgraphapplikationen bieten.

Literatur

[1] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne et al. „The FAIR Guiding Principles for scientific data management and stewardship“. In: *Scientific data* 3 (2016).

[2] Lars Vogt. *Toward an easy-to-use framework for developing FAIREr Knowledge Graphs with Knowledge Graph Building Blocks*. Juli 2022. DOI: 10.13140/RG.2.2.13151.12967.

[3] Johannes Frey, Kay Müller, Sebastian Hellmann, Erhard Rahm und Maria-Esther Vidal. „Evaluation of metadata representations in RDF stores“. In: *Semantic Web* 10.2 (2019), S. 205–229.

[4] Olaf Hartig. „Foundations of RDF* and SPARQL*“. In: *Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, CEUR-WS. org*. 2017.

[5] Hartig et al. *RDF-star and SPARQL-star. W3C Draft Community Group Report*. https://w3c.github.io/rdf-star/cg-spec/2021-07-01.html. 2021.

[6] Tobias Kuhn Lars Vogt. *Semantic Units: Organizing knowledge graphs into semantically meaningful units of representation*. 2022. URL: https://doi.org/10.13140/RG.2.2.13742.59203.

[7] Lars Vogt. *We need FAIREr Knowledge Graphs: Cognitive Interoperability and the Human-Explorability of Knowledge Graphs*. 2022. URL: http://dx.doi.org/10.13140/RG.2.2.22809.49769.

[8] Mohamad Yaser Jaradeh, Allard Oelen, Kheir Eddine Farfar, Manuel Prinz, Jennifer D'Souza, Gábor Kismihók, Markus Stocker und Sören Auer. „Open research knowledge graph: next generation infrastructure for semantic scholarly knowledge“. In: *Proceedings of the 10th International Conference on Knowledge Capture*. 2019, S. 243–246.

[9] Souripriya Das, Seema Sundara und Richard Cyganiak. *R2RML: RDB to RDF Mapping Language W3C Recommendation 27 September 2012*. https://www.w3.org/TR/r2rml/. Eingesehen am 20.06.2022.

[10] Richard Cyganiak, David Wood, Markus Lanthaler, Graham Klyne, Jeremy J. Carroll und Brian McBride. *RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation 25 February 2014*. https://www.w3.org/TR/rdf11-concepts/. Eingesehen am 06.06.2022.

[11] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer und Andrés Taylor. „Cypher: An evolving query language for property graphs“. In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, S. 1433–1445.

[12] *Cypher Query Language*. https://neo4j.com/developer/cypher/. Eingesehen am 06.06.2022.

[13] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martın Ugarte und Domagoj Vrgoč. „Foundations of JSON schema“. In: *Proceedings of the 25th International Conference on World Wide Web*. 2016, S. 263–273.

[14] Deborah L. McGuinness und Frank van Harmelen. *OWL Web Ontology Language Overview W3C Recommendation 10 February 2004*. https://www.w3.org/TR/owl-features/. Eingesehen am 15.06.2022.

[15] Holger Knublauch und Dimitris Kontokostas. *Shapes Constraint Language (SHACL) W3C Recommendation 20 July 2017*. https://www.w3.org/TR/shacl/. Eingesehen am 20.06.2022.

[16] Eric Prud'hommeaux, Jose Emilio Labra Gayo und Harold Solbrig. „Shape expressions: an RDF validation and transformation language“. In: *Proceedings of the 10th International Conference on Semantic Systems*. 2014, S. 32–40.

[17] *RDF 1.1 Turtle Terse RDF Triple Language W3C Recommendation 25 February 2014*. https://www.w3.org/TR/turtle/. Eingesehen am 20.06.2022.

[18] Directorate General for Research European Commission und PwC EU Services Innovation. *Cost-benefit analysis for FAIR research data: cost of not having FAIR research data*. https://data.europa.eu/doi/10.2777/02999. Eingesehen am 28.06.2022.

