Bachelorarbeit

Modernisierung eines Legacy-Systems zur Datenqualitätsprüfung und Entwicklung eines Testdatenmanagement-Tools im Kontext von Linked Open Data

Anna-Lena Buccoli

Abgabe am 3. August 2023
Eingereicht bei Dr. Karsten Tolle

Institut für Informatik
Goethe-Universität Frankfurt am Main

Inhaltsverzeichnis

1 Einleitung

Im Projekt „Corpus Nummorum“ werden antike Münzen in einer Forschungsdatenbank erfasst und können über ein Webportal abgerufen werden. Zum Zeitpunkt des Verfassens dieser Arbeit sind über 28.000 Münzen und über 11.000 zugehörige Typen abrufbar. Dieser Datenbestand wird fortlaufend aktualisiert und ergänzt. [1]

Bei der Erfassung solcher Datenmengen durch eine Vielzahl von Personen über einen langen Zeitraum können immer wieder Inkonsistenzen, Datenlücken und andere Fehler auftreten. Aus diesem Grund wurde bereits vor einigen Jahren das Programm „Data Quality SPARQL Rules“ [2] entwickelt, mit dem derartige Fehler ausfindig gemacht werden können. Da das Projekt „Corpus Nummorum“ Linked Open Data nutzt [1], ist es möglich, mit der Abfragesprache SPARQL in dem Programm Regeln zu definieren, um den Datenbestand systematisch nach derartigen Fehlern zu durchsuchen. Mit den Ergebnissen der ausgeführten SPARQL-Abfragen wird eine Excel-Datei erstellt, die die gefundenen Fehler dokumentiert. Dieser Fehlerbericht wird an Projektmitglieder zur Analyse und Datenkorrektur weitergeleitet. Auf diese Weise wird die Datenqualität sukzessive verbessert.

1.1 Motivation

Das Programm „Data Quality SPARQL Rules“ [2] zur Datenqualitätsprüfung wurde im Rahmen von zwei Bachelorarbeiten weiterentwickelt. Diese Weiterentwicklungen ergänzen das Programm um sinnvolle Funktionalitäten. Allerdings können sie für den praktischen Einsatz weiter verbessert werden, da manche Funktionen nicht wie ursprünglich konzipiert eingesetzt werden können. Zudem befinden sich die verwendeten Ressourcen nach wie vor auf dem Stand zum Zeitpunkt der jeweiligen Implementierung.

Für die Datenqualitätsprüfung ist die Korrektheit der SPARQL-Abfragen in den verwendeten Regeln zur Datenanalyse von zentraler Bedeutung. Auch wenn die, von einer SPARQL-Abfrage gefundenen, Ergebnisse scheinbar dem gewünschten Ergebnis entsprechen, kann die Korrektheit nicht geprüft werden. Werden beispielsweise für eine Regel keine Ergebnisse im Fehlerbericht dokumentiert, ist unklar, ob der Fehler tatsächlich nicht auftritt oder die Regel nicht korrekt formuliert ist. Bei der großen Datenmenge ist es kaum möglich, dies manuell zu verifizieren.

1.2 Aufgabenstellung

Das Ziel dieser Arbeit ist es, das bestehende Programm „Data Quality SPARQL Rules“ [2] zu überarbeiten und weiterzuentwickeln. Dabei sollen die Komponenten aktualisiert, eine weitere Verbesserung der Nutzbarkeit erreicht und, durch die Implementierung einer Testfunktionalität für SPARQL-Abfragen, die Qualität der Ergebnisse gesteigert werden.

In der ersten Phase dieser Arbeit soll das System modernisiert werden. Es geht insbesondere darum, die genutzten Ressourcen und Komponenten auf den aktuellen Stand zu bringen sowie den uneingeschränkten Einsatz aller bereits implementierten Funktionalitäten zu ermöglichen.

In der zweiten Phase soll das Programm um ein Testdatenmanagement-Tool erweitert werden. Dieses Tool soll es ermöglichen, die Korrektheit der abgefragten Regeln, auch ohne Programmierkenntnisse, validieren zu können. Dazu soll das Erstellen und Verwalten von Testfällen sowie die Testausführung ermöglicht werden.

1.3 Struktur der Arbeit

In Abschnitt 2 werden zunächst einige Grundlagen erläutert. Die Themen Java-Build Tool, Reverse Engineering, Linked Open Data und Legacy-System werden besprochen. Weiterhin wird das Thema Testdatenmanagement mit einigen zugehörigen Begriffen vorgestellt.

In Abschnitt 3 wird ein allgemeiner Überblick über die Funktionsweise des Programms „Data Quality SPARQL Rules“ [2] gegeben. Zudem wird der Stand der aktuellen Version des Programms vorgestellt.

In Abschnitt 4 wird die Modernisierung des Programms zur Datenqualitätsprüfung beschrieben. Das aktuelle System wird mit der Methode des Reverse Engineering untersucht, um die Programmstruktur zu analysieren und zu verstehen. Hierzu werden mehrere Reverse Engineering Tools genutzt und deren Ergebnisse verglichen. Basierend auf dem gewonnenen Verständnis über die Struktur des Programms erfolgt die eigentliche Aktualisierung der Anwendung sowie die Überarbeitung der bisher vorhandenen Komponenten. Um die vorgenommenen Änderungen zu testen, wird eine Funktion implementiert, die automatisiert die Ergebnisse der Datenqualitätsprüfung vor und nach der Modernisierung vergleichen kann.

Abschnitt 5 beschäftigt sich mit der Entwicklung eines Testdatenmanagement-Tools. Damit soll die Möglichkeit geschaffen werden, die in SPARQL definierten Regeln zu überprüfen und sicherzustellen, dass sie die gesuchten Daten finden. Dafür werden Anforderungen definiert und die Konzeption beschrieben. Auf dieser Basis wird die Umsetzung der verschiedenen Funktionalitäten erläutert. Schließlich wird das entwickelte Testdatenmanagement-Tool genutzt, um die vorhandenen Regeln zu testen.

Abschließend folgt in Abschnitt 6 eine Zusammenfassung der vorgenommenen Maßnahmen und erreichten Ergebnisse im Hinblick auf die gesetzten Ziele.

2 Grundlagen

In diesem Kapitel werden einige Themen und Begriffe erläutert, die in dieser Bachelorarbeit als Grundlagen dienen werden.

2.1 Java Build-Tool

Ein Build-Tool ist ein Programm, das den Erstellungsprozess von Software aus Programmcode übernimmt. Dabei ist es auch möglich, die Einbindung der benötigten Bibliotheken über ein Dependency-Management zu regeln. Die beiden bekanntesten Build-Tools für Java sind Apache Maven und Gradle. [3]

Maven ist ein Open-Source Projekt von Apache. Zentraler Bestandteil ist eine XML-Datei namens „project object model (POM)“. In dieser Datei werden alle für den Build-Prozess benötigten Informationen spezifiziert. Auch die Angabe der Bibliotheken erfolgt in der POM. [3]

Gradle ist ebenfalls ein Open-Source-Tool und basiert auf dem Konzept von Maven. Im Unterschied zu Maven nutzt Gradle keine XML-basierte Datei zur Konfiguration. Die Spezifikationen erfolgen mit Hilfe einer speziell entwickelten „domain-specific language (DSL)“. Gradle bietet ebenfalls ein Dependency-Management. [3]

2.2 Reverse Engineering

Reverse Engineering bezeichnet den Prozess, ein vorhandenes System zu analysieren. Es dient dazu, ein Verständnis für die Komponenten des Systems und ihre Zusammenhänge zu gewinnen. Vor allem bei Legacy-Systemen, wenn beispielsweise keine Dokumentation vorhanden ist, kann das Verfahren eingesetzt werden. Es hilft, den Aufbau zu verstehen, um Verbesserungen und Weiterentwicklungen vornehmen zu können. [4]

2.3 Linked Open Data

Das Konzept Linked Open Data ist charakterisiert durch die Verknüpfung von global zugänglichen Daten im Internet. Große Datenmengen sollen über ein Standardformat zugänglich gemacht werden. Dabei werden auch die Beziehungen zwischen den Daten abgebildet. Somit stellt Linked Open Data eine Sammlung miteinander verknüpfter Datensätze dar. Die Grundlage dafür bildet das „Ressource Description Framework (RDF)“ (Unterunterabschnitt 2.3.1). Um Daten im RDF-Format abfragen und bearbeiten zu können, wurde die Abfragesprache SPARQL (Unterunterabschnitt 2.3.2) entwickelt. [5, 6]

2.3.1 RDF

RDF steht für „Ressource Description Framework“ und ist ein Modell für den Austausch von Daten im Internet. RDF nutzt URIs (Uniform Resource Identifier), um Verknüpfungen zwischen Ressourcen eindeutig darzustellen. Diese Verknüpfungen werden als Triples beschrieben. Die Elemente eines Triples werden als Subjekt, Prädikat und Objekt bezeichnet. Die Struktur von Daten im RDF-Format entspricht einem gerichteten, beschrifteten Graphen. Wie in Abbildung 1 zu sehen ist, stellen Subjekt und Objekt die Knoten des Graphen dar. Das Prädikat entspricht der Kantenbeschriftung und beschreibt die Beziehung von Subjekt nach Objekt. [7, 8]

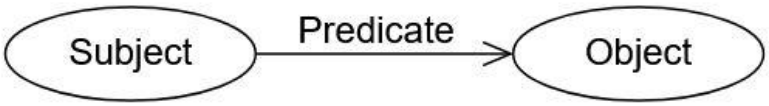

Abbildung 1: RDF-Graph mit Subjekt, Prädikat und Objekt. Quelle: [8]

Die Darstellung von RDF-Daten ist in verschiedenen Formaten möglich. Sie können beispielsweise im XML-Format oder im Turtle-Format, das der Struktur von SPARQL-Abfragen ähnelt, dargestellt werden. Zur vereinfachten Darstellung können Präfixe definiert werden, um URIs in Triples verkürzt anzugeben. [9, S. 23ff]

2.3.2 SPARQL

SPARQL steht für „SPARQL Protocol and RDF Query Language“ und ist eine Abfragesprache für Daten im RDF-Format. SPARQL ermöglicht es, die oben beschriebene Triples-Struktur des RDF nachzubilden. Bei der Abfrage wird nach einem passenden Muster in den vorhandenen Daten gesucht. Dabei ist die Verwendung von Variablen, URIs oder konkreten Werten möglich. Das Ergebnis kann zum Beispiel durch verschiedene Bedingungen oder Aggregationen eingeschränkt werden. [9]

Um Daten abzufragen oder zu bearbeiten, bietet SPARQL verschiedene Möglichkeiten. Zu den Abfrageformen gehören SELECT-, CONSTRUCT-, ASK- und DESCRIBE-Abfragen. Diese sind ähnlich aufgebaut und liefern Ergebnisse die dem, in der Abfrage definierten, Muster entsprechen. Sie unterscheiden sich in der Form, wie die Ergebnisse zurückgegeben werden. Zur Bearbeitung des RDF-Graphen gibt es beispielsweise INSERT-, DELETE-, LOAD- und CLEAR-Operationen. [9]

Die Spezifizierung der gesuchten Daten einer Abfrage erfolgt in einem WHERE-Teil. Abbildung 2 zeigt den Ausschnitt der SPARQL Syntax zur Definition des WHERE-Teils. Daraus lässt sich ableiten, dass der WHERE-Teil einer Abfrage, neben einem SubSelect, aus zwei Teilen bestehen kann. Im TriplesBlock werden Triples definiert. Diese entsprechen dem Muster des RDF-Graphen und enthalten alle gesuchten Attribute. Bei der Ausführung einer Abfrage wird ein Datensatz gesucht, der diesem Muster entspricht. Die gefundenen Werte werden in den Variablen der Triples gespeichert. Das Ergebnis kann durch Bedingungen (GraphPatternNotTriples), wie beispielsweise FILTER, eingegrenzt werden. Diese Bedingungen schränken das Ergebnis auf Grundlagen der, in den Triples definierten, Variablen ein. [10]

```
[17] WhereClause            ::= 'WHERE'? GroupGraphPattern
[53] GroupGraphPattern      ::= '{' ( SubSelect | GroupGraphPatternSub ) '}'
[54] GroupGraphPatternSub   ::= TriplesBlock? ( GraphPatternNotTriples '.'? TriplesBlock? )*
```

Abbildung 2: Ausschnitt der SPARQL Syntax. Quelle: [10]

Zudem bietet das „Graph Store HTTP Protocol“ die Möglichkeit, Operationen via HTTP auszuführen. Mit der GET-Methode können Abfragen ausgeführt werden, die Daten zurückliefern. Mit der Methode POST können Abfragen ausgeführt werden, die Daten in den RDF-Graphen einfügen. [11]

2.4 Legacy-System

Der Begriff Legacy-System bezeichnet im Allgemeinen ein veraltetes Computersystem, dass noch verwendet wird. Dabei können sowohl Software-, Hardware-Systeme,

Dateiformate oder Programmiersprachen gemeint sein. Gründe für eine Modernisierung können beispielsweise sein, dass veraltete Technologien eingesetzt werden oder das Programm zu unübersichtlich und daher schwierig weiterzuentwickeln ist. [12]

2.5 Testdatenmanagement

Testen ist ein zentraler Bestandteil bei der Entwicklung von Softwaresystemen. Die Qualität eines Tests hängt dabei maßgeblich von der Qualität der Testdaten ab. Um ein gutes Testergebnis zu erhalten ist es wichtig, eine gute Auswahl von Testdaten zu treffen. Diese Testdaten werden in Testfällen organisiert. Sind zu wenig Testfälle definiert, werden Fehler möglicherweise nicht gefunden. Aber auch wenn viele Testfälle definiert sind, führt das nicht zwangsläufig zu einem zuverlässigen Ergebnis. Entscheidend ist, dass für jedes Testobjekt möglichst viele unterschiedliche Fälle, wie Normal-, Fehler-, Grenz- und Sonderfälle, abgedeckt sind. [13, S. 10f]

Eine zentrale Frage bei der Erstellung von Testdaten ist die Entscheidung, ob Echtdaten oder synthetische Daten verwendet werden. Der Vorteil von Echtdaten ist, dass diese bereits vorliegen und die Realität abbilden. Allerdings kann nicht sichergestellt werden, dass die Echtdaten beziehungsweise ein Auszug aus den Echtdaten alle notwendigen Testfälle abdecken. Synthetische Daten zu erzeugen, die alle Testfälle abdecken, ist oftmals komplex und bedeutet einen hohen Zeit- und Arbeitsaufwand. [13, S. 88ff]

Um das Erstellen und Verwalten von Testdaten zu ermöglichen, kann ein Testdatenmanagement-Tool verwendet werden. Ein solches Tool soll einige Aufgaben des Testdatenmanagement vereinfachen und übernehmen.

Für den Begriff Testdatenmanagement existieren verschiedene Definitionen. In einem engeren Verständnis gehören zu den Aufgaben des Testdatenmanagements vor allem die Erstellung und Verwaltung von Testdaten und Testfällen, sowie die Bereitstellung der Testdaten zur Testausführung. Zudem soll mit einem solchen Tool die Archivierung der Testdaten und somit die Wiederholbarkeit von Tests ermöglicht werden. [13, S. 163ff]

Testdaten sollen mit einem möglichst geringen Aufwand erzeugt und gewartet werden können. Dabei ist eine gute Zugänglichkeit wichtig, damit sie leicht auffindbar und eindeutig identifizierbar sind. Um die Testausführung zu ermöglichen, muss die Bereitstellung der Testdaten zum Testzeitpunkt gewährleistet sein. Die Archivierung der Testdaten ermöglicht eine gute Nachvollziehbarkeit sowie die Reproduzierbarkeit der Tests. Durch das Wiederholen von Testfällen können iterativ die Ursachen für gefundene Fehler identifiziert und behoben werden. [13, S. 39ff, 130f]

Bei der Testausführung werden die Testfälle für ein Testobjekt getestet. Das Testergebnis kann durch einen Soll-Ist-Vergleich ermittelt werden. Das Soll-Ergebnis gibt das erwartete Ergebnis an, wenn der Test erfolgreich verläuft. Das Ist-Ergebnis ist das tatsächliche Ergebnis, das der Testlauf liefert, und kann vom erwarteten Ergebnis abweichen. Je nachdem, ob Soll-Ergebnis und Ist-Ergebnis übereinstimmen, kann der Test als erfolgreich beziehungsweise bestanden („passed“) oder fehlgeschlagen („failed“) gewertet werden. [13, S. 21]

Zu einer erweiterten Definition von Testdatenmanagement gehören auch Themen wie Datenschutz oder Rollen- und Zugriffsberechtigungen. Diese sind insbesondere relevant, wenn personenbezogene Daten verarbeitet werden oder ein Programm von mehreren Personen gleichzeitig getestet wird. [13, S. 163ff]

3 Programm „Data Quality SPARQL Rules“

Im Projekt „Corpus Nummorum“ liegen zahlreiche Daten über antike Münzen vor [1]. Diese Daten können verschiedene Fehler, wie beispielsweise Inkonsistenzen, enthalten. Um solche Fehler ausfindig zu machen, wurde das Programm „Data Quality SPARQL Rules“ [2] entwickelt. Ziel des Programms ist es, im Rahmen einer Datenqualitätsprüfung einen detaillierten Fehlerbericht zu erstellen, um Korrekturen in dem Datenbestand vornehmen zu können.

Um die Projektdaten für die Analyse lokal bereitzustellen, wird der SPARQL-Server „Apache Jena Fuseki“ [14] genutzt. Auf dem Server kann ein benanntes Dataset erstellt werden, das den Endpoint für die SPARQL-Abfragen bildet. Über den Endpoint können RDF-Daten, beispielsweise aus einer Datei, geladen werden.

Das Programm „Data Quality SPARQL Rules“ [2] ist in Java implementiert. Daher wird für sämtliche Weiterentwicklungen im Rahmen dieser Bachelorarbeit ebenfalls Java verwendet.

3.1 Datenqualitätsprüfung

Für die Abfrage der Projektdaten wird die Abfragesprache SPARQL genutzt. Dazu sind in dem Programm Regeln als SPARQL-Abfragen definiert, die bestimmte Fehler finden sollen. Solche Fehler sind beispielsweise Inkonsistenzen, fehlende Daten oder außergewöhnlich große Abweichungen. Eine Regel besteht dabei immer aus einer SELECT-Abfrage und kann mehrere, mit dem UNION-Operator verknüpfte, Unterregeln enthalten.