[19] European Commission, Directorate-General for Research und Innovation. *Realising the European open science cloud : first report and recommendations of the Commission high level expert group on the European open science cloud*. Publications Office, 2016. DOI: doi/10.2777/940154.

[20] Aidan Hogan et al. „Knowledge Graphs“. In: *ACM Comput. Surv.* 54.4 (Juli 2021). ISSN: 0360-0300. DOI: 10.1145/3447772. URL: https://doi.org/10.1145/3447772.

[21] Xiaojun Chen, Shengbin Jia und Yang Xiang. „A review: Knowledge reasoning over knowledge graph“. In: *Expert Systems with Applications* 141 (2020), S. 112948.

[22] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen und S Yu Philip. „A survey on knowledge graphs: Representation, acquisition, and applications“. In: *IEEE Transactions on Neural Networks and Learning Systems* 33.2 (2021), S. 494–514.

[23] Aidan Hogan et al. „Knowledge Graphs“. In: *ACM Comput. Surv.* 54.4 (Juli 2021). ISSN: 0360-0300. DOI: 10.1145/3447772. URL: https://doi.org/10.1145/3447772.

[24] Lars Vogt, Roman Baum, Philipp Bhatty, Christian Köhler, Sandra Meid, Björn Quast und Peter Grobe. „SOCCOMAS: a FAIR web content management system that uses knowledge graphs and that is based on semantic programming“. In: *Database* 2019 (2019).

[25] Renzo Angles. „The Property Graph Database Model.“ In: *AMW*. 2018.

[26] Thomas Frisendal. *Property Graphs: The Swiss Army Knife of Data Modeling*. https://www.dataversity.net/property-graphs-swiss-army-knife-data-modeling/. Eingesehen am 12.07.2022. 2017.

[27] European Commission, Directorate-General for Research, Innovation, O Corcho, M Eriksson, K Kurowski, M Ojsteršek, C Choirat, M Sanden und F Coppens. *EOSC interoperability framework : report from the EOSC Executive Board Working Groups FAIR and Architecture*. Publications Office, 2021. DOI: doi/10.2777/620649.

[28] Sandra Collins, Francoise Genova, Natalie Harrower, Simon Hodson, Sarah Jones, Leif Laaksonen, Daniel Mietchen, Rūta Petrauskaitė und Peter Wittenburg. „Turning FAIR into reality: Final report and action plan from the European Commission expert group on FAIR data“. In: (2018).

[29] Carlos Buil Aranda et al. *SPARQL 1.1 Overview. W3C Recommendation 21 March 2013*. https://www.w3.org/TR/sparql11-overview/. Eingesehen am 06.06.2022.

[30] Evan Wallace David Booth. *2nd U.S. Semantic Technologies Symposium 2019*. https://us2ts.org/2019/posts/program-session-x.html. Eingesehen am 23.06.2022.

[31] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro und Riccardo Rosati. „Ontologies and databases: The DL-Lite approach“. In: *Reasoning Web International Summer School*. Springer. 2009, S. 255–356.

[32] Timea Bagosi, Diego Calvanese, Josef Hardi, Sarah Komla-Ebri, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, Mindaugas Slusnys und Guohui Xiao. „The ontop framework for ontology based data access“. In: *Chinese Semantic Web and Web Science Conference*. Springer. 2014, S. 67–77.

[33] Guohui Xiao, Davide Lanti, Roman Kontchakov, Sarah Komla-Ebri, Elem Güzel-Kalaycı, Linfang Ding, Julien Corman, Benjamin Cogrel, Diego Calvanese und Elena Botoeva. „The virtual knowledge graph system ontop“. In: *International Semantic Web Conference*. Springer. 2020, S. 259–277.

[34] Diego Calvanese, Benjamin Cogrel, Elem Guzel Kalayci, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro und Guohui Xiao. „OBDA with the Ontop Framework.“ In: *SEBD*. Citeseer. 2015, S. 296–303.