Listing 1 zeigt die Regel „Portrait Obverse“. Diese Regel soll prüfen, ob in den Daten für die Vorderseite jeder Münze und ihres zugehörigen Typs jeweils mindestens ein Porträt angegeben ist und diese Porträts übereinstimmen. Im SELECT-Teil einer Regel können Variablen und, wie in diesem Beispiel, Funktionen und Aliase angegeben werden. Der WHERE-Teil dieser Abfrage besteht aus zwei Unterregeln, die mit dem UNION-Operator verknüpft werden. Jede Unterregel besteht aus mehreren Triples und weiteren Angaben mit OPTIONAL-, FILTER-, MINUS- und VALUES-Statements. Zudem enthält die Regel eine Gruppierung der Ergebnisse nach der Variablen „?coin“.

```
1  SELECT (group_concat(distinct ?reason) as ?reasons) ?coin (
       group_concat(distinct ?c_portrait) as ?c_portraits) (
       group_concat(distinct ?c_description) as ?c_descriptions) (
       group_concat(distinct ?type) as ?types) (group_concat(distinct ?
       t_portrait) as ?t_portraits)  (group_concat(distinct ?
       t_description) as ?t_descriptions) (group_concat(distinct ?cmint
       ) as ?mints)
2  WHERE {
3    {?type rdf:type nmo:TypeSeriesItem.
4      ?type nmo:hasObverse ?t_obv.
5      ?t_obv nmo:hasPortrait ?t_portrait.
6      OPTIONAL {?t_obv dcterms:description ?t_description.
7          FILTER(langMatches(lang(?t_description), "EN"))}
8      ?coin nmo:hasTypeSeriesItem ?type.
9      OPTIONAL {?coin nmo:hasObverse ?c_obv.}
10     OPTIONAL {?c_obv dcterms:description ?c_description.
11         FILTER(langMatches(lang(?c_description), "EN"))}
12     OPTIONAL {?coin nmo:hasMint ?cmint.}
13     MINUS {?coin nmo:hasObverse ?c_obv.
14         ?c_obv nmo:hasPortrait ?t_portrait.}.
```

```
    VALUES ?reason {"Typ hat Porträt auf Obverse, Münze nicht."}.
  } UNION {
    ?coin nmo:hasTypeSeriesItem ?type.
    ?coin nmo:hasObverse ?c_obv.
    ?c_obv nmo:hasPortrait ?c_portrait.
    OPTIONAL {?coin nmo:hasMint ?cmint.}
    OPTIONAL {?c_obv dcterms:description ?c_description.
        FILTER(langMatches(lang(?c_description), "EN"))}
    ?type nmo:hasObverse ?t_obv.
    OPTIONAL {?t_obv nmo:hasPortrait ?t_portrait.}
    OPTIONAL {?t_obv dcterms:description ?t_description.
        FILTER(langMatches(lang(?t_description), "EN"))}
    MINUS {?type nmo:hasObverse ?t_obv.
        ?t_obv nmo:hasPortrait ?c_portrait.}.
    VALUES ?reason {"Münze hat Porträt auf Obverse, Typ nicht oder
   anderes."}.
  }
}
GROUP BY ?coin
```

Listing 1: Im Programm [2] definierte Regel „Portrait Obverse“ mit zwei Unterregeln

Bei der Ausführung der Datenqualitätsprüfung werden alle Regeln als SPARQL-Abfragen an den Server gestellt. Das Ergebnis wird als Fehlerbericht in eine Excel-Datei geschrieben. Dieser Fehlerbericht wird an Projektmitglieder weitergeleitet, die nun an der Korrektur der Fehler arbeiten und Bemerkungen in der Excel-Datei hinzufügen können.

Ein Fehlerbericht besteht aus einer Übersichtsseite (Abbildung 3) und mehreren Regelseiten (Abbildung 4). In Abbildung 3 ist die Übersichtsseite eines Fehlerberichts zu sehen. Dort werden alle ausgeführten Regeln mit weiteren Informationen, wie beispielsweise einer Beschreibung und der Anzahl gefundener Fehler, aufgelistet. Zu jeder Regel wird ein separates Excel-Blatt mit einer detaillierten Auflistung aller aufgetretenen Fehler erstellt. Abbildung 4 zeigt einen Ausschnitt einer Regelseite. Die gefundenen Fehler sind zeilenweise aufgelistet und neben den, in der SPARQL-Abfrage definierten, Variablen sind Spalten für Kommentare vorgesehen.

Regel	Name	Beschreibung	Anzahl Fälle	Reference Query	Reference Size	Fallquotient	Query Type
1	Portrait Obverse	mmigkeiten von Portraitmerkmalen zwischen Münzen und ihren Typ	155	coins_type	26435	0,005863439	Inconsistent
2	Portrait Reverse	immigkeiten von Portraitmerkmalen zwischen Münzen und ihren Ty	39	coins_type	26435	0,001475317	Inconsistent
3	Start Date and End Date fitting	Prüft, ob die Münz-Datierung außerhalb der Type-Datierung liegt.	1065	coins_type	26435	0,040287498	Inconsistent
4	Start Date after End Date	Prüft, ob das Start Date logisch vor dem End Date liegt	0	coins AND types	39272	0	Inconsistent
5	Denomination	ıal haben wie der Typ (bei Zuordung zu einem Subtype wird der Wer	531	coins_type	26435	0,020087006	Inconsistent
6	Mint	ıt hat wie der Typ (bei Zuordung zu einem Subtype wird der Wert vc	28	coins_type	26435	0,001059202	Inconsistent
7	Material	erial hat wie der Typ (bei Zuordung zu einem Subtype wird der Wert	120	coins_type	26435	0,004539436	Inconsistent
8	Diameter	Prüft MinDiameter und MaxDiameter Angaben.	0	coins	28013	0	Inconsistent
9	Diameter Weight existing	uf Vollständigkeit der Daten bezogen auf Diameter (Min, Max) und V	16868	coins	28013	0,602149002	Missing
10	Start - End Date - missing	Prüft welche Start und End-Dates bei Münzen und Typen fehlen.	91	coins_type	26435	0,003442406	Missing
11	Tests Diameter Weight	oder negative Werte bezogen auf Diameter (Min, Max) und Weight	0	coins	28013	0	Missing
12	Diameter Weight	Prüft auf Extremfälle bei Diameter und Weight relationen.	86	coins	28013	0,003070003	Outlier

Abbildung 3: Übersichtsseite des Fehlerberichts [15]

NR	?coin	?weight	?dia_max	?dia_min	?reason	?cmint	Datum	erledigt	in Bearbeitung	unauffindbar	kein Fehler
1	https://www.corpus-nummorum.eu/CN_27053		19.8	19.3	Gewicht nicht eingetragen	nm:cypsela	2021/03/16 12:24:34				
2	https://www.corpus-nummorum.eu/CN_27050		13.8	12.8	Gewicht nicht eingetragen	nm:cypsela	2021/03/16 12:24:34				
3	https://www.corpus-nummorum.eu/CN_27045		18.8	18.2	Gewicht nicht eingetragen	nm:cypsela	2021/03/16 12:24:34				
4	https://www.corpus-nummorum.eu/CN_27297		12.1	11.5	Gewicht nicht eingetragen	nm:sigeum	2019/05/20 11:53:58				
5	https://www.corpus-nummorum.eu/CN_27293		12.8	12.2	Gewicht nicht eingetragen	nm:sigeum	2019/05/20 11:53:58				
6	https://www.corpus-nummorum.eu/CN_27252		20.4	18.5	Gewicht nicht eingetragen	nm:sigeum	2019/05/20 11:53:58				
7	https://www.corpus-nummorum.eu/CN_27251		21.7	20	Gewicht nicht eingetragen	nm:sigeum	2019/05/20 11:53:58				

Abbildung 4: Regelseite des Fehlerberichts mit Kommentarspalten [15]

3.2 Weiterentwicklungen

Die ursprüngliche Programmversion wurde im Rahmen von zwei Bachelorarbeiten weiterentwickelt.

Die erste Bachelorarbeit „Information Propagation across Versions in Context of a SPARQL-Rules System“ von Kateryna Kvasnytsia [16] ermöglicht es, alte Fehlerberichte mit Kommentaren als Eingabe für die nächste Datenqualitätsprüfung zu nutzen. Dadurch werden die Kommentare aus den alten Versionen in den neuen Fehlerbericht übernommen. Zu diesem Zweck wurden explizite Kommentarspalten eingeführt, wie in Abbildung 4 zu sehen ist.

Im Rahmen der zweiten Bachelorarbeit „Benutzerschnittstelle für die Erstellung von Datenqualitätsabfragen in SPARQL“ von Elif Tugba Dichter und Vladyslav Matsuyev [17] wurde die Bearbeitung der Regeln vereinfacht. Ursprünglich mussten die abgefragten SPARQL-Regeln im Programmcode definiert werden. Um eine Bearbeitung ohne Programmierkenntnisse zu ermöglichen, wurde eine Benutzungsoberfläche entwickelt, über die Regeln und Unterregeln erstellt und bearbeitet werden können. Zudem werden die Regeln in einer Datei gespeichert und bei Programmstart eingelesen. Die im Code definierten Regeln dienen als Ersatzregeln, falls keine Regeldatei vorliegt.

3.3 Benutzungsoberfläche

Die Benutzungsoberfläche des Programms wurde mit Java Swing [18] implementiert. Abbildung 5 zeigt die Benutzungsoberfläche des Programms. In dem Tab „File Chooser“ erfolgt die Dateiauswahl für die Eingabedatei und Speicherung des Fehlerberichts sowie der Start der Datenqualitätsprüfung. Der Tab „Rule Creator“ enthält das Regelmanagementsystem mit allen zugehörigen Funktionen.

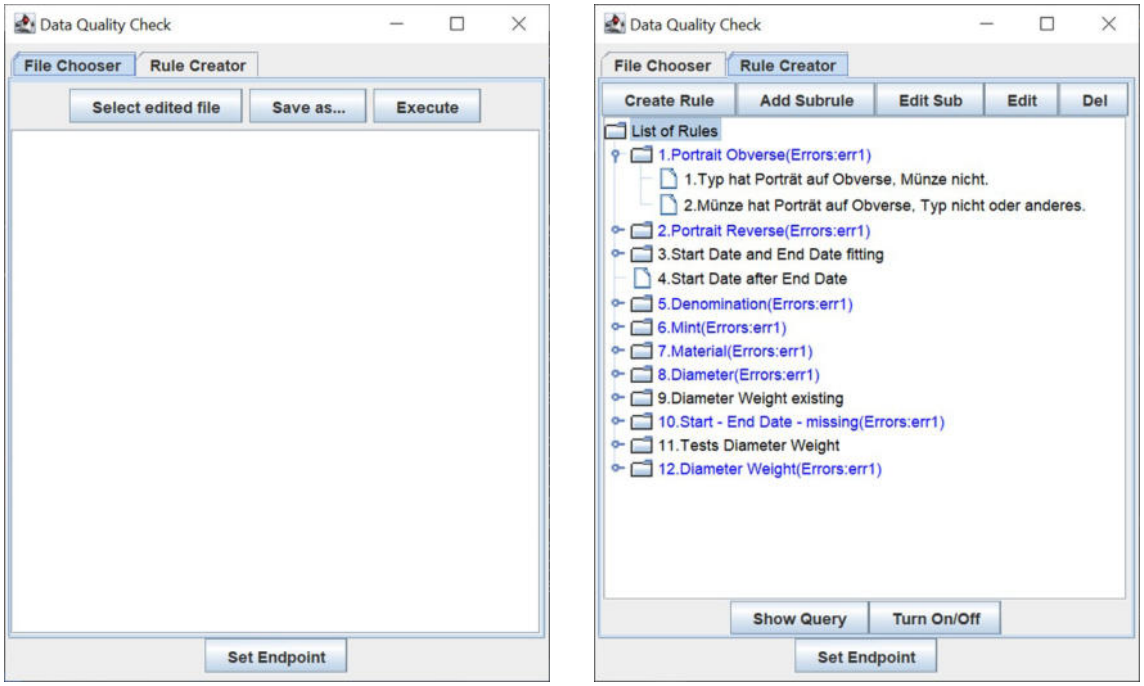

Abbildung 5: Benutzungsoberfläche der Ausgansgversion des Programms [2]

4 Modernisierung des Legacy-Systems

Bei dem Programm „Data Quality SPARQL Rules“ [2] handelt es sich um ein historisch gewachsenes System. Es wurde vor einigen Jahren entwickelt und im Rahmen von zwei Bachelorarbeiten um weitere Funktionen ergänzt. Allerdings sind einige verwendeten Ressourcen mittlerweile veraltet. Zudem treten beim praktischen Einsatz Fehler auf, die die Nutzung einiger Funktionen des Programms einschränken.

In diesem Abschnitt geht es um die Modernisierung des vorhandenen Systems zur Datenqualitätsprüfung. Dazu wird in Unterabschnitt 4.1 der Aufbau des Programms „Data Quality SPARQL Rules“ [2] mit Hilfe verschiedener Tools analysiert und untersucht. In Unterabschnitt 4.2 wird die Aktualisierung der Komponenten des Systems geschildert. Dabei wird die Verwendung eines Build-Tools eingeführt, wobei die Tools Apache Maven und Gradle infrage kommen. Weiterhin werden in Unterabschnitt 4.3 einige Verbesserungen beschrieben, die vorgenommen werden, um alle implementierten Funktionalitäten nutzen zu können. Zur Überprüfung der vorgenommenen Änderungen wird zum Abschluss der Modernisierung eine Funktion zum automatisierten Vergleich der vom Programm erstellten Fehlerberichte entwickelt. Damit soll ein Vergleich der Fehlerberichte der ursprünglichen Programmversion und der modernisierten Version ermöglicht werden. In Unterabschnitt 4.4 werden die Anforderungen, Konzeption und Implementierung dieser Funktion erläutert.

4.1 Untersuchung des aktuellen Systems

Bevor Aktualisierungen oder Verbesserungen durchgeführt werden, ist es notwendig, die Implementierung und den genauen Aufbau des Systems zu verstehen. Dazu soll der Prozess des Reverse Engineering genutzt werden. Zu diesem Zweck soll ein Tool eingesetzt werden, dass automatisch ein UML-Diagramm aus Java-Code erstellt. Auf dem Markt gibt es eine Reihe von Tools, die diese Funktion anbieten. Daher wird im Folgenden eine Auswahl von vier Tools (Visual Paradigm, Enterprise Architect, UML Lab und ESS-Model) untersucht und verglichen. Beispielhafte Übersichten solcher Tools sind auf verschiedenen Websites zu finden [19].

Name	Anbieter	Erstveröffentlichung	Verwendete Version	Proprietäre Lizenz
Visual Paradigm [20]	Visual Paradigm	2002	17.1 (Mai 2023)	Ja
Enterprise Architect [21, 22]	SparxSystems Software GmbH	2000	16.1 (März 2023)	Ja
UML Lab [23, 24]	Yatta Solutions GmbH	2010	1.32.0 (März 2023)	Ja
ESS-Model [25]	Eldean AB (über SourceForge)	2001	2.2 (Januar 2003)	Nein (Open-Source)

Tabelle 1: Übersicht der ausgewählten Reverse Engineering Tools

Tabelle 1 zeigt eine Übersicht der ausgewählten Tools. Auf den Websites von Visual Paradigm und Enterprise Architect sind zahlreiche Unternehmen als Referenzen angegeben. Daher werden diese beiden bekannten Tools in den Vergleich miteinbezogen. Des Weiteren wird UML Lab für den Vergleich ausgewählt, da auf der Website dieses Tools eine besondere Funktion zur gleichzeitigen Bearbeitung von UML-Diagramm und Code hervorgehoben wird. ESS-Model wird ausgewählt,

da es sich um ein Open-Source Tool handelt, obwohl es bereits älter ist. Da Visual Paradigm, Enterprise Architect und UML Lab lizenzpflichtig sind, werden im Rahmen dieser Arbeit kostenlose Testversionen verwendet. Zum Vergleich sind in den folgenden Abbildungen von Ausschnitten der UML-Diagramme jeweils die Klassen Rule, Subrule und Query zu sehen.

4.1.1 Visual Paradigm

Visual Paradigm bietet die Erstellung eines Klassendiagramms an. Dabei ist es möglich, die Klassen nach Packages zu gruppieren. Weiterhin kann die Erstellung eines Packagediagramms ausgewählt werden. Neben Vererbung werden verschiedene Assoziationstypen verwendet, um die Beziehungen zwischen den Klassen zu visualisieren. Die Assoziationen werden beschriftet und mit Kardinalitäten versehen. Des Weiteren bietet Visual Paradigm die Möglichkeit Änderungen im UML-Diagramm, beispielsweise das Hinzufügen einer neuen Klasse, in Code umzuwandeln. Abbildung 6 zeigt einen Ausschnitt des UML-Klassendiagramms, erstellt mit Visual Paradigm.

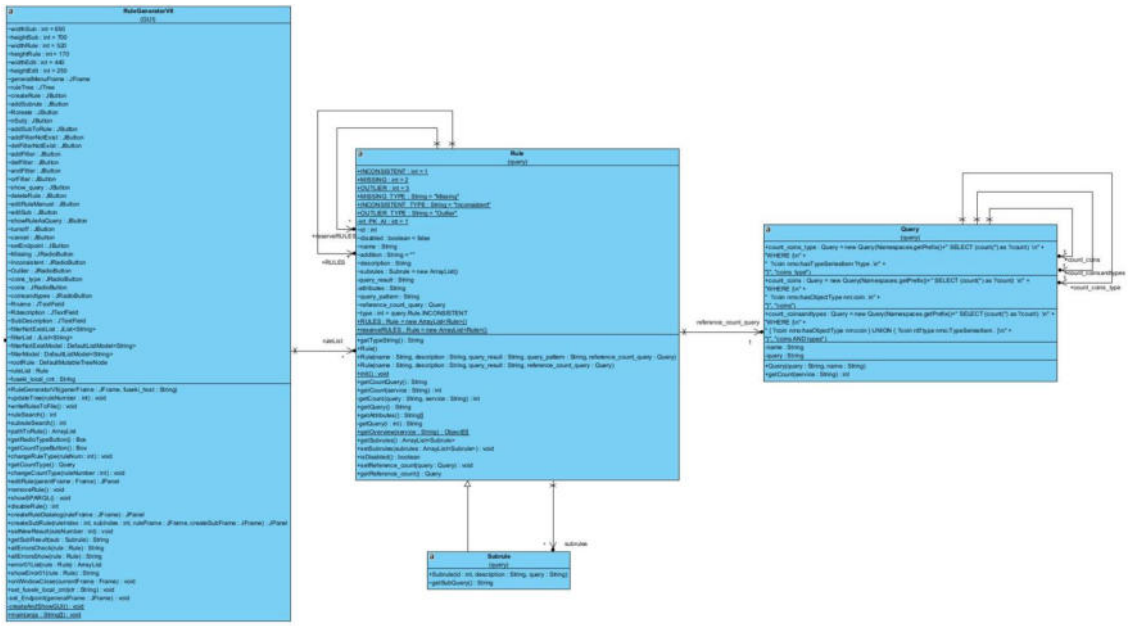

Abbildung 6: Ausschnitt des UML-Diagramms, erstellt mit Visual Paradigm [20]

4.1.2 Enterprise Architect

Mit Enterprise Architect ist die Erstellung eines Klassendiagramms möglich. Es wird für jedes Package ein separates UML-Diagramm erstellt. Dabei werden Vererbung und normale Assoziationen genutzt. Die Assoziationen werden beschriftet, aber nur wenige Assoziationen werden mit Kardinalitäten angezeigt. Auch Enterprise Architect bietet die Umwandlung von Änderungen im UML-Diagramm zu Code an. Abbildung 7 zeigt einen Ausschnitt des mit Enterprise Architect erstellten UML-Diagramms des Package „query“.

Abbildung 7: Ausschnitt des UML-Diagramms, erstellt mit Enterprise Architect [21]

4.1.3 UML Lab

Mit UML Lab kann ein Klassendiagramm erstellt werden. Es ist möglich, alle Klassen in einem UML-Diagramm darzustellen oder einzelne Packages auszuwählen, für die ein Diagramm erstellt werden soll. Im Diagramm werden Vererbungen und normale Assoziationen angezeigt. Die Assoziationen werden beschriftet und mit Kardinalitäten versehen. Zudem gibt es die Möglichkeit, Änderungen im UML-Diagramm in Code umzuwandeln. Eine Besonderheit dabei ist, dass Änderungen im Code direkt innerhalb des Tools vorgenommen werden können. In einem Split-Screen-Modus können UML-Diagramm und Code gleichzeitig bearbeitet und synchronisiert werden. Abbildung 8 zeigt einen Ausschnitt des UML-Diagramms des Package „query“, das mit UML Lab erstellt wurde.