[35] *The Enterprise Knowledge Graph Platform | Stardog*. https://www.stardog.com/platform/. Eingesehen am 15.06.2022.

[36] Peter Haase, Daniel M. Herzig, Artem Kozlov, Andriy Nikolov und Johannes Trame. „metaphactory: A platform for knowledge graph management“. In: *Semantic Web* 10 (2019), S. 1109–1125.

[37] *Metaphactory - Get Started*. https://metaphacts.com/get-started. Eingesehen am 22.06.2022.

[38] *LinkML - Linked data Modeling Language - LinkML - Linked data Modeling Language*. https://linkml.io/. Eingesehen am 16.05.2022.

[39] Sierra Moxon et al. „The Linked Data Modeling Language (LinkML): A General-Purpose Data Modeling Framework Grounded in Machine-Readable Semantics“. English (US). In: *CEUR Workshop Proceedings* 3073 (2021), S. 148–151. ISSN: 1613-0073.

[40] Oren Ben-Kiki, Clark Evans und Brian Ingerson. „Yaml ain't markup language (yaml™) version 1.1“. In: *Working Draft 2008-05* 11 (2009).

[41] Alistair Cockburn. *Hexagonal architecture - Alistair Cockburn*. https://alistair.cockburn.us/hexagonal-architecture/. Eingesehen am 19.05.2022. 2005.

[42] Juan Manuel Garrido de Paz. *Ports and Adapters Pattern (Hexagonal Architecture)*. https://jmgarridopaz.github.io/content/hexagonalarchitecture.html. Eingesehen am 09.06.2022. 2018.

[43] Martin Fowler. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley, 2012.

[44] Richard G Côté, Philip Jones, Rolf Apweiler und Henning Hermjakob. „The Ontology Lookup Service, a lightweight cross-platform tool for controlled vocabulary queries“. In: *BMC bioinformatics* 7.1 (2006), S. 1–7. URL: https://doi.org/10.1186/1471-2105-7-97.

[45] *The Eclipse RDF4J Framework*. https://rdf4j.org/about/. Eingesehen am 06.06. 2022.

[46] Jeremy J Carroll, Christian Bizer, Pat Hayes und Patrick Stickler. „Named graphs“. In: *Journal of Web Semantics* 3.4 (2005), S. 247–267.

[47] *Neo4j Docs - Getting Started Guide*. https://neo4j.com/docs/getting-started/current/. Eingesehen am 06.06.2022.

[48] *Indexes for search performance - Neo4j Cypher Manual*. https://neo4j.com/docs/cypher-manual/current/indexes-for-search-performance/. Eingesehen am 17.07.2022.

[49] *Thymeleaf*. https://www.thymeleaf.org/. Eingesehen am 17.07.2022.

[50] *Spring*. https://spring.io/. Eingesehen am 17.07.2022.

[51] Gerard O'Regan. *A practical approach to software quality*. Springer Science & Business Media, 2002.

[52] Nicolas Fränkel. *Integration Testing from the Trenches*. Leanpub, 2015.