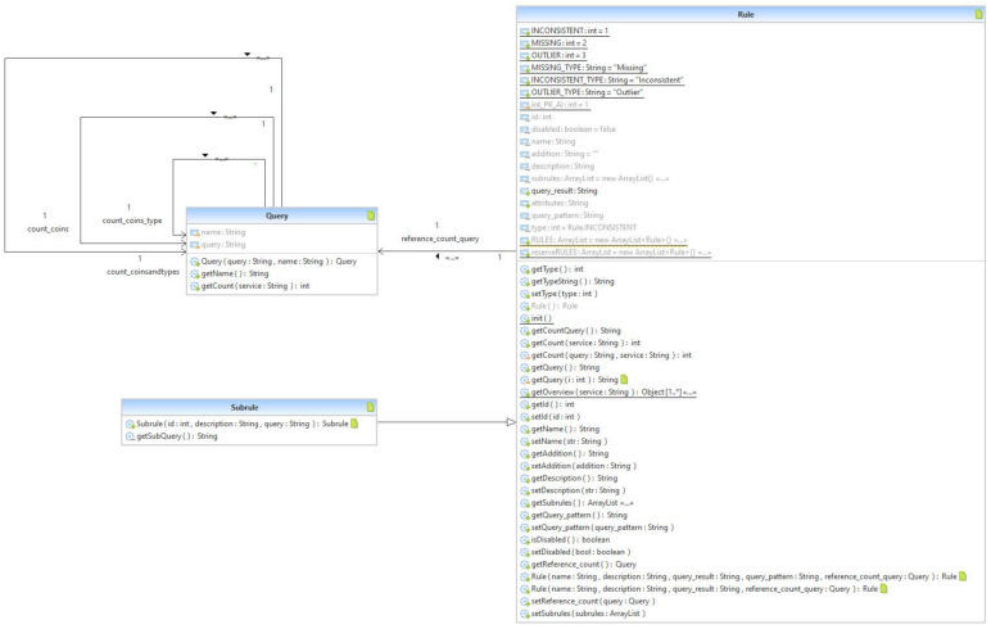

Abbildung 8: Ausschnitt des UML-Diagramms, erstellt mit UML Lab [23]

4.1.4 ESS-Model

Mit ESS-Model kann ein Klassendiagramm erstellt werden. Dabei ist es möglich zu wählen, ob alle Klassen in einem Diagramm dargestellt werden oder für jedes Package ein separates UML-Diagramm erstellt wird. Zudem wird eine Auswahl zwischen „nur Vererbung“ und „Vererbung und Assoziationen“ geboten. Es werden nur normale Assoziationen ohne Beschriftungen und ohne Kardinalitäten angezeigt. Im Vergleich zu den Diagrammen, die mit den anderen Tools erstellt wurden, werden weniger Assoziationen angezeigt. Daher ist anzunehmen, dass ESS-Model nicht alle Zusammenhänge zwischen den Klassen erkennt. Außerdem ist es nicht möglich, Änderungen im UML-Diagramm automatisch in Code zu überführen. Eine Besonderheit des Tools ist, dass bei den Diagrammen für einzelne Packages neben den Klassen im Code auch Interfaces und Vererbung von externen Klassen angezeigt werden. Abbildung 9 zeigt einen Ausschnitt des Package „query“ des mit ESS-Model erstellten UML-Diagramms. Dort sind auch ein Interface und die Vererbung von einer externen Klasse zu sehen.

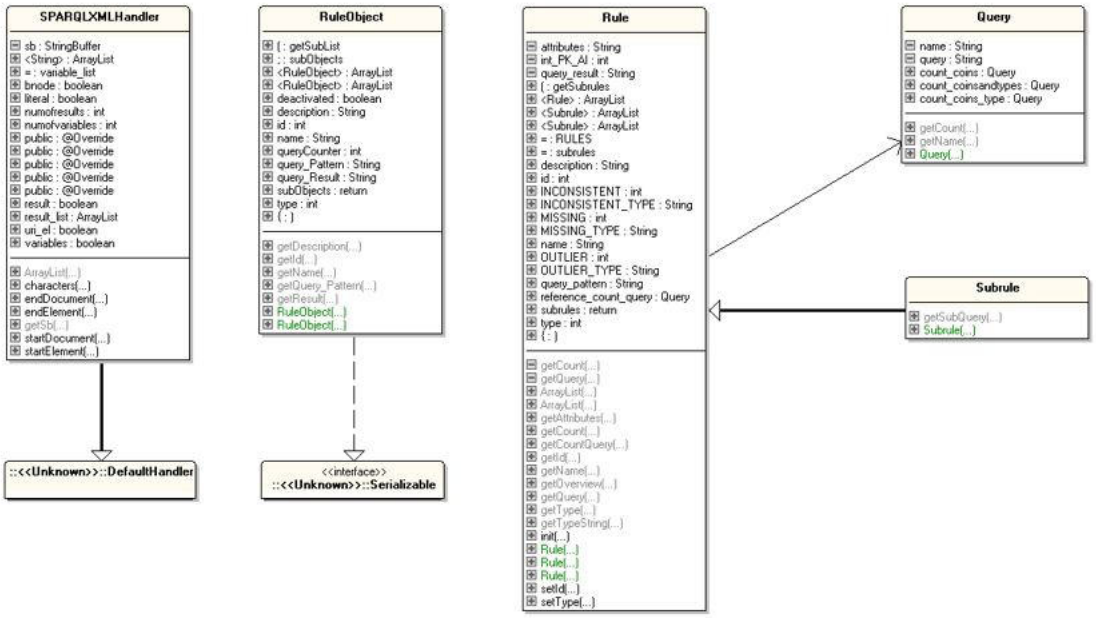

Abbildung 9: Ausschnitt des UML-Diagramms, erstellt mit ESS-Model [25]

4.1.5 Ergebnisse

Tabelle 2 fasst die Unterschiede und Gemeinsamkeiten der vier Reverse Engineering Tools zusammen.

Alle vier Programme führen den Prozess des Reverse Engineering automatisiert aus. Als besonders hilfreich zum Verständnis des Programms erwies sich das UML-Diagramm von Visual Paradigm. Die Abbildung aller Klassen in einem Diagramm und die gleichzeitige Gruppierung nach Packages ermöglicht es, einen umfassenden Überblick über die Zusammenhänge innerhalb und zwischen den Packages zu gewinnen. Zudem sind die dargestellten Assoziationen sehr detailliert. Beispielsweise erkennen alle Tools, dass die Klasse Subrule von der Klasse Rule erbt. Jedoch nur das Diagramm von Visual Paradigm zeigt, dass ein Attribut der Klasse Rule (Attribut „subrules“) Objekte der Klasse Subrule enthält. Ein weiteres Beispiel ist das

	Diagramm	Assoziationstypen	UML zu Code	Anmerkungen
Visual Paradigm	Klassen- & Package-diagramm	verschiedene Assoziationstypen, Beschriftung, Kardinalitäten	Ja	alle Klassen in einem Diagramm; Gruppierung nach Packages möglich; erkennt vergleichsweise die meisten Assoziationen
Enterprise Architect	Klassen-diagramm	normale Assoziationen, Beschriftung, Kardinalitäten	Ja	Diagramme für alle Packages einzeln
UML Lab	Klassen-diagramm	normale Assoziationen, Beschriftung, Kardinalitäten	Ja (Split-Screen)	alle Klassen in einem Diagramm oder alle Packages einzeln; synchrone Bearbeitung von Code und UML-Diagramm
ESS-Model	Klassen-diagramm	normale Assoziationen, keine Beschriftung, keine Kardinalitäten	Nein	alle Klassen in einem Diagramm oder alle Packages einzeln (mit Interfaces und Vererbung von externen Klassen); Assoziationen unvollständig

Tabelle 2: Vergleich der Reverse Engineering Tools

Attribut „ruleList“ der Klasse RuleGeneratorV8. Nur das Diagramm von Visual Paradigm zeigt, dass dieses Attribut Objekte der Klasse Rule enthält. Da dies eine Package-übergreifende Assoziation ist, kann sie mit separaten Diagrammen für jedes Package nicht dargestellt werden. Beides ist in dem in Abbildung 6 (Seite 10) dargestellten Ausschnitt des UML-Diagramms zu sehen.

Insgesamt können die UML-Diagramme dazu beitragen, die Struktur des Programms zu verstehen. Insbesondere während der Überarbeitung bestehender Klassen ist es hilfreich die Zusammenhänge zu betrachten. So beispielsweise für das Einlesen der Regeleigenschaften aus einer Regeldatei, wofür Änderungen in mehreren Klassen durchgeführt werden müssen (Unterunterabschnitt 4.3.2). Aber auch beim Erstellen einer neuen Klasse für die Konfiguration von Einstellungen, die in verschiedenen Klassen benötigt werden, ist es hilfreich einen Überblick über die Zusammenhänge zu gewinnen (Unterunterabschnitt 5.3.5).

4.2 Aktualisierung verwendeter Komponenten

Im ersten Schritt der Modernisierung wurden die verwendeten Komponenten aktualisiert. Die Umstellung des Programms auf die aktuelle Java-Version 20 unter Verwendung von OpenJDK [26] verursachte keine Probleme.

Die Aktualisierung der verwendeten Bibliotheken führte jedoch zu einem größeren Aufwand. Da bislang kein Build-Tool verwendet wurde, mussten die Bibliotheken einzeln heruntergeladen und dem Programm manuell hinzugefügt werden. Bei dem Versuch, aktuelle Versionen der notwendigen Bibliotheken zu verwenden, zeigte sich, dass die Zahl der Abhängigkeit von weiteren Bibliotheken in den neuen Versionen erheblich größer war. Daher würde das manuelle Aktualisieren der Bibliotheken einen

erheblichen Zeit- und Arbeitsaufwand bedeuten. Zudem ist zu erwarten, dass dies auch zukünftig immer aufwändiger wird. Aus diesem Grund soll für den Erstellungsprozess des Programms ein Build-Tool ergänzt werden. Dies vereinfacht aktuelle und zukünftige Aktualisierungen der Java-Version und Bibliotheken.

Wie in Unterabschnitt 2.1 erläutert, bieten Maven und Gradle ähnliche Funktionen und ermöglichen beide ein Dependency-Management. Somit eignen sich beide Tools als Build-Tool für das vorliegende Programm „Data Quality SPARQL Rules“ [2]. Da Maven aufgrund der XML-basierten Konfiguration einen einfachen Einstieg ermöglicht, fiel die Wahl auf die Verwendung von Maven [27].

Zunächst wurde der gesamte Programmcode in ein neu angelegtes Maven-Projekt kopiert. Anschließend erfolgte die Konfiguration in der Datei pom.xml (POM).

Über das Element „properties“ können in der POM die zu verwendende Version des Maven-Compilers sowie die gewünschte Java-Version spezifiziert werden [28]. Listing 2 zeigt die Spezifikationen der Java-Version und der Version des Maven-Compilers in der Datei pom.xml.

```
<properties>
    <java.version>20</java.version>
    <maven.compiler.version>3.11.0</maven.compiler.version>
    <maven.compiler.source>20</maven.compiler.source>
    <maven.compiler.target>20</maven.compiler.target>
</properties>
```

Listing 2: In der Datei pom.xml definierte Properties

Über das Element „dependencies“ werden die Bibliotheken eingebunden. Dabei wird neben dem Namen auch die Versionsnummer angegeben. Wenn eine neue Version einer Bibliothek verfügbar ist, kann diese durch das Ändern der Versionsnummer aktualisiert werden. Durch die explizite Angabe der Versionsnummer kann sichergestellt werden, dass für das Programm, insbesondere während der Entwicklung, eine stabile, definierte Umgebung besteht. Maven lädt automatisch alle angegebenen Bibliotheken sowie deren Abhängigkeiten [28].

Unter den bisher manuell eingebundenen Bibliotheken wurden einige Bibliotheken nur aufgrund von Abhängigkeiten anderer Bibliotheken des Programms hinzugefügt. Diese mussten somit nicht explizit in der POM spezifiziert werden, da sie von Maven automatisch geladen werden.

Listing 3 zeigt beispielhaft die Einbindung der Bibliothek Apache Jena ARQ [29] unter Angabe der aktuellen Versionsnummer.

```
<dependency>
    <groupId>org.apache.jena</groupId>
    <artifactId>jena-arq</artifactId>
    <version>4.9.0</version>
</dependency>
```

Listing 3: Beispiel zur Einbindung einer Bibliothek in der Datei pom.xml

Nachdem die neuesten Versionen aller Bibliotheken eingebunden waren, mussten noch ein paar Anpassungen im Code vorgenommen werden. Einige Funktionen und import-Statements waren in den neuen Versionen veraltet und mussten ersetzt werden. Beispielsweise musste die in Listing 4 auskommentierte Funktion zur Ausführung einer SPARQL-Abfrage durch die folgende Funktion ersetzt werden.

```
//QueryExecutionFactory.sparqlService(service, query).execSelect();
QueryExecution.service(service).query(query).build().execSelect();
```

Listing 4: Beispiel einer veralteten Funktion, die ersetzt wurde

Des Weiteren kann mit dem „Maven Assembly Plugin“ eine ausführbare JAR-Datei erstellt werden, die alle benötigten Bibliotheken enthält. Dazu muss das Plugin in der POM spezifiziert werden. Die Ausführung des Befehls „mvn package“ erstellt eine JAR-Datei, mit der das Programm über den Kommandozeilenbefehl in Listing 5 unabhängig von einer Entwicklungsumgebung gestartet werden kann. [30]

Um die Programmausführung zu vereinfachen, wurde eine Batch-Datei „DataQualitySPARQLRules.bat“ erstellt, die den Befehl aus Listing 5 enthält.

```
java -jar DataQualitySPARQLRules-1.0-SNAPSHOT-jar-with-dependencies
    .jar
```

Listing 5: Kommandozeilenbefehl zum Ausführen der JAR-Datei

Beim Erstellen einer JAR-Datei ist zu beachten, dass die passende Java-Version spezifiziert wird. Ist die in der POM angegebene Java-Version, aktuell Java 20 [26] (Listing 2), nicht installiert, kann die JAR-Datei nicht ausgeführt werden.

Abschließend wurde auch das Design modernisiert, um eine optische Abgrenzung zur alten Version zu erreichen. Zum Vergleich ist die alte Benutzungsoberfläche in Abbildung 5 (Seite 8) zu sehen. Abbildung 10 zeigt die Benutzungsoberfläche des Programms mit dem Tab „File Chooser“ und dem Regelmanagementsystem mit dem neuen Design unter Verwendung der Bibliothek FlatLaf [31].

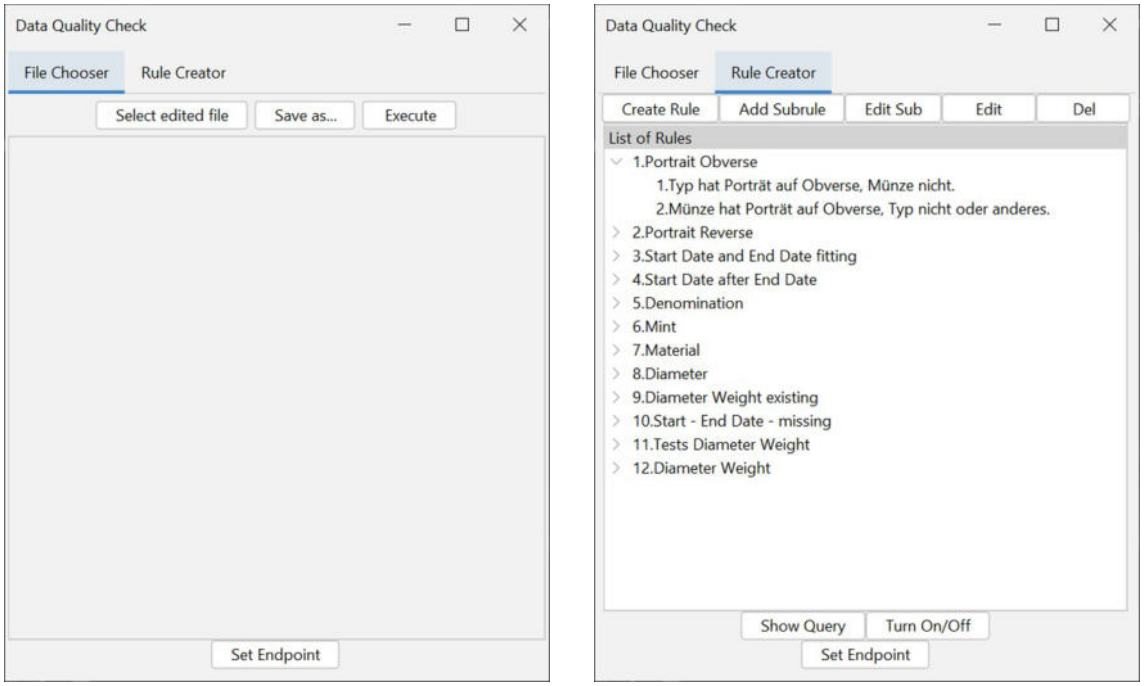

Abbildung 10: Benutzungsoberfläche des Programms [2] mit Bibliothek FlatLaf [31]

4.3 Fehlerbehebung und Verbesserungen

Nach der Aktualisierung aller verwendeten Komponenten, mussten einige Änderungen im Programm vorgenommen werden. Insbesondere die Funktionalitäten, die in der zweiten Bachelorarbeit [17] ergänzt wurden, konnten in der Praxis nicht vollständig genutzt werden. Darin wurde ein Regelmanagementsystem entwickelt, das es ermöglichen soll, Regeln zu erstellen, zu bearbeiten und in einer Datei zu speichern. Es waren im Wesentlichen drei Verbesserungen notwendig, die im Folgenden beschrieben werden.

4.3.1 Regel ohne Unterregel

Bei der Ausführung der Datenqualitätsprüfung mit der Programmversion, die zu Beginn dieser Arbeit vorlag, kam es zu einem Fehler. Aus diesem Grund wurde in der Praxis eine ältere Version ohne das Regelmanagementsystem genutzt. Dadurch mussten die SPARQL-Regeln wieder im Code bearbeitet werden. Das Regelmanagementsystem sollte die Bearbeitung der Regeln vereinfachen, damit Regeln auch ohne Programmierkenntnisse definiert werden können.

Das Konzept des Regelmanagementsystems sieht vor, dass eine Regel aus verschiedenen Eigenschaften wie zum Beispiel Name und Beschreibung sowie mindestens einer Unterregel besteht. Jede Unterregel enthält neben einer Beschreibung die eigentliche SPARQL-Abfrage. Mehrere Unterregeln werden mit dem UNION-Operator zu einer Regel verknüpft. Die im Programmcode definierten Regeln nutzt das Regelmanagementsystem als Ersatzregeln, wenn keine Regeldatei vorliegt.

Der Fehler bei der Ausführung der Datenqualitätsprüfung wurde durch eine Regel in den Ersatzregeln verursacht. Die Regel „Start Date after End Date“ war als Regel ohne Unterregel definiert und enthielt die SPARQL-Abfrage direkt. Aber gemäß dem oben beschriebenen Konzept des Regelmanagementsystems muss eine Regel mindestens eine Unterregel enthalten. Somit muss eine Regel ohne Unterregel als Regel mit einer Unterregel dargestellt werden, um sie mit dem Regelmanagementsystem nutzen zu können. Diese Änderung kann vorgenommen werden, ohne dass die Ausdrucksfähigkeit der Regel verändert wird.

Aus diesem Grund musste die Regel „Start Date after End Date“ in eine Regel mit Unterregel umgeschrieben werden. Die Darstellung der Abfrage als Unterregel verändert das interne Zusammensetzen der Regel bei der Ausführung. Dadurch ändert sich die Formatierung der Regel, aber die Bedeutung bleibt bestehen. Diese Änderung wurde direkt in den Ersatzregeln vorgenommen, damit der Fehler nicht erneut auftritt und die Regel ausgeführt werden kann. Listing 6 zeigt die Regel im Code nach der beschriebenen Änderung. Die SPARQL-Abfrage wurde als Unterregel hinzugefügt.

```
1  Rule start_end_wrong = new Rule("Start Date after End Date",
2      "Prüft, ob das Start Date logisch vor dem End Date liegt",
3      "?reason ?coin ?c_startdate ?c_enddate ?cmint",
4      Query.count_coinsandtypes);
5  start_end_wrong.subrules.add(new Subrule(1,
6      "Startdate liegt nach Enddate!",
7      "    ?coin nmo:hasStartDate ?c_startdate.\n" +
8      "    ?coin nmo:hasEndDate ?c_enddate.\n" +
9      "    ?coin nmo:hasMint ?cmint.\n" +
10     "    FILTER (?c_startdate > ?c_enddate )"));
11 start_end_wrong.setType(Rule.INCONSISTENT);
```

Listing 6: Ersatzregel als Regel mit einer Unterregel

Zusätzlich wurde im Programm eine Prüfung ergänzt, die bei der Datenqualitätsprüfung sicherstellt, dass nur Regeln ausgeführt werden, die mindestens eine Unterregel enthalten. Sind alle Unterregeln einer Regel gelöscht oder deaktiviert, wird diese Regel übersprungen, um den zuvor beschriebenen Fehler zu vermeiden.

4.3.2 Bearbeitung der Regeleigenschaften

Ein weiteres Problem in der Ausgangsversion des Programms gab es bei der Bearbeitung der Regeleigenschaften. Eine Regel besteht, neben den Unterregeln mit

den eigentlichen SPARQL-Abfragen, aus weiteren Eigenschaften. Für drei dieser Regeleigenschaften gab es beim Erstellen oder Bearbeiten keine Möglichkeit diese anzugeben. Das betraf die Eigenschaften „Query Result“, „Attributes“ und „Additions“.

Bei „Query Result“ handelt es sich um den SELECT-Teil der SPARQL-Abfrage. Im SELECT-Teil können Variablen verwendet werden, die in der SPARQL-Abfrage in den Triples definiert werden. Es ist auch möglich, beispielsweise Funktionen oder Aliase zu verwenden. Mit dem Keyword DISTINCT können Duplikate im Ergebnis eliminiert werden. Mit der Eigenschaft „Attributes“ können die Variablen für den Fehlerbericht definiert werden. Dies ist notwendig, falls der SELECT-Teil der Abfrage nicht nur Variablen enthält, sondern beispielsweise Aliase nutzt. In „Additions“ können Modifikationen für SPARQL-Abfragen, wie beispielsweise GROUP BY oder LIMIT, definiert werden. Die Eigenschaften „Attributes“ und „Additions“ sind bei der Definition einer Regel optional.

Über die Benutzungsoberfläche des Regelmanagementsystems konnten bei der Erstellung und Bearbeitung einer Regel lediglich die Eigenschaften „Name“, „Beschreibung“, „Rule Type“ und „Count Type“ angegeben werden. Abbildung 11 zeigt das Fenster zum Erstellen einer neuen Regel mit den genannten Regeleigenschaften. Das „Query Result“ wurde automatisch aus den Variablen in den Unterregeln erstellt und das Keyword DISTINCT immer eingefügt. Es war möglich, das „Query Result“ manuell über das Bearbeiten einer Regel zu ändern und beispielsweise Funktionen hinzuzufügen. Diese wurden allerdings beim Erstellen oder Bearbeiten einer Unterregel mit dem automatisch erzeugten „Query Result“ überschrieben. Die optionalen Eigenschaften „Attributes“ und „Additions“ konnten nicht angegeben werden.

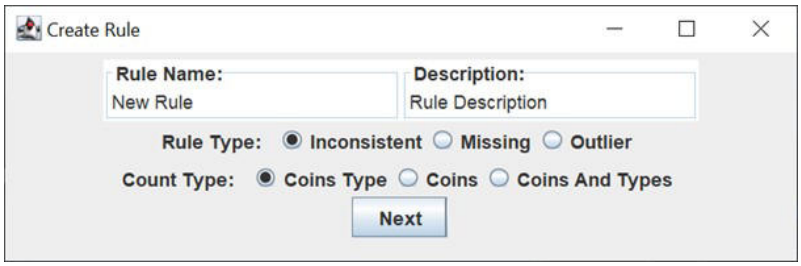

Abbildung 11: Erstellen einer neuen Regel in der alten Programmversion

Aus diesem Grund konnte das Regelmanagementsystem nicht mit komplexeren Regeln, die zum Beispiel Funktionen im SELECT-Teil nutzen, verwendet werden. Ein Beispiel für eine solche Regel, ist die in Listing 1 (Seite 6) gezeigte Regel. Diese enthält sowohl Funktionen und Aliase im SELECT-Teil als auch den Zusatz GROUP BY.

Um dieses Problem zu beheben, musste zunächst die automatische Generierung des „Query Result“ deaktiviert werden, damit das manuell definierte „Query Result“ nicht überschrieben wird. Zudem musste die Benutzungsoberfläche erweitert werden, um beim Erstellen einer neuen Regel alle notwendigen Eigenschaften angeben zu können. In ähnlicher Weise musste auch die Benutzungsoberfläche zum Bearbeiten einer Regel ergänzt werden.

Wie in Abbildung 12 zu sehen ist, können auf der neuen Benutzungsoberfläche alle Regeleigenschaften definiert werden. Die Abbildung zeigt beispielhaft das Fenster zur Bearbeitung der Regeleigenschaften für die Regel aus Listing 1 (Seite 6). In dem Textfeld „Attributes for Excel sheet“ können die Attribute für den Fehlerbericht

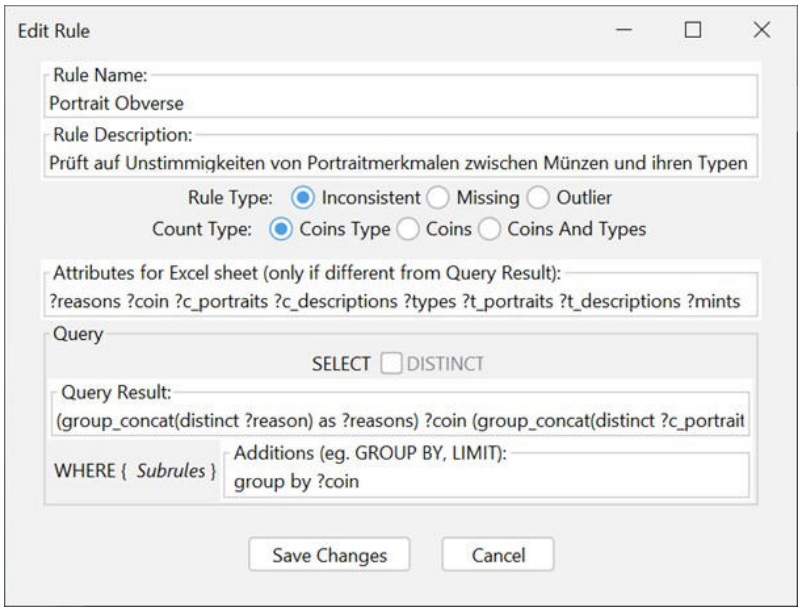

Abbildung 12: Bearbeitung der Regeleigenschaften in der neuen Programmversion

angegeben werden, falls diese von dem „Query Result“ abweichen. Für die Definition des „Query Result“ und der „Additions“ wurden ebenfalls Textfelder eingefügt. Zudem ist es über eine Checkbox möglich, dass Keyword DISTINCT auszuwählen. Um die Eingabe verständlich zu gestalten, wurde die Anordnung dem Aufbau einer SPARQL-Abfrage nachempfunden.

Des Weiteren werden beim Speichern einer Regel alle definierten Regeleigenschaften in die Regeldatei geschrieben. Da die Regeleigenschaften „Attributes“ und „Additions“ im Regelmanagementsystem bisher nicht vorgesehen waren, wurden sie nicht in die Regeldatei gespeichert und nicht ins Programm eingelesen. Daher wurden die SPARQL-Abfragen bei der Ausführung der Datenqualitätsprüfung ohne „Additions“ ausgeführt, was bei manchen Regeln einen Fehler verursachte. Durch das Fehlen der „Attributes“ wurde die Auflistung der Fehler im Fehlerbericht für die Regeln, die im SELECT-Teil Funktionen und Aliase verwenden, nicht korrekt dargestellt. Um diese Probleme zu lösen, mussten entsprechende Funktionen zum Speichern und Einlesen der Regeleigenschaften „Attributes“ und „Additions“ in den Klassen RuleObjectMaker, RuleObject und Rule ergänzt werden. Somit kann die Datenqualitätsprüfung ausgeführt und ein Fehlerbericht erstellt werden, der alle geforderten Attribute enthält.

4.3.3 Darstellung im SPARQLBuilder

Für das Erstellen und Bearbeiten von Unterregeln wurde für das Regelmanagementsystem ein spezielles Tool namens SPARQLBuilder entwickelt. Dabei kann über eine Eingabeoberfläche eine SPARQL-Abfrage erstellt werden. Aufgrund der Komplexität der SPARQL-Syntax ist es kaum möglich, eine Benutzungsoberfläche zu entwickeln, auf der die gesamte Syntax dargestellt werden kann. Daher wurden für den SPARQLBuilder die wesentlichen Funktionen implementiert, die für die Definition von Unterregeln benötigt werden: Triples, Optionale Triples, Filter- und Filter Not Exists-Statements. Diese können über die Benutzungsoberfläche spezifiziert werden und der SPARQLBuilder erstellt daraus die SPARQL-Abfrage.

Bei einigen Unterregeln besteht ein Problem bei der Darstellung der Abfrage

im SPARQLBuilder. Wenn komplexere Abfragen mit dem SPARQLBuilder geöffnet werden, können diese nicht richtig dargestellt werden. Der Grund dafür ist, dass wie zuvor erläutert, nicht die gesamte SPARQL-Syntax abgedeckt ist. Daher gibt es in dem Regelmanagementsystem die Möglichkeit, komplexe Abfragen manuell in einem Editor zu bearbeiten. Allerdings wurden auch einfache Abfragen falsch dargestellt, wenn Keywords kleingeschrieben waren oder die Abfrage nicht richtig formatiert war, insbesondere wenn Zeilenumbrüche nicht korrekt gesetzt waren. Wenn eine Unterregel falsch dargestellt wurde, war somit unklar, ob dies nur an der Formatierung oder an der Komplexität der Abfrage lag. In der Praxis konnten daher nur wenige Unterregeln aus den im Code definierten Ersatzregeln mit dem SPARQLBuilder korrekt dargestellt und bearbeitet werden.

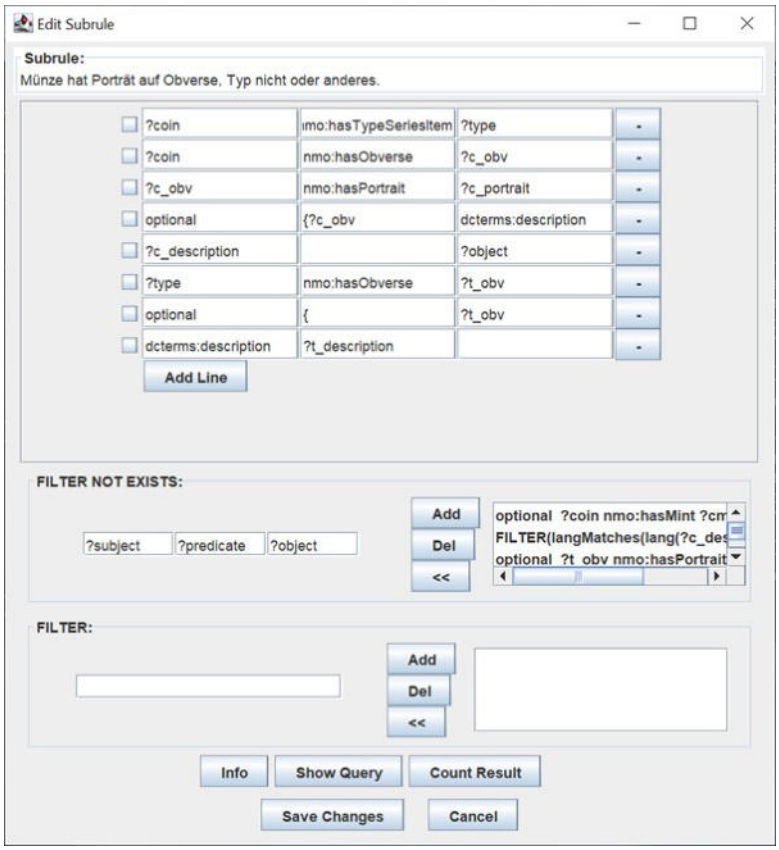

Abbildung 13: Falsche Darstellung einer Regel mit dem SPARQLBuilder

Abbildung 13 zeigt eine Unterregel der Regel in Listing 1 (Seite 6), geöffnet mit dem SPARQLBuilder in der Ausgangsversion des Programms. Diese Unterregel kann nicht korrekt dargestellt werden. Das kleingeschriebene Keyword OPTIONAL wird nicht als solches erkannt. Zudem verursacht der in Listing 7 dargestellte Ausschnitt der Unterregel Probleme. Mehrere Statements in einem Statement, hier ein Triples und ein Filter in einem OPTIONAL-Statement, können auf der Benutzungsoberfläche nicht korrekt dargestellt werden. Diese Unterregel ist also zu komplex, um sie mit dem SPARQLBuilder zu bearbeiten.

```
1 OPTIONAL {?c_obv dcterms:description ?c_description.
2     FILTER(langMatches(lang(?c_description), "EN"))}
```

Listing 7: Ausschnitt einer SPARQL Unterregel aus Listing 1 (Seite 6)

Um dieses Problem zu lösen, musste das Programm um eine Überprüfung ergänzt werden. Mit dieser soll ermittelt werden, ob eine ausgewählte Unterregel mit

dem SPARQLBuilder bearbeitet werden kann. Falls ja, soll sichergestellt werden, dass die Formatierung der Abfrage den im SPARQLBuilder implementierten Bedingungen entspricht. Dazu wurde eine Funktion implementiert, die eine Prüfung der Unterregel durchführt, bevor diese mit dem SPARQLBuilder geöffnet wird. Wenn die Überprüfung ergibt, dass die Unterregel zu komplex ist, um sie mit dem SPARQLBuilder zu öffnen, wird eine Warnung ausgegeben und auf die manuelle Bearbeitung hingewiesen. Andernfalls wird die Formatierung der Unterregel angepasst, sodass diese mit dem SPARQLBuilder geöffnet werden kann. Weiterhin wurde im SPARQLBuilder die Erkennung kleingeschriebener Keywords ergänzt.

4.4 Automatisierter Fehlerberichtsvergleich

Nachdem das Programm aktualisiert und diese Probleme behoben sind, kann das Programm „Data Quality SPARQL Rules“ [2] mit dem Regelmanagementsystem genutzt werden. Das Ausführen der Datenqualitätsprüfung und eine umfassende Bearbeitung der Regeln über die Benutzungsoberfläche ist möglich.

Um die modernisierte Version des Programms „Data Quality SPARQL Rules“ [2] zu überprüfen, soll eine Funktion zum Vergleich der Fehlerberichte implementiert werden. Das Ziel ist es, prüfen zu können, dass die vorgenommenen Änderungen zu keiner Veränderung des Fehlerberichts geführt haben. Wenn die gleichen Regeln und die gleichen Daten wie in der alten Version verwendet werden, sollen im Fehlerbericht die gleichen Ergebnisse dokumentiert werden.

4.4.1 Anforderungen

Die Funktion zum automatisierten Ergebnisvergleich soll zwei Fehlerberichte auf inhaltliche Gleichheit prüfen. Eine solche Testfunktion eignet sich für verschiedene Anwendungsfälle.

Codeänderungen prüfen: Dies ist der ursprünglich beschriebene Einsatzzweck. Dabei soll geprüft werden, dass Änderungen am Programmcode keine Auswirkungen auf das, im Fehlerbericht dokumentierte, Ergebnis haben. Dazu sollen die Fehlerberichte verschiedener Versionen des Programms miteinander verglichen werden können.

Versionsvergleich: Für eine neue Ausführung der Datenqualitätsprüfung kann eine alte, kommentierte Version eines Fehlerberichts als Eingabe genutzt werden. Die alte Version kann mit dem neu erstellten Fehlerbericht verglichen werden. Dabei kann geprüft werden, ob alle Kommentare korrekt übernommen wurden und wie sich das Ergebnis seit der letzten Programmausführung verändert hat. Dazu sollen der Fehlerbericht, der als Eingabe genutzt wurde und der neue Fehlerbericht verglichen werden können.

Regeländerung nachvollziehen: Nach Änderung der SPARQL-Abfrage einer Regel kann verglichen werden, wie sich das Ergebnis verändert hat. Dazu sollen Fehlerberichte vor und nach einer Regeländerung verglichen werden können.

Diese drei beispielhaften Einsatzzwecke verdeutlichen, dass ein automatisierter Vergleich der Fehlerberichte vielseitig eingesetzt werden kann. Um diese Anwendungsfälle zu realisieren, werden folgende Anforderungen definiert:

A1. Benutzungsfreundliche Oberfläche: Die Ausführung des Vergleichs soll über eine einfache Benutzungsoberfläche aufgerufen werden können. Damit soll die Vergleichsfunktion auch ohne Programmierkenntnisse genutzt werden können.

A2. Zwei Exceldateien als Eingabe: Auf der Benutzungsoberfläche sollen zwei Exceldateien als Eingabe für den Vergleich ausgewählt werden können.

A3. Dokumentation der Ergebnisse: Neben dem Vergleichsergebnis sollen alle vorhandenen Unterschiede gefunden und dokumentiert werden.

A4. Automatisierte Prüfung auf inhaltliche Gleichheit: Die Ausführung des Gleichheitstests soll automatisiert erfolgen. Dabei ist es wichtig, dass die Prüfung auf inhaltliche Gleichheit abzielt. Unterschiede in der Reihenfolge der Fehlerauflistung sollen nicht als Unterschiede berücksichtigt werden.

4.4.2 Konzeption

Anforderung A1, eine benutzungsfreundliche Oberfläche, soll erfüllt werden, indem ein Fenster implementiert wird, in dem alle Funktionen aufgerufen werden können. Damit soll die Vergleichsfunktion auch ohne Programmierkenntnisse genutzt werden können. Alle ausgewählten Dateien sowie das Vergleichsergebnis sollen auf der Benutzungsoberfläche angezeigt werden.

Anforderung A2, die Auswahl von zwei Exceldateien, soll auf dieser Benutzungsoberfläche ermöglicht werden. Dort sollen über Buttons zwei Dateien zum Vergleich ausgewählt werden können.

Für Anforderung A3, die Dokumentation der Ergebnisse, ist es erforderlich, eine Datei zum Speichern des Vergleichsergebnis (gleich oder nicht gleich) und der gefundenen Unterschiede auszuwählen.

Anforderung A4, die Prüfung auf inhaltliche Gleichheit, bedeutet insbesondere, dass die Reihenfolge der aufgelisteten Fehler des Fehlerberichts keine Beachtung finden darf. Es ist zu berücksichtigen, dass nicht jeder Wert einer Zeile relevant ist. Beispielsweise sind die Ergebnisse mit einer laufenden Nummer versehen, die für den inhaltlichen Vergleich irrelevant ist.

Der Fehlerbericht enthält weitere Attribute, die je nach Anwendungsfall im Vergleich ignoriert werden können. Dazu gehören das Datum in den Regelseiten und die SPARQL-Abfrage auf der Übersichtsseite. Um diese Attribute zu ignorieren, sollen Optionen hinzugefügt werden.

Die Option „Datum ignorieren“ kann beispielsweise bei dem Anwendungsfall „Codeänderungen prüfen“ nützlich sein. Der Eintrag jedes gefundenen Fehlers enthält ein Feld mit Datum und Uhrzeit der Programmausführung. Wenn zwei Fehlerberichte nach Änderungen am Programmcode verglichen werden, soll nur geprüft werden, dass die gleichen Fehler gefunden werden. Die Berücksichtigung des Datums würde zu einem unübersichtlichen Ergebnis führen, da die Zelle Datum für jede Zeile als Unterschied erkannt wird. Bei dem Anwendungsfall „Versionsvergleich“ ist eine Berücksichtigung des Datums wichtig, um die beiden Versionen des Fehlerberichts zu vergleichen. Wenn eine alte Version eines Fehlerberichts als Eingabe genutzt wird, übernimmt das Programm das älteste Datum in die neue Version. Damit wird festgehalten, wann der Fehler zuerst aufgetreten ist.