A Anhang

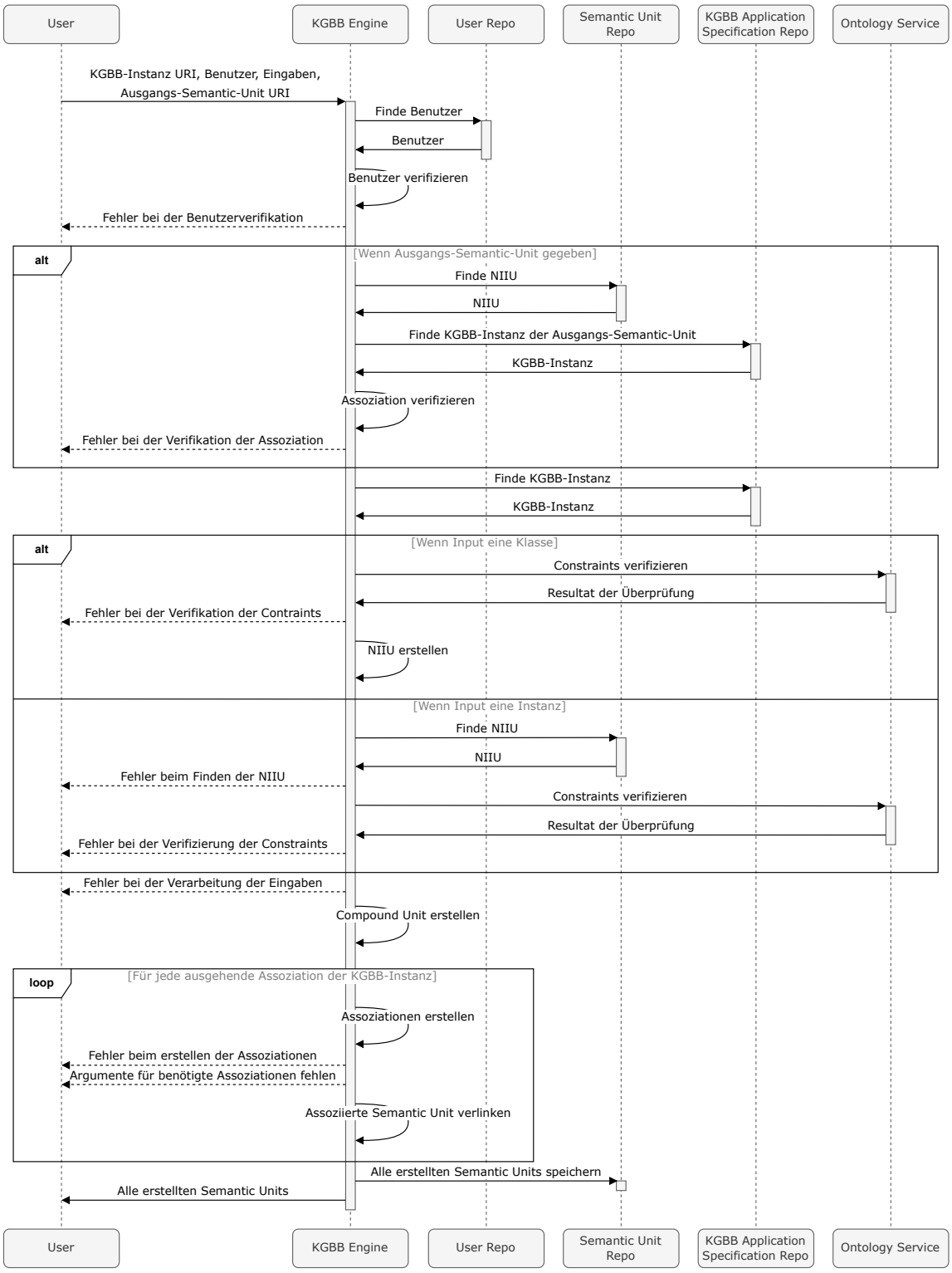


Abbildung A.1: Sequenzdiagramm des Prozesses zum Erstellen einer Compound Unit.