Die Option „SPARQL-Abfrage ignorieren“ kann bei dem Anwendungsfall „Regeländerung nachvollziehen“ hilfreich sein. Da die Übersichtsseite jedes Fehlerberichts

die gesamte SPARQL-Abfrage enthält, werden diese bei unterschiedlichen Abfragen in der Unterschiedsdokumentation ausgegeben. Ist bekannt, dass die Abfragen verschieden sind, wird das Ergebnis übersichtlicher, wenn diese nicht berücksichtigt werden.

4.4.3 Implementierung

Nachdem alle Anforderungen definiert und die Umsetzung konzipiert waren, wurde die Vergleichsfunktion implementiert. Die Funktion kann über den Button „Compare files“ im Tab „File Chooser“ aufgerufen werden. Wenn die Datenqualitätsprüfung zuvor ausgeführt wurde, werden die Dateipfade der Eingabedatei und der Ergebnisdatei als Vorauswahl für den Vergleich übernommen. Der Vergleich dieser beiden Dateien entspricht dem Anwendungsfall „Versionsvergleich“ und kann nun direkt ausgeführt werden. Damit wird eine gute Einbindung in das Programm erreicht.

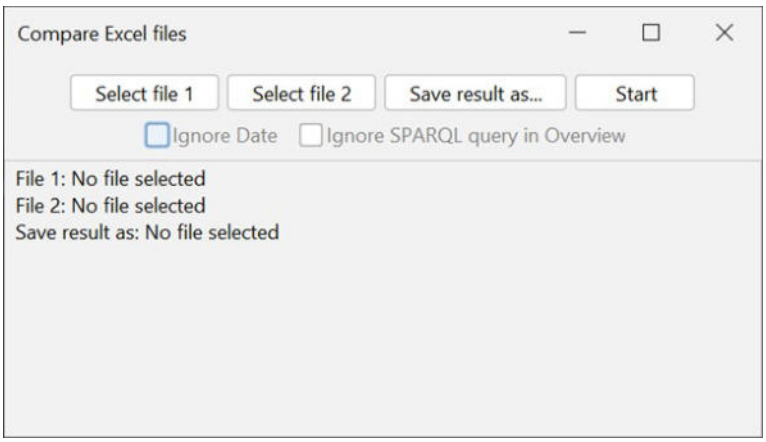

Abbildung 14: Benutzungsoberfläche für den Fehlerberichtsvergleich

Abbildung 14 zeigt die Benutzungsoberfläche für den Vergleich. Über die Buttons „Select file 1“ und „Select file 2“ können die Dateien ausgewählt beziehungsweise geändert werden. Um das Ergebnis zu speichern, muss eine Datei über den Button „Save result as...“ angegeben werden. Alle ausgewählten Dateien werden auf der Benutzungsoberfläche angezeigt. Mit zwei Checkboxen können die Optionen „ignore Date“ und „ignore SPARQL query in Overview“ ausgewählt werden. Über den Button „Start“ wird der Vergleich gestartet. Das Ergebnis des Vergleichs wird auf der Benutzungsoberfläche angezeigt.

Die Implementierung der Benutzungsoberfläche und aller zugehörigen Funktionen ist in der Klasse FileComparison zu finden. Für die Vergleichsausführung wurde die Klasse CompareExcel implementiert.

Bei der Ausführung des Vergleichs wird in drei Schritten, entsprechend den möglichen Fehlerarten, vorgegangen. Dabei ist grundlegend zu beachten, dass die Einträge im Fehlerbericht zeilenweise aufgelistet werden. Es kann zwischen drei Arten von Fehlern unterschieden werden:

- Fehlende Excel-Blätter
- Fehlende Einträge
- Gleiche Einträge mit unterschiedlichen Zelleninhalten

Im ersten Schritt werden die Blätter verglichen. Dafür werden jeweils die Namen aller Blätter der beiden Dateien verglichen. Wenn diese nicht übereinstimmen, wird geprüft, welche Blätter in welcher Datei fehlen.

Im zweiten Schritt werden für jedes Blatt, das in beiden Dateien vorkommt, die Einträge verglichen. Beim Vergleich der Regelseiten muss beachtet werden, dass die Reihenfolge der Einträge verschieden sein kann. Die Ergebnisse der SPARQL-Abfragen können zwar durch den Zusatz ORDER BY sortiert werden [9, S. 95ff], aber bei den im Code definierten Ersatzregeln wird kein ORDER BY verwendet. Somit muss beim Vergleich der Einträge davon ausgegangen werden, dass die Reihenfolge in beiden Dateien nicht übereinstimmen muss. Aus diesem Grund ist ein sequenzieller Vergleich der Zeilen anhand der Zeilennummern nicht möglich. Stattdessen werden zuerst alle Einträge zeilenweise mit den Attributnamen und Werten eingelesen. Wenn die Optionen „Datum ignorieren“ oder „SPARQL-Abfrage ignorieren“ ausgewählt wurden, werden diese Attribute übersprungen. Anschließend werden die Einträge ohne Berücksichtigung der Reihenfolge verglichen. Dabei wird geprüft, welche Einträge in der jeweils anderen Datei ungleich sind oder fehlen.

Bei den, im zweiten Schritt gefundenen, unterschiedlichen Zeilen kann es sich um Einträge handeln, die in der jeweils anderen Datei komplett fehlen oder gleiche Einträge, die sich in mindestens einem Wert unterscheiden. Letzteres ist beispielsweise der Fall, wenn der gleiche Fehlereintrag in nur einer Datei mit einem Kommentar versehen ist. Aus diesem Grund wird im letzten Schritt geprüft, um welche Unterschiede es sich handelt.

Dabei besteht die Herausforderung darin, gleiche Einträge mit unterschiedlichen Werten in einzelnen Zellen zu finden. Dazu muss definiert werden, welche Attribute gleiche Einträge identifizieren. Listing 8 zeigt die Definition dieser Attribute in der Klasse CompareExcel.

```
1 String[] identifierInOverview = {"Name"};
2 String[] identifierInRulesSheets = {"?coin", "?reason", "?reasons",
      "?type", "?types"};
```

Listing 8: Definition der Attribute zur Identifizierung gleicher Einträge

Für die Übersichtsseite wird der Regelname gewählt. Dies ist ausreichend um Regeln zu identifizieren, da keine Regeln mit demselben Namen erstellt werden können. Für die Regelseiten ist die Auswahl geeigneter Attribute schwieriger, da jede Regel unterschiedliche Variablen enthalten kann. Aus diesem Grund wurde im Programm eine Liste definiert, die mehrere Variablen enthält, die zur Identifizierung gleicher Einträge benötigt werden. Dafür werden die Variablen, die die Münz-ID, die Fehlerbeschreibung und den Typ angeben, genutzt. Um gleiche Einträge zu finden, werden für jede Regel möglichst viele dieser Attribute ausgewählt, die in dieser Regel vorkommen. Diese Attribute werden nicht nur für den Vergleich benötigt. Bei der Ausgabe werden sie genutzt, um fehlende oder unterschiedliche Zeilen zu identifizieren.

Mit diesen drei Schritten wird eine Prüfung auf inhaltliche Gleichheit durchgeführt ohne die Reihenfolge der Elemente zu berücksichtigen. Alle Unterschiede werden in einer Ergebnisdatei dokumentiert.

In Abbildung 15 ist ein beispielhaftes Ergebnis eines Fehlerberichtsvergleichs zu sehen, der alle drei Fehlerarten enthält. Zu Beginn der Datei wird dokumentiert, welche Dateien verglichen und welche Optionen ausgewählt wurden. Darunter wird das Vergleichsergebnis angegeben. Im Abschnitt „Sheets missing“ werden alle fehlenden Blätter aufgelistet. Danach folgt für jedes Blatt, das Unterschiede enthält, eine

```
File 1: D:\Ressources\Fehlerbericht1.xlsx
File 2: D:\Ressources\Fehlerbericht2.xlsx
Ignore date is selected
Ignore SPARQL query in Overview is selected

The files are not equal!

===== Sheets missing

Missing in file 1: "Diameter Weight";

============================


=====  Sheet "Übersicht"

--- Rows missing in file 1

Row identified by: [Name: Diameter Weight];

============================


=====  Sheet "Portrait Obverse"

--- Rows with differences in cells

Row identified by: [?coin: https://www.corpus-nummorum.eu/CN_3752, ?reasons: Typ hat
Porträt auf Obverse, Münze nicht., ?types: https://www.corpus-nummorum.eu/types/1992]
        Different cell: "erledigt" (file 1 "", file 2 "Test");

============================
```

Abbildung 15: Vergleichsergebnis mit allen drei Fehlerarten

Auflistung der fehlenden oder unterschiedlichen Einträge. Die Zeilen dieser Fehlereinträge werden anhand der zuvor beschriebenen Attribute identifiziert. Dabei wird angegeben, ob Zeilen in einer Datei komplett fehlen oder es gleiche Zeilen mit Unterschieden in einzelnen Zellen gibt.

4.4.4 Evaluation

Mit der zuvor beschriebenen Funktion soll nun die modernisierte Version des Programms „Data Quality SPARQL Rules“ [2] getestet werden. Ein Fehlerbericht der modernisierten Programmversion soll mit einem Fehlerbericht der ursprünglichen Programmversion verglichen werden. Dazu müssen zwei Fehlerberichte generiert werden, die anschließend mit der Vergleichsfunktion verglichen werden. Dieser Test ist nicht allgemeingültig, aber zeigt beispielhaft, wie die Ergebnisse der Datenqualitätsprüfung getestet werden können. In diesem Fall soll insbesondere überprüft werden, dass die Ergebnisse mit den gleichen verwendeten Daten [32] und den gleichen Regeln nicht durch die Modernisierung verändert wurden.

Der erste Fehlerbericht wird mit einer älteren Programmversion, die bisher in der Praxis genutzt wurde, erzeugt. Dafür wird das Programm in der Version vor der zweiten Bachelorarbeit, also ohne das Regelmanagementsystem, ausgeführt. Der Fehlerbericht wird in der Datei „Fehlerbericht_Altsystem.xlsx“ gespeichert.

Der zweite Fehlerbericht wird mit der modernisierten Programmversion mit dem Regelmanagementsystem und nach allen, in diesem Abschnitt beschriebenen Ände-

rungen, erzeugt. Dieser Fehlerbericht wird in der Datei „Fehlerbericht_aktualisiertes-System.xlsx“ gespeichert.

Für die Erstellung der beiden Fehlerberichte werden die gleichen Daten [32] und die gleichen Regeln (die Ersatzregeln im Code) verwendet. Daher ist das erwartete Ergebnis des Vergleichs, dass die Dateien übereinstimmen. Mögliche Unterschiede wären auf die Verwendung des Regelmanagementsystems oder die Modernisierung zurückzuführen.

Nun wird der Dateivergleich mit diesen beiden Fehlerberichten als Eingabe ausgeführt. Es wird die Option „ignore Date“ ausgewählt, da der Zeitpunkt der Erstellung verschieden ist. Zudem ist das Datum für diesen Zweck unerheblich, da nur festgestellt werden soll, ob die gleichen Fehler gefunden werden. Die Option „ignore SPARQL query in Overview“ wird ebenfalls ausgewählt. Es werden zwar die gleichen Regeln ausgeführt, aber durch die zuvor erläuterten Änderungen gibt es Unterschiede bei der Formatierung. Durch die Darstellung der Regel „Start Date after End Date“ in Unterunterabschnitt 4.3.1 (Listing 6, Seite 16) als Regel mit einer Unterregel in den Ersatzregeln wurde die Formatierung dieser Regel verändert. Abbildung 16 zeigt das Ergebnis des Vergleichs.

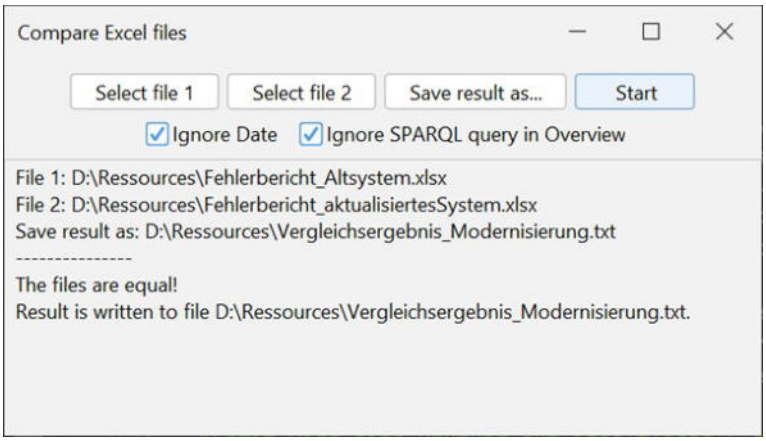

Abbildung 16: Ergebnis des Fehlerberichtsvergleichs

Der Vergleich zeigt, dass die Modernisierung zu keiner Veränderung des Ergebnisses geführt hat. Das Programm „Data Quality SPARQL Rules“ [2] kann nun in der aktuellen Version mit allen Funktionalitäten genutzt werden.

4.5 Zusammenfassung

In diesem Abschnitt wurde die Modernisierung des historisch gewachsenen Programms „Data Quality SPARQL Rules“ [2] beschrieben. Dafür wurde der Ausgangszustand des Programms untersucht. Zuerst wurde ein Vergleich verschiedener Reverse Engineering Tools durchgeführt, die halfen ein genaueres Verständnis des Programmaufbaus zu gewinnen. Daraufhin wurden die verwendeten Komponenten aktualisiert, wobei die Verwendung des Build-Tools Apache Maven eingeführt wurde. Um die Nutzung der aktuellen Programmversion mit allen implementierten Funktionalitäten zu ermöglichen, mussten einige Änderungen vorgenommen werden. Abschließend wurde ein automatisierter Vergleich der Fehlerberichte implementiert. Damit wurde für einen Testfall geprüft und nachgewiesen, dass die modernisierte Programmversion die gleichen Ergebnisse liefert wie die zuvor verwendete, ältere Version.

5 Testdatenmanagement-Tool

Im Rahmen der Ausführung des Programms „Data Quality SPARQL Rules“ [2] werden mit SPARQL definierte Regeln auf eine große Datenmenge ausgeführt. Das Ergebnis wird in einem Fehlerbericht dokumentiert. Abbildung 17 zeigt einen Ausschnitt der Übersichtsseite eines Fehlerberichts [15]. In der Spalte „Anzahl Fälle“ wird angegeben, wie viele Treffer die jeweilige SPARQL-Abfrage gefunden hat. Während in diesem Fall für die Regel „Diameter Weight existing“ über 16.000 Fehler gefunden wurden, betrug die Anzahl gefundener Fehler für drei Regeln Null („Start Date after End Date“, „Diameter“ und „Test Diameter Weight“).

Regel	Name	Beschreibung	Anzahl Fälle
1	Portrait Obverse	Prüft auf Unstimmigkeiten von Portraitmerkmalen zwischen Münzen und ihren Typen (Obverse).	155
2	Portrait Reverse	Prüft auf Unstimmigkeiten von Portraitmerkmalen zwischen Münzen und ihren Typen (Reverse).	39
3	Start Date and End Date fitting	Prüft, ob die Münz-Datierung außerhalb der Type-Datierung liegt.	1065
4	Start Date after End Date	Prüft, ob das Start Date logisch vor dem End Date liegt	0
5	Denomination	ıs gleiche Nominal haben wie der Typ (bei Zuordung zu einem Subtype wird der Wert vom entsprec	531
6	Mint	ı die gleiche Mint hat wie der Typ (bei Zuordung zu einem Subtype wird der Wert vom entsprechen	28
7	Material	das gleiche Material hat wie der Typ (bei Zuordung zu einem Subtype wird der Wert vom entsprech	120
8	Diameter	Prüft MinDiameter und MaxDiameter Angaben.	0
9	Diameter Weight existing	Prüft auf Vollständigkeit der Daten bezogen auf Diameter (Min, Max) und Weight.	16868
10	Start - End Date - missing	Prüft welche Start und End-Dates bei Münzen und Typen fehlen.	91
11	Tests Diameter Weight	Prüft ob 0 oder negative Werte bezogen auf Diameter (Min, Max) und Weight vorliegen.	0
12	Diameter Weight	Prüft auf Extremfälle bei Diameter und Weight relationen.	86

Abbildung 17: Ausschnitt der Übersichtsseite des Fehlerberichts [15]

Eine Kontrolle dieser Ergebnisse ist kaum möglich. Wenn für eine Regel keine Fehler gefunden werden, kann nicht geprüft werden, ob dieser Fehler tatsächlich nicht in den Daten vorkommt. Möglicherweise ist die SPARQL-Abfrage nicht korrekt definiert und findet Daten, die den gesuchten Fehler enthalten, nicht. Aus diesem Grund soll das Programm um ein Testdatenmanagement-Tool erweitert werden, mit dem die definierten Regeln getestet werden können.

Dieser Abschnitt beschäftigt sich mit der Entwicklung dieses Tools. In Unterabschnitt 5.1 werden die Anforderungen für das Testdatenmanagement-Tool formuliert. Daraufhin werden in Unterabschnitt 5.2 die Konzeptionen entsprechend den Anforderungen beschrieben. In Unterabschnitt 5.3 wird die Implementierung und Nutzung der verschiedenen Funktionen erläutert. Abschließend wird in Unterabschnitt 5.4 die Testung der vorhandenen Regeln mit dem entwickelten Tool analysiert.

5.1 Anforderungen

Die Anforderungen an ein Testdatenmanagement-Tool variieren je nach Einsatzzweck stark. In diesem Fall sollen die SPARQL-Regeln im Programm „Data Quality SPARQL Rules“ [2] getestet werden, um sicherzustellen, dass die zu untersuchenden Datenfehler gefunden werden. Dafür werden folgende Anforderungen definiert:

TDM-A1. Benutzungsfreundliche Oberfläche: Alle Funktionen des Testdatenmanagement-Tools sollen über eine einfache Benutzungsoberfläche genutzt werden können. Damit soll es auch ohne Programmierkenntnisse möglich sein, die erstellten Regeln zu testen.

TDM-A2. Testobjektspezifische Testfälle: Das Ziel ist es, jede Unterregel separat zu testen. Eine Unterregel ist somit ein Testobjekt. Es soll möglich sein, Testfälle für jede Unterregel, also testobjektspezifisch, zu erstellen und zu bearbeiten.

TDM-A3. Testfälle erstellen: Es soll eine Möglichkeit geben, neue Testfälle zu erstellen. Die Testfälle sollen aus einer Beschreibung, Testdaten und einem Soll-Ergebnis bestehen.

TDM-A4. Testfälle verwalten: Es soll möglich sein, bestehende Testfälle zu bearbeiten oder zu löschen.

TDM-A5. Testdaten erzeugen: Beim Erstellen von Testfällen soll eine Vorauswahl mit Testdaten erzeugt werden. Diese sollen die abgefragten Attribute enthalten und je nach Anforderung angepasst werden können.

TDM-A6. Archivierung: Um eine Archivierung zu ermöglichen, soll die Speicherung der Testdaten in einer RDF-Datei realisiert werden. Die Testfälle sollen ebenfalls in einer Datei gespeichert werden.

TDM-A7. Reproduzierbarkeit: Um Tests reproduzieren zu können, soll es eine Möglichkeit geben, die gespeicherten Testfälle und Testdaten für erneute Tests aus Dateien laden zu können.