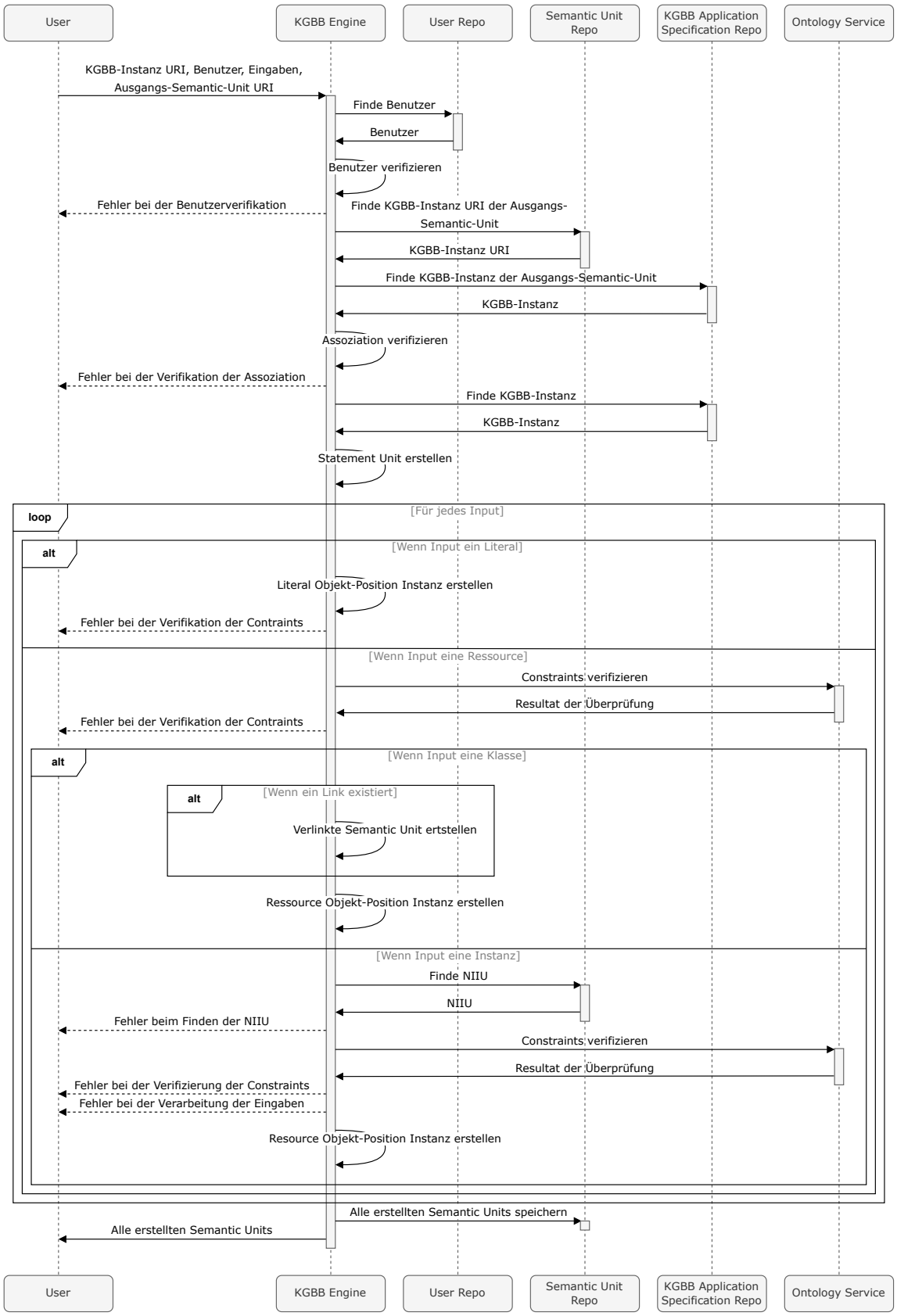


Abbildung A.2: Sequenzdiagramm des Prozesses zum Erstellen einer Statement Unit.