TDM-A8. Bereitstellung der Testdaten zur Testausführung: Um die Tests der Unterregeln durchzuführen, ist es notwendig, dass die erstellten Testdaten zur Abfrage zur Verfügung stehen. Dazu sollen sie auf einen Server geladen und über einen Endpoint abgerufen werden können. Auch das Laden der Testdaten aus einer RDF-Datei soll ermöglicht werden.

TDM-A9. Testausführung: Über das Testdatenmanagement-Tool soll die Testausführung gestartet werden können. Bei der Ausführung soll mithilfe der definierten Testfälle ein Soll-Ist-Vergleich durchgeführt werden.

TDM-A10. Dokumentation der Testergebnisse: Die Ergebnisse der Testausführung sollen in einer Datei dokumentiert werden. Dies bietet eine gute Nachvollziehbarkeit und ermöglicht es, die Ursache für Fehler zu finden.

5.2 Konzeption

Um Anforderung TDM-A1, eine einfache Bedienung über eine Benutzungsoberfläche, zu erfüllen, soll auf der Benutzungsoberfläche des Programms ein neuer Tab für das Testdatenmanagement-Tool hinzugefügt werden. Dort soll eine Übersicht der Regeln und Unterregeln angezeigt werden. Die verschiedenen Funktionen des Testdatenmanagement-Tools sollen über Buttons aufgerufen werden können. Damit kann das Tool auch ohne Programmierkenntnisse genutzt werden.

Anforderung TDM-A2, die testobjektspezifische Testung, kann erfüllt werden, indem alle Funktionalitäten jeweils auf Ebene der Unterregeln konzipiert werden. Es muss also möglich sein, für jede Unterregel Testfälle zu erstellen und zu verwalten. Auch bei der Testausführung muss jede Unterregel separat getestet werden.

Das Erstellen von Testfällen (Anforderung TDM-A3) soll über eine Benutzungsoberfläche für jede Unterregel möglich sein. Dabei sollen eine Beschreibung, Testdaten und ein Soll-Ergebnis angegeben werden können. Abbildung 18 veranschaulicht das Verhältnis zwischen Unterregeln, Testfällen und Testdaten. Zu jeder Unterre-

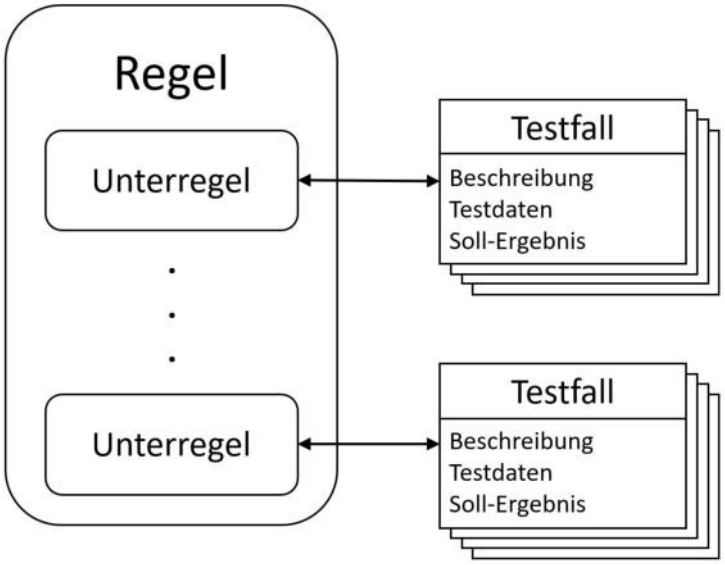

Abbildung 18: Zuordnung von Testdaten zu Testfällen und Unterregeln

gel sollen mehrere Testfälle erstellt werden können. Die Testdaten sind dabei ein Bestandteil eines Testfalls.

Anforderung TDM-A4, das Verwalten von Testfällen, soll für jede Unterregel über eine Benutzungsoberfläche möglich sein. Dort sollen alle Testfälle zu einer ausgewählten Unterregel angezeigt werden. Bei der Auswahl eines Testfalls werden die zugehörigen Informationen angezeigt. Für diesen Testfall können die Beschreibung, die Testdaten und das Soll-Ergebnis bearbeitet und gespeichert werden. Außerdem soll es möglich sein, den ausgewählten Testfall zu löschen.

Für die Testdatenerzeugung (Anforderung TDM-A5) wird ein halbautomatisierter Prozess entwickelt. In Unterabschnitt 2.5 wurde bereits die Frage nach der Verwendung von Echtdaten oder synthetischen Daten betrachtet. Es wurde deutlich, dass beide Möglichkeiten Nachteile mit sich bringen. Insbesondere für die Erstellung von Testdaten für SPARQL-Abfragen, die Fehler in Daten finden sollen, ist es nicht ausreichend einen Auszug aus den Echtdaten zu verwenden. Es kann nicht sichergestellt werden, dass die gesuchte Fehlerkonstellation in den Daten vorkommt. Genau dieses Problem soll mit dem Testdatenmanagement-Tool gelöst werden. Ein Verfahren zur Erstellung synthetischer Testdaten zu entwickeln, wäre in diesem Zusammenhang nicht zielführend. Zum einen liegt eine umfangreiche Menge Echtdaten vor, auf die zurückgegriffen werden kann. Zudem kann bei einer vollautomatisierten Testdatengenerierung kein Soll-Ergebnis erstellt werden und es müsste manuell beurteilt werden, ob die von einer Unterregel gefundenen Testdaten den Anforderungen entsprechen. Aus diesem Grund soll eine Mischform aus Echtdaten und synthetischen Daten erstellt werden. Dabei soll eine Vorauswahl aus den Echtdaten getroffen werden, die manuell bearbeitet werden kann.

Insgesamt ist für das Erstellen von Testfällen und Testdaten folgendes Vorgehen geplant:

1. Vorauswahl von Testdaten aus den Echtdaten erzeugen
    - Passenden Datensatz anhand der Triples einer SPARQL-Abfrage konstruieren
2. Optional eine Beschreibung für den Testfall angegeben

3. Werte der Testdaten entsprechend den gewünschten Anforderungen bearbeiten
4. Auswahl des Soll-Ergebnisses
    - „Positiv“: Die Testdaten dieses Testfalls müssen von der Unterregel gefunden werden.
    - „Negativ“: Die Testdaten dieses Testfalls dürfen von der Unterregel nicht gefunden werden.
5. Speichern des Testfalls

Dieses Vorgehen mit den beschriebenen 5 Schritten bietet einige Vorteile. Zum einen ist es möglich, die Testdaten gezielt zu bearbeiten. Dadurch können systematisch Testfälle, wie Normal-, Fehler-, Grenz- und Sonderfälle, definiert werden. Des Weiteren ermöglicht die Auswahl des Soll-Ergebnisses bei der Testausführung einen Soll-Ist-Vergleich durchzuführen. Erst dieser Soll-Ist-Vergleich erlaubt eine Bewertung der SPARQL-Abfragen und führt zu einem verlässlichen Urteil über die Korrektheit der Regeln.

Weiterhin soll die Archivierung, Anforderung TDM-A6, ermöglicht werden. Dazu sollen alle erstellten Testdaten in einer ausgewählten Datei im RDF-Format gespeichert werden. Zusätzlich sollen alle Testfälle in einer zweiten Datei gespeichert werden. Dabei muss sichergestellt werden, dass die Zuordnung der Testdaten zu den Testfällen erhalten bleibt.

Anforderung TDM-A7, die Reproduzierbarkeit der Tests, soll umgesetzt werden, indem eine Möglichkeit geschaffen wird, die gespeicherten Dateien mit Testfällen in das Programm und Testdaten auf den Server zu laden. Damit kann der Zustand der Testdaten, der beim Speichern der Daten vorlag, jederzeit wiederhergestellt werden. Vorteilhaft für die Sicherstellung der Reproduzierbarkeit ist, dass bei der Testausführung nur SELECT-Abfragen genutzt werden und damit die Testdaten unverändert bleiben. Somit können die Testdaten bei wiederholten Testausführungen weiterverwendet und müssen nicht vor jedem Testlauf neu geladen werden.

Um Anforderung TDM-A8, die Bereitstellung der Testdaten zur Testausführung, umzusetzen, sollen alle erstellten und bearbeiteten Testdaten auf den Server geladen werden. Dabei ist eine Trennung von Testdaten und Originaldaten zu beachten, damit keine versehentliche Veränderung oder Beschädigung der Originaldaten erfolgt. Daher muss für die Testdaten ein separater Endpoint angegeben werden.

Bei der Testausführung (Anforderung TDM-A9) soll es zwei Optionen geben. Zum einen soll ein Test für eine einzelne Unterregel ausgeführt werden können, um den Entwicklungsprozess einer Unterregel zu unterstützen. Weiterhin soll es möglich sein, alle Unterregeln in einem Testlauf zu testen. Mit den in den Testfällen definierten Soll-Ergebnissen soll bei der Testausführung ein Soll-Ist-Vergleich durchgeführt werden.

Anforderung TDM-A10, die Dokumentation der Testergebnisse, resultiert unmittelbar aus den Ergebnissen der Testausführung. Neben dem Gesamtergebnis des Tests (erfolgreich, fehlgeschlagen oder nicht getestet), sollen auch die Testergebnisse für jede Unterregel angegeben werden. Ist der Test einer Unterregel fehlgeschlagen, soll zudem dokumentiert werden, welche Testfälle nicht erfolgreich waren. Des Weiteren sollen verschiedene Kennzahlen, wie beispielsweise die Anzahl der erstellten Testfälle pro Unterregel, erfasst werden. Diese sollen einen Überblick über die Qualität des Tests ermöglichen.

5.3 Implementierung

Im Folgenden wird die Implementierung gemäß den zuvor beschriebenen Anforderungen und Konzeptionen beschrieben. Dabei wird die Nutzung der verschiedenen Funktionalitäten anhand der Benutzungsoberfläche erläutert.

Um das Testdatenmanagement-Tool in das bestehende Programm einzubinden wurde ein neuer Tab auf der Benutzungsoberfläche des Programms hinzugefügt, in dem alle Funktionen des Tools abrufbar sind. Die Benutzungsoberfläche und die entsprechenden Funktionen wurden in der Klasse TestDataManagement implementiert. Abbildung 19 zeigt die Benutzungsoberfläche des Tools beim Aufruf über den Tab „Test Data Management“.

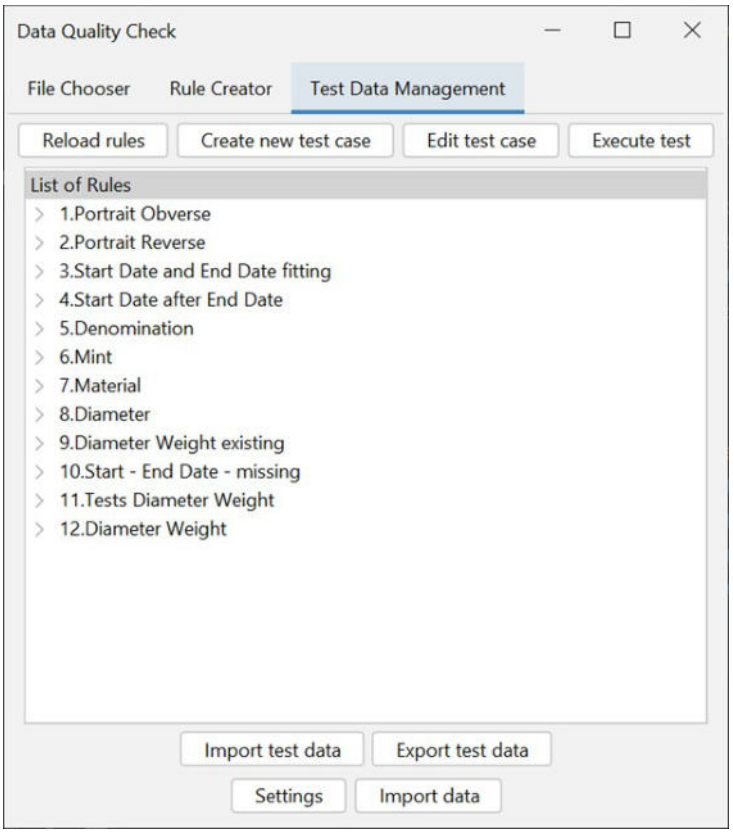

Abbildung 19: Benutzungsoberfläche des Testdatenmanagement-Tools

Die Auswahl der Unterregeln wurde analog zum Regelmanagementsystem implementiert. Um die bereits vorhandenen Funktionen für die Regelübersicht zu nutzen, wurden alle dafür notwendigen Funktionen aus der Klasse RuleGeneratorV8 in eine neue Klasse RuleTree überführt. Auch das Einlesen der Regeldatei zum Erstellen der Regelübersicht wurde in die Klasse RuleTree übernommen. Damit können beide Tools, das Regelmanagement und das Testdatenmanagement, ein Objekt der Klasse RuleTree erstellen und dieselben Funktionen nutzen. Durch diese neue Klasse wird eine Codeduplizierung vermieden, vorhandener Code wiederverwendet und so die Wartbarkeit verbessert. Zudem wird die Benutzungsoberfläche einheitlich gestaltet, was die Nutzung vereinfacht. Listing 9 zeigt das initiale Erstellen eines Objekts der Klasse RuleTree für die Regelübersicht und den Funktionsaufruf zum Einlesen der Regeln aus der Regeldatei. Beim Einlesen wird die Regelübersicht aktualisiert und eine Liste aller Regeln zurückgegeben.

```
RuleTree tree = new RuleTree(false);
ArrayList<Rule> ruleList = tree.readRulesFromFile(false);
```

Listing 9: Erstellen eines RuleTree und Einlesen der Regeln

Alle Funktionen, die zur Verwaltung der Testfälle benötigt werden, wurden in der Klasse TestCaseHandler implementiert. Dort wird auch die Zuordnung der Testfälle zu den Unterregeln verwaltet. Listing 10 zeigt den initialen Aufruf zum Erstellen eines Objekts der Klasse TestCaseHandler. Dabei wird die Liste der Regeln übergeben, um für alle Unterregeln Testfälle anlegen zu können. Für die Testfälle wurde die Klasse TestCase implementiert.

```
TestCaseHandler testCasesList = new TestCaseHandler();
testCasesList.initTestCaseList(ruleList);
```

Listing 10: Initialisieren eines Objekts der Klasse TestCaseHandler

Die Regelübersicht auf der Benutzungsoberfläche kann über den Button „Reload rules“ aktualisiert werden. Änderungen im Regelmanagementsystem werden mit dem Aktualisieren der Regelübersicht übernommen. Da beispielsweise das Löschen einer Unterregel nach dem Aktualisieren auch zum Löschen der zugehörigen Testfälle führt, muss die Aktualisierung explizit über einen Button durchgeführt werden.

Die weiteren Funktionen werden im Folgenden beschrieben.

5.3.1 Neue Testfälle erstellen

Nach Auswahl einer Unterregel öffnet der Button „Create new test case“ ein Fenster zum Erstellen eines neuen Testfalls und neuer Testdaten. Da eine Vorauswahl aus den Echtdaten getroffen wird, ist eine Verbindung zum Server notwendig.

Beispielhaft ist in Abbildung 20 die Benutzungsoberfläche zum Erstellen eines neuen Testfalls zu sehen. Dieses Beispiel zeigt einen Testfall für eine Unterregel der Regel in Listing 1 (Seite 6) mit einer Vorauswahl aus den Echtdaten im XML-Format.

Im oberen Bereich des Fensters ist die Beschreibung und die SPARQL-Abfrage der Unterregel zu sehen. Unterhalb der Abfrage kann in einem Textfeld eine Beschreibung für den neu angelegten Testfall hinzugefügt werden. Darunter wird in einem Textfeld die Vorauswahl aus den Echtdaten angezeigt. Diese Daten können manuell bearbeitet werden. Anschließend muss das Soll-Ergebnis ausgewählt und der Testfall gespeichert werden.

Die Vorauswahl der Testdaten erfolgt anhand der SPARQL-Abfrage der ausgewählten Unterregel. Die Herausforderung bei dem in Unterabschnitt 5.2 beschriebenen Konzept zur Testdatenerzeugung besteht darin, eine geeignete Vorauswahl mit Testdaten auszugeben. Diese Vorauswahl soll alle in einer Unterregel abgefragten Attribute enthalten, sodass alle Werte gemäß der Anforderung an den Testfall manuell bearbeitet werden können. Dazu soll ein passender Auszug aus den Originaldaten gefunden werden. Zu diesem Zweck muss eine neue SPARQL-Abfrage konstruiert werden. Diese muss alle Triples der Abfrage der Unterregel enthalten, darf aber aus keinen weiteren Einschränkungen, wie beispielsweise FILTER, bestehen. Dazu wurde eine Funktion implementiert, die aus der Abfrage der Unterregel alle Einschränkungen (FILTER, FILTER NOT EXISTS, etc.) entfernt. Somit entsteht eine Abfrage, nur bestehend aus den Triples, die in der Unterregel definiert sind. So ist es möglich in den Echtdaten einen Datensatz zu finden, der alle notwendigen Attribute aus der SPARQL-Abfrage der Unterregel enthält. Es wird eine CONSTRUCT-Abfrage

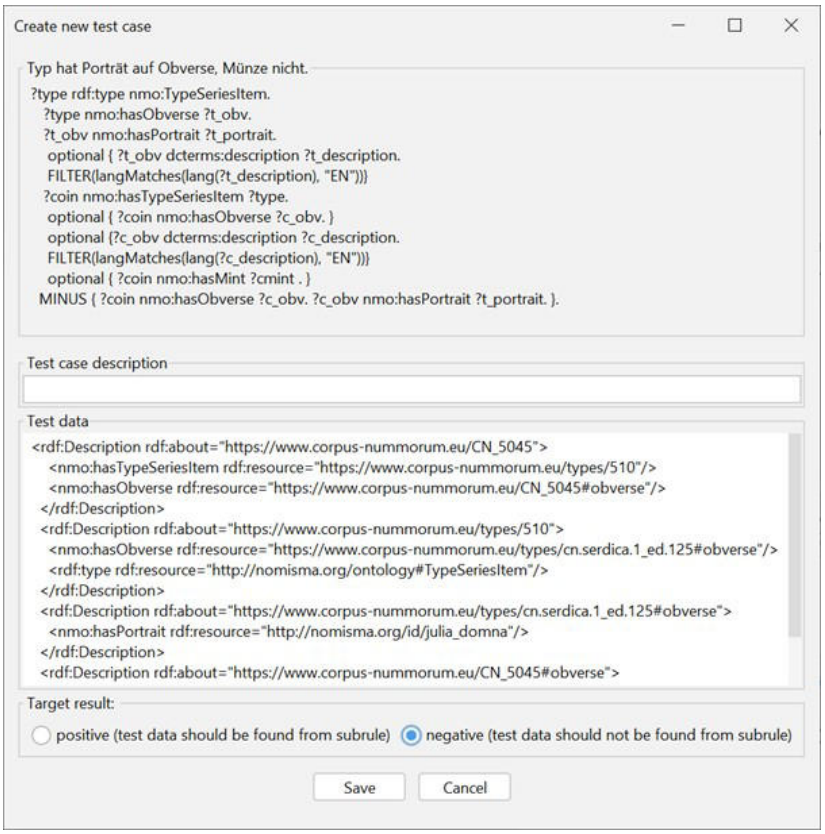

Abbildung 20: Benutzungsoberfläche zum Erstellen eines neuen Testfalls

genutzt, um aus den gefundenen Daten einen vollständigen Datensatz zu erstellen. Da nur ein Datensatz benötigt wird, ist die Anzahl der Ergebnisse durch LIMIT 1 begrenzt. Durch wiederholtes Erstellen eines neuen Testfalls wird jeweils ein weiterer Datensatz aus den Originaldaten geladen.