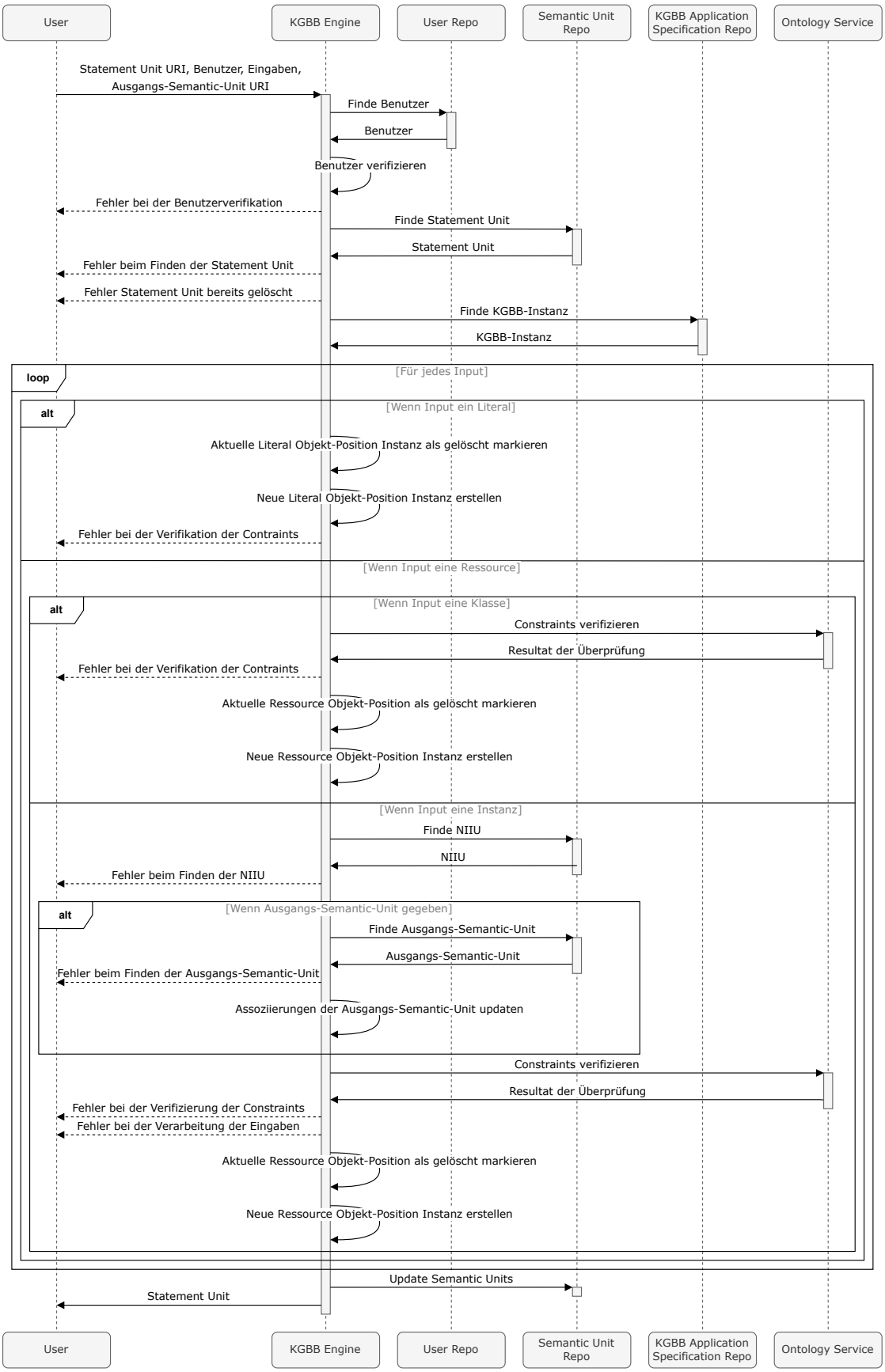


Abbildung A.3: Sequenzdiagramm des Prozesses zum Aktualisieren einer Statement Unit.

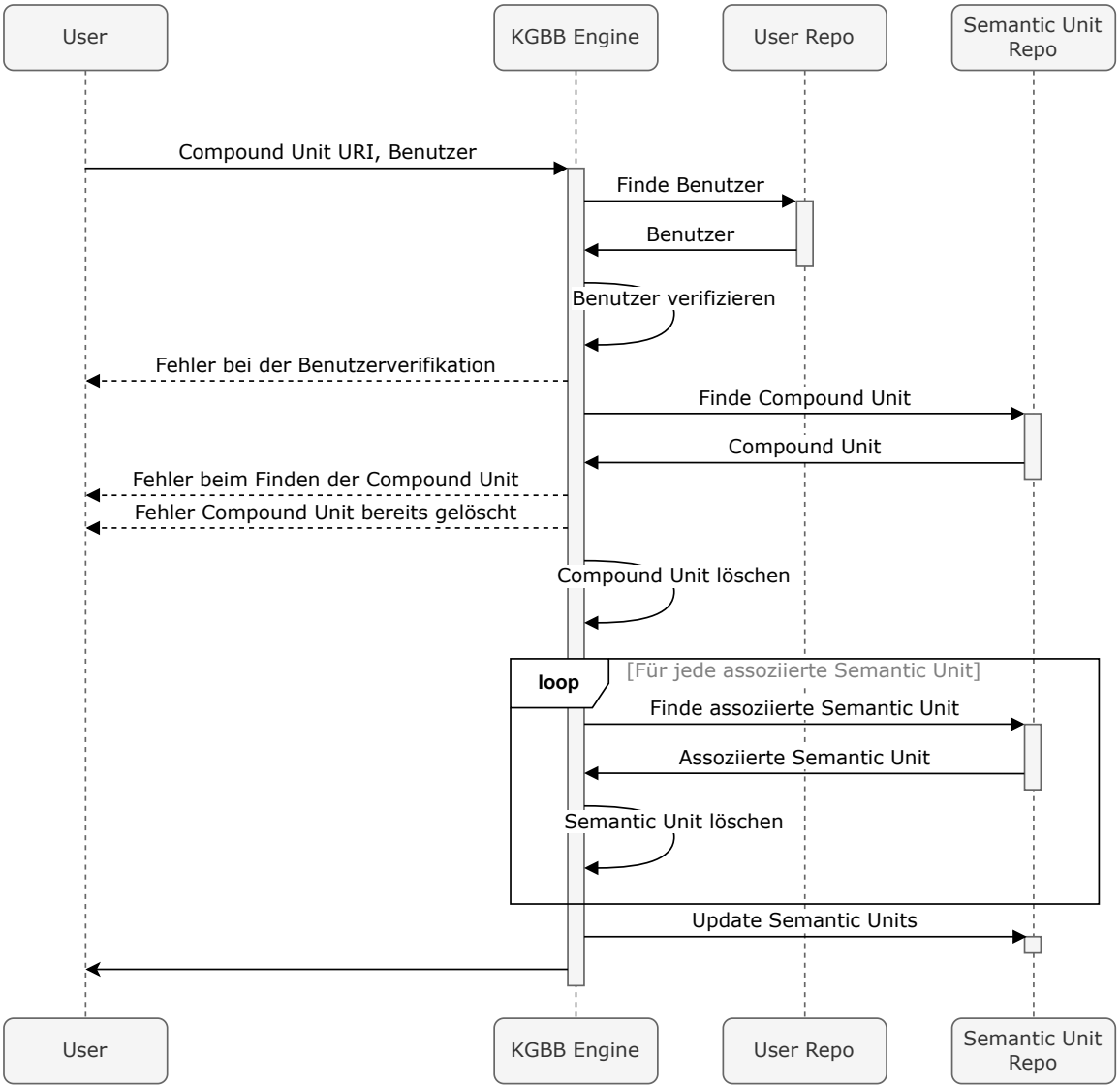


Abbildung A.4: Sequenzdiagramm des Prozesses zum Löschen einer Compound Unit.