Listing 11 zeigt beispielhaft ein neu generiertes SPARQL-Abfragemuster, um einen Datensatz mit einer Vorauswahl aus den Originaldaten abzufragen. Diese Abfrage wurde aus der in Abbildung 20 dargestellten Unterregel erzeugt und enthält nur die Triples dieser Unterregel. Dieses Abfragemuster wird im CONSTRUCT-Teil und WHERE-Teil der Abfrage eingesetzt. Ein beispielhaftes Ergebnis ist der Datensatz in Abbildung 20 im Textfeld „Test data“.

```
1 ?type rdf:type nmo:TypeSeriesItem.
2 ?type nmo:hasObverse ?t_obv.
3 ?t_obv nmo:hasPortrait ?t_portrait.
4 ?coin nmo:hasTypeSeriesItem ?type.
5 ?coin nmo:hasObverse ?c_obv.
6 ?c_obv nmo:hasPortrait ?t_portrait.
```

Listing 11: Beispielhaftes SPARQL-Abfragemuster zur Vorauswahl von Testdaten

Falls kein passender Datensatz in den Echtdaten gefunden wird, können Testdaten auch manuell erstellt werden. Dabei ist zu beachten, dass alle notwenigen Präfixe angegeben werden müssen.

Beim Speichern eines Testfalls werden die neu erstellten Testdaten auf den Server geladen, damit sie zur Testausführung zur Verfügung stehen. Dafür werden die Subjekte der Testdaten intern umbenannt, sodass diese eindeutig den Testfällen zugeordnet werden können. Vor der INSERT-Anfrage wird durch ASK-Anfragen an den Endpoint mit den Testdaten sichergestellt, dass keine Subjekte mit der gleichen

Bezeichnung mehrfach in den Testdaten vorkommen. Dabei wird auf die Konsistenz der Bezeichnungen im neu erstellten Datensatz geachtet, damit die Beziehungen zwischen den Daten korrekt erhalten bleiben. Somit kann bei der Testausführung geprüft werden, dass die gefundenen Ergebnisse tatsächlich den in den jeweiligen Testfällen definierten Testdaten entsprechen. Zudem wird dem Testfall beim Speichern eine ID zugewiesen, damit dieser beim Bearbeiten und in der Testdokumentation identifiziert werden kann.

5.3.2 Testfälle verwalten

Um Testfälle und Testdaten zu bearbeiten oder zu löschen, muss eine Unterregel und der Button „Edit test case“ ausgewählt werden. Auch hierfür ist eine Serververbindung notwendig.

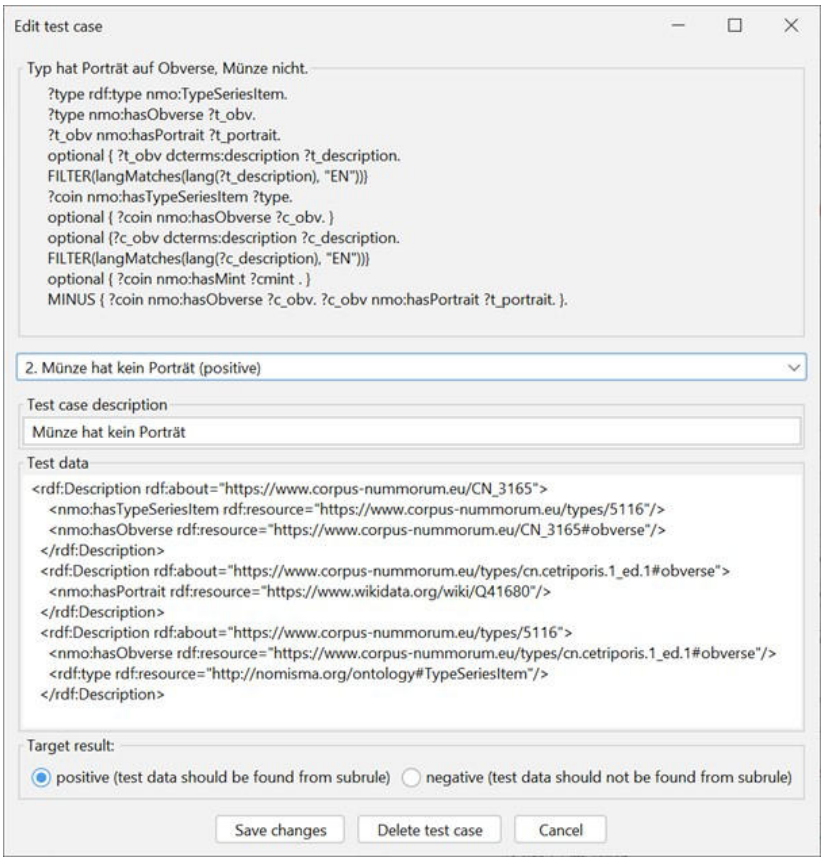

Abbildung 21: Benutzungsoberfläche zum Bearbeiten eines Testfalls

Abbildung 21 zeigt die Benutzungsoberfläche zum Bearbeiten oder Löschen eines Testfalls. In diesem Beispiel ist ein Testfall für eine Unterregel der Regel in Listing 1 (Seite 6) zu sehen.

In dem Fenster zum Bearbeiten der Testfälle ist zunächst die Beschreibung und die SPARQL-Abfrage der Unterregel zu sehen. Darunter befindet sich eine Dropdown-Liste zur Auswahl der Testfälle. Nach Auswahl eines Testfalls werden die Beschreibung, die Testdaten und das Soll-Ergebnis angezeigt. Daraufhin kann dieser Testfall bearbeitet werden. Mit dem Button „Delete test case“ kann der ausgewählte Testfall gelöscht werden. Mit dem Button „Save changes“ werden die Änderungen gespeichert. Die Änderungen werden in den Testdaten auf dem Server vorgenommen, damit die Testdaten zur Testausführung auf dem aktuellen Stand sind.

5.3.3 Testausführung und Dokumentation

Die Testausführung kann über den Button „Execute test“ gestartet werden. Dabei muss zuerst ein Speicherort für die Dokumentation der Testergebnisse ausgewählt werden. Wenn zuvor eine Unterregel selektiert wurde, ist es möglich auszuwählen, ob der Test für alle Unterregeln oder nur für die selektierte Unterregel durchgeführt werden soll. Ist keine Unterregel ausgewählt, wird der Test für alle Unterregeln durchgeführt. Die Funktionsweise ist in beiden Fällen grundsätzlich identisch. Sollen alle Unterregeln getestet werden, wird die Funktion zum Testen einer Unterregel in einer Schleife für alle Unterregeln aufgerufen.

Für die Testausführung wurde die Klasse TestExecution implementiert. Bei der Testung einer Unterregel wird diese SPARQL-Abfrage in Form einer SELECT-Abfrage an den Endpoint mit den Testdaten gestellt. Mit dem Ergebnis der Abfrage wird der Soll-Ist-Vergleich durchgeführt.

Abbildung 22 veranschaulicht die Vorgehensweise während des Tests, um das Testergebnis für eine Unterregel zu ermitteln. Zuerst wird geprüft, ob für die entsprechende Unterregel Testfälle definiert sind. Wenn für eine Unterregel keine Testfälle vorliegen, gilt diese Unterregel als nicht getestet. Sind Testfälle definiert, wird geprüft, dass die Daten aller Testfälle mit positivem Soll-Ergebnis in dem Ist-Ergebnis vorhanden sind. Anschließend wird geprüft, dass keine Daten eines Testfalls mit negativem Soll-Ergebnis in dem Ist-Ergebnis vorhanden sind. Sind diese Bedingungen erfüllt, wird das Testresultat für diese Unterregel als erfolgreich ausgegeben. Falls Daten eines positiven Testfalls fehlen oder Daten eines negativen Testfalls in den Ergebnissen vorkommen, gilt der Test als fehlgeschlagen. In diesem Fall wird in der Testdokumentation ausgegeben, welche Testfälle fehlgeschlagen sind.

Die Abbildung zeigt die Vorgehensweise zur Ermittlung des Testergebnisses und nicht den gesamten Ablauf des Soll-Ist-Vergleichs. Für die Ausgabe der fehlgeschlagenen Testfälle ist es notwendig, alle Testfälle zu prüfen. Insbesondere kann die Ausführung des Soll-Ist-Vergleichs nicht beendet werden, sobald ein fehlgeschlagener Testfall gefunden wurde, da für die Fehlerdokumentation alle fehlgeschlagenen Testfälle ermittelt werden sollen.

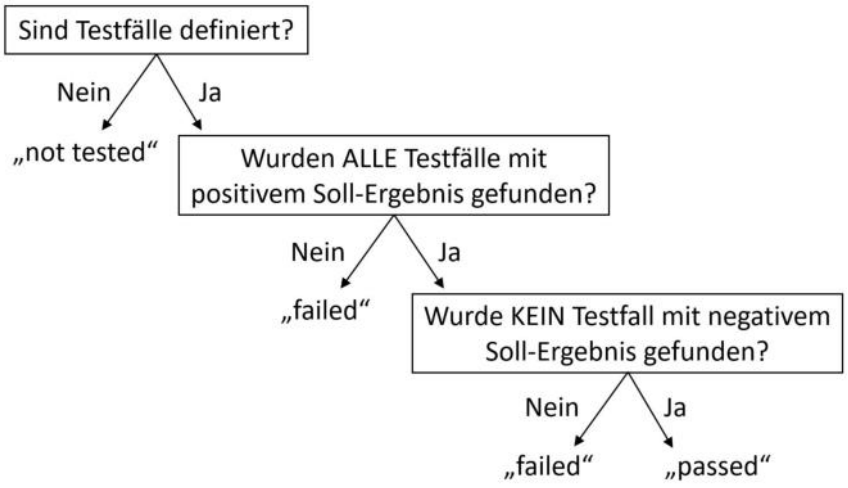

Abbildung 22: Vorgehensweise zur Ermittlung des Testergebnisses

Bei der Testausführung aller Unterregeln wird das Gesamtergebnis als erfolgreich gewertet, wenn die Tests aller getesteten Unterregeln erfolgreich verlaufen. Wenn der Test für mindestens eine Unterregel fehlgeschlagen ist, wird das Gesamtergebnis als

fehlgeschlagen gewertet. Unterregeln, für die keine Testfälle definiert sind, haben keinen Einfluss auf das Gesamtergebnis.

Des Weiteren werden während der Testausführung und Auswertung des Soll-Ist-Vergleichs verschiedene Kennzahlen erfasst. Dabei werden einige Kennzahlen für jede Unterregel separat erfasst. Bei der Ausführung aller Unterregeln werden zusätzlich Kennzahlen zum Gesamtergebnis erfasst. Zu den dokumentierten Kennzahlen gehören:

- Für jede Unterregel:
  - Anzahl der Testfälle mit positivem Soll-Ergebnis
  - Anzahl der Testfälle mit negativem Soll-Ergebnis
  - Anzahl erfolgreich getesteter Testfälle
- Gesamtergebnis (bei der Testausführung für alle Unterregeln):
  - Anzahl getesteter Unterregeln
  - Anzahl nicht getesteter Unterregeln (aufgrund fehlender Testdaten)
  - Anzahl erfolgreich getesteter Unterregeln
  - Anzahl nicht erfolgreich getesteter Unterregeln

Das Ergebnis der Tests, die gegebenenfalls fehlgeschlagenen Testfälle und die Kennzahlen werden als Testdokumentation in die zuvor angegebene Datei geschrieben. Um einen schnellen Überblick über die Testergebnisse zu ermöglichen, werden in der Testdokumentation erfolgreich getestete Unterregeln mit einem Plus, nicht erfolgreich getestete Unterregeln mit einem Minus und nicht getestete Unterregeln mit einem Fragezeichen gekennzeichnet. Abbildung 23 zeigt einen beispielhaften Ausschnitt aus einer Testdokumentation. Dort sind die verschiedenen Kennzahlen, das Gesamtergebnis des Tests sowie die Testergebnisse der einzelnen Unterregeln zu sehen. Die Unterregeln zeigen die drei möglichen Testergebnisse „passed“, „not tested“ und „failed“. Bei der letzten Unterregel ist auch ein fehlgeschlagener Testfall dokumentiert.

```
Test for all Subrules
Tested Subrules: 2; Not tested Subrules: 38
Successfully tested Subrules: 1; Not successfully tested Subrules: 1
Test result: failed

============================

Rule "Portrait Obverse"
+ Subrule 1. "Typ hat Porträt auf Obverse, Münze nicht." - Test result: passed
        Positive test cases: 2; Negative test cases: 2; Successful test cases: 4
? Subrule 2. "Münze hat Porträt auf Obverse, Typ nicht oder anderes." - Test result: not tested
        Positive test cases: 0; Negative test cases: 0; Successful test cases: 0

============================

Rule "Portrait Reverse"
- Subrule 1. "Münze hat Porträt auf Reverse, Typ nicht oder anders." - Test result: failed
        Positive test cases: 2; Negative test cases: 2; Successful test cases: 3
----- Failed test cases -----
ID 2, Description "Münze hat kein Porträt" (target: positive - actual: negative)
----------------------------
```

Abbildung 23: Ausschnitt einer beispielhaften Testdokumentation

5.3.4 Testdaten speichern und laden

Durch Funktionen zum Speichern und Laden von Testdaten aus einer Datei wird die Archivierung und Reproduzierbarkeit ermöglicht. Dabei sollen alle definierten Testfälle sowie die Testdaten aller Unterregeln in jeweils einer Datei gespeichert werden.

Zum Speichern muss der Button „Export test data“ gewählt werden. Daraufhin muss ein Dateipfad ausgewählt und ein Dateiname angegeben werden. Dort werden die Testdaten als RDF-Datei gespeichert. Die zugehörige Zuordnung der Testdaten zu den Testfällen und Unterregeln wird unter demselben Dateinamen als JSON-Datei gespeichert.

Zum Laden der Testdaten und Testfälle muss der Button „Import test data“ gewählt werden. Dann muss die gewünschte RDF-Datei ausgewählt werden. Die Testdaten in dieser Datei werden nun auf den Server geladen. Zudem muss eine JSON-Datei mit dem gleichen Dateinamen vorliegen, die die Zuordnung der Testdaten zu den Testfällen und Unterregeln enthält. Wenn diese Datei nicht vorliegt, können die Testdaten nicht genutzt werden. Um den gleichen Zustand wie beim Speichern der Testdaten wiederherzustellen, werden beim Importieren alte Testdaten am angegeben Endpoint des Servers und Testfälle im Programm überschrieben. Damit wird sichergestellt, dass keine Daten von früheren Tests das Ergebnis verfälschen.

Die Funktionen zum Importieren und Exportieren der Testdaten sind in der Klasse ImportExportData zu finden. Dazu werden die HTTP-Methoden GET und POST genutzt.

Die Funktion zum Laden einer RDF-Datei wurde generisch gehalten und kann daher nicht nur für Testdaten genutzt werden. Dies ermöglicht es, die Funktion auch für das Hochladen der Originaldaten auf den Server zu nutzen. Dazu wurde der Button „Import data“ hinzugefügt, der aus allen Tabs des Programms aufgerufen werden kann. So kann direkt auf der Benutzungsoberfläche des Programms eine RDF-Datei ausgewählt und für die Datenqualitätsprüfung an den angegebenen Endpoint geladen werden.

5.3.5 Einstellungen

Bisher war es über den Button „Set Endpoint“ möglich, den Endpoint zu den Originaldaten für die SPARQL-Abfragen der Datenqualitätsprüfung zu konfigurieren. Wie in Unterabschnitt 5.2 erläutert wurde, ist für das Testdatenmanagement-Tool eine klare Unterscheidung zwischen dem Endpoint für die Originaldaten und dem Endpoint für die Testdaten notwendig. Um auch den Endpoint für die Testdaten und weitere Konfigurationen ändern zu können, wurde die Klasse Settings implementiert. Die Einstellungen können über den Button „Settings“ aus allen Tabs des Programms aufgerufen werden.

Abbildung 24 zeigt die Benutzungsoberfläche zum Konfigurieren der Einstellungen. In diesen Einstellungen kann, wie zuvor, der Endpoint für die Originaldaten geändert werden. Dies ist in dem obersten Textfeld möglich. Im zweiten Textfeld kann der Endpoint für die Testdaten konfiguriert werden.

Wie aus den vorigen Erläuterungen deutlich wird, ist es an mehreren Stellen des Programms notwendig eine Datei auszuwählen. In der alten Programmversion war dies bisher nur bei der Ausführung der Datenqualitätsprüfung notwendig. Daher war ein Standardpfad zur Dateiauswahl in der Klasse Request definiert. Durch die Erweiterungen wird eine Dateiauswahl zusätzlich beim Importieren und Exportieren

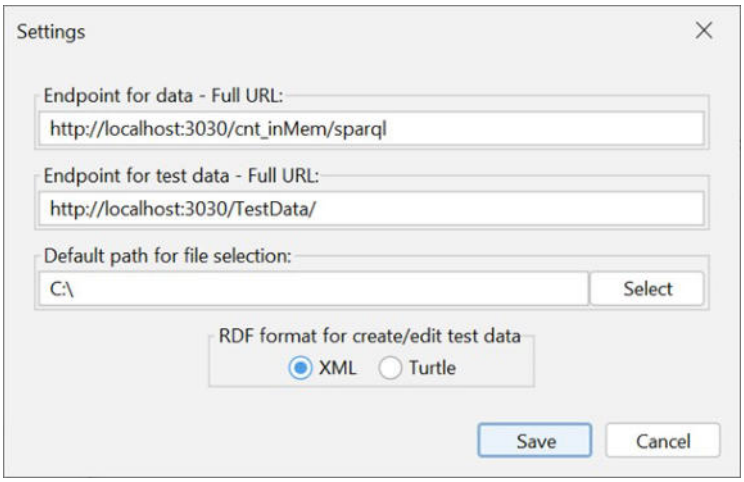

Abbildung 24: Benutzungsoberfläche zum Bearbeiten der Einstellungen

der Daten sowie zum Speichern der im Testdatenmanagement-Tool erzeugten Testdokumentation und für den automatisierten Fehlerberichtsvergleich benötigt. Aus diesem Grund wurde die Konfiguration des Pfades in der Klasse Settings hinzugefügt. Die Auswahl des Pfades kann manuell in einem Textfeld oder über den Button „Select“ erfolgen.

Zusätzlich wurde eine Auswahl des Anzeigeformats für die Testdaten beim Erstellen und Bearbeiten der Testfälle ergänzt. Als Standard werden die RDF-Daten im XML-Format angezeigt. Optional ist es auch möglich, das Turtle-Format zur Anzeige auszuwählen.

5.4 Evaluation

Zur Evaluation des Testdatenmanagement-Tools wurden für alle im Code definierten Ersatzregeln Testfälle erstellt. Dabei wurden beim Erstellen neuer Testfälle für manche Unterregeln keine Beispieldaten als Vorauswahl gefunden. Dafür kann es mehrere Ursachen geben. Wenn beispielsweise die Originaldaten nicht am angegebenen Endpoint abgefragt werden können, weil sie nicht hochgeladen oder der Endpoint falsch konfiguriert wurde, kann keine Vorauswahl gefunden werden. In diesem Fall lagen die Originaldaten vor, aber es konnte kein passender Datensatz gefunden werden. Die Ursache war, dass ein in den Unterregeln verwendetes Prädikat im Datensatz nicht vorkam. Listing 12 zeigt das Triple, zu dem keine Daten gefunden wurden.

```
1 ?st skos:broader ?type.
```

Listing 12: Triple, zu dem keine Vorauswahl mit Testdaten gefunden wurde

Alle erstellten Testfälle wurden in den Dateien testdata.rdf (Testdaten) und testdata.json (Zuordnung der Testfälle) gespeichert.

```
Test for all Subrules
Tested Subrules: 37; Not tested Subrules: 3
Successfully tested Subrules: 28; Not successfully tested Subrules: 9
Test result: failed
```

Abbildung 25: Ausschnitt der Testdokumentation

Abbildung 25 und Abbildung 26 zeigen Ausschnitte aus der Testdokumentation bei der Testausführung für alle Unterregeln. Abbildung 25 zeigt das Testergebnis und einige Kennzahlen. Dort ist zu sehen, dass der Test fehlgeschlagen ist („Test result: failed“). Die Kennzahlen zeigen, dass drei Unterregeln nicht getestet wurden. Der Grund dafür ist das Triple in Listing 12, für das keine Testdaten erstellt werden konnten. Für neun Unterregeln ist der Test fehlgeschlagen.