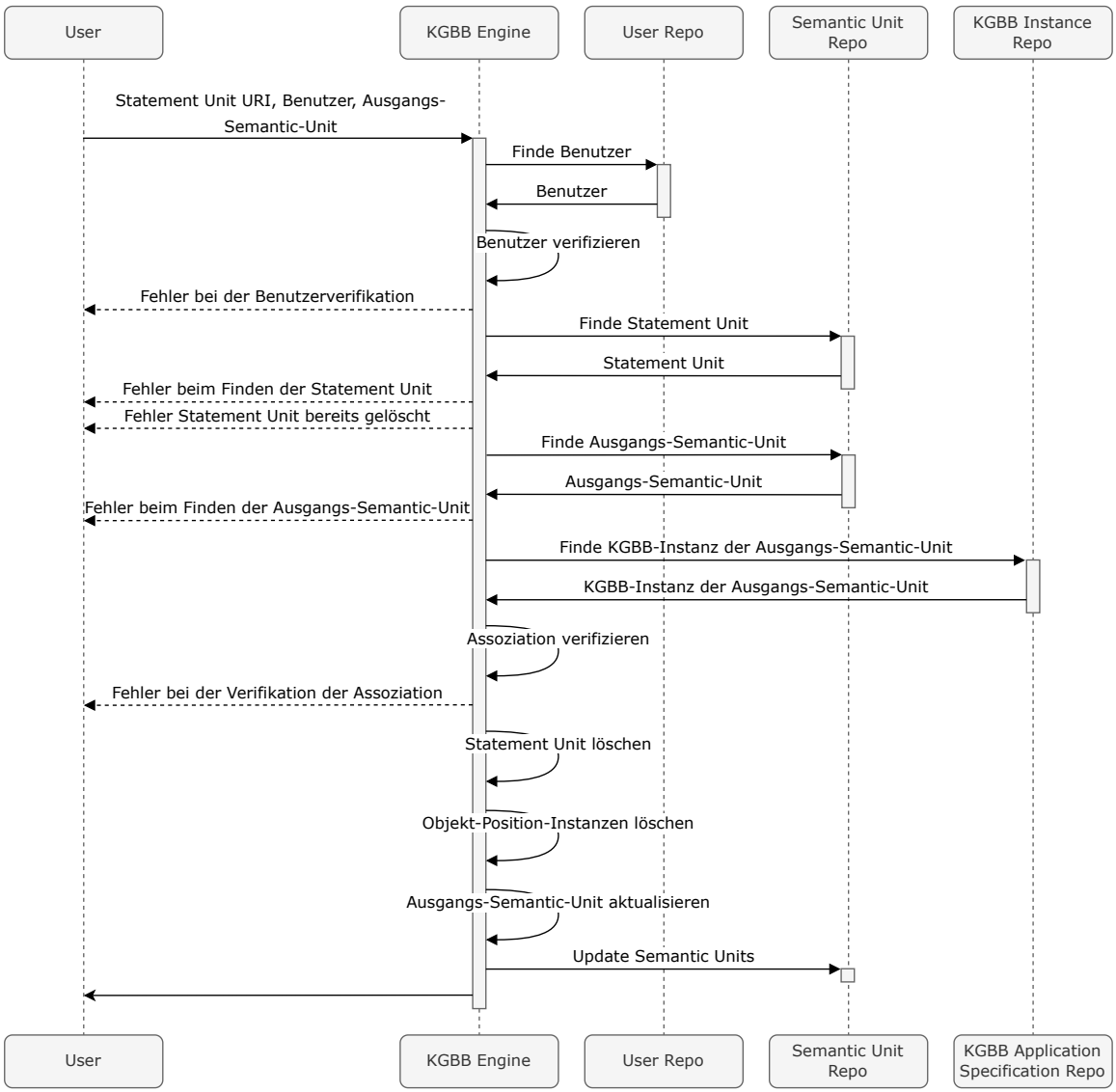


Abbildung A.5: Sequenzdiagramm des Prozesses zum Löschen einer Statement Unit.