Nach einer genaueren Untersuchung der fehlgeschlagenen Testfälle konnten die Fehlerursachen gefunden werden:

- In fünf Unterregeln wurde ein falscher Vergleichsoperator genutzt.
- Zwei Unterregeln enthielten Schreibfehler.
- In einer Unterregel wurden Triples mit falschen Variablen definiert.
- Ein Fehler wurde verursacht, da zwei Unterregeln derselben Regel die gleiche ID hatten und daher die Testfälle nicht korrekt zugeordnet wurden.

Beispielhaft ist in Abbildung 26 zu sehen, dass bei der Regel „Tests Diameter Weight - Unterregel 3“ die Testfälle „Gewicht negativ“ und „Min-Diameter negativ“ fehlgeschlagen sind. Wenn man die Unterregel betrachtet, zeigt sich, dass statt dem Kleiner-gleich-Zeichen (<= 0) nur das Gleichheitszeichen (= 0) genutzt wurde. Dadurch wurden negative Werte nicht als Fehler erkannt.

```
Rule "Tests Diameter Weight"
+ Subrule 1. "Angabe 0 oder negativ." - Test result: passed
        Positive test cases: 6; Negative test cases: 1; Successful test cases: 7
+ Subrule 2. "Angabe 0 oder negativ." - Test result: passed
        Positive test cases: 4; Negative test cases: 2; Successful test cases: 6
- Subrule 3. "Angabe 0 oder negativ." - Test result: failed
        Positive test cases: 4; Negative test cases: 2; Successful test cases: 4
----- Failed test cases -----
ID 4, Description "Gewicht negativ" (target: positive - actual: negative)
ID 6, Description "Min-Diameter negativ" (target: positive - actual: negative)
----------------------------
```

Abbildung 26: Fehlgeschlagene Testfälle

Da mit Hilfe der Testdaten die Ursachen für die Fehler gefunden wurden, konnten diese behoben werden. Die Korrekturen wurden direkt in den im Code definierten Ersatzregeln vorgenommen.

Ein erneutes Ausführen der Tests mit den gleichen Testdaten zeigte, dass die Korrekturen die Fehler behoben haben. Abbildung 27 zeigt einen Ausschnitt der Testdokumentation. Es ist zu sehen, dass der Test erfolgreich war („Test result: passed“), da alle getesteten Unterregeln das erwartete Ergebnis lieferten.

```
Test for all Subrules
Tested Subrules: 37; Not tested Subrules: 3
Successfully tested Subrules: 37; Not successfully tested Subrules: 0
Testresult: passed
```

Abbildung 27: Ausschnitt der Testdokumentation nach den Regelaktualisierungen

Um abschließend zu prüfen, wie sich das Ergebnis durch die Regeländerungen verändert hat, soll der automatisierte Fehlerberichtsvergleich aus Unterabschnitt 4.4 genutzt werden.

Als Referenz für den Vergleich wird der Fehlerbericht in der Datei „Fehlerbericht_aktualisiertesSystem.xlsx“ (Unterunterabschnitt 4.4.4) verwendet, der mit den ursprünglichen Regeln erstellt wurde. Dieser wird mit einem Fehlerbericht verglichen, der mit den neuen Regeln generiert und in der Datei „Fehlerbericht_neueRegeln.xlsx“ gespeichert wird.

```
=====  Sheet "Start - End Date - missing"

--- Rows missing in file 1

Row identified by: [?coin: https://www.corpus-nummorum.eu/CN_40541, ?reason: Münze hat kein Enddate,
Type schon, ?type: https://www.corpus-nummorum.eu/types/11919];

Row identified by: [?coin: https://www.corpus-nummorum.eu/CN_39120, ?reason: Münze hat kein Enddate,
Type schon, ?type: https://www.corpus-nummorum.eu/types/12183];

Row identified by: [?coin: https://www.corpus-nummorum.eu/CN_39438, ?reason: Münze hat kein Enddate,
Type schon, ?type: https://www.corpus-nummorum.eu/types/12183];

Row identified by: [?coin: https://www.corpus-nummorum.eu/CN_40543, ?reason: Münze hat kein Enddate,
Type schon, ?type: https://www.corpus-nummorum.eu/types/8249];

Row identified by: [?coin: https://www.corpus-nummorum.eu/CN_23539, ?reason: Münze hat kein Enddate,
Type schon, ?type: https://www.corpus-nummorum.eu/types/12784];
```

Abbildung 28: Ausschnitt des Fehlerberichtsvergleichs nach Regeländerungen

Der Vergleich zeigt, dass mit den korrigierten Regeln mehr Fehler gefunden werden als bisher. Abbildung 28 zeigt einen beispielhaften Ausschnitt des Vergleichsergebnis. Dort ist zu sehen, dass für die Regel „Start - End Date - missing“ in Datei 1 („Fehlerbericht_aktualisiertesSystem.xlsx“) Zeilen, also Fehlereinträge, fehlen.

Dieses Ergebnis verdeutlicht, dass das Testdatenmanagement-Tool dabei unterstützen kann, Fehler in den SPARQL-Abfragen zu beheben. Damit können bisher unbekannte Fehler in den Daten gefunden werden.

5.5 Zusammenfassung

Bei der bisher verwendeten Programmversion gab es keine Möglichkeit die SPARQL-Abfragen zu testen. Daher musste man sich auf die Korrektheit der definierten Regeln verlassen.

Das Testdatenmanagement-Tool ermöglicht es nun, jede Unterregel zu testen. Dafür können gezielt Testfälle für jede Unterregel erstellt und bearbeitet werden. Das Erstellen passender Testdaten wird durch eine Vorauswahl aus den Echtdaten vereinfacht. Des Weiteren wurde die Testausführung sowie die Dokumentation der Testergebnisse implementiert. Durch die ausführliche Dokumentation wird die Nachvollziehbarkeit der Tests erhöht und das Finden der Ursachen für Fehler ermöglicht. Zudem gibt es Funktionen zum Import und Export der Testdaten und Testfälle für die Archivierung und Reproduzierbarkeit von Tests. Die Funktion zum Importieren von Daten konnte ebenfalls eingesetzt werden, um das Laden der Originaldaten zu vereinfachen. Weiterhin wurde mit einer Benutzungsoberfläche für Einstellungen die Konfiguration der wichtigsten Angaben ermöglicht.

Mit diesen Funktionen kann mit dem Testdatenmanagement-Tool überprüft werden, dass die Regeln das gewünschte Resultat liefern. Dadurch kann die Qualität der Fehlerberichte verbessert werden. So konnten beim Ausführen einiger Testfälle bereits Fehler in den bisherigen Regeln gefunden und behoben werden. Mit den neuen Regeln können bisher unentdeckte Fehler in den Daten gefunden werden.

6 Zusammenfassung

Diese Arbeit verfolgte zwei wesentliche Ziele. Zuerst sollte die veraltete Version des Programms „Data Quality SPARQL Rules“ [2] modernisiert werden. Anschließend galt es, ein Testdatenmanagement-Tool zu entwerfen und implementieren, mit dem die definierten Regeln zur Datenqualitätsprüfung getestet werden können.

Der Ausgangszustand des Programms ist auf eine historisch gewachsene Entwicklung zurückzuführen. Einige Ressourcen waren veraltet und durch Fehler, die beim praktischen Einsatz entstanden, waren manche Funktionen nur eingeschränkt nutzbar. Daher war das Ziel, die bisher verwendete Programmversion zu überarbeiten, um alle Komponenten auf den aktuellen Stand zu bringen und die Nutzung aller Funktionalitäten zu ermöglichen.

In der ersten Phase dieser Arbeit wurden die notwendigen Modernisierungen durchgeführt. Dazu wurden zuerst verschiedene Reverse Engineering Tools verwendet und verglichen. Diese halfen, einen umfassenden Überblick über den Aufbau des Programms zu gewinnen. Daraufhin wurde die Verwendung des Build-Tools Maven eingeführt. Dies ermöglichte die einfache Umstellung auf die aktuelle Java-Version sowie die aktuellen Versionen der verwendeten Bibliotheken. Zudem konnte mit Maven eine ausführbare JAR-Datei erstellt werden, sodass das Programm auch ohne Entwicklungsumgebung ausgeführt werden kann. Um alle implementierten Funktionalitäten nutzen zu können, mussten einige Verbesserungen umgesetzt werden. Die größten Änderungen betrafen die Einbindung des Regelmanagementsystems in das Programm. Den Abschluss der Modernisierung bildete die Entwicklung einer Funktion, die zwei, von dem Programm erstellte, Fehlerberichte automatisiert vergleicht.

Bei der Entwicklung der Vergleichsfunktion bestand die Herausforderung darin, gleiche Einträge mit Unterschieden in einzelnen Zellen zu identifizieren. Die Suche nach verschiedenen Werten in einzelnen Einträgen und explizite Unterscheidung zu komplett fehlenden Einträgen ermöglicht eine ausführliche Unterschiedsdokumentation. Diese detaillierte Dokumentation erlaubt eine genaue Analyse der Unterschiede.

Schließlich wurde mit dem automatisierten Fehlerberichtsvergleich getestet, ob der Fehlerbericht der neuen Programmversion mit dem Fehlerbericht der älteren Version, die bisher in der Praxis eingesetzt wurde, übereinstimmt. Durch die Übereinstimmung wurde gezeigt, dass die vorgenommenen Maßnahmen das Ergebnis nicht veränderten und diese Programmversion in der Praxis eingesetzt werden kann.

Die zweite Phase dieser Arbeit beschäftigte sich mit der Entwicklung eines Testdatenmanagement-Tools. Die Notwenigkeit dafür zeigt sich an den Ergebnissen der Datenqualitätsprüfung. Bisher konnte nicht getestet werden, ob die ausgeführten Regeln den gesuchten Fehler tatsächlich finden. Wurden beispielsweise für eine Regel keine Fehler dokumentiert, konnte nicht überprüft werden, ob dieser Fehler nicht existiert oder die SPARQL-Abfrage den Fehler nicht findet. Eine manuelle Überprüfung ist bei der Datenmenge kaum realisierbar. Das Ziel war daher die Erweiterung des Programms, um die Unterregeln testen zu können. Damit sollte eine einfache Möglichkeit geschaffen werden, die definierten SPARQL-Abfragen zu überprüfen.

Mit Hilfe des entwickelten Testdatenmanagement-Tools ist es nun möglich, jede Unterregel zu testen. Dafür können zielgerichtet Testfälle erstellt und bearbeitet werden. Die erstellten Testdaten werden auf einem Server zur Testausführung bereitgestellt. Bei der Testausführung ist sowohl das Testen aller Unterregeln oder einer einzelnen Unterregel möglich. Die Ergebnisse der Testausführung werden in einer detaillierten Testdokumentation gespeichert. Weiterhin können die Testdaten und

Testfälle gespeichert und für erneuten Testläufe geladen werden. Zusätzlich wurde das Laden der Originaldaten an einen angegebenen Endpoint, sowie die Konfiguration verschiedener Einstellungen ermöglicht.

Die größte Herausforderung bei der Entwicklung des Testdatenmanagement-Tools war die Konzeption und Implementierung eines Verfahrens zum Erstellen einer passenden Vorauswahl von Testdaten. Der entwickelte halbautomatisierte Prozess ermöglicht eine einfache und zielgerichtete Generierung von Testdaten für jede Unterregel. Die manuelle Bearbeitung der Testdaten ermöglicht es, systematisch Testfälle anzulegen, um beispielsweise auch Grenzfälle abzudecken. Des Weiteren wird durch die manuelle Auswahl eines Soll-Ergebnisses für jeden Testfall der Soll-Ist-Vergleich während der Testausführung und damit eine Bewertung der Korrektheit der definierten Regeln ermöglicht.

Damit können Tests entworfen werden, um alle Unterregeln zu testen und so mögliche Fehler in den SPARQL-Abfragen zu finden und die Qualität der Ergebnisse zu verbessern. Durch den Einsatz des Testdatenmanagement-Tools konnten bereits in dieser Arbeit Fehler in den Regeln gefunden und behoben werden, sodass mit den korrigierten Regeln weitere bisher unbekannte Fehler in den Originaldaten gefunden werden können. Zukünftig kann das Tool bei der Erstellung neuer Regeln oder Bearbeitung der bestehenden Regeln genutzt werden, um die Korrektheit der SPARQL-Abfragen zu überprüfen.

6.1 Ausblick

Die in dieser Bachelorarbeit erarbeitete Modernisierung und Erweiterung am Programm „Data Quality SPARQL Rules“ [2] ermöglichen eine umfassende Nutzung des Programms. Dennoch bieten sich einige Möglichkeiten zur Weiterentwicklung.

Es ist wichtig, auch in Zukunft regelmäßige Aktualisierungen durchzuführen. Beispielsweise können Sicherheitspatches in den Bibliotheken Aktualisierungen notwendig machen. Durch die, in dieser Arbeit eingeführte, Nutzung von Maven wird die Aktualisierung der verwendeten Bibliotheken und der Java-Version vereinfacht.

Auch die Vergleichsfunktion für Fehlerberichte kann weiterentwickelt werden, um sie beispielsweise für weitere Einsatzzwecke nutzen zu können. So könnte es, je nach Anwendungsfall notwendig sein, weitere Optionen hinzuzufügen. Auch der Vergleich von mehr als zwei Dateien könnte implementiert werden, da es bei der Datenqualitätsprüfung möglich ist, mehrere Fehlerberichte als Eingabe auszuwählen.

Das Testdatenmanagement-Tool könnte ebenfalls um weitere Funktionen ergänzt werden. Eine Möglichkeit wäre eine detailliertere Erfassung von Kennzahlen während des Tests sowie die Erstellung einer Historie, um einen Überblick über erreichte Verbesserungen zu erhalten. Zudem könnten weitere Optionen für die Testausführung erstellt werden. Beispielsweise könnte die Erstellung einer Gruppierung von Testfällen, die in einem Testlauf ausgeführt werden sollen, ergänzt werden. Damit könnte gezielt eine Gruppe von Testfällen ausgewählt werden, die dem Testziel entsprechen.

Diese und weitere Ergänzungen können eine umfangreichere Nutzung des Programms ermöglichen. Zusammenfassend wurde das Programm „Data Quality SPARQL Rules“ [2] umfassend überarbeitet und erweitert. Die aktuelle Version bietet die Möglichkeit, Regeln für die Datenqualitätsprüfung zu erstellen, zu testen und auszuführen. Somit ist die vollständige Nutzung aller konzipierten Funktionen für den praktischen Einsatz möglich, um zur Verbesserung der Datenqualität der Münzdaten im Projekt „Corpus Nummorum“ [1] beizutragen.

Literaturverzeichnis

[1] *Corpus Nummorum*. https://www.corpus-nummorum.eu/. Zuletzt abgerufen am 04.06.2023.

[2] *Java Code des Programms „Data Quality SPARQL Rules“*. Erhalten von Dr. Karsten Tolle.

[3] Erica Vartanian. *Java build tools: Maven vs Gradle*. https://www.educative.io/blog/java-build-tools-maven-vs-gradle. Zuletzt abgerufen am 18.06.2023.

[4] *Reverse Engineering*. https://t2informatik.de/wissen-kompakt/reverse-engineering/. Zuletzt abgerufen am 05.06.2023.

[5] *Linked Open Data*. https://www.w3.org/egov/wiki/Linked_Open_Data. Zuletzt abgerufen am 04.06.2023.

[6] *Linked Data*. https://www.w3.org/standards/semanticweb/data. Zuletzt abgerufen am 04.06.2023.

[7] *RDF*. https://www.w3.org/RDF/. Zuletzt abgerufen am 05.06.2023.

[8] *RDF 1.1 Concepts and Abstract Syntax*. https://www.w3.org/TR/rdf11-concepts/. Zuletzt abgerufen am 02.07.2023.

[9] Bob DuCharme. *Learning SPARQL: Querying and Updating with Sparql 1.1*. 2. Edition. Sebastopol: O'Reilly Media, Inc, 2013.

[10] *SPARQL 1.1 Query Language*. https://www.w3.org/TR/sparql11-query/. Zuletzt abgerufen am 01.07.2023.

[11] *SPARQL 1.1 Graph Store HTTP Protocol*. https://www.w3.org/TR/sparql11-http-rdf-update/. Zuletzt abgerufen am 01.07.2023.

[12] Nick Barney. *Definition - legacy system (legacy application)*. https://www.techtarget.com/searchitoperations/definition/legacy-application. Zuletzt abgerufen am 04.06.2023.

[13] Janet Albrecht-Zölch. *Testdaten und Testdatenmanagement*. 1. Auflage. Heidelberg: dpunkt.verlag GmbH, 2018.

[14] *Apache Jena Fuseki*. https://jena.apache.org/documentation/fuseki2/. Zuletzt abgerufen am 27.06.2023.

[15] *Fehlerbericht der Datenqualitätsprüfung*. 2023_04_7_Comments.xlsx. Erhalten von Dr. Karsten Tolle.

[16] Kateryna Kvasnytsia. *Information Propagation across Versions in Context of a SPARQL-Rules System*. Bachelorarbeit. 2018.

[17] Elif Tugba Dichter und Vladyslav Matsuyev. *Benutzerschnittstelle für die Erstellung von Datenqualitätsabfragen in SPARQL*. Bachelorarbeit. 2019.

[18] *Trail: Creating a GUI With Swing*. https://docs.oracle.com/javase/tutorial/uiswing/. Zuletzt abgerufen am 26.06.2023.

[19] *List of Unified Modeling Language tools*. https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools. Zuletzt abgerufen am 13.07.2023.

[20] *Visual Paradigm*. https://www.visual-paradigm.com/. Zuletzt abgerufen am 12.06.2023.

[21] *Enterprise Architect*. https://www.sparxsystems.de/. Zuletzt abgerufen am 12.06.2023.

[22] *Enterprise Architect*. https://www.sparxsystems.com/. Zuletzt abgerufen am 12.06.2023.

[23] *UML Lab*. https://www.uml-lab.com/. Zuletzt abgerufen am 12.06.2023.

[24] *UML Lab Modeling IDE*. https://marketplace.eclipse.org/content/uml-lab-modeling-ide. Zuletzt abgerufen am 12.06.2023.

[25] *ESS-Model*. https://essmodel.sourceforge.net/. Zuletzt abgerufen am 12.06.2023.

[26] *OpenJDK JDK 20.0.1 General-Availability Release*. https://jdk.java.net/20/. Zuletzt abgerufen am 20.06.2023.

[27] *Apache Maven Project*. https://maven.apache.org/. Zuletzt abgerufen am 20.06.2023.

[28] *Apache Maven Project - POM Reference*. https://maven.apache.org/pom.html. Zuletzt abgerufen am 20.06.2023.

[29] *MVN Repository - Apache Maven ARQ 4.9.0*. https://mvnrepository.com/artifact/org.apache.jena/jena-arq/4.9.0. Zuletzt abgerufen am 12.07.2023.

[30] Torsten Horn. *Maven 3.5*. https://www.torsten-horn.de/techdocs/maven.htm. Zuletzt abgerufen am 26.06.2023.

[31] *FlatLaf - Flat Look and Feel*. https://www.formdev.com/flatlaf/. Zuletzt abgerufen am 22.06.2023.

[32] *RDF-Datei mit Originaldaten*. dump_2023_04_07_TypeNames_2.rdf. Erhalten von Dr. Karsten Tolle.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings