Fachbereich Informatik und Medien

Masterarbeit

Entwicklung eines Frameworks zur Erstellung und Durchführung von Ontologietests im Kontext der Health Intelligence Platform

Vorgelegt von:

Jessica Truber

Am: 30. April 2017

zum
Erlangen des akademischen Grades

Master of Science
(M.Sc.)

Erstbetreuer: Prof. Dr.-Ing. Jochen Heinsohn
Zweitbetreuer: Dr. med. Christian Seebode
Gutachter: Dipl.-Inform. Ingo Boersch

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit zum Thema

Entwicklung eines Frameworks zur Erstellung und Durchführung von Ontologietests im Kontext der Health Intelligence Platform

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Berlin, den 30. April 2017

Unterschrift

Danksagung

Ich möchte mich recht herzlich bei meinen Betreuern Herrn Christian Seebode, Herrn Jochen Heinsohn und Herrn Ingo Boersch für die technischen und fachlichen Anregungen und ihre Geduld bei der Realisierung dieser Arbeit bedanken. Auch den Mitarbeitern der Ortec medical GmbH danke ich für ihre Hilfsbereitschaft bei allen Fragen.

Zusammenfassung

Aufgrund des ständigen Wandels der Ressource *Wissen* müssen auch Wissensrepräsentationssysteme wie Ontologien jederzeit flexibel anpassbar sein. Häufige Änderungen im Zuge der Ontologieevolution sorgen allerdings für eine Erhöhung ihres Fehlerpotenzials, wodurch Probleme im Anwendungsbereich auftreten können. Um dies zu vermeiden, rückt das Thema der Qualitätssicherung in den Vordergrund.

Auch für die Ontologie in der Health Intelligence Platform (HIP) wird eine effiziente Möglichkeit benötigt, um an sie gestellte Qualitätskriterien überprüfen zu können. Aus Mangel an passenden öffentlichen Werkzeugen zeigt diese Arbeit die Konzeption und Umsetzung eines Ontologietest-Frameworks, welches an die Anforderungen der HIP angepasst ist und sowohl strukturelle als auch semantische Fehler aufdeckt.

Als Ergebnis der Arbeit ist eine über Maven gebaute Klassenbibliothek entstanden, die in Form eines Java Archives in HIP-Entwicklungsprojekte eingebunden werden kann, um auf der Grundlage des Frameworks Ontologietests zu implementieren. Zukünftig wird eine Weiterentwicklung des Frameworks dafür sorgen, die Testgenerierung und die Nutzung an sich weiter zu vereinfachen.

Abstract

Knowledge as a resource is faced with a process of constant change. For this reason knowledge representation systems like ontologies have to be flexible and provide an ontology evolution process. Frequent ontological changes enhance the potential of errors and can cause problems in the application area. Thus, the importance of quality assurance gains in importance.

The ontology of the Health Intelligence Platform (HIP) requires efficient possibilities for the examination of imposed quality criteria. Due to the lack of suitable tools this thesis shows the conception and implementation of an ontology test framework, which is adapted to the requirements of the HIP and able to detect structural and semantic errors.

The outcome of this thesis is a class library in the Java Archive (JAR) format, which was build with Maven and contains the framework. This JAR can be integrated into HIP development projects in order to provide a basis for the implementation of ontology tests. Future developments on the framework will simplify its usage and the test generations.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ABox	Assertional-Box
API	Application Programming Interface
BDD	Behaviour Driven Development
BL	Beschreibungslogiken
CDR	Clinical Data Repository
DQTP	Data Quality Test Pattern
GUI	Graphical User Interface
HION	Health Intelligence Ontology
HIP	Health Intelligence Platform
HL7	Health Level 7
HRFW	HIP Resource Framework
ICD-10	International Classification Of Diseases
IDE	Integrated Development Environment
IT	Informationstechnik
KI	Künstliche Intelligenz
KIS	Krankenhausinformationssystem
NLP	Natural Language Processing
ODM	Ontologie-Designmuster
OE	Ontology Engineering
OQuaRE	Ontology Quality Evaluation Framework
OPS	Operationen- und Prozedurenschlüssel

OWL	Web Ontology Language
POM	Project Object Model
RDF(S)	Resource Description Framework (Schema)
SPARQL	SPARQL Protocol and RDF Query Language
SQuaRE	Software Quality Evaluation Framework
SWB	Semantic Workbench
TBox	Terminological-Box
TDD	Test Driven Development
UML	Unified Modeling Language
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium

1. Einleitung

Wissen ist ein kulturelles und fachliches Gut, welches unendlich wachsen kann und einem stetigen Wandel unterliegt. Aufgrund dessen ist es schwierig, dieses in einer statischen Form festzuhalten. Eine Wissensrepräsentation muss flexibel sein und sich an Änderungen anpassen können. Ontologie-Entwickler gehen mit dem Thema der Ontologieevolution schon in die richtige Richtung, allerdings gibt es neben einigen Methoden bisher keine verbreitete Umsetzung eines Werkzeuges zur Unterstützung dieser Evolution.
Genauso sieht es im verwandten Bereich der Qualitätssicherung von Ontologien aus. Wenn sich eine Wissensrepräsentation dem ständigen Wandel und der Veränderung öffnet, ist es umso wichtiger, jederzeit eine hohe Qualität sicherstellen zu können. Doch auch dieser Bereich ist weit von Standards entfernt. Zusätzlich wird schon die Definition des Wortes "Qualität" im Zusammenhang mit Ontologien erschwert, da mögliche Qualitätskriterien stark von den jeweiligen Anwendungsfällen einer Ontologie abhängen.

1.1. Motivation

Trotz der Schwierigkeiten bei der Qualitätssicherung von Ontologien überwiegt die Dringlichkeit, eine technische Unterstützung zur Überprüfung bestimmter Qualitätskriterien verwenden zu können. Dabei geht es in erster Linie nicht um eine Qualitätseinschätzung anhand von Skalen, sondern um die Abwesenheit von Fehlern in einer Ontologie.
Die Health Intelligence Platform (HIP) besitzt die Hauptaufgabe, medizinische Fakten aus Daten und Texten zur Weiterverarbeitung zu extrahieren. Dies wird unter anderem durch eine medizinische Ontologie ermöglicht, welche immer auf dem richtigen Entwicklungsstand existieren sollte. Viele und häufige Ontologieänderungen bedeuten einen hohen Testaufwand, um Fehler zu vermeiden. Daher wird ein Werkzeug benötigt, welches das Schreiben von Ontologietests vereinfacht und in verschiedenen Stadien der Ontologieevolution eingesetzt werden kann.

1.2. Aufgabenstellung

Diese Arbeit umfasst die Konzeption und Entwicklung eines Frameworks zur Erstellung und Ausführung von Ontologietests in der HIP. Es werden Qualitätseigenschaften von Ontologien erarbeitet, die in einem zweiten Schritt der Ausarbeitung einer Testsystematik dienen. Die softwaretechnische Umsetzung hat zum Ziel, die schnelle und systematische Erstellung von Testszenarien während der HIP-Entwicklung aber auch zur Laufzeit während des Ontology Engineering Zyklus in der Semantic Workbench (SWB) zu unterstützen. Die Software des Frameworks ist auf den Einsatz in der HIP ausgelegt und wird an deren Softwareumgebung angepasst. Dazu gehört unter anderem die Adaption des sogenannten HIP Resource Frameworks. Der Nutzen der Testsystematik und des Frameworks wird durch eine beispielhafte Umsetzung im Kontext der HIP demonstriert.

Ziel der gesamten Arbeit ist es, eine Systematik und ein Framework zur Sicherung der Qualität von Ontologien zu realisieren, die für die Entwicklung und den Betrieb von wissensbasierten Systemen wie der HIP essentiell sind. Das Framework soll dazu genutzt werden, um die Wissensrepräsentation der HIP kontinuierlich auf Fehler oder mögliche Probleme zu testen. Diese Entwicklung ist stark motiviert durch die Tatsache, dass die Entwicklungswerkzeuge für wissensbasierte Systeme noch nicht die gleiche Qualität und Marktpräsenz haben wie Werkzeuge der Softwareentwicklung.

Mit Hilfe der finalisierten Software werden die Ontologieerstellung im Rahmen neuer Kundenaufträge und der Evolutionszyklus vorhandener Ontologien in der HIP unterstützt und Fehlerquellen stark vermindert. Das Framework liefert somit einen wichtigen Beitrag zur Unterstützung der gesamten Softwarequalität der HIP.

1.3. Aufbau der Arbeit

Der Aufbau dieser schriftlichen Ausarbeitung richtet sich stark nach der Vorgehensweise zur Konzeption und Implementierung des Frameworks. Der Einleitung folgt das zweite Kapitel, welches sich mit Grundlagen zu den Bereichen Wissen und Testbarkeit beschäftigt und die HIP thematisiert. Im dritten Kapitel wird die Wissensverwaltung in der HIP analysiert, um feststellen zu können, wie ihre Qualität definiert und überprüft werden kann. Zusätzlich werden vorhandene Methoden, Werkzeuge und sogar Module aus der HIP selbst auf ihre Eignung hin untersucht, in das Framework integriert zu werden. Es folgt die Konzeption des Test-Frameworks und der Testsystematik, die mit einer Anforderungsanalyse beginnt und das Framework in drei Teilen umschreibt: es gibt die strukturellen Tests, die semantischen Tests und das zusammenführende Rahmenwerk darum. Auf der Grundlage dieser Planung fand die Entwicklung des Frameworks statt, welche im nachfolgenden Kapitel beschrieben ist. Zuletzt wird darauf eingegangen, wie das Framework während der Entwicklung getestet wurde. Die dabei entstandenen Tests zeigen gleichzeitig, wie konkrete Testfälle durch die Nutzung des Frameworks aussehen und wie sie ausgeführt werden. Abschließend wird das Ergebnis dieser Arbeit zusammengefasst und betrachtet, welche Verbesserungen in Zukunft am Ontologietest-Framework vorgenommen werden können oder sollten.

2. Grundlagen

2.1. Wissen und Wissensmanagement

Als Teilgebiet der Informatik beschäftigt sich die Künstliche Intelligenz (KI) mit der Nachbildung von intelligentem Handeln durch Algorithmen beziehungsweise der Formalisierung und Implementierung menschlicher Informationsverarbeitungsprozesse. Seit dem Anfang der 1970er Jahre werden KI-Systeme immer mehr mit anwendungsspezifischem Wissen ausgestattet, da diese Vorgehensweise ein wesentlich höheres Problemlösungspotenzial besitzt als domänenunabhängige Verfahren [Rei91]. Die Grundlage der KI bildet somit das Wissen an sich. Eine einheitliche Definition des Begriffs ist nicht zu finden, jedoch lassen sich die meisten durch folgende Beschreibungen zusammenfassen: die Basis für erfolgreiches oder zweckmäßiges Verhalten einer Person [SS09], die Menge der Kenntnisse eines Menschen [Dud14] oder die Gesamtheit der von einer Person als richtig angenommenen Aussagen, die tatsächlich wahr sind [Rei91].
Um dieses Wissen im großen Umfang nutzen zu können, zum Beispiel in Unternehmen oder in der Forschung, gibt es das sogenannte Wissensmanagement, welches sich auch mit der Sammlung und Speicherung von Informationen befasst. Wissensmanagement sorgt vor allem dafür, dass das richtige Wissen zur richtigen Zeit am richtigen Ort zur Verfügung gestellt wird. Im Laufe dieser Arbeit wird lediglich das elektronische Wissensmanagement thematisiert, welches einige Hürden und Schwierigkeiten mit sich bringt und in den nachfolgenden Kapiteln näher beleuchtet wird.

2.1.1. Wissensrepräsentationen

Die wohl wichtigste und grundlegende Frage des Wissensmanagements ist, wie das Wissen formal und maschinell verarbeitbar dargestellt werden kann. Die Art der Wissensmodellierung beziehungsweise der Wissensrepräsentation ist ausschlaggebend für die anschließenden Nutzungsmöglichkeiten und Problemlösungspotenziale der darin gespeicherten Informationen. Je nach Anwendungsfall und dessen Anforderungen muss somit eine passende Repräsentationsart gewählt werden.

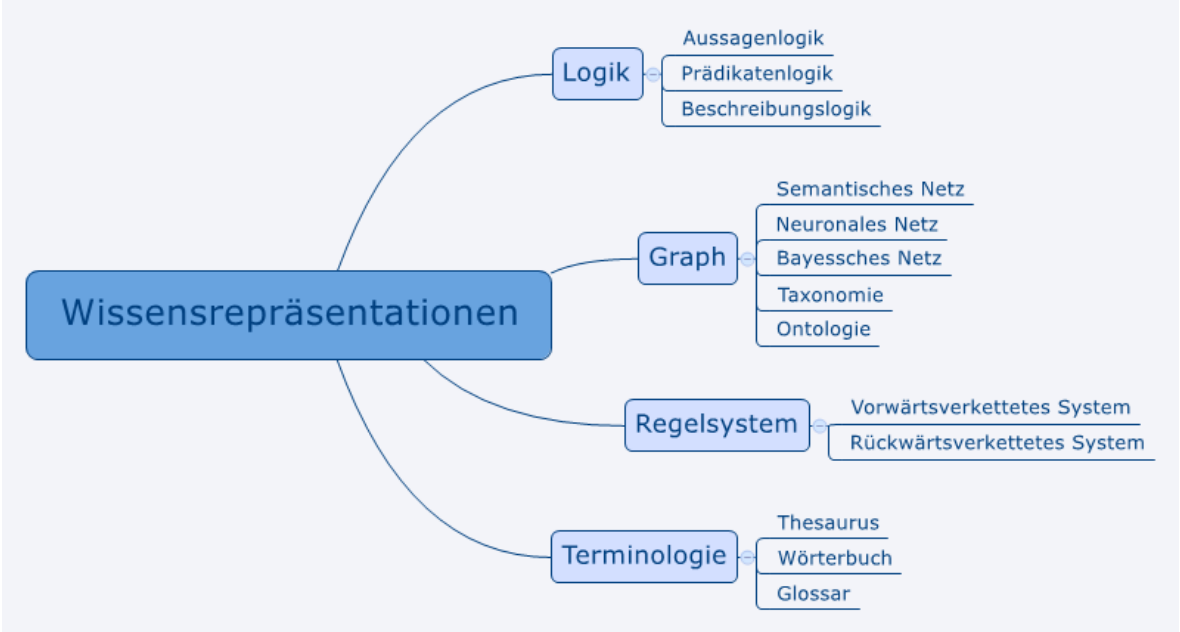

Abbildung 2.1.: Auswahl möglicher Wissensrepräsentationen, Quelle: Eigene Abbildung

In Abbildung 2.1 ist eine Auswahl möglicher Wissensrepräsentationen dargestellt, die über die Zeit entwickelt wurden. Einige der dargestellten Repräsentationen lassen sich nicht klar voneinander abgrenzen oder sind sogar Bestandteil einer anderen. Aufgrund der Bedeutsamkeit für diese Arbeit werden nachfolgend nur die Präsentationsformen der semantischen Netze, Beschreibungslogiken und Ontologien näher betrachtet.

Semantische Netze

Nach [Bei04] ist: "Ein semantisches Netz [...] eine geordnete Zusammenstellung von Begriffen und deren Bezeichnungen, deren Zusammenhang über beliebige Beziehungen miteinander definiert wird. Sowohl Begriffe als auch Beziehungen sind typisierbar und es existiert eine Grammatik für deren Verwendung." Diese Wissensrepräsentation kann sowohl durch menschliche Benutzer, als auch durch IT-Systeme verstanden beziehungsweise verarbeitet werden. Wichtig ist, dass es sich hierbei nicht um eine einfache formale Darstellung handelt. Im Gegensatz dazu besitzt sie die Fähigkeit mit den Eigenschaften der menschlichen Sprache, wie unscharfen Formulierungen, Homonymen, Synonymen und anderen umzugehen. Mit semantischen Netzen können zum Beispiel Abstraktionen oder Konkretisierungen dargestellt werden [Rei10].

Beschreibungslogiken

Die Beschreibungslogiken (BL) gehören zu den Wissensrepräsentationssprachen und sind auf die Beschreibung von Ontologien ausgerichtet. Sie bilden eine Teilmenge der Prädikatenlogik erster Ordnung und sind durch ihre Einschränkungen im Gegensatz zu dieser und anderen Logiken entscheidbar. Somit wird wiederum das Schließen beziehungsweise Inferieren von implizitem Wissen aus einer Wissensbasis ermöglicht [Stu11].
Formal unterteilen BLs das von ihnen umfasste Wissen in eine TBox (Terminological-Box) und eine ABox (Assertional-Box). Während die TBox terminologisches Wissen über Konzepte und ihre Definitionen enthält, wird in der ABox Wissen über Entitäten beziehungsweise Instanzen und deren Beziehungen festgehalten und somit der Zustand der modellierten Welt repräsentiert. Die Ausdruckskraft dieser Repräsentation unterscheidet sich nach Art der Beschreibungslogik, da jede unterschiedliche Konstruktoren zur Verfügung hat, um aus einfachen Konzepten komplexe zu bilden [Kar16].

2.1.2. Ontologien

Die Einsatzbereiche von Ontologien sind vielfältig. Sie reichen von Information Retrieval und intelligenter Informationsintegration über Datenbank Design und E-Commerce zur Künstlichen Intelligenz [Iqb13]. Was sich jedoch dahinter verbirgt, wird in den nächsten Abschnitten erläutert.

Definition von Ontologien

Ontologien können als Repräsentation komplexer Wissensbeziehungen einer Domäne, also eines Wissens- oder Sachgebiets, beschrieben werden. Auch semantische Netze können als semi-formale Ontologien ("lightweight ontologies") bezeichnet werden, die allerdings nicht die Möglichkeiten der Beschreibungslogiken ausnutzen. Im Gegensatz dazu gibt es die "heavyweight ontologies", die über BLs Regeln einschließen und somit logisches Schlussfolgern ermöglichen, was wiederum hoher logischer Konsistenz und Widerspruchsfreiheit bedarf [Bei04].
Die wohl bekannteste Definition von Ontologien stammt von [SBF98]: "An ontology is a formal, explicit specification of a shared conceptualization." Wissen soll also in einer Art und Weise abgebildet werden, die von verschiedensten Software-Applikationen und Personengruppen erfolgreich wiederverwendet werden kann.
Es gibt folgende typische Bestandteile einer Ontologie:

- **Instanzen:** (auch Individuen) Objekte der realen Welt;
- **Relationen:** (auch Rollen oder Properties) Beziehungen zwischen Instanzen;
- **Attribute:** ordnen Instanzen Werte zu;
- **Klassen:** (auch Konzepte oder Begriffe) Mengen von Individuen;
- **Semantische Relationen:** Beziehungen zwischen Klassen;
- **Konzeptdefinitionen:** setzen Konzepte/Klassen in logische Beziehungen, um neue Konzepte zu definieren;
- **Regeln:** können zur Konzeptdefinition als Beziehung zwischen zwei Klassen genutzt werden;
- **Restriktionen:** Beschränkungen für die erlaubten Konstellationen von Individuen, Klassen, Relationen und vor allem Attributen.

Ontologiesprachen

Um Ontologien schlussendlich komplett maschinenverarbeitbar zu machen, haben sich verschiedene Ontologiesprachen entwickelt. Die Bekanntesten und zu einem Standard gereiften werden nachfolgend näher betrachtet.

Das Resource Description Framework (RDF) ist eine vom W3C (World Wide Web Consortium) spezifizierte Syntax, um Metadaten zur Beschreibung von Webinhalten und strukturierten Informationen bereitzustellen und somit den Grundbaustein für das Semantic Web zu bilden. Dieses wiederum ist eine Erweiterung des Internets, die versucht, den Sinn der im Internet gespeicherten Daten zu hinterlegen und diese somit maschinenlesbar zu machen [BP06]. RDF bietet die Möglichkeit, Ressourcen im Internet eindeutig über sogenannte URIs (Uniform Resource Identifier) zu identifizieren, mit Eigenschaften zu beschreiben und Beziehungen zwischen ihnen zu definieren [Kar16]. Mit Hilfe von RDF können allerdings nur Aussagen in der ABox formuliert werden. Zur Erweiterung des Frameworks wurde die Schema-Sprache RDFS (Resource Description Framework Schema) eingeführt, die auch einfache Definitionen der TBox, also eine Art Basisvokabular für eine Domäne, ermöglicht. RDF und RDFS stellen gemeinsam jedoch nur eine einfache Ontologiesprache dar, da ihre Ausdrucksstärke für komplexe Konzeptdefinitionen nicht ausreicht. Um die Beschreibungsmöglichkeiten zu erhöhen, wurde die Web Ontology Language (OWL) entwickelt [Kar16].
OWL und die mittlerweile erweiterte OWL2 erleichtern die Maschinenlesbarkeit für Webinhalte durch die erhöhte Ausdruckskraft seines Vokabulars. Die Basis bildet die Beschreibungslogik SHOIN(D), welche folgende Komponenten beinhaltet:

- Konzeptnamen,
- Rollennamen,
- Konstruktoren:
  - Negation,
  - Konjunktion (UND),
  - Disjunktion (ODER),
  - Restriktionen,
  - transitive Rollen,
  - Rollenhierarchien,
  - Nominale (Aufzählungen),
  - inverse Rollen,
  - Zahlenrestriktionen,
  - Datentypen.

Die Teilsprache OWL DL umfasst die höchstmögliche Ausdruckskraft mit diesen Komponenten, ohne rechnergestützte Vollständigkeit und Entscheidbarkeit aufzugeben. Um das zu erreichen, gibt es allerdings einige RDF-Einschränkungen. Beispielsweise darf eine Klasse nicht als Instanz einer anderen Klasse definiert werden [Bru05].
Neben den Ontologie-Repräsentationssprachen werden auch Abfragesprachen benötigt, die es ermöglichen, auf RDF-Daten zuzugreifen. Das W3C stellte zu diesem Zweck die standardisierte Sprache SPARQL Protocol and RDF Query Language (kurz SPARQL) vor, die der bekannten Structured Query Language (SQL) zur Datenbankabfrage ähnelt [VBH17]. SPARQL besitzt eine festgelegte Syntax, die aus drei Teilen besteht:

- der Definition der Namensräume,
- der Definition des Ausgabeformats,
- der Definition der Anfrage selbst.

2.1.3. Ontology Engineering

Der Begriff des Ontology Engineerings (OE) umfasst Prinzipien, Methoden, Aktivitäten und Werkzeuge zur Erstellung und zum Erhalt von Ontologien [Iqb13].

Lebenszyklus von Ontologien

Die Arbeit mit Ontologien umfasst verschiedene Schritte, von denen die wichtigsten Aufgaben nachfolgend aufgeführt sind [FGJ97]:

- Spezifikation,
- Wissensakquise,
- Konzeptualisierung,
- Implementierung,
- Integration,
- Evaluation,
- Dokumentation,
- Instandhaltung/Evolution.

Die Abarbeitung der Schritte ist dabei kein linearer Prozess, denn jede Aufgabe kann so oft wiederholt werden, wie es nötig ist. Es findet eine ständige Verbesserung statt [Rou11]. Aufgrund dessen kann dies auch als Lebenszyklus von Ontologien angesehen werden.
OE umfasst alle diese Schritte und befasst sich vor allem mit deren technischer Unterstützung in Form von Methoden und Werkzeugen.

Ontologieänderungsmanagement und Ontologieevolution

Durch den mittlerweile großen Einsatz von Ontologien in Informationssystemen wird deren Aktualität immer wichtiger. Veränderungen an Ontologien können nötig werden, wenn sich Wissen ändert oder vermehrt oder wenn sich die Benutzeranforderungen an die Wissensrepräsentation oder das dazugehörige System ändern. Das sogenannte Ontologieänderungsmanagement (oder auch Ontology Change Management) umfasst diese Instandhaltung und Pflege, hält die sich ändernde Ontologie konsistent und ist ein Teil des Ontology Engineerings. Eine Ontologie wird generell als konsistent betrachtet, wenn alle Konzepte, Relationen und Regeln nach der Ontologiesprache korrekt definiert sind und alle Aussagen über Integritäts- und Inferenzregeln in sich selbst und untereinander konsistent und schlüssig sind [Ditt07]. Die Ontologieevolution ist neben der Versionierung, der Integration (Zusammensetzen von Ontologien verschiedener Domänen) und dem Merging (Zusammensetzen von Ontologien der gleichen Domäne) Bestandteil des Ontologieänderungsmanagements.
Eine Automatisierung dieser Aufgaben wird immer wichtiger, um manuelle Interventionen und Änderungen zu vermeiden, die nicht nur zeitintensiv sind, sondern auch ein hohes Fehlerpotenzial mit sich bringen. Trotzdem gibt es bis jetzt kein Komplettsystem, welches die Evolution und das Änderungsmanagement von Ontologien unterstützt.

Ontology Engineering Methoden

Im Laufe der Zeit, in der Ontologien benutzt wurden, entstanden aus den gesammelten Erfahrungen zum OE verschiedene Methoden, bestehend aus einer Reihe an Aktivitäten und Richtlinien. Trotz dessen gibt es bisher keine weit verbreitete oder akzeptierte ganzheitliche Methode zur Unterstützung von OE. Der Hauptgrund dafür ist, dass der jeweilige Nutzen sehr projektspezifisch und dadurch schwer nachvollziehbar ist, was die Wiederverwendung oder Anpassung einer vorhandenen Methode stark einschränkt. Auch hier ist somit die Anwendungsspezifik ein Hindernis für eine Standardisierung. Zusätzlich verändern sich Ontology Engineering Methoden auch mit der Weiterentwicklung anderer Bereiche, wie Ontologiesprachen oder nutzbarer Ontologie-Werkzeuge.

Ontologie-Designmuster

In der Softwareentwicklung ist es mittlerweile weit verbreitet, Entwurfsmuster (Design Pattern) als Lösungsstrategie für wiederkehrende Designprobleme einzusetzen. Die domänenunabhängigen Muster entstehen durch Abstraktionen von Expertenerfahrungen in speziellen Gebieten und können als erwiesene Problemlösungen den Entwicklungsprozess einer Software stark beschleunigen [KA16].
Auch im Wissensmanagement oder genauer im OE wurde dieser Ansatz in den letzten Jahren verfolgt. Die Idee ist es, Ontologie-Designmuster (ODM) einzuführen, welche Lösungen bei wiederkehrenden Ontologie-Entwicklungsproblemen oder Modellierungsherausforderungen darstellen. Auch in diesem Informatikgebiet können Muster somit ein aufstrebender Ansatz für die Wiederverwendung kodierter Erfahrungen und bewährter Methoden sein [Ruy15]. Designmuster können auch als vorgefertigte Ontologiebausteine verstanden werden, aus denen komplexe Ontologien zusammengesetzt werden.
Nach [GP09] lassen sich ODMs in verschiedene Arten unterteilen:

- **Struktur-ODMs** Muster zur allgemeinen Form einer Ontologie oder Muster als Kompositionen logischer Konstrukte;
- **Korrespondenz-ODMs** Muster zur Übertragung von konzeptionellen Modellen in Ontologien oder Muster, um Ontologien miteinander zu verbinden;
- **Inhalts-ODMs** Muster zu Problemen in bestimmten Domänen;
- **Schlussfolgerungs-ODMs** logische Designmuster, um bestimmte Schlussfolgerungen zu erhalten;

- **Präsentations-ODMs** Muster zur Verwendbarkeit und Lesbarkeit von Ontologien aus Benutzersicht;
- **Lexico-Syntaktische-ODMs** Muster, um linguistische Strukturen zu unterstützen.

Die verschiedenen Designmuster finden in unterschiedlichen OE-Phasen statt [Ruy15].
Der erste Ansatz, ODMs öffentlich zu sammeln und zugänglich zu machen, ist die Internetseite "ontologydesignpatterns.org"[1], welche von der Association for Ontology Design & Patterns ins Leben gerufen wurde. Auf dieser Seite finden sich Kataloge zu den meisten Arten von Entwurfsmustern. Die eigentliche Idee besteht darin, dass jeder Nutzer neue Muster vorschlagen kann und diese durch ein Komitee bewertet werden. Bei einer guten Bewertung wird das Muster in den Katalog übernommen. Bisher hat sich dieses Prinzip allerdings noch nicht durchgesetzt, da zwar schon einsehbare Vorschläge eingegangen sind (vor allem an Inhalts-ODMs), jedoch keine Bewertung in den letzten Jahren stattfand. Die Seite scheint aktuell nicht gepflegt zu werden.

[1] http://ontologydesignpatterns.org/wiki/Community:ListPatterns

2.1.4. Qualität

Die Definition des Wortes *Qualität* wird oft als Gesamtheit charakteristischer Eigenschaften verstanden oder mit der Güte eines Objektes gleichgestellt [Dud14]. In dieser Arbeit soll es als Kennzahl für die Übereinstimmung von Ansprüchen mit Leistungen genutzt werden. Die Qualität eines Objektes sagt somit aus, ob es seinen Anforderungen gerecht wird.

Qualität von Wissensrepräsentationen

Ein weit verbreitetes Thema ist die Datenqualität elektronischer Informationen und deren Überprüfung auf Fehler, wie beispielsweise fehlende oder falsche Werte. Viel seltener werden dagegen die "Quellen", aus denen Daten entstehen können, überprüft. Ein Beispiel für diese Quellen, welches von Bedeutung für diese Arbeit ist, sind Ontologien. Es gibt viele frei zugängliche Ontologien im Internet, die wiederverwendet werden können. Im biomedizinischen Bereich wäre hier auf die BioPortal Webseite[2] hinzuweisen, die viele Ontologien in dieser Domäne referenziert. Das Problem dabei ist, dass dieselbe Domäne bei zwei Anwendungsfällen unterschiedlich modelliert werden würde. Dementsprechend muss es Möglichkeiten geben, eine Ontologie auf ihre Qualität und Verwendbarkeit für einen bestimmten Anwendungsfall hin zu untersuchen. Diese Aufgabe übernimmt die Evaluation und Validierung im OE-Prozess, welche als Unterziele die Erstellung einer Ontologie-Rangordnung und die Einschätzung von Qualität oder Korrektheit hat [Ram13].
Es können somit nicht nur für Daten, sondern auch für Wissensrepräsentationen Anforderungen und Kriterien aufgestellt werden, mit dessen Hilfe die Qualität realer Wissensbasen ermittelt werden kann.

Qualitätskriterien für Wissensrepräsentationen

In der Forschung gibt es mittlerweile einige Ansätze, wie die Qualität von Ontologien am besten gemessen werden kann, Standards wurden bisher allerdings noch nicht entwickelt [Ram13]. Im folgenden werden daher einige Ansätze vorgestellt.
Als erstes wird das Ontology Quality Evaluation Framework (OQuaRE)[3] nach [Ram13] näher betrachtet. Es handelt sich dabei um eine Methode zur automatischen Ontologiequalitätsevaluation, die Best Practices aus der Software Qualitätsevaluation, genauer gesagt aus den SQuaRE (Standards für Software Produktqualität), übernimmt und anpasst. Dabei sollen

[2] http://bioportal.bioontology.org/
[3] http://miuras.inf.um.es/oquarewiki/index.php5/Main_Page

Stärken und Schwächen verschiedener Ontologien aufgezeigt werden können. Basierend auf dem SQuaRE-Ansatz benötigt die Methode ein Qualitätsmodell und Qualitätsmetriken. Ein Modell besteht aus einer Menge von Qualitätskriterien, die wiederum jeweils eine Menge von Unterkriterien enthalten. Für jedes Unterkriterium wird eine oder mehrere Metriken angegeben. Die Ausprägung dieses Modells sollte mit allen Kriterien in einer Gemeinschaft als Standard festgelegt werden. Da dies jedoch nicht der Fall ist, gibt es momentan kein festes Modell und die Evaluationskriterien aus SQuaRE wurden übernommen. Dazu gehört: Zuverlässigkeit, Funktionsfähigkeit/Bedienbarkeit, Wartbarkeit, Kompatibilität, Übertragbarkeit, funktionale Angemessenheit und die dazu gehörenden Unterkriterien. Nur strukturelle Kriterien mussten hinzugefügt werden [Ram13]. Eine Aufstellung ausgewählter Evaluationskriterien ist in Tabelle A.1 im Anhang zu finden.
Eine weitere Methode zur Qualitätsanalyse von Ontologien wird in [Tar05] unter dem Namen OntoQA beschrieben. Sie umfasst eine Liste von Metriken (Tabelle 2.1), die in mehrere Gruppen unterteilt ist und anhand derer eine Qualitätseinschätzung vorgenommen werden soll. Das Modell ist darauf ausgelegt, sowohl die TBox einer Ontologie (wird als Schema bezeichnet) als auch die ABox (Wissensbasis genannt) zu bewerten. Darauf sind auch die vorgestellten Metriken ausgerichtet. Geplant war laut [Tar05] ein Online-Werkzeug aus diesem Modell zu entwickeln. Da eine Recherche nichts darüber zum Vorschein bringt, wurde der Plan anscheinend verworfen. Auch wird nicht weiter darauf eingegangen, wie die Überprüfung umgesetzt werden kann.

Gruppe	**Bezeichnung**	**Beschreibung**
Schema Metriken	Beziehungsreichtum	Verhältnis der Anzahl an Relationen zur Anzahl an SubClass-Relationen
Schema Metriken	Attributreichtum	Durchschnittliche Menge an Attributen pro Klasse
Schema Metriken	Vererbungsreichtum	Durchschnittliche Menge an Unterklassen pro Klasse
Instanz Metriken, Wissensbasis Metriken	Klassenreichtum	Verhältnis der Anzahl an Klassen, die in der Wissensbasis genutzt werden, zu der Anzahl an Klassen im Schema
Instanz Metriken, Wissensbasis Metriken	Durchschnittliche Population	Verhältnis der Anzahl an Instanzen in der Wissensbasis zur Anzahl an Klassen im Schema

Gruppe	Bezeichnung	Beschreibung
Instanz Metriken, Wissensbasis Metriken	Kohäsion	Anzahl separater Komponenten, wenn die Wissensbasis als Graph dargestellt wird
Instanz Metriken, Klassen Metriken	Wichtigkeit einer Klasse	Anzahl der Instanzen, die im Teilbaum einer Klasse vorkommen
Instanz Metriken, Klassen Metriken	Vollständigkeit einer Klasse	Verhältnis der Anzahl der Instanzen, die im Teilbaum einer Klasse vorkommen, zur erwarteten Anzahl
Instanz Metriken, Klassen Metriken	Vererbungsreichtum einer Klasse	Durchschnittliche Menge an Unterklassen pro Klasse im Teilbaum einer bestimmten Klasse
Instanz Metriken, Klassen Metriken	Beziehungsreichtum einer Klasse	Verhältnis der Anzahl genutzter Relationen der Instanzen einer Klasse zur Anzahl der im Schema für diese Klasse definierten Relationen
Instanz Metriken, Klassen Metriken	Konnektivität einer Klasse	Anzahl der Instanzen anderer Klassen, die mit Instanzen einer bestimmten Klasse verbunden sind
Instanz Metriken, Klassen Metriken	Lesbarkeit einer Klasse	Summe aller Attribute einer Klasse, die Kommentare oder Labels sind

Tabelle 2.1.: Metriken zur Ontologiequalitätsanalyse nach OntoQA, Quelle: Angelehnt an [Tar05]

Eine große Schwachstelle der beschriebenen Methoden ist, dass die darin enthaltenen Kriterien sehr allgemein gehalten sind und nicht jedes Qualitätsmerkmal für jede Ontologieumsetzung geeignet beziehungsweise sinnvoll ist. So spielt zum Beispiel die Interoperabilität einer Ontologie in der HIP keine große Rolle.

Die vorgestellten Methoden zur Qualitätssicherung von Ontologien umfassen Kriterien, mit deren Hilfe eine Qualitätseinschätzung vorgenommen werden kann. Beschreibungen oder Vorschläge, wie diese Kriterien in der Praxis getestet werden können oder sollen, gibt es jedoch nicht. Auch konkrete Systeme, in denen eine Methode angewendet wird, sind zumindest nicht öffentlich zugänglich. Tests, die im besten Fall sogar automatisch die Qualität einer Ontologie überprüfen, müssen daher selbst entwickelt werden. Eine ausgeprägte Werkzeugunterstützung wie bei der Qualitätssicherung in der Softwareentwicklung ist im Bereich der Ontologien bisher nicht zu finden.

2.2. Health Intelligence Platform

In den folgenden Abschnitten wird das Produkt der HIP näher beleuchtet. Sie bildet die Grundlage und Motivation dieser Arbeit und ermöglicht die praktische Anwendung des Themas der Ontologietests.

2.2.1. Motivation

Die Ortec medical GmbH begann die Entwicklung der heutigen HIP als Projekt mit dem Namen *Berliner Forschungsplattform Gesundheit* in Zusammenarbeit mit der Charité-Universitätsmedizin Berlin und der Vivantes Netzwerk für Gesundheit GmbH. Ursprünglich sollten IT-Lösungen in Form von Plattformsystemen in Krankenhäusern zur Unterstützung von klinischen Studien entstehen, die Ein- und Ausschlusskriterien mit den vorhandenen Patientendaten vergleichen und mögliche Probanden vorschlagen. Die aktuelle HIP bietet mittlerweile eine Sammlung von Anwendungen, welche auf der Verarbeitung und Verwaltung von strukturierten und unstrukturierten Patientendaten basieren.

2.2.2. Funktion

Eine Kernkompetenz der HIP stellt die sprachwissenschaftliche - linguistische - Analyse von klinischen Dokumenten dar, welche in Krankenhausinformationssystemen (KIS) archiviert sind. Die Schriftstücke repräsentieren maßgebliche Quellen für medizinisch-klinische Informationen in Freitextform. Das heißt, dass jeder benötigte Fakt in Handarbeit durch einen medizinischen Mitarbeiter herausgesucht werden müsste. Aufgrund der Zeitintensität dieser Aufgabe werden die Dokumente in aller Regel für die klinische Forschung nur sehr eingeschränkt genutzt. Die linguistische Verarbeitungskette der HIP (im weiteren Linguistische

Pipeline genannt) erschließt die klinischen Textdokumente semantisch und speichert die gewonnenen Fakten in das sogenannte Clinical Data Repository (CDR). Dabei handelt es sich um die zentrale Datenhaltung der HIP, welche die Möglichkeit bietet, alle patientenbezogenen Informationen nach Fällen gruppiert (auch Fallakten genannt) zu speichern. Um die Fakten innerhalb des Textes zu erkennen, wurde eine spezielle medizinisch-semantische Wissensbasis in Form einer Ontologie modelliert, welche implizites Wissen von Ärzten und weiterem medizinischen Personal beinhaltet. Eine nähere Erörterung der Informationsverarbeitung findet sich in Kapitel 3.1.4.
Das CDR bildet die Basis zur Suche nach Ein- und Ausschlusskriterien von Studien, um daraus eine Vorschlagsliste möglicher passender Probanden zu generieren. Diese und darauf aufbauende Funktionen, wie die Verwaltung von Studien, werden dem Benutzer in Form von Webapplikationen zur Verfügung gestellt [Tru16].

2.3. Entwicklungsprozess der HIP

Das spätere Test-Framework soll sich in den Entwicklungsprozess der HIP integrieren lassen. Um mögliche Anforderungen von dieser Seite zu überblicken, beschäftigt sich das folgende Kapitel mit dem Prozess an sich und den darin genutzten Werkzeugen.

2.3.1. Werkzeuge

Die folgenden Werkzeuge werden innerhalb des Entwicklungsprozesses der HIP genutzt und zum Verständnis kurz definiert. Die hier dargestellten Informationen stammen von den gekennzeichneten Produktseiten im Internet.

Confluence

Die kommerzielle Web-Plattform Confluence[4] der Firma Atlassian besitzt die Hauptaufgabe der Verwaltung von Projekten. Sie unterstützt vor allem die Zusammenarbeit im Team, erleichtert Dokumentationen und kann als Wissensplattform und zentrale Informationsquelle benutzt werden. Zusätzlich ermöglicht die Software die Freigabe von Unternehmensinformationen, um Kunden den Self-Service zu erleichtern.

[4] https://de.atlassian.com/software/confluence

JIRA Software

Eine weitere Plattform der Firma Atlassian ist JIRA Software[5]. Dabei handelt es sich um ein Entwicklungswerkzeug für agile Teams. Die Plattform umfasst sowohl die Planung einer Software mit Hilfe verschiedener Methoden (Scrum, Kanban, gemischte, ...) als auch Möglichkeiten zur Nachverfolgung, Ausführung und Erstellung von Berichten. Zur Erweiterung der Funktionalitäten können von den Nutzern viele Add-ons eingebunden werden.
Bei der gleichzeitigen Nutzung von Confluence und JIRA Software findet eine automatische Verknüpfung von JIRA-Vorgängen mit der dazugehörigen Softwaredokumentation im Confluence statt.

Git und GitHub

Git[6] ist eine Open-Source-Software zur schnellen und effizienten Versionsverwaltung in Entwicklungsprojekten. Der Hauptvorteil dieses System ist die Möglichkeit, während der Entwicklung einen sogenannten neuen Branch zu eröffnen, auf dem alle Änderungen vorgenommen werden. Erst wenn alles abgeschlossen und getestet wurde, werden die Neuerungen in den eigentlichen Produktbranch (Masterbranch genannt) überführt.
Das System kann entweder über einen eigenen Server oder einen externen Hosting-Dienst laufen. Einer dieser auf dem Markt verfügbaren Dienste ist GitHub[7]. Mit Hilfe dieser Entwicklungsplattform kann Code leicht gehostet, geprüft und überarbeitet werden.

Maven

Von der Apache Software Foundation stammt die Software Maven[8]. Dabei handelt es sich um ein Werkzeug zum Management Java-basierter Software-Projekte, welches seinen Fokus auf den Build, den Report und die Dokumentation des Projektes legt. Immer wieder anfallende Prozeduren in der Softwareentwicklung sollen automatisiert und vereinfacht werden. Dabei spielt das sogenannte POM (Project Object Model) eine große Rolle, welches die zentrale Projektsteuerungsdatei bildet und Metadaten zum Projekt enthält.

[5] https://de.atlassian.com/software/jira
[6] https://git-scm.com//
[7] https://github.com/
[8] https://maven.apache.org/

JUnit

Bei JUnit[9] handelt es sich um ein einfaches Open-Source-Rahmenwerk zum Programmieren wiederholbarer Tests für Java-Software, welches auf der sogenannten xUnit Architektur für Unittests basiert. Das Framework umfasst unter anderem Erweiterungen für das Ausführen von Tests und das Überprüfen eines erwarteten Ergebnisses.

JBehave

Ein weiteres Open-Source-Test-Framework ist JBehave[10]. Es handelt sich dabei um ein Werkzeug, welches das sogenannte Behaviour Driven Development (BDD) unterstützt, eine Weiterentwicklung des Test Driven Developments (TDD) und des acceptance-test driven Designs. Beim BDD geht es darum, eine intuitive Nutzung und Erstellung von Tests voranzutreiben und das gemeinsame Verständnis zwischen Entwickler und Kunde durch leichte Lesbarkeit zu gewährleisten. Benutzer-Stories werden in JBehave daher textuell als Szenarien in einer Gegeben-Wenn-Dann-Form definiert. Anschließend werden diese Schritte auf Java-Methoden abgebildet, über Maven ausgeführt und ausgewertet. Innerhalb einer integrierten Entwicklungsumgebung (IDE) können die Tests auch mittels JUnit ausgeführt werden.
Das Test-Framework kann über das Add-on: JBehave for JIRA[11] als Werkzeug zum Schreiben agiler Benutzer-Stories auf der Plattform JIRA Software genutzt werden. JIRA ermöglicht die direkte Definition der Stories und stellt Story Run Reports zur Verfügung.

[9] http://junit.org/junit4/
[10] http://jbehave.org/
[11] https://marketplace.atlassian.com/plugins/com.jbehaveforjira.plugin/server/overview

2.3.2. Ablauf des Entwicklungsprozesses in der HIP

Der Entwicklungsprozess in der HIP beginnt mit einer Benutzeranforderung, einem gemeldeten Fehler oder einer Änderungsidee des Entwicklers. Die betreffenden Anforderungen werden als Erstes tabellarisch im Confluence dokumentiert. Anschließend wird auf der JIRA Plattform einer neue Story erzeugt, mit der Confluence-Dokumentation verknüpft und nach den Regeln des Scrums einem Sprint zugeordnet. Beim Scrum handelt es sich um das in der HIP benutzte agile Entwicklungskonzept, bei dem Aufgaben in zeitlich begrenzten Iterationen (Sprints) bearbeitet und am Ende zusammen mit den Stakeholdern geprüft werden [Rub14]. In der JIRA-Story wird, nach dem Vorbild des BDD und unter Zuhilfenahme des JBehave for JIRA Add-ons, eine JBehave-Story erstellt, die eine oder mehrere Szenarien enthält und somit die Akzeptanz- beziehungsweise Abnahmetests beschreibt. Diese werden anschließend in ihren Java-Methoden implementiert. Es folgt die eigentliche Implementierung der in den Szenarien beschriebenen Funktionen, nachdem dafür ein neuer Branch in Git über GitHub angelegt wurde. Mit der ersten Übertragung von Code ins Git wird die Komponente über Maven gebaut und JUnit-Tests sowie JBehave-Tests ausgeführt. Sind diese erfolgreich, erfolgt das Zusammenlegen alter und neuer Softwarekomponenten im sogenannten Integrationsbranch, gefolgt von einem weiteren Build und Tests. Mit dem Abschluss der Implementierung der Story werden die Änderungen endgültig übernommen und der spezifische Branch gelöscht [Gri16].

Auch die Entwicklung innerhalb der Wissensverwaltung der HIP soll an diesen Prozess angelehnt werden. Somit ist es auch für die Konzeption des Test-Frameworks wichtig, die einzelnen Schritte zu kennen. In Zukunft soll das Framework dort ansetzen, wo die Definition der JBehave-Tests aufhört und zur anschließenden Implementierung der Szenarien eingesetzt werden. Dementsprechend sollte es möglich sein, aus den textuellen JBehave-Anforderungen mit Hilfe des Frameworks funktionierende Tests zu formulieren.

2.4. Testbarkeit

Jede Software und Softwarekomponente sollte sowohl während der Entwicklung als auch bei der Nutzung darauf überprüft beziehungsweise getestet werden, ob sie den gestellten Anforderungen gerecht wird. Dafür werden Tests benötigt, die diese Überprüfung übernehmen und wiederholt unabhängig voneinander ausgeführt werden können.

2.4.1. Softwaretests

Softwaretests können als kleine Programme angesehen werden, die ohne Benutzerkontrolle automatisch über Code oder deren Bestandteile laufen und zeigen, dass dieser ein bestimmtes gewünschtes Verhalten aufweist. Ein bestimmter Testfall zeigt also, ob ein Programm die Vorgaben aus der vorher entstandenen Spezifikation erfüllt, allerdings kann dadurch kein Test die vollkommene Abwesenheit von Fehlern zeigen. Aufgrund des hohen Zeitaufwandes bei der Erstellung von Softwaretests sind sie nicht sehr beliebt, obwohl sie einige Vorteile haben. Da Software oft permanenter Änderung oder Verbesserung unterliegt, auch Refactoring genannt, können Tests nach jeder Änderung ausgeführt werden, um eventuell eingebaute Fehler zu erkennen. Zusätzlich führt das Schreiben von testbarer Software zwangsläufig zu besserem Design. Um die Entwicklung der Tests einfacher, schneller und strukturierter zu gestalten, können und sollten Test-Bibliotheken genutzt werden. Damit sehen alle Testfälle gleich aus, es müssen immer die gleichen Schritte ausgeführt werden und die Testtreiber erleichtern sogar die Ausführung und Auswertung der vorhandenen Tests [Ull14].

2.4.2. Ontologietests

Ontologietests können durchaus mit Softwaretests verglichen werden. Der Unterschied besteht nur darin, dass überprüft wird, ob eine Ontologie (als Teil einer Software) definierte Anforderungen erfüllt. Als Schwierigkeiten stellen sich die Identifizierung dieser Anforderungen und das Fehlen von Test-Bibliotheken dar.

2.4.3. Test-Frameworks

Test-Frameworks, oder auch Modultestsoftware genannt, können zur Erstellung und Ausführung von Tests genutzt werden, da sie Infrastrukturen bilden, die die gesamte Testumgebung bereitstellen. Mittlerweile gibt es für fast jede verbreitete Programmiersprache ein Framework. Gemeinsamkeiten sind jedoch in den Komponenten zu finden. Allgemein besteht eine Modultestsoftware aus folgenden Elementen [Sch10]:

Software Testobjekt

Test Suite Sammlung von konkreten Testfällen

Überprüfungsmechanismus Prüft, ob tatsächliche Ergebnisse mit Erwartungswerten übereinstimmen

Test Runner Sorgt für nötige Vorbedingungen und führt Tests aus

Test Reporter Auswertung erfolgreicher und fehlgeschlagener Tests.

Bei der Erstellung eines neuen Test-Frameworks sollten diese Punkte beachtet und umgesetzt werden.

3. Stand der Forschung und Technik

Dieses Kapitel beschäftigt sich mit dem aktuellen Stand der Wissensverwaltung in der HIP, um das Test-Framework optimal an die internen Anforderungen anpassen zu können. Zusätzlich wird nachfolgend ein Überblick über bestehende Werkzeuge und Methoden zur Qualitätsüberprüfung von Wissensrepräsentationen geschaffen und eingeschätzt, ob und in welchem Umfang bestehende Ansätze in die strukturellen Tests des Frameworks eingegliedert werden können. Außerdem werden die sogenannten Pipelinetests näher betrachtet, um zu entscheiden, ob es möglich ist, Module für die semantischen Tests zu nutzen.

3.1. Wissensverwaltung in der HIP

Dieses Kapitel umfasst alles wissenswerte darüber, wie Wissen in der HIP gespeichert, bearbeitet und verarbeitet wird und setzt sich mit den Schwierigkeiten auseinander, denen sich die Entwickler im Rahmen der Wissensverwaltung stellen müssen. Zusätzlich wird die Frage nach der Qualität der HIP-Wissensrepräsentation thematisiert.

3.1.1. Wissensrepräsentation in der HIP

Die Wissensmodellierung im Zuge der HIP wurde von Beginn an anwendungsorientiert ausgerichtet. Es war schnell klar, dass eine theoretische Definition von Wissensrepräsentationen in der Praxis schwierig ist, da diese sich immer an den jeweiligen Anwendungskontext und Nutzen anpassen sollten. Mit dem Ziel die Art und Weise ihrer Problemlösung festzuhalten und somit auch Anforderungen an die benötigte Wissensbasis zu erhalten, wurden zahlreiche Interviews mit medizinischem Personal und Beobachtungen dieser Experten durchgeführt. Die Erkenntnis, dass vor allem nach charakteristischen Begriffen von Krankheitsbildern, häufig in Verbindung mit Attributen, gesucht wird, hat die Umsetzung der Wissensrepräsentation stark beeinflusst [Tru16].

Es wurde entschieden, eine Ontologie zu entwickeln, deren Konzepte mit linguistischen Informationen (Wörtern) annotiert sind, um abstrakte Konzepte des Wirklichkeitsbereiches mit ihren logischen Zusammenhänge darzustellen und mit linguistischen Ausdrücken zu verknüpfen.
Zu Beginn des Projektes wurden zuerst bestehende medizinische Ontologien und Terminologien auf ihre Verwendbarkeit im Rahmen der HIP untersucht, um Arbeit zu sparen. Dazu gehörten unter anderem UMLS (Unified Medical Language System), Mesh (Medical Subject Headings), SNOMED (Systematized Nomenclature of Medicine), ICD-10 (International Classification of Diseases), OPS (Operationen- und Prozedurenschlüssel) und OpenGalen. Allerdings konnte bei jeder Alternative mindestens ein Punkt identifiziert werden, der dem Einsatz entgegenstand. Gründe für den Ausschluss waren Lizenzbedingungen, Intransparenz, fehlende Eindeutigkeit durch Doppelklassifikationen, zu hohe Komplexität oder ein unzureichender Detaillierungsgrad [Gei12].
Schlussendlich wurde auf Grundlage der Interviews und der Auswertung medizinischer Studien und den dazugehörigen Ein- und Ausschlusskriterien eine eigene Ontologie erstellt, die direkt an ihren Anwendungsfall angepasst ist und im weiteren Verlauf HION (Health Intelligence Ontology) genannt wird. Sie dient zur Identifikation von Begriffen in Texten wie Arztbriefen und radiologischen Befunden, so dass diese korrekt als Diagnose, Befund, Symptom, Medikament, Schweregrad, Lokalisation und so weiter erkannt werden können. Zum anderen ermöglicht die Repräsentation der inhaltlichen Zusammenhänge zwischen Begriffen eine semantische Suche nach Synonymen sowie nach Formulierungen, die Konkretisierungen eines allgemein eingegeben Studienkriteriums darstellen.

Modellierungssprache

Zur Umsetzung der HION wurde aufgrund der leichten Beherrschbarkeit und Verfügbarkeit von Triplestores zuerst RDFS gewählt. Triplestores oder auch RDF-Stores sind Datenbanken zur Speicherung von semantischen Aussagen in Form von Tripeln (Subjekt, Prädikat, Objekt). Im Hinblick auf die eingeschränkte Ausdruckskraft von RDFS und dass so nur das taxonomische Grundgerüst einer Ontologie repräsentierbar ist, wurde eine Erweiterung mit OWL eingeführt. Bisher wird diese Ontologiesprache jedoch nicht ausgenutzt, sondern lediglich einige Sprachkonstrukte benutzt, die größtenteils Typisierungen umfassen. Zum Beispiel ist jedes Konzept als owl:Class typisiert. Inhaltliche Zusammenhänge zwischen Konzepten werden über Relationen/Properties definiert, wobei diese entweder aus dem RDF(S)-Vokabular stammen oder selbstdefinierte Annotation-Properties aus OWL sind [Tru16].
Im Rahmen der HION wird außerdem die sogenannte Metamodellierung eingesetzt. Dabei werden Klassen beziehungsweise Konzepte als Instanzen betrachtet und Properties werden für Paare von Klassen und nicht für Paare von Instanzen formuliert. Dies wird dadurch ermöglicht, dass RDFS Klassen selbst als Instanzen der Meta-Klasse rdfs:Class angesehen und formuliert werden. Auch der Einsatz von OWL ändert an diesem Phänomen in der HION nichts.
Der folgende Codeausschnitt aus der Orthopädie-Ontologie zeigt beispielhaft, wie einzelne Konzepte definiert werden und wie die Ontologiesprachen eingesetzt werden:

Ausschnitt aus der Orthopädie-Ontologie:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

@prefix : <http://ontology.de/dd-orthopaedics#> .
@prefix c: <http://ontology.de/core#>.
@prefix obv: <http://ontology.de/observationvalues#> .


:Hueftpfannenrekonstruktion rdf:type owl:Class ;
              rdfs:subClassOf c:Beobachtung ;
              c:standard_label "Hueftpfannenrekonstruktion" .

:HWSSyndrom rdf:type owl:Class ;
              rdfs:subClassOf c:Beobachtung ;
              c:standard_label "HWS-Syndrom" ;
              c:synonym "Halswirbelsaulensyndrom" ,
                        "Zervikalsyndrom"  .

:Klumpfuss rdf:type owl:Class ;
          rdfs:subClassOf c:Beobachtung ;
          c:standard_label "Klumpfuss" ;
          c:hatBeobachtungsAttribut obv:Sinister ,
                                    obv:Dexter ,
                                    obv:Bilateral .
```

Aufbau

Die HION besteht außerhalb der Laufzeit aus mehreren Dateien in der Turtle-Syntax (einem bestimmten Serialisierungsformat von RDF), die aufeinander aufbauende Teilontologien enthalten und während der Laufzeit in einen gemeinsamen Triplestore geladen werden. Durch diesen Aufbau wird die Modularität, Wartbarkeit und Übersichtlichkeit der HION gewahrt. Die einzelnen Dokumente stellen allgemeine medizinische Begriffe, verschiedene Fachbereiche, Labor- und Medikamentenbegriffe und allgemeine Konzepte zur Strukturierung dar. Über Importe nach OWL und die Benutzung von Namensräumen werden Verbindungen zwischen den Teilontologien aufgebaut. Dabei gilt: eine Ontologie A importiert eine Ontologie B, wenn sie diese erweitert. Ein Namensraum ist als Präfix mehrerer URIs zu verstehen und bildet somit eine Art Vokabular. Im obenstehenden Codeausschnitt ist dies gut zu erkennen. Dateien, denen der gleiche Namensraum zugeordnet ist, gehören zu einer Teilontologie [Tru16]. Das Konzept der einzelnen Ontologien soll zusätzlich den Gedanken unterstützen, die HION für jeden einzelnen Kunden individuell nach seinen Anwendungswünschen zusammenstellen zu können. In Zukunft soll es möglich sein, verschiedene Teilontologien zu einer großen zusammenzustecken, die die geforderten Fachbereiche abdeckt, jedoch nicht überdimensioniert ist.

Die HION, als Zusammenschluss der Teilontologien, ist einfach strukturiert und besitzt eine feste Basishierarchie, die in Abbildung 3.1 über die ersten zwei Hierarchiestufen dargestellt ist. Es handelt sich dabei um einen Baum, der die Wurzelkonzepte *BaseConcept, CodeConcept, DiagnosisType, Fall, Fallattribut, DocumentTerm* und *ZeitlichesKonzept* beinhaltet. Zusätzlich ist gut zu erkennen, dass in der HION die Mehrfachvererbung genutzt wird. So hat beispielsweise das Konzept *Beobachtung* die Überklassen *BaseConcept, Fallattribut* und *ZeitlichesKonzept*. Die abgebildete Basishierarchie bildet die Grundlage für jede Anwendung der HION und darf daher nicht verändert werden.

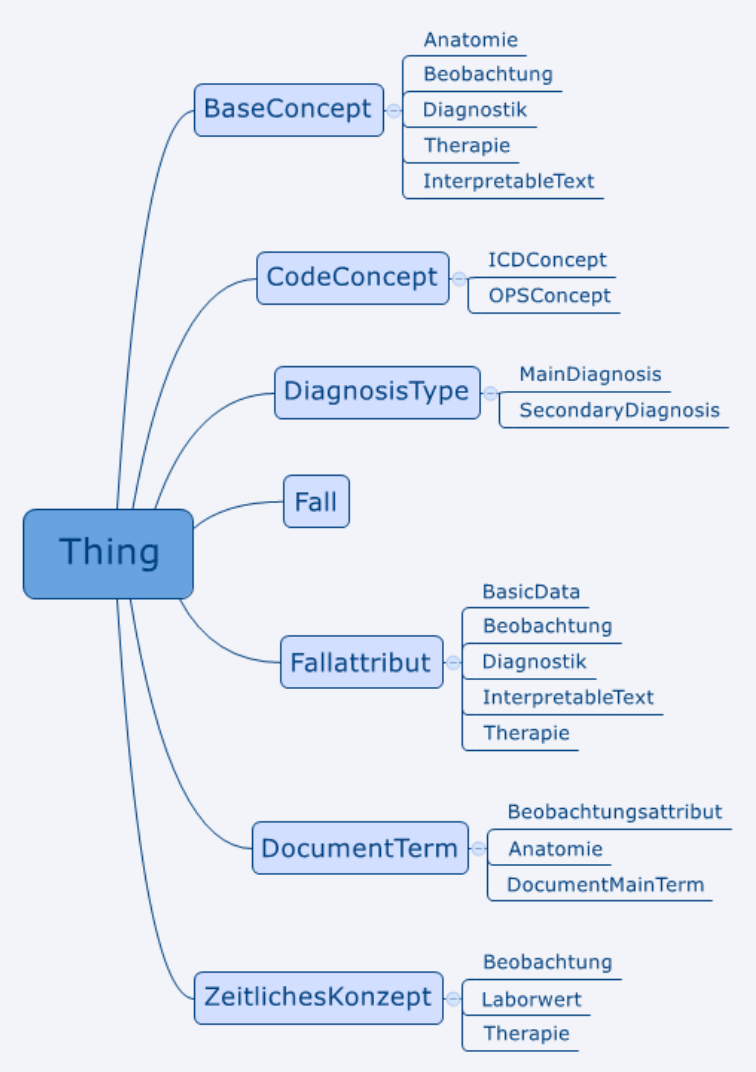

Abbildung 3.1.: Basishierarchie der HION, Quelle: Eigene Abbildung

Unter der Oberklasse *CodeConcept* werden ICD-10 und OPS Klassifikationen umgesetzt, die in den Krankenhausinformationssystemen gefunden werden. In Verbindung mit *DiagnosisType* kann zusätzlich über jeden gefundenen Fakt eine Aussage darüber gemacht werden, ob es sich um eine Hauptdiagnose eines Patienten handelt oder nicht. Auch die Oberklassen *Fall* und *Fallattribut* erfüllen ihre Funktion erst bei der Faktengenerierung, auf die in Kapitel 3.1.4 näher eingegangen wird. *ZeitlichesKonzept* wird wiederum genutzt, um einem bestimmten Zeitpunkt ein oder mehrere Konzepte zuzuordnen.
Alle medizinischen Konzepte und deren Zusammenhänge werden als Unterklasse des Wurzelkonzepts *BaseConcept* definiert. Alle Attribute, die den Beobachtungen, Therapien und so

weiter zugeordnet werden können, werden unter der Klasse *BeobachtungsAttribut* unter *DocumentTerm* zusammengefasst. Mit anderen Worten wird jede fachliche Veränderung an der HION nur in diesen Bereichen vorgenommen und jedes neue Konzept muss einer Unterklasse von *DocumentTerm* oder *BaseConcept* angehören. Letztere werden fortlaufend als Basiskonzepte bezeichnet. Die Tabelle 3.1 gibt eine genauere Übersicht über diese Basiskonzepte und welche Art von fachlichem Wissen welchem Konzept zugeordnet werden soll.

Klasse	**Beschreibung**
Anatomie	Klasse für Körperteile, Organe, Gewebe und Zellen. Die anatomischen Einheiten dienen u.a. für die Angabe von Lokalisationen.
Beobachtung	Klasse für alle Krankheitsbilder, Diagnosen, Befunde, Symptome und nicht krankheitswertige Beobachtungen am Patienten.
Diagnostik	Klasse für durchgeführte diagnostische Prozeduren und Tests.
Therapie	Klasse für therapeutische Prozeduren aller Art, insbesondere der Gabe von Medikamenten.
InterpretableText	Klasse, die die analysierten Texte enthält (keine semantische Bedeutung)

Tabelle 3.1.: Übersicht über die Basiskonzepte (BaseConcept) der HIP-Modellierung, Quelle: Eigene Darstellung

Inhaltliche beziehungsweise semantische Zusammenhänge zwischen verschiedenen Konzepten werden über Relationen abgebildet, die für die Benutzung in der HION definiert sind. Die Tabelle 3.2 zeigt eine Übersicht über alle benutzerdefinierten Relationen, die in der HION genutzt werden, um medizinische Zusammenhänge darzustellen und wichtige Angaben über Konzepte vorzunehmen. Diese werden im Weiteren semantische Relationen genannt. Es gibt noch einige andere Properties, die allerdings nur zur Funktionalität beitragen und somit in keinem Fall bearbeitet werden dürfen. Eine Überprüfung derer ist daher zumindest im Rahmen des hier bearbeiteten Frameworks nicht notwendig. Ausnahmen sind dabei die Relationen rdfs:subClassOf, :synonym und :standard_label, die genutzt und teilweise benötigt werden, um ein Konzept korrekt zu definieren. Deshalb können auch Nutzer auf sie zugreifen, womit die Notwendigkeit von Tests indiziert wird. Zusätzlich gibt es noch Relationen, die nur innerhalb der ABox, also im CDR, genutzt werden. Auch diese sind für das Framework nicht von Bedeutung.

Klasse	Beschreibung
hatBeobachtung	Relation zwischen einem Fall und einer zutreffenden Beobachtung.
hatBeobachtungs-Attribut	Relation zwischen einer Beobachtung und einem dazugehörigen Attribut.
hatEinheit	Relation zwischen einem Konzept und einem String, der eine Einheit repräsentiert.
hatLokalisation	Relation zwischen einer Beobachtung und einer anatomischen Lokalisation.
hatRelevanten-Begriff	Relation zwischen zwei Beobachtungen, die oft zusammen auftreten.
hatTherapie	Relation zwischen einem Fall und einer zutreffenden Therapie.
hatTherapie-Attribut	Relation zwischen einer Therapie und einem dazugehörigen Attribut.
hatWert	Relation zwischen einem Konzept und einem konkreten Wert.
hatZeitpunkt	Relation zwischen einem ZeitlichenKonzept und einem konkreten Zeitwert.
hatRisikofaktor	Relation zwischen einer Beobachtung und einer als Risiko angesehenen Beobachtung.
hatTeil	Relation zwischen zwei Anatomiekonzepten.

Tabelle 3.2.: Übersicht über die semantischen Relationen der Modellierung, Quelle: Eigene Darstellung

Wissen im Clinical Data Repository

Zur Wissensrepräsentation innerhalb der HIP gehört auch das CDR. Das darin gespeicherte Wissen bildet eine zweite, auf der HION basierende, Ontologie und kann als ABox angesehen werden. Dort werden somit die Instanzen und Fakten der modellierten Welt, hier Diagnosen, Symptome und so weiter eines Falles, festgehalten. Das CDR enthält ausschließlich über das Concept-Mapping generierte Fakten und wird daher im Weiteren nicht referenziert, wenn Aussagen über die HION getroffen werden. Diese umfasst grundsätzlich nur die TBox. Wie das Concept-Mapping funktioniert, wird in Kapitel 3.1.4 näher erklärt. Innerhalb der HIP können Anwendungen über SPARQL-Abfragen auf das CDR zugreifen und Informationen abfragen.

Das im CDR gespeicherte Wissen basiert auf der gleichen Modellierungssprache wie die HION und ist innerhalb des Repositories als eine Ansammlung von Triplen in der Subjekt-Prädikat-Objekt-Form repräsentiert. Pro Patient wird eine Instanz des HION-Konzeptes *Fall* angelegt und mit allen gefundenen Informationen verknüpft. Werden dort falsche Fakten abgelegt, deutet dies auf einen semantischen Fehler hin, der seinen Grund im Concept-Mapping, also in der Zusammenarbeit der HION und der Linguistischen Pipeline, findet. Ontologietests sollten somit dazu in der Lage sein, zu überprüfen, ob auf der Basis einer Datenquelle die richtigen Fakten entstehen.

HIP Resource Framework

Das HIP Resource Framework (HRFW) verfolgt die Absicht, einen frameworkunabhängigen Zugriff auf ein RDF Model zu ermöglichen. Es vereinheitlicht Eigenschaften bekannter RDF-Frameworks wie Apache JENA[1], Apache Commons RDF[2] und Eclipse RDF4J[3], zu denen aktuell Adaptionen entwickelt werden. Komponenten dieser Basis-Frameworks wie zum Beispiel Reasoner, Serialisierer und Triple Stores sollen leicht in die HIP eingebunden werden können. Die Serialisierungen der RDF Modelle und ihre Syntax sind vom W3C standardisiert, allerdings unterscheiden sich die APIs (Application Programming Interface) und In-Memory Modelle von Framework zu Framework, obwohl sie das gleiche Ziel verfolgen. Das HRFW verfolgt somit das Ziel, den Zugriff auf diese Modelle zu vereinfachen und dafür eine eigene interne Repräsentation für die HIP-Fakten bereitzustellen, die an RDF angelehnt ist. Sie trägt den Namen HIP Model.
Das HRFW ist momentan in Entwicklung und kann erst zu einem späteren Zeitpunkt mit den HIP Ontologietests und dem hier vorgestellten Test-Framework integriert werden. Trotzdem spielt das HIP Model in der Konzeption des Frameworks und der dortigen Repräsentation von Fakten eine entscheidende Rolle.

[1] https://jena.apache.org/
[2] https://commons.apache.org/proper/commons-rdf/
[3] http://rdf4j.org/about/

3.1.2. Ontology Engineering in der HIP

Das folgende Kapitel beschreibt den Prozess des OEs im Umfeld der HIP. Es geht dabei vor allem um die einzelnen Arbeitsschritte und die Identifikation der verschiedenen Einsatzbereiche des Test-Frameworks.

Ontology Engineering Zyklus in der HIP

In der Abbildung 3.2 ist der Wissensmanagementzyklus der HIP dargestellt. Als erstes werden alle Rohdaten und die benötigten Ontologien in Form der HION in das System geladen, damit daraus über Data Mining Prozesse und die Linguistische Pipeline Informationen extrahiert werden können. Anschließend ist es als Nutzer oder Entwickler möglich, fehlende oder falsche Konzepte oder Relationen in den Datenquellen zu erkennen und temporär zur Wissensbasis hinzuzufügen oder Änderungen gemäß des gefundenen Fehlers vorzunehmen. Jede Erneuerung der Ontologien wird im vierten Schritt auf die vorhandenen Daten angewendet, um ihre Effekte analysieren zu können. Zusätzlich sollten an diesem Punkt des Prozesses Ontologietests zum Einsatz kommen, um sicherzustellen, dass alle strukturellen und semantischen Anforderungen eingehalten werden und um den ausführenden Nutzer in seiner Arbeit zu unterstützen. Für den Fall, dass kein Test fehlschlägt, werden alle Änderungen endgültig an die HION übergeben. Anschließend kann nach einer erneuten Ausführung der Informationsextraktionsprozesse auf den Rohdaten der Effekt auf die Faktenbasis analysiert werden. Der letzte Schritt umfasst die Freigabe des neuen Domänenmodells, nachdem strukturelle und semantische Inkonsistenzen ausgeschlossen wurden. Der Zyklus startet danach wieder von vorn [Tru15b].

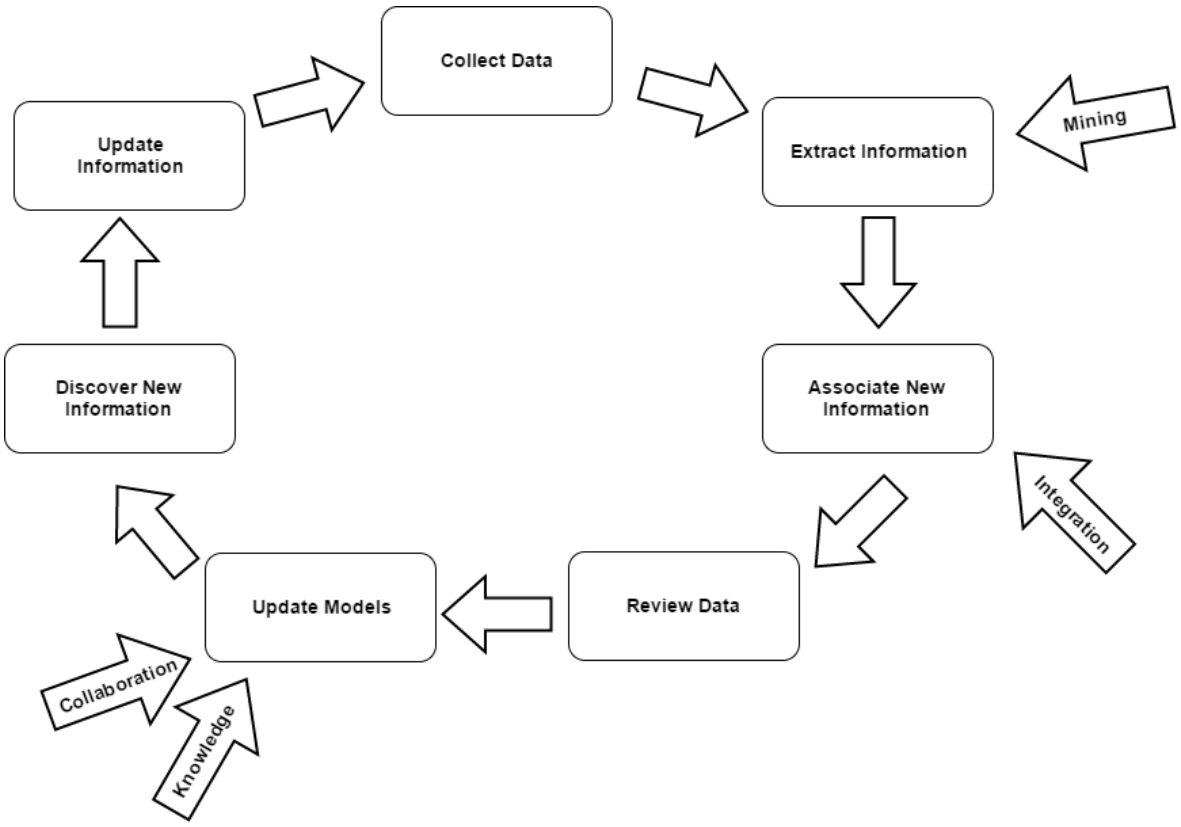

Abbildung 3.2.: Ontology Engineering Prozess in der HIP, Quelle: Angelehnt an: [Tru15b]

Semantic Workbench

Zur Unterstützung des dargestellten Wissensmanagementprozesses ist die Entwicklung eines Werkzeuges geplant, welches den Namen Semantic Workbench (SWB) trägt. Es soll eine umfassende Verwaltung semantischer Wissensresourcen erleichtern und die Ontologiemodellierung durch Ontologie- und Domänenexperten ermöglichen. Die Erstellung, Pflege und Verwaltung von medizinischfachlichem aber auch linguistisch-terminologischem Wissen steht dabei im Vordergrund. Der Nutzer soll mit Hilfe der SWB Ontologien im Kontext eines konkreten Textbezuges erweitern und ändern können und somit die Spezialisierung für den Anwendungsfall der Texterkennung weiter vorantreiben [Tru15a].

Die SWB wird in erster Linie zur Unterstützung der Ontologieevolution in der HIP benötigt. Die HION oder ein definierter Teil aller vorhandenen Ontologien wird manipuliert und im Anschluss daran werden die Effekte der Manipulation angezeigt. Da es mit wachsender Komplexität der Ontologien immer schwieriger wird, zu erkennen, ob die durchgeführte Änderung den gewünschten Effekt aber keine unerwünschte Auswirkungen hatte, sind auch an dieser Stelle Tests notwendig. Damit ergibt sich ein weiterer Anwendungsfall für das Test-Framework.
Der Begriff *Änderung* an einer Ontologie umfasst im weiteren Verlauf der Arbeit jede mögliche Manipulation. Dazu gehört unter anderem das Hinzufügen, Löschen oder Ändern von Konzepten oder Relationen, genauso wie die Neuordnung der Hierarchie.

3.1.3. Ontologieevolution in der HIP

Der aktuelle Workflow in der Erstellung und Bearbeitung der HION für eine Instanz der HIP bei einem bestimmten Kunden beruht auf manueller Arbeit, die Zeit und Ressourcen kostet. Gibt es einen neuen Kunden, wird auf der Grundlage von Datenbankausschnitten und/oder Beispieldatensätzen händisch eine HION zusammengestellt. Dies geschieht durch das Zusammenschließen der allgemeinen Ontologien mit mehreren einzelnen Fachbereichsontologien und eventuellen Ergänzungen. Mit vorhandenen Kunden gibt es eine stete Kommunikation darüber, ob ihnen Fehler in der Text- beziehungsweise Faktenerkennung aufgefallen sind. Da die Kunden selbst keine Möglichkeit haben, die HION zu bearbeiten, müssen sie warten, bis die Entwickler das Problem analysiert und behoben haben und eine neue Version der HIP zur Verfügung stellen. Zusätzlich ist Wissen nie statisch und unterliegt einem ständigen Wandel, wodurch ebenfalls immer wieder Aktualisierungen oder Erweiterungen der Ontologien notwendig werden. Hinzu kommt, dass alle diese Arbeiten größtenteils über Text-Editoren erledigt werden, da auf dem Markt verfügbare Werkzeuge mehrere Probleme aufweisen.

Der bekannte Ontologie-Editor Protégé[4] hat zum Beispiel Schwierigkeiten bei der Darstellung von Metarelationen und deren Vererbung und unternimmt eine Umstrukturierung der Datei, die über die Texteditoren allerdings in bestimmter Weise geordnet wurden. Zusätzlich ist es Editoren oft nicht möglich, mehrere Ontologie-Dateien zusammen anzuzeigen, was die Übersicht verbessern und die Bearbeitung erleichtern würde. Allerdings sind auch Text-Editoren keine ideale Lösung, da die Bearbeitung der einzelnen Dateien sehr umständlich ist und Syntaxfehler weder überprüft noch erkannt werden können. Auch eine graphische Darstellung der Konzepte und Hierarchien wären in vielen Situationen hilfreich.
Der beschriebene Workflow für die momentane Ontologieevolution soll in Zukunft anders geregelt werden. Es werden zwei verschiedene Wege eingeführt, wie die Evolution von Seiten der Kunden und der Entwickler vereinfacht wird.

Evolution im Entwicklungsprozess

Als Erstes muss es natürlich weiterhin den Entwicklern möglich sein, die einzelnen Ontologien wie auch eine bei Kunden genutzte HION zu bearbeiten. Zukünftig soll auch die Bearbeitung im Entwicklungsprozess über die SWB ausgeführt werden können. Die einzelnen Schritte sind im Prozessmodell in Abbildung A.1 dargestellt, die sich im Anhang befindet.
Der Evolutionsprozess beginnt mit einer Änderungsidee des Entwicklers, welche in manchen Fällen durch einen Änderungswunsch oder eine Fehlermeldung des Kunden initialisiert worden ist. Anschließend werden die in Kapitel 2.3 beschriebenen Schritte ausgeführt, um mit Hilfe der Plattformen Confluence und JIRA Anforderungen an die durchzuführenden Änderungen zu definieren und nach dem Prinzip des BDD Tests zu implementieren. An dieser Stelle wird das Test-Framework Einsatz finden und die Formulierung der Ontologietests vereinfachen. Als nächstes findet die eigentliche Evolution statt, in dem der Entwickler über die Semantic Workbench eine beliebige Ontologie-Zusammenstellung lädt und diese über den dazugehörigen Ontologie-Editor bearbeitet. Sind die gewünschten Änderungen vorgenommen, werden die formulierten Ontologietests ausgeführt. Es folgt eine iterative Veränderung der geladenen Ontologie, bis alle Tests erfolgreich durchlaufen, um anschließend die Überführung in den Integrationsbranch vorzunehmen und nach weiteren Tests die Änderungen zu übernehmen. Damit ist ein Evolutions-Zyklus beendet.

[4] http://protege.stanford.edu/

Evolution im laufenden Betrieb

Über die SWB soll die Möglichkeit eingeführt werden, dass auch die Nutzer der HIP an der Ontologieevolution teilnehmen können. Aufgrund der zu erwartenden fehlenden Expertise im Bereich der Ontologien, ist es sehr wichtig, dem Nutzer feste Vorgaben zu bieten. Die SWB wird zum Beispiel so entwickelt, dass für den Nutzer Einschränkungen in den Änderungsmöglichkeiten vorgenommen oder Handlungsvorschläge gegeben werden. So werden beim Hinzufügen eines neuen Konzeptes und einer dazugehörigen Relation mögliche Prädikate vorgeschlagen, um eine Verletzung einer Range-Beschränkung zu verhindern. Trotz dieser Hilfestellungen kann nicht auf Ontologietests nach einer Änderung verzichtet werden.
Für den Nutzer gibt es zwei Möglichkeiten, an der Evolution teilzuhaben. Als Erstes wird ihm die Gelegenheit geboten, einen Texterkennungsfehler, der im täglichen Umgang mit der HIP auffällt, durch eigene Maßnahmen zu beseitigen und die Plattform somit schnell wieder fehlerfrei nutzen zu können. Falls dies nicht funktionieren sollte, wird der Fehler an das Entwicklungsteam weitergeleitet. Abbildung A.2 umfasst ein Prozessmodell, welches die einzelnen Schritte eines Nutzers zur Fehlerbeseitigung darstellt. Nach einer Fehlerentdeckung öffnet der Nutzer die SWB und lädt sowohl die von ihm benutze HION als auch ein Beispieldokument, um erst den Fehler und zum Schluss die korrigierte Texterkennung zu überprüfen. Im Anschluss wird der Ontologie-Editor geöffnet und die Ontologie geändert. Die danach auszuführenden Tests müssen automatisch über die SWB und das Test-Framework generiert werden. Beim Fehlschlagen eines Tests wird eine weitere Änderungs- und Testiteration durchgeführt, andernfalls lässt sich der Nutzer erneut die Pipelineannotationen mit den Änderungen anzeigen und kann so das Ergebnis überprüfen. Anschließend können weitere Änderungen in einer Schleife vorgenommen werden oder der Nutzer gibt die Neuerungen frei und beendet den Prozess.

Auf der anderen Seite kann der Nutzer auch Ontologieänderungen innerhalb eines Annotationsprozesses vornehmen. Dieser ist in Abbildung A.3 im Anhang zu erkennen. Annotationen dienen dazu, zu erkennen, welche Themengebiete und Konzepte innerhalb der genutzten Ontologie abgedeckt sind und welche hinzugefügt werden müssen. Der Prozess beginnt also mit einem Annotationswunsch des Nutzers. Dieser öffnet die SWB, lädt eine Ontologiekonfiguration und eine zu annotierende Datei und markiert dort relevante Wortgruppen, Wörter und Relationen. Zu diesen Markierungen können passende Konzepte oder Relationen gesucht und zugeordnet werden, die in der Ontologie vorhanden sind. Für den Fall, dass kein passendes Konzept gefunden wird, kann der Nutzer den Ontologie-Editor öffnen und dieses hinzufügen. Auch diese Änderung wird von automatisch generierten Tests überprüft und muss bei Fehlern bearbeitet werden. Wenn alle Tests erfolgreich waren, können entweder weitere Annotationen vorgenommen werden, oder der Prozess wird durch die Freigabe möglicher Änderungen beendet.

3.1.4. Linguistische Pipeline

Klinische Dokumente, die als Freitexte in Krankenhausinformationssystemen vorliegen, enthalten relevante medizinische Fakten. Für digitale Systeme ist es in diesen Texten allerdings schwer zu erkennen, wo ein bestimmtes medizinisches Faktum steht, um welches es sich handelt oder gar wie viele Fakten vorhanden sind. Somit unterscheiden sich Freitexte deutlich von strukturierten Daten, in denen Attribut-Wert-Paare dargestellt sind (wie z.B. "Blutdruck: 120/80“).
Ziel der Linguistischen Pipeline ist es, Freitexte so aufzuarbeiten, dass die Informationen in ihnen genauso erschließbar sind wie in strukturierten Texten. Den Menschen ist es aufgrund von impliziten linguistischen Strukturen möglich, Sprache und Texte zu verstehen. Die Linguistische Pipeline soll diese Strukturen durch Annotation explizit machen. Dabei wird der Text schrittweise mit immer mehr semantischen und linguisitischen Informationen versehen. Diese Schritt-für-Schritt-Annotation oder auch inkrementelle Annotation verwendet verschiedene Komponenten, die nacheinander genutzt werden. Dabei ist die Ausgabe der vorhergehenden Komponente die Eingabe der folgenden. Aus diesem Grund wird für eine solche linguistische Verarbeitungskette auch das Bild der 'Rohrleitung' (Pipeline) gebraucht, bei der verschiedene Teile aneinandergereiht ein funktionierendes Ganzes ergeben.

Funktionalität der Linguistischen Pipeline

Wie genau die Textverarbeitung in der HIP abläuft, wird an einem Beispielsatz erläutert. Es wird angenommen, dass ein Dokument über einen Patienten in die HIP eingepflegt wird und den Satz: "Er berichtete von einem torakalen Druckgefühl." enthält. Der Text wird innerhalb der Linguistischen Pipeline zuerst einer morphosyntaktischen Analyse unterzogen, welche ihn auf der Grundlage des Natural Language Processings (NLP) in Satzfragmente gliedert. Die Verarbeitung des Beispieltextes ist in Tabelle 3.3 dargestellt.

Schritte	**Erklärung**	**Beispiel**
Tokenisierung	Erkennung einzelner Wörter und Satzzeichen	Er — berichtete — von — einem — thorakalen — Druckgefühl — .
Sentence-Splitting	Markierung der Satzgrenzen (weitere Analyse auf Satzebene)	Erkennung, dass es sich nur um einen Satz handelt
Spell-Checker	Fehlerkorrektur nach dem Damerau-Levenshtein-Algorithmus	torakalen → thorakalen
Morphologie-Lookup	Zerlegung von Komposita, Rückführung zur Grundform	Er — berichten — von — ein — thorakal — Druckgefühl
POS-Tagger	Erkennung der Wortarten	Er (irreflexibles Personalpronomen) — berichten (finites Verb) — von (Präposition) — ein (Artikel) — thorakal (attributives Adjektiv) — Druckgefühl (normale Nomina)
Parser	Zerlegung in linguistische Bestandteile, Erkennung von Negation	Er [berichtet] [von einem [thorakalen Druckgefühl]].

Tabelle 3.3.: Schritte des NLP in der Linguistischen Pipeline anhand eines Beispiels, Quelle: Angelehnt an [Reu13]

Im Anschluss an das Parsing findet das Concept-Mapping statt, welches aus dem annotierten Text die eigentlichen Konzepte extrahiert, sofern sie in der HION vorhanden sind. Das Concept-Mapping umfasst somit die komplette Interaktion zwischen der Linguistischen Pipeline und der HION. Aufgrund dessen ist für die Definition der Anforderungen an die Gesamtheit der Ontologien in der HIP hauptsächlich dieser Schritt wichtig und zu beachten. Als Eingabe erhält der Prozess des Concept-Mappings einen vollständig annotierten Text und die HION. Jede im Text gefundene Nominalphrase wird über eine SPARQL-Abfrage in der HION gesucht. Wird sie nicht gefunden, bilden ihre Einzelteile neue Phrasen, die abgefragt werden. Bei einer Übereinstimmung wird ebenfalls über SPARQL-Abfragen überprüft, ob es sich bei dem gefundenen Konzept um einen MainTerm handelt (dazu zählen Diagnosen, Symptome, Medikamente und Beobachtungen) und wenn ja, welcher RDFS-Klasse es entspricht und ob durch Relationen weitere Übereinstimmungen gefunden werden können. Der dadurch entstandene Fakt wird als eine Anzahl von Triplen ins CDR eingepflegt. Ein Fakt entspricht einer konkreten Ausprägung einer abstrakten Klasse, also hier einer Aussage über einen Patienten. Nominalphrasen ohne Entsprechung in der Ontologie werden ignoriert [Bec12].
Der folgende Code-Ausschnitt zeigt, welche Fakten mit Hilfe des Concept-Mappings aus dem annotierten Beispiel generiert werden können und wie diese im CDR gespeichert werden:
Generierte Fakten im CDR:

```
1 :Fall_1 rdf:type :Fall.
2 :Druckgefühl_1 rdf:type :Druckgefühl.
3 :Fall_1 :hatBeobachtung :Druckgefühl_1.
4 :Thorakal_1 rdf:type :Thorakel.
5 :Druckgefühl_1 :hatBeobachtungsAttribut :Thorakal_1.
```

Als Erstes wird ein neuer Fall als Instanz des Konzeptes :Fall angelegt, dem eine Instanz von :Druckgefühl als Beobachtung mit dem Attribut :Thorakal_1 zugeordnet wird.

Rolle in der HIP

Die Linguistische Pipeline nimmt in der HIP einen wichtigen Platz in der Verarbeitung unstrukturierter Daten ein. Abbildung 3.3 stellt den Datenfluss innerhalb der HIP dar und zeigt den Einsatzort der Pipeline. Der Datenverarbeitungsprozess beginnt mit dem Einlesen von strukturierten und unstrukturierten Daten, die meistens über eine HL7 (Health Level 7)-Schnittstelle oder direkt aus dem KIS kommen. Die grünen Pfeile in der Abbildung zeigen den Weg der strukturierten Daten, welche nach der Pseudonymisierung sofort im CDR abgespeichert werden. Die pseudonymisierten unstrukturierten Daten (blaue Pfeile) werden an die Linguistische Pipeline weitergeleitet und dort analysiert und verarbeitet. Die entstandenen Satzfragmente werden anschließend mit der medizinischen Wissensbasis abgeglichen, um zu neuen Informationen wie Diagnosen oder Symptomen zu gelangen, die ebenfalls im CDR gespeichert werden. Allgemein wird dieser Schritt Anreicherung genannt, das heißt auf der Basis von bereits bekanntem Wissen (den strukturierten Informationen) wird die Fallakte um neue Fakten erweitert beziehungsweise angereichert. Das CDR und alle darin enthaltenen Daten können von den Anwendungen der HIP während der Laufzeit genutzt und abgefragt werden.
Die Linguistische Pipeline nimmt somit in Zusammenarbeit mit der Wissensbasis einen großen Stellenwert in der Datenverarbeitung der HIP ein: die Erkennung und damit das Verständnis von Daten aus Freitexten.

Rolle in der Wissensverwaltung

Um Faktenwissen über einen bestimmten Fall aus einem als Text gegebenen Befund extrahieren zu können, wird medizinisches Wissen benötigt. Im Rahmen der HIP wird solches Wissen in Form von Ontologien bereitgestellt, welche die Linguistische Pipeline verwendet, um einzelne Nominalphrasen (oder in ihnen enthaltene Wortgruppen) als Diagnose, Befund, Symptom, Medikament, Therapie und so weiter zu identifizieren. Hinzu kommen ergänzende Eigenschaften wie beispielsweise eine Lagebezeichnung im Körper. Neben der semantischen Annotation einzelner Elemente soll die Struktur der HION die Linguistische Pipeline unterstützen, gefundene Informationen sinnvoll zu verknüpfen und zu vernetzen. Da dieser Schritt während des Concept-Mappings vonstattengeht, gibt genau dieser Prozess Anforderungen an die genutzten Ontologien vor.

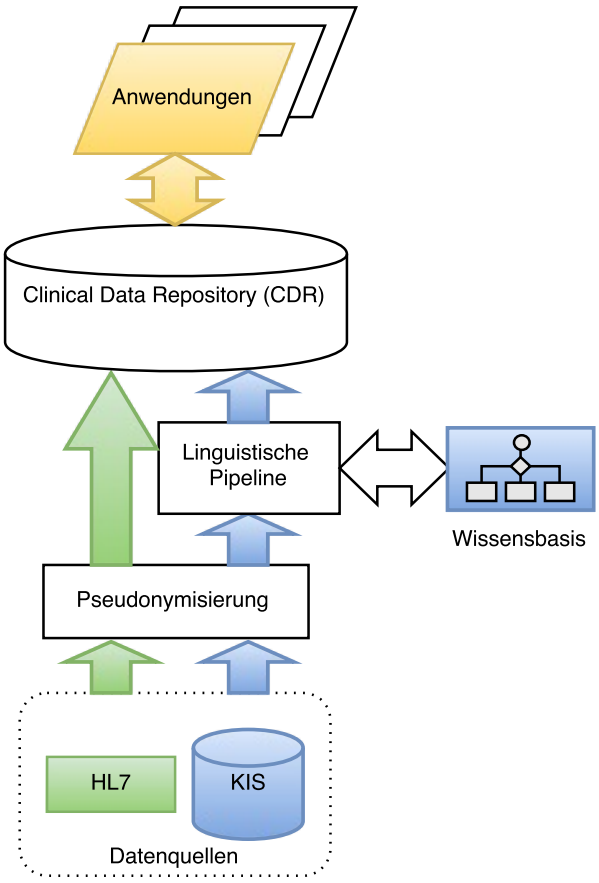

Abbildung 3.3.: Datenfluss in der HIP, Quelle: Eigene Abbildung

In erster Linie zählt die Vielfalt der medizinischen Konzepte und die Struktur, in der sie eingebettet sind. Es muss für die Linguistische Pipeline klar erkennbar sein, zu welcher Oberklasse ein Konzept zugeordnet wird und ob es einen MainTerm bildet. Zusätzlich haben auch die semantischen Relationen in der HION einen wichtigen Stellenwert. Sie müssen genutzt werden, damit bestimmte Beziehungen durch das Concept-Mapping erkannt werden können. So muss in der HION zum Beispiel die Relation :hatLokalisation zwischen den Konzepten :Infarkt und :Niere vorhanden sein, um auf Basis eines passenden Textes zu erkennen, dass nicht nur ein Infarkt festgestellt wurde, sondern auch dass dieser in der Niere lokalisiert ist. Das gilt auch für alle anderen semantisch relevanten Relationen. Zusätzlich gibt es Relationen, die von der Pipeline genutzt, aber nicht semantisch benötigt werden. Dazu gehören :synonym und :standard_label, welches pro Konzept nur einmal vergeben werden darf und einzigartig sein muss, da es als eine Art Identifikation des Konzeptes angesehen wird. Diese Besonderheiten muss die HION in jedem Fall beibehalten.
Zusammenfassend spielt die Linguistische Pipeline mit dem Concept-Mapping eine entscheidende Rolle für die Qualität der HION. Sie setzt sowohl strukturelle Anforderungen zur Aufrechterhaltung ihrer Funktionalität als auch Anforderungen an die Ergebnisqualität des Anwendungsfalls um.

3.1.5. Qualität in der HIP-Wissensrepräsentation

Nachdem in Kapitel 2.1.4 verschiedene Qualitätsmodelle für Ontologien vorgestellt wurden, muss anschließend festgelegt werden, was Qualität im Sinne der HIP-Ontologien bedeutet.

Qualitätskriterien

In Anlehnung an die verschiedenen Qualitätsmodelle und unter Berücksichtigung der aufgeführten Informationen zur Wissensrepräsentation in der HIP werden folgende Qualitätsmerkmale beziehungsweise -kriterien für eine beim Kunden laufende HION eingeführt:

- Angemessenheit und Aussagekraft,
- Übersichtlichkeit und Modularität,
- Erweiterbarkeit,
- Überprüfbarkeit/Testbarkeit,
- Vollständigkeit (im Sinne der Open World Assumption),
- Funktionale Korrektheit,
- Strukturelle Korrektheit,
  - Redundanz,
  - Zyklen,
  - Konsistenz (logisch und strukturell),
  - Strukturelle Genauigkeit - Einhaltung von Domain und Range Beschränkungen,
  - Einhaltung der Basishierarchie.

Es fällt auf, dass Metriken zu Konzepten beziehungsweise Klassen an sich, wie im OntoQA-Modell, komplett fehlen. Der Grund dafür ist, dass diese Kriterien keine Aussage darüber liefern können, ob die HION den an sie gerichteten Anforderungen innerhalb der HIP gerecht wird. Im Gegensatz dazu konnten die Qualitätsmerkmale des OQuaRE Frameworks teilweise übernommen werden.

Messbarkeit der Qualität

Aufgrund des bereits festgestellten Mangels an Werkzeugunterstützung im Bereich der Ontologiequalität beschäftigt sich diese Arbeit damit, einen eigenen Ansatz zu entwickeln, um die HION zu testen.
Viele der genannten Qualitätskriterien sind nur schwer direkt durch Tests abzufangen, da sie eher oberflächlich formuliert sind. Aufgrund dessen wird nachfolgend in Form von Beispielen darauf eingegangen, wie die Kriterien genau in Verbindung mit den HIP-Ontologien definiert werden.
Angemessenheit und Aussagekraft: Die Konzepte und Strukturen der HION entsprechen der Semantik in der realen Welt beziehungsweise im aktuellen Anwendungsfall. Dieses Kriterium wird indirekt durch die Überprüfung der funktionalen Korrektheit getestet.
Übersichtlichkeit und Modularität: Darunter zählt zum einen die Trennung der medizinischen Fachbereiche in einzelne Dateien. Dieses Merkmal kann während der Nutzung der HION nur über die Zugehörigkeit zu den Namensräumen überprüft werden. Zum anderen gehört zu diesem Punkt auch die inhaltliche Trennung der Konzepte durch die Einordnung in die Hierarchie. So gelingt es beispielsweise, Krankheitsbilder von Therapien zu trennen.
Erweiterbarkeit: Das Thema Erweiterbarkeit bezieht sich auf die Möglichkeit, dass die Ontologie weiter wachsen kann und sich daraus ein Wachstum der Funktionen ergibt. Dieses Kriterium wird allein dadurch überprüft, dass zum Beispiel ein neues Konzept hinzugefügt wird und im Anschluss daran alle Qualitätstests erfolgreich ablaufen.
Überprüfbarkeit/Testbarkeit: Dass dieses Kriterium erfüllt ist, wird mit der Implementierung und Anwendung des Test-Frameworks bewiesen.
Vollständigkeit: Der Begriff vollständig ist hier im Sinne der Open World Assumption zu verstehen. Demnach kann ein Objekt in der realen Welt existieren, obwohl es nicht in der Ontologie definiert ist. Vollständig ist die HION immer dann, wenn im Rahmen der Texterkennung alle erwarteten Konzepte gefunden werden. Zur Erfüllung dieses Kriteriums wird die SWB eingeführt.
Strukturelle Korrektheit: Dieses Qualitätsmerkmal nimmt einen großen Teil der testbaren Kriterien ein. Alle Unterpunkte müssen innerhalb des Test-Frameworks überprüft werden, um die Qualität der HION zu gewährleisten. Sie werden weiterhin als strukturelle Tests bezeichnet.

Funktionale Korrektheit: Im Fall der HIP kann dieses Kriterium auch als semantische Korrektheit bezeichnet werden, denn im momentanen Zustand der Plattform hat die Ontologie die Funktion, eine semantische Textverarbeitung zu unterstützen und Inhalte zu isolieren. Auch dieses Kriterium muss in Form von Tests im Framework überprüft werden. Sie werden semantische Tests genannt.
Abschließend ist festzuhalten, dass das zu erarbeitende Framework vor allem Tests zur Verfügung stellen muss, um die strukturelle und semantische Korrektheit der Ontologie zu überprüfen.

3.2. Werkzeuge und Methoden zur Qualitätsüberprüfung von Wissensrepräsentationen

In Kapitel 2.1.4 wurden bereits verschiedene Methoden zu Qualitätskriterien für Wissensrepräsentationen vorgestellt, allerdings ohne darauf einzugehen, wie diese überprüft und getestet werden können. Nachfolgend werden verschiedene Methoden und Werkzeuge vorgestellt, die sich genau mit dieser Frage beschäftigen.

3.2.1. Vergleich von Werkzeugen und Methoden

Test-Driven Linked-Data Quality Methodology

Diese Methode wird in [Kon14] beschrieben und thematisiert die testgetriebene Qualitätsüberprüfung von Linked-Data, die durch testgetriebene Softwareentwicklung inspiriert ist. Da das Vorgehen teilweise auf die Qualitätsüberprüfung von Wissensrepräsentationen übertragbar ist, wird es an dieser Stelle näher beleuchtet. Das Paper beschäftigt sich mit der Qualität von Linked-Open-Data und damit wie einzelne, sich eventuell entwickelnde, Datensätze mit einem bestimmten Anwendungsfall auf dem gleichen Qualitätslevel gehalten werden können. Zusätzlich wird die Aussage formuliert, dass alle Vokabularien, Ontologien und Wissensbasen immer von Testfällen begleitet werden sollten, um ein Basislevel an Qualität aufrecht zu erhalten.
In Abbildung 3.4 sind die einzelnen Komponenten der Methode zu erkennen. Der erste große Schritt umfasst den Aufbau einer Musterbibliothek (Pattern Library), die sogenannte Data Quality Test Patterns (DQTP) enthält. Dabei handelt es sich um verschiedene Testmuster, die über parametrisierte SPARQL-Abfragen bestimmte Qualitätsaspekte überprüfen können. Zusammengestellt wird diese Bibliothek aus generalisierten und kategorisierten bekannten

Fehlern in Datensätzen und Integritätsbeschränkungen durch OWL und RDFS, also dem benutzten Schema. Die Muster sind generisch und lassen sich auf verschiedene Datensätze anwenden. Sie können zur schnellen Entwicklung von Testfällen manuell instanziiert werden. Zusätzlich umfasst die Methode eine automatisierte Instanziierung der Muster auf Basis einer konkreten Datenquelle. Dieser Schritt wird Test Pattern Binding genannt und resultiert in Data Quality Test Cases. Damit sind alle konkreten Testfälle gemeint, die im Anschluss daran auf der Datenquelle ausgeführt werden können und Aufschluss über deren Datenqualität geben.

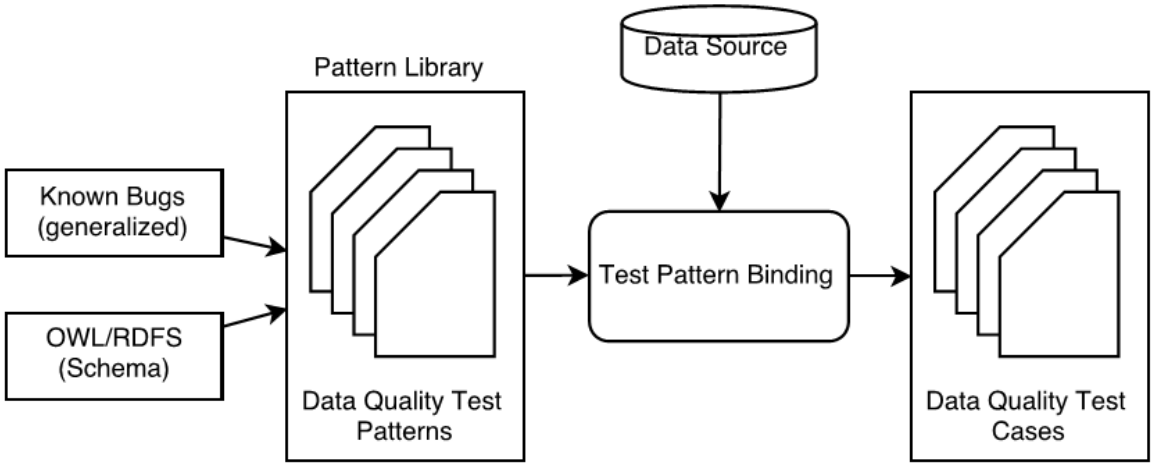

Abbildung 3.4.: Test-Driven Linked-Data Quality Methodology, Quelle: Angelehnt an [Kon14]

Im Anhang des Papers befindet sich eine Sammlung von 17 DQTPs, die im Zuge der Bearbeitung entstanden sind. Jedes Muster wird kurz beschrieben und zeigt die dazugehörige parametrisierte SPARQL-Abfrage. Zusätzlich sind jeweils einige Beispiele aufgeführt, an welchen Stellen das Muster eingesetzt werden könnte. Es fällt auf, dass einige DQTPs so allgemein formuliert sind, dass sie nicht nur zur Überprüfung von Datenqualität sondern auch in leicht veränderter Form für eine Wissensrepräsentation an sich genutzt werden könnten. Ein Beispiel dafür ist das Muster mit dem Namen *RDFSRANGED Pattern*. Es überprüft die Einhaltung der Domain-Beschränkung aus der Ontologiesprache RDFS, also ob der Datentyp eines Objektes in einem Triple zu den Beschränkungen der Relation passt. Bei der Überprüfung des Ontologieschemas der HIP könnte dieses Testmuster dahin gehend verändert werden, dass anstatt des Datentyps die Art des Konzeptes, also welche Oberklasse es besitzt, kontrolliert wird.

OntoLoki

Die OntoLoki Methode nach [Goo15] spezialisiert sich vor allem auf biologische und medizinische OWL-Ontologien, die keine formalen Restriktionen enthalten, wann welche Instanzen zu welchen Klassen gehören. Somit gibt es keine automatischen Methoden zur Messung der Konsistenz der Instanzzuordnungen. Genau diese wird jedoch im Modell als wichtige Qualitätseigenschaft angesehen. Die Grundidee von OntoLoki besteht darin, aus einer Wissensbasis, die aus einer TBox und einer ABox (Instanzen und dazugehörige Properties) besteht, Regeln und Muster abzuleiten, anhand derer die Konsistenz der Instanzzuordnungen gemessen werden kann. Als erstes entstehen Property-Muster, die Instanzen einer Klasse von den Instanzen anderer Klassen unterscheiden können. Zusätzlich dazu ist jedem Muster ein Score als sogenannte Klassifikationskonsistenz zugeordnet, die einschätzt, mit welcher Sicherheit die Zugehörigkeit einer Instanz zur richtigen Klasse vorhergesagt werden kann.
Der Algorithmus beginnt mit einer Tabelle, die für jede Instanz eine Zeile besitzt und die Zugehörigkeit zu den verschiedenen Klassen sowie die Nutzung von Properties durch die Werte 0 oder 1 angibt. Die Angaben werden dabei nach der Closed World Assumption gemacht und es gilt: wenn eine Instanz zu einer bestimmten Klasse gehört, zählt sie auch zu deren Überklassen. Anschließend wird durch die Anwendung von Algorithmen zum überwachten Lernen aus dem Softwaretool WEKA (Waikato Environment for Knowledge Analysis)[5] ein Klassifikationsverfahren auf 90% der vorhandenen Daten trainiert, um eine Instanz einer Klasse zuordnen zu können. Die restlichen 10% werden zum Testen der Genauigkeit des Verfahrens verwendet. Innerhalb dieser Algorithmen entstehen die Property-Muster und Scores, welche durch das Klassifikationsverfahren genutzt werden. So kann bei Anwendung auf eine andere Wissensbasis eine Messung der Konsistenz der Instanzzuordnungen durchgeführt werden.

OntologyTest

OntologyTest ist eine Methode und eine Java Anwendung zum dynamischen Testen von OWL-DL-Ontologien hinsichtlich ihrer funktionalen Spezifikation und wird in [GOF09] beschrieben. Es ermöglicht den Nutzern Tests zur Funktion zu definieren, auszuführen und die Ergebnisse zu kontrollieren. Jeder Test umfasst dabei eine optionale Menge an Instanzen, eine Abfrage und ein erwartetes Ergebnis und kann zu jeder Zeit im Ontologie-Zyklus ausgeführt werden. Abfragen werden in erster Linie manuell aus Kompetenzfragen entwickelt und in SPARQL-Abfragen überführt. Kompentenzfragen sind natürlichsprachliche Fragestellungen, die durch

[5] http://www.cs.waikato.ac.nz/ml/weka/

eine spezielle Ontologie beantwortet werden sollten.
Die Verwendung des Werkzeuges startet mit dem Anlegen eines neuen Prozesses und dem Festlegen der zu evaluierenden Ontologie. Voraussetzung ist, dass diese konsistent und syntaktisch korrekt sein muss. Ist dies nicht der Fall, wird eine Fehlermeldung angezeigt. Wie diese Überprüfung stattfindet, wird nicht beschrieben. Anschließend können die Testabfragen zu den funktionalen Anforderungen definiert werden. In einigen Fällen ist es nötig, Instanzen zu bestimmen, damit eine Abfrage erfolgreich sein kann. Als letztes muss der Nutzer das erwartete Ergebnis angeben, um dieses mit dem tatsächlichen Ergebnis einer Anfrage vergleichen zu können. Eine Sammlung von Tests kann in einer XML-Datei gespeichert werden, um sie an weitere Entwickler weiterzugeben.

3.2.2. Einschätzung zur möglichen Integration der Werkzeuge und Methoden

Zusammenfassend kann über jede der vorgestellten Methoden und Werkzeuge festgehalten werden, dass keine komplett auf den Anwendungsfall des Testens der HION anwendbar ist. OntoLoki konzentriert sich auf die Konsistenz von Instanzzuordnungen. Dieser Ansatz ist in der HION schwer umzusetzen, da die ABox im CDR, die die eigentlichen Instanzen enthält, nicht getestet werden soll. Im Hinblick auf die relativ kleine Anzahl an Relationen und die Annahme der Closed World Assumption ist es außerdem fragwürdig, ob in diesem Zusammenhang ein aussagekräftiges Klassifikationsverfahren entstehen würde. Zusätzlich ist OntoLoki ausschließlich auf OWL-Ontologien ausgelegt. Nach der Analyse der HION ist jedoch klar, dass diese nicht den OWL-Richtlinien entspricht.
Die OntologyTest-Anwendung verfolgt den auch für die HION interessanten Ansatz zur Überprüfung der funktionalen Spezifikation, also des Anwendungsfalls einer Ontologie. Das entspricht den semantischen Tests des Frameworks. Auch hier besteht das Problem, dass ausschließlich OWL-DL-Ontologien vorgesehen sind. Dementsprechend ist die Applikation nicht nutzbar. Allerdings kann das Modell an sich, in dem der Nutzer eine Erwartungshaltung an einen bestimmten Test formuliert, damit diese mit dem tatsächlichen Verhalten der Ontologie verglichen werden kann, adaptiert werden.
Die Data Quality Methodology hat das Ziel Datenqualität zu überprüfen. Allerdings ist das Modell an sich, in dem eine Bibliothek für Testmuster zur Verfügung gestellt wird, um daraus konkrete Testfälle zu instanziieren, gut auf die strukturellen Tests anwendbar.

3.3. Qualitätsüberprüfung der Linguistischen Pipeline

Im Zuge der Entwicklung der Linguistischen Pipeline wurden Tests implementiert, die sicherstellen sollen, dass die Texterkennung der Plattform im gewünschten Umfang funktioniert. Da am Erkennungsprozess ebenso die HION beteiligt ist, wird in einigen Tests auch diese auf ihre Funktionalität im Sinne ihrer Anwendung überprüft. Im Weiteren wird dieser Teil der Pipelinetests, OntologyPipelineTests genannt, analysiert, um zu entscheiden, ob sie innerhalb des Test-Frameworks wiederverwendet werden können.

3.3.1. Analyse

Die Pipelinetests wurden mit Hilfe von JUnit implementiert und befinden sich direkt im Projekt der Linguistischen Pipeline. Falls Änderungen an dieser vorgenommen werden, können die Tests überprüfen, ob die gewünschten Verhaltensweisen und Ergebnisse trotz allem erzielt werden. Es gibt für jede NLP-Phase in der Pipeline eigene Tests, allerdings prüfen erst die OntologyPipelineTests anhand von Beispielen am Ende, ob der komplette Texterkennungsprozess funktioniert.

Aufbau der Erwartungshaltungen

Den Input für die Tests bilden Dateien, die Sätze oder Wortgruppen und die dazugehörigen Fakten enthalten, die erkannt werden sollen. Als Ergebnis wird für jeden Satz angezeigt, ob alles richtig erkannt wurde (als “yes”) oder Abweichungen entdeckt wurden (“no”).
Ein Beispiel einer Inputdatei zeigt der folgende Abschnitt:

```
1 Er berichtete von Brustschmerzen.
    ==> [hatBeobachtung:Thoraxschmerz[]]
2 Er berichtete von Brustenge.
    ==> [hatBeobachtung:AnginaPectoris[]]
3 Er berichtete von einem thorakalen Druckgefühl.
    ==> [hatBeobachtung:Druckgefühl[hatBeobachtungsAttribut:
    Thorakal[]]]
4 Patient hat LVEF >= 0.40
    ==> [hatBeobachtung:LVEF[hatWert:0.40[]]]
```

Es ist zu erkennen, dass die Erwartungen in einer Art Baumstruktur aufgebaut sind, die für die Anwendung innerhalb der Pipelinetests konzipiert wurde. Die Wurzel wird durch eine Relation gebildet (meist :hatBeobachtung), der ein Wurzelkonzept zugeordnet ist. Anschließend ist es möglich, Konzepte oder auch Werte über weitere Relationen anzuhängen und ineinander zu verschachteln. Eine Erwartungshaltung kann somit aus folgenden Elementen zusammengesetzt werden:

- Wurzel,
- Konzept,
- Relation,
- Wert,
- Einheit.

Aufbau der OntologyPipelineTests

Die wichtigsten drei Komponenten in diesem Teil der Pipelinetests sind: der OntologyPipelineTestRunner, der Analyzer und der Matcher.
Der Analyzer ist ein Modul, welches sich um die Ausführung der Texterkennung durch die Linguistische Pipeline und die Ontologien kümmert. Er sorgt dafür, dass die Beispielsätze, die als Input dienen, mit Hilfe der Pipeline verarbeitet (siehe Kapitel 3.1.4) und anschließend über das Concept-Mapping Fakten generiert werden.
Der Matcher hat die Aufgabe, die entstandenen RDF-Fakten in die beschriebene Baumstruktur umzuformen und die Ergebnisse anschließend mit den Erwartungshaltungen zu vergleichen. Für jeden einzelnen Satz wird die Übereinstimmung geprüft und entweder ein "true" oder bei gefundenen Abweichungen ein "false" zurückgegeben.
Zusammengeführt werden diese beiden Schritte über den sogenannten OntologyPipelineTestRunner. Dieser umfasst einen JUnit-Test, der zur Ausführung gestartet wird und die anderen Komponenten aufruft. Der Runner sorgt dafür, den richtigen Input zu definieren und dem Nutzer aussagekräftige Ausgaben zu bieten.

3.3.2. Einschätzung zur möglichen Integration der OntologyPipelineTests

Die OntologyPipelineTests sind momentan nur ein kleiner Teil der Pipelinetests, die überprüfen, ob der gesamte Texterkennungsprozess in der HIP so funktioniert, wie er sollte. Bei Abweichungen könnte der Fehler sowohl in den Ontologien als auch in der Pipeline lokalisiert sein. Diese gewisse Unsicherheit kann momentan nicht beseitigt werden. Nur wenn davon ausgegangen wird, dass die Tests vor einer Ontologieänderung erfolgreich waren, kann der Fehler sicher in der HIOn gesucht werden. Ausgeführt werden die OntologyPipelineTests ebenfalls nur im Rahmen der Pipelinetests. Durch die beschriebenen unterschiedlichen Module wird allerdings die Möglichkeit geboten, auf diese zuzugreifen und sie zu nutzen. Eine andere Möglichkeit wäre es, die wichtigen Module zu kapseln und direkt in die Ontologietests einzubinden. In jedem Fall muss das Framework dazu in der Lage sein, die Linguistische Pipeline anzusteuern und zu nutzen. Dieser Aspekt wird im Zuge dieser Arbeit nicht umsetzbar sein, da das Test-Framework vorerst modular und unabhängig entstehen soll, um es auch in zukünftigen Projekten der Firma einsetzen zu können.

4. Konzeption des Frameworks

Es folgt die Planung des Frameworks. Diese beinhaltet eine Anforderungsanalyse, die Konzeption der einzelnen Komponenten sowie die Beschreibung ihres Zusammenspiels.

4.1. Anforderungsanalyse

Wie zu Beginn jeder Softwareentwicklung wurde eine Anforderungsanalyse durchgeführt, um das Endprodukt zu spezifizieren und dessen Konzeption zu ermöglichen. Als Ergebnis der Analyse sind Anforderungen an das Framework entstanden, welche im Laufe der Entwicklung im Rahmen dieser Arbeit sowie in der Weiterentwicklung und Integration in die HIP umgesetzt wurden.

4.1.1. Anwendungsfälle

Zur Identifikation von Anforderungen mussten zuerst die Anwendungsfälle für das Framework innerhalb der HIP betrachtet und analysiert werden, um das Produkt perfekt anzupassen und den Integrationsaufwand so gering wie möglich zu halten.

Evolution der Ontologien

In erster Linie wird das Test-Framework zur Unterstützung der Ontologieevolution sowohl in der Entwicklung als auch während der Laufzeit benötigt. Die unterschiedlichen Einsatzmöglichkeiten sind im Kapitel 3.1.3 und den dazugehörigen Prozessmodellen formuliert.
Die erste Anwendung soll das Framework im Rahmen der Evolution im Entwicklungsprozess finden, da dies die momentan praktizierte Lösung ist. Um den Entwicklern der Ontologien die Möglichkeit zu geben, jegliche Änderungen nicht nur manuell zu überprüfen, soll das Test-Framework zum Einsatz kommen. Dabei sollen nicht nur strukturelle, sondern auch semantische Fehler aufgedeckt werden. Das Framework muss so konzipiert werden, dass es in der Zukunft an JIRA angeschlossen werden kann und mit Hilfe der dort hinterlegten konkreten Anforderungen als JBehave-Stories die in diesem Fall passenden Tests generiert.

Der Aufwand des Entwicklers an dieser Generierung soll so gering wie möglich gehalten werden. Der genaue Umfang wird jedoch erst im Anschluss an diese Arbeit bei der Integration des Frameworks in die HIP-Strukturen festgelegt. An diesen konkreten Anwendungsfall wurde die Konzeption und Entwicklung des Test-Frameworks innerhalb der Arbeit vorrangig angelehnt. Allerdings fällt der Aufwand für den Entwickler in der aktuellen Umsetzung aufgrund der fehlenden Anbindung an die HIP und ihren Entwicklungsprozess höher aus, als in der endgültigen Produkt-Version des Frameworks.
Die Möglichkeit für Nutzer der Plattform, Änderungen an den HIP-Ontologien im laufenden Betrieb vorzunehmen, führt zu hohen Fehlerpotenzialen, da diese in den meisten Fällen kein bis wenig Wissen über Ontologien besitzen. Das heißt, den Benutzern muss eine Hilfestellung gegeben werden, um Änderungen vorzubeugen, die zu Fehlern im HIP-System führen. Abgesehen davon, dass die geplante SWB schon über die Nutzeroberfläche bestimmte Änderungen einschränken soll, ist es trotzdem notwendig, im Anschluss die wichtigsten Qualitätskriterien und Ontologieanforderungen zu überprüfen. Um dies zu ermöglichen, soll das Test-Framework zukünftig in die SWB integriert und das Testen der geänderten Ontologien erleichtert werden. Wie der genaue Einsatz an dieser Stelle funktioniert, kann allerdings erst nach der endgültigen Planung und Umsetzung der SWB festgelegt werden und hat daher auf die Konzeption des Test-Frameworks keinen Einfluss. Auch die automatische Generierung der Ontologietests mit Hilfe des Frameworks im Rahmen der Evolution durch den Nutzer spielt im weiteren Verlauf keine Rolle.

Entwicklung von Ontologien

Es ist nicht auszuschließen, dass es bei der Entwicklung einer HIP-Version für neue Kunden nötig sein kann, auch eine komplett neue HION aufzusetzen, um die individuellen Informationsstrukturen so gut wie möglich nutzen zu können. Trotz allem muss sichergestellt werden, dass die neue Ontologie auch den Anforderungen der HIP entspricht und fehlerfrei mit der Linguistischen Pipeline zusammenarbeiten kann. Zusätzlich soll durch die Modularität der HIP-Ontologiedateien ermöglicht werden, neue Ontologie-Versionen aus verschiedenen medizinischen Fachbereichen zusammenzufügen, um individuellen Kundenansprüchen gerecht zu werden. Auch auf diese Weise neu entwickelte Ontologien müssen auf ihre Qualität im Sinne der HIP überprüft werden. Mit Hilfe des Test-Frameworks, welches in diesem Fall im Entwicklungsprozess von den Ortec-Mitarbeitern genutzt wird, sollen die Anforderungen der HIP überprüfbar werden.

Der von den Entwicklern zu durchlaufende Prozess in diesem Anwendungsfall ähnelt dem Prozessmodell in Abbildung A.1. Der Unterschied besteht darin, dass keine Ontologiekonfiguration geladen wird, da die zu überprüfende erst entsteht. Ansonsten kann über den OntoEditor eine neue Ontologie aufgesetzt und diese im Anschluss mit Hilfe des Frameworks getestet werden.

Qualitätsmanagement der Ontologien

Zusätzlich soll das Framework die Möglichkeit geben, zu jeder Zeit die Qualität der vorhandenen Ontologien einschätzen zu können. Dabei wird unter dem Wort "Qualität" in diesem Fall verstanden, was schon in Kapitel 3.1.5 definiert wurde. Vorausschauend soll ein übergreifendes Ontologiemanagement in der HIP realisiert werden, welches sowohl in der Entwicklung als auch während der Laufzeit Anwendung findet. Durch viele Änderungen und die Existenz verschiedener HION-Versionen bei den Kunden wird es immer wichtiger, ein einfaches und effektives Qualitätsmanagement für die Ontologien zur Verfügung zu stellen. Auch wenn dieser Anwendungsfall momentan noch keine Anforderungen an das Test-Framework stellt, bleibt es im Sinne des Ausblicks nicht unerwähnt.

4.1.2. Anforderungen aus den Anwendungsfällen

Aus den zuvor erläuterten Anwendungsfällen, vor allem aus der Ontologie-Evolution durch die Entwickler, ergeben sich einige Anforderungen an das Test-Framework. Als Input werden durch den Entwickler definierte Erwartungshaltungen an die Ontologien festgelegt (später als JBehave-Stories), für die er passende Tests auswählen und instanziieren muss, um konkrete Testfälle zu generieren. Diese bilden den Output des Frameworks. Eine Erwartungshaltung entspricht hierbei auf der einen Seite einer bestimmten Faktenerkennung und bezieht sich daher auf semantische Korrektheit. Auf der anderen Seite müssen durch den Nutzer auch strukturelle Anforderungen festgelegt werden. Die Hauptaufgabe des Frameworks sollte es sein, die semantische und strukturelle Testgenerierung zu unterstützen. Die konkreten Tests sollten für den Entwickler leicht ausführbar sein und zurückgeben, ob und wo ein Fehler und somit eine Verletzung der Ontologie-Anforderungen gefunden wurde. Der Ausführende sollte durch die Anzeige des fehlgeschlagenen Tests eingrenzen können, welche Konzepte oder Relationen er überprüfen sollte.

4.1.3. Technische Anforderungen

Die Auslegung des Test-Frameworks auf die HIP und die zukünftige Integration birgt auch technische Anforderungen, die im Folgenden erläutert werden.
Der geplante Anschluss an den HIP-Entwicklungsprozess legt die Nutzung des Frameworks JUnit und der damit verbundenen Programmiersprache Java nahe. Grund dafür sind die Werkzeuge JIRA und JBehave, die ebenfalls auf JUnit basieren. Um den neuesten Stand der Technik nutzen zu können, wurden alle Werkzeuge in ihrer aktuellsten Version zum Zeitpunkt der Entwicklung verwendet.
Zusätzlich spielt das Java-Framework Eclipse RDF4J (früher Sesame) eine große Rolle in der Entwicklung des Test-Frameworks, da es Methoden zur Verarbeitung von RDF-Daten bietet. Aufgrund der Nutzung von Sesame in anderen HIP-Modulen, wie der Linguistischen Pipeline, wurde es auch in diesem Projekt eingebaut.

4.1.4. Zusammenfassung der Anforderungen

Die folgende Liste bildet eine Zusammenfassung der an das Test-Framework gestellten Anforderungen:

- Input: Erwartungshaltung des Entwicklers, Ontologie auf der die Tests ausgeführt werden sollen;
- Output: Ontologietests;
- Überprüfung der semantischen Korrektheit;
- Überprüfung der strukturellen Korrektheit;
- Tests sind leicht ausführbar;
- Anzeige, welche Tests fehlgeschlagen sind oder erfolgreich waren;
- Überprüfung der Qualitätskriterien möglich;
- Nutzung von Java und JUnit;
- Nutzung von RDF4J.

4.2. Konzeptionelles Modell des Frameworks

Die Aufgabe des Frameworks ist es, unterschiedliche Testarten zusammenzuführen und ein Medium zu bilden, mit dessen Hilfe alle wichtigen Bereiche und Anforderungen an Ontologien gleichzeitig überprüft werden können. Dazu werden die Tests an sich benötigt und eine Schicht, in der diese ausgewählt, instanziiert und ausgeführt werden können. Das Framework selbst bietet Strukturen und Unterstützung, damit der Tester diese Aufgaben schnell und einfach erledigen kann. Die konkreten ausführbaren Tests entstehen daher erst bei der Verwendung des Frameworks. Das im folgenden beschriebene konzeptionelle Modell des Test-Frameworks gibt seine Struktur und Repräsentation im Sinne seiner Anwendungsfälle vor und berücksichtigt dabei die gegebenen Randbedingungen.

4.2.1. Aufbau und Komponenten des Frameworks

Das Framework besteht aus mehreren Komponenten, welche in Verbindung mit ihren Interaktionen in Abbildung 4.1 dargestellt sind. Die Grundlage bildet ein In-Memory-Repository beziehungsweise Datenspeicher, in den zur Laufzeit die zu testenden Ontologien (also das Testobjekt) geladen werden. Sowohl die strukturellen als auch die semantischen Tests führen alle Operationen auf diesem Zwischenspeicher durch. Der Nutzer selbst legt den Umfang der gespeicherten Ontologien fest. So ist es möglich, auch einzelne Ontologieteile mit Hilfe des Frameworks zu überprüfen. Die verschiedenen Tests werden wiederum von einer gemeinsamen Zugriffsschicht angesprochen, in welcher die Auswahl und Instanziierung der benötigten Testfälle über sogenannte Test Suiten stattfindet. Diese umfassen eine Sammlung von Tests, die nacheinander ausgeführt werden. Die Zugriffsschicht gehört nicht zum Framework, sondern fungiert als Schnittstelle zu diesem. Dort werden die konkreten Testfälle implementiert und ausgeführt. Über eine graphische Benutzeroberfläche (GUI) sollen vor allem die Nutzer der HIP mit der Zugriffsschicht kommunizieren können. Die GUI gehört allerdings nicht zum Entwicklungsumfang dieser Arbeit.
Die strukturellen und semantischen Tests innerhalb des Frameworks sollen festlegen, wie die jeweiligen konkreten Tests aufgebaut werden und welche Funktionen sie anbieten. Beide Testarten stellen dafür abstrakte Klassen zur Verfügung. Die genauen Konzeptionen sind in den nächsten Abschnitten zu finden.

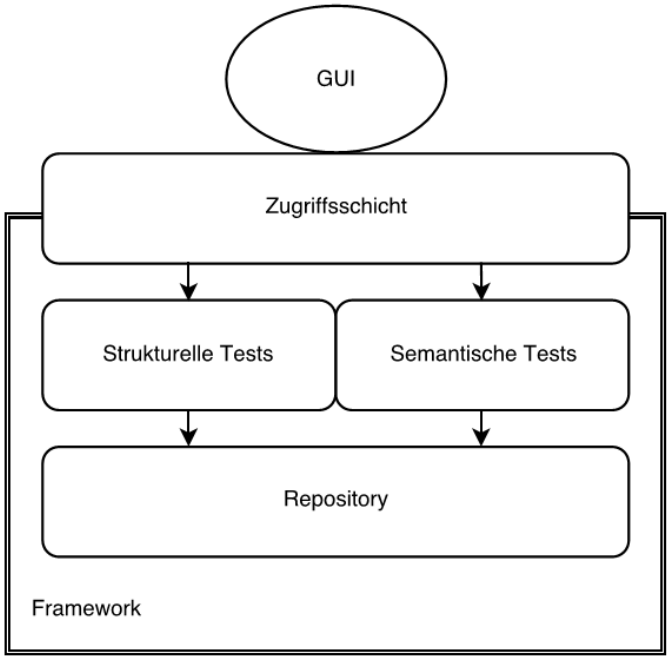

Abbildung 4.1.: Komponenten des Frameworks, Quelle: Eigene Abbildung

Die in Kapitel 2.4.3 beschriebenen Framework-Elemente des Test Runners, Reporters und des Überprüfungsmechanismus finden im Test-Framework direkt keine Anwendung. Es gibt allerdings auch keine ausführbaren Tests, die diese Elemente nötig machen würden. Erst bei der Implementierung der Tests unter Verwendung des Frameworks wird JUnit zur Überbrückung dieser Aufgaben genutzt.

4.2.2. Nutzung des Frameworks

Die medizinischen Nutzer sollen später über eine GUI und die Zugriffsschicht mit dem Framework interagieren und Ontologietests in bestimmten Situationen generieren und ausführen können. Die Voraussetzungen dafür sind durch den allgemeinen Aufbau des Frameworks gegeben. Die dieser Arbeit entspringende Version ist grundsätzlich nur zur Nutzung durch Programmierer vorgesehen, die sich mit der HIP auskennen und setzt diese Kenntnisse auch voraus.
Um das Test-Framework nutzen zu können, muss es als Erstes in das aktuelle Entwicklungsprojekt integriert werden. Um das zu ermöglichen, liegt es dem Nutzer in Form eines Java Archive (JAR-Datei), also einer Java-Klassenbibliothek, vor. Dieses wird über Maven aus dem Entwicklungsprojekt des Frameworks erzeugt. Der Nutzer muss die JAR-Datei in sei-

nem Maven-Projekt installieren und anschließend als Bibliothek einbinden. Später ist genau diese direkt im HIP-Projekt enthalten, sodass keine Schritte zur Integration notwendig sind. Anschließend kann jedes Modul und jede Klasse des Frameworks referenziert und initialisiert werden. Eine Veränderung am Rahmenwerk an sich, ist so jedoch nicht mehr möglich.
Der erste Schritt zur Generierung von Ontologietests ist die Festlegung einer Ontologiekonfiguration durch den Nutzer, die in das In-Memory-Repository geladen wird. Zur Umsetzung des Repositories und zur Kommunikation mit diesem wird das Java Framework RDF4J genutzt.
Abbildung 4.2 zeigt die Idee hinter der Nutzung des Frameworks genauer. Ein Entwickler hat die Aufgabe, Ontologietests zu schreiben und nutzt dafür das Test-Framework. Er muss zum einen die abstrakten semantischen Klassen instanziieren und unter Angabe eines Textes und seiner Erwartungshaltung ausführen. Zum anderen wählt er aus der Menge der strukturellen Testmuster, die auf den abstrakten strukturellen Klassen basieren, zu seinen Anforderungen passende aus und führt auch diese mit Angabe der benötigten Parameter aus. Die Ausführungen an sich werden in einer Test Suite zusammengefasst stattfinden.

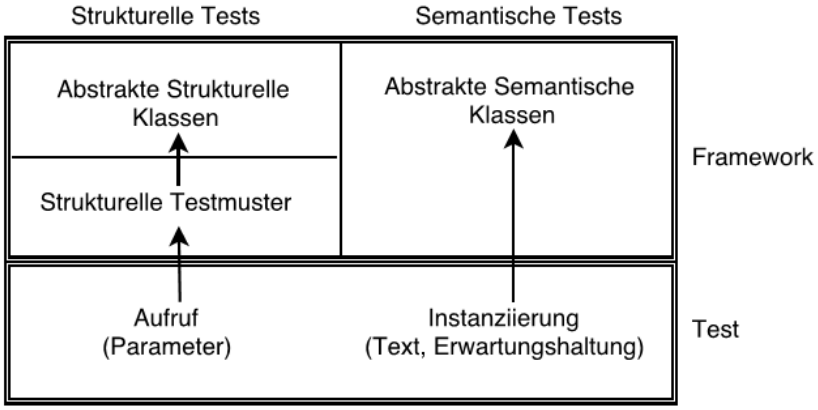

Abbildung 4.2.: Übersicht über die Nutzung des Test-Frameworks, Quelle: Eigene Abbildung

4.3. Konzeptionelles Modell der strukturellen Tests

Der kommende Abschnitt befasst sich mit dem Aufbau und der Konzeption der sogenannten strukturellen Tests, die das Ziel haben zu überprüfen, ob die vorliegende Ontologie die erwartete Struktur aufweist beziehungsweise die aufgestellten strukturellen Qualitätskriterien erfüllt. Was genau erwartet wird, hängt bei der HIP stark mit ihrem Anwendungsfall und der darauf aufbauenden momentanen Ordnung der HION zusammen. Zur Ausführung dieser Art von Tests werden eine zu überprüfende Ontologie und eine Auswahl von Testfälle erwartet, die mit den richtigen Parametern aufgerufen sein müssen. Das Ergebnis wird als eine Anzahl von Testberichten dargestellt, welche wiedergeben, ob Tests fehlgeschlagen sind oder nicht.

4.3.1. Integration einer Evaluationsmethode

Die Konzeption der strukturellen Ontologietests des Frameworks ist angelehnt an die *Test-Driven Linked-Data Quality Methodology*, die in Kapitel 3.2.1 beschrieben ist. Obwohl in dieser Methode Datenqualität überprüft wird, kann das allgemeine Vorgehen auf die Evaluation der HION übertragen werden. In Abbildung 4.3 ist die abgeleitete Konzeption zu erkennen. Auch die Benennung der einzelnen Komponenten ist an die Methode zur Messung von Datenqualität angepasst. So enstand eine Bibliothek aus sogenannten Ontology Quality Test Patterns (kurz Testmuster), die aus mehreren Informationsquellen zusammengestellt wurde. In erster Instanz wurden die Testmuster aus den Anforderungen und den strukturellen Qualitätskriterien der HION abgeleitet (Ontology Requirements). Zusätzlich dienten Ontologie Designmuster (Ontology Design Patterns) sowie Beschränkungen der genutzten Ontologiesprache als Vorbilder, solange sie auch in der HION Anwendung finden. Auch diese Muster können durch das Test Pattern Binding in konkrete Testfälle, genannt Ontology Quality Test Cases, umgewandelt werden. Diese Instanziierung bedarf Interaktion des Nutzers und gehört nicht zum eigentlichen Framework. Er stellt im Zuge des Entwicklungsprozesses Anforderungen zusammen, denen die Ontologie gerecht werden muss und die festlegen, welche Testmuster wie genutzt werden. Die Instanziierung an sich umfasst im Fall der strukturellen Tests eher einer Auswahl zum Fall passender Muster und deren Aufruf in der eigentlichen Testausführung, also in einer Test Suite.

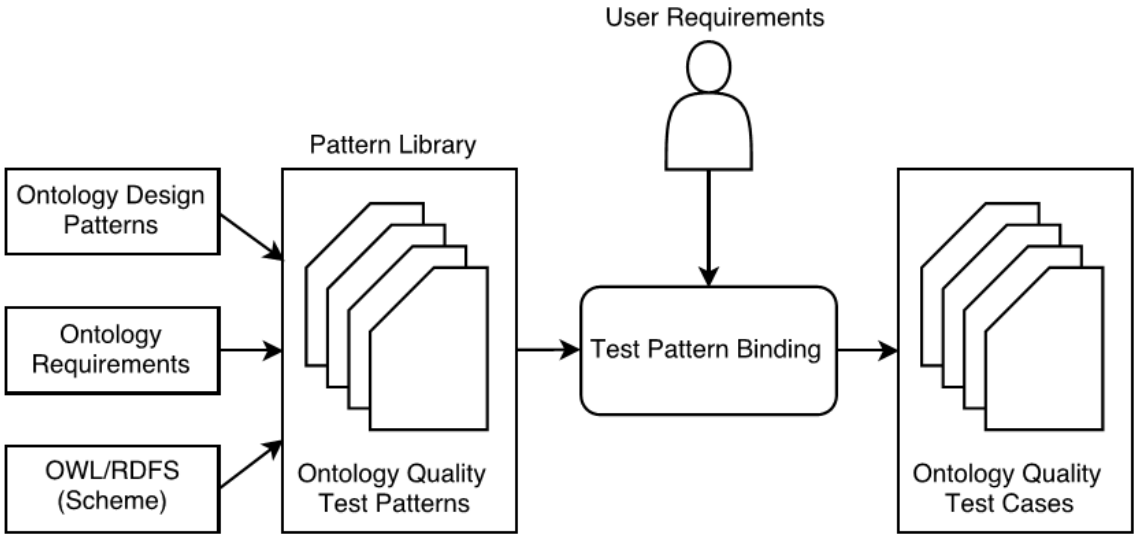

Abbildung 4.3.: Konzeption der strukturellen Ontologietests, Quelle: Eigene Abbildung

4.3.2. Identifizierung der Ontologie Testmuster-Bibliothek

Ein wichtiger Schritt in der Konzeption der strukturellen Tests war somit die Identifizierung der Testmuster-Bibliothek. Um Strukturen innerhalb der HION überprüfen zu können, wird für jede semantische Relation ein eigenes Testmuster eingeführt. Zusätzlich werden auch die wichtigen Relationen rdfs:subClassOf, :synoynm und :standard_label und ihre Eigenschaften überprüft. Hinzu kommen weitere Tests zur Überprüfung der strukturellen Korrektheit, die in der momentanen Version vor allem aus dem OQuaRE-Framework nach [Ram13] und der Test-Driven Linked-Data Quality Methodology nach [Kon14] abgeleitet wurden.
Die folgende Tabelle 4.1 zeigt alle identifizierten Muster mit einer kurzen Beschreibung und der Angabe der dafür benötigten Parameter, die für die parametrisierten SPARQL-Anfragen benötigt werden.

Test	**Beschreibung**	**Parameter**
ExistenceTest	Test, ob ein Konzept in der Ontologie vorhanden ist	Konzept
SubClassOfTest	Test, ob ein Konzept Unterklasse eines Anderen ist	Unterklassekonzept, Überklassekonzept
HatBeobachtung-Test	Test, ob die Relation HatBeobachtung zwischen zwei bestimmten Konzepten vorhanden ist	Quell-Konzept, Ziel-Konzept
HatLokalisation-Test	Test, ob die Relation HatLokalisation zwischen zwei bestimmten Konzepten vorhanden ist	Quell-Konzept, Ziel-Konzept
HatBeobachtungs-AttributTest	Test, ob die Relation HatBeobachtungsAttribut zwischen zwei bestimmten Konzepten vorhanden ist	Quell-Konzept, Ziel-Konzept
HatEinheitTest	Test, ob die Relation HatEinheit zwischen zwei bestimmten Konzepten vorhanden ist	Quell-Konzept, Ziel-Konzept
HatLaborwert-KennzeichenTest	Test, ob die Relation HatLaborwertKennzeichen zwischen zwei bestimmten Konzepten vorhanden ist	Quell-Konzept, Ziel-Konzept
HatRelevanten-BegriffTest	Test, ob die Relation HatRelevantenBegriff zwischen zwei bestimmten Konzepten vorhanden ist	Quell-Konzept, Ziel-Konzept
HatTherapie-AttributTest	Test, ob die Relation HatTherapieAttribut zwischen zwei bestimmten Konzepten vorhanden ist	Quell-Konzept, Ziel-Konzept
HatWertTest	Test, ob die Relation HatWert zwischen zwei bestimmten Konzepten vorhanden ist	Quell-Konzept, Ziel-Konzept
HatZeitpunktTest	Test, ob die Relation HatZeitpunkt zwischen zwei bestimmten Konzepten vorhanden ist	Quell-Konzept, Ziel-Konzept

Test	**Beschreibung**	**Parameter**
HatRisikofaktor-Test	Test, ob die Relation HatRisikofaktor zwischen zwei bestimmten Konzepten vorhanden ist	Quell-Konzept, Ziel-Konzept
HatTeilTest	Test, ob die Relation HatTeil zwischen zwei bestimmten Konzepten vorhanden ist	Quell-Konzept, Ziel-Konzept
DomainTest	Test, ob ein Statement die Domain-Beschränkung der Relation einhält	Quell-Konzept, Relation, Ziel-Konzept
RangeTest	Test, ob ein Statement die Range-Beschränkung der Relation einhält	Quell-Konzept, Relation, Ziel-Konzept
SynonymTest	Test, ob ein Konzept ein bestimmtes Synonym hat	Konzept, String
StandardLabelTest	Test, ob ein Konzept ein standard_Label hat	Konzept
UniqueLabelTest	Test, ob das standard_Label jedes Konzeptes einzigartig ist	keine
OneLabelTest	Test, ob jedes Konzept nicht mehr als ein standard_Label hat	keine
HierarchyTest	Test, ob die grundlegende Basishierarchie eingehalten ist	keine
RedundancyTest	Test, ob ein doppeltes Konzept gefunden wird (vergleicht Label mit Synonymen und umgekehrt)	keine
CycleTest	Test, ob Zyklen in einer Relation bestehen	Relation
BaseConceptTest	Test, ob ein Konzept Unterklasse von Base-Concept ist	Konzept
IrreflexiveTest	Test, ob eine Relation irreflexiv ist	Relation
AsymmetricTest	Test, ob eine Relation asymmetrisch ist	Relation

Test	Beschreibung	Parameter
NamespaceTest	Test, ob ein/eine Konzept/Relation in einem bestimmten Namensraum ist	Konzept/Relation, Namensraum

Tabelle 4.1.: Liste der strukturellen Testmuster mit Beschreibung und Parameter, Quelle: Eigene Darstellung

Diese Tabelle ist nicht endgültig und kann sich im Laufe der Evolution der HION und auch der Weiterentwicklung der HIP ändern. Zum momentanen Zeitpunkt enthält sie alle notwendigen Ontology Quality Test Patterns.
Jedes Testmuster enthält eine SPARQL-Abfrage, die zur eigentlichen Kommunikation mit der Ontologie genutzt wird. Die Tabelle 4.1 zeigt in der letzten Spalte an, welche Parameter für die verschiedenen Tests benötigt werden. Testmuster ohne Parameter können als allgemeine Tests angesehen werden. Diese laufen über die komplette Ontologie und überprüfen jedes Konzept. Auch die SPARQL-Abfragen dieser Tests sind nicht parametrisiert. Ein Beispiel ist der OneLabelTest, der überprüft, ob jedes Konzept nicht mehr als ein :standard_label besitzt. Er umfasst folgende SPARQL-Abfrage:

```
1 PREFIX c: <...>
2 SELECT ?a
3 WHERE
4 {
5  ?a c:standard_label ?v1.
6  ?b c:standard_label ?v2.
7  FILTER((str(?v1)!=str(?v2))&&(?a=?b))
8 }
```

Für den Fall, dass ein Konzept zwei unterschiedliche :standard_label-Relationen besitzt, wird dieses zurückgegeben. Sonst bleibt das Anfrageergebnis leer. Die anderen Tests enthalten hingegen auch parametrisierte SPARQL-Abfragen, so wie der HatLokalisationTest:

```
1 PREFIX x: <...>
2 PREFIX y: <...>
3 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4 SELECT ?a ?c
5 WHERE
6 {
7  ?b y:hatLokalisation ?a.
8  OPTIONAL {?c rdfs:subClassOf+ ?a.}
9  FILTER(?b = x:%B%)
10 }
```

Der Parameter ist hier durch %B% gekennzeichnet. Diese Anfrage gibt alle Konzepte a und deren Unterkonzepte c zurück, die in der :hatLokalisation-Relation zu einem bestimmten Konzept b stehen, welches durch das Quell-Konzept festgelegt wurde. Das Ergebnis muss im Anschluss mit dem Parameter des Ziel-Konzepts verglichen werden. Ist das Konzept in der Ergebnismenge enthalten, endet der HatLokalisationTest erfolgreich.

4.3.3. Nutzung der strukturellen Testmuster

Das Test Pattern Binding findet innerhalb der Test Suiten statt und gehört daher zur Nutzung des Frameworks. In einem konkreten Testfall wird eine neue Suite erstellt und dort die gewünschten Testmuster mit den richtigen Parametern aufgerufen. Die Suite sorgt mit der Hilfe des JUnit-Frameworks dafür, dass alle gesammelten Tests nacheinander ausgeführt werden und dass für jeden der aktuelle Status angezeigt wird.

4.4. Konzeptionelles Modell der semantischen Tests

Der nächste Abschnitt thematisiert die Konzeption der semantischen Tests innerhalb des Test-Frameworks. Mit deren Hilfe wird die funktionelle beziehungsweise semantische Korrektheit der HION überprüft. In der momentanen Ausprägung der HIP muss daher überprüft werden, ob der implementierte Texterkennungsprozess, der die Ontologie nutzt, die erwarteten Ergebnisse liefert.
Im Falle einer Erweiterung der HIP-Funktionalität kann es sein, dass es in Zukunft neben der Texterkennung weitere Anwendungsfälle gibt, die ebenfalls als semantische Tests implementiert werden müssen.

4.4.1. Integration der OntologyPipelineTests

In Kapitel 3.3 wurden die OntologyPipelineTests vorgestellt, die die Möglichkeit geben, die funktionale Korrektheit der HION zu überprüfen, in dem sie Beispielsätze durch die Linguistische Pipeline verarbeiten lassen und die Ergebnisse mit vorher definierten Erwartungshaltungen vergleichen.
Die komplette Integration dieser Tests wurde im Zuge der Arbeit nicht umgesetzt, um die Modularität des Test-Frameworks beizubehalten und die direkte Abhängigkeit von der HIP zu vermeiden. Dennoch wurde das Framework so vorbereitet, dass eine Integration in Zukunft möglich ist. Um dies zu gewährleisten, finden die Module des Analyzers und des Matchers einen Platz innerhalb des Frameworks und werden in ihrer analysierten Funktion eingesetzt. Abbildung 4.4 zeigt den Gedanken hinter der Nutzung und Integration der OntologyPipelineTests. Innerhalb der semantischen Tests werden die Module als ISemanticAnalyzer und ISemanticMatcher eingeführt, welche als Platzhalter für die spätere Integration der funktionierenden Module der OntologyPipelineTests dienen. Sie benötigen als Input einen Text und einen dazu passenden Erwartungswert und geben "true-“ oder "false-“Werte als Matching-Ergebnis zurück. Die Erwartungswerte müssen in einem sogenannten Semantischen Modell definiert sein.
Die Module werden über eine semantische Teststruktur in das Framework eingebaut, die auch den Aufbau der konkreten semantischen Testfälle über teilweise abstrakte Klassen vorgibt. Diese Struktur ist ebenfalls für die Benutzung des Semantischen Modells zuständig.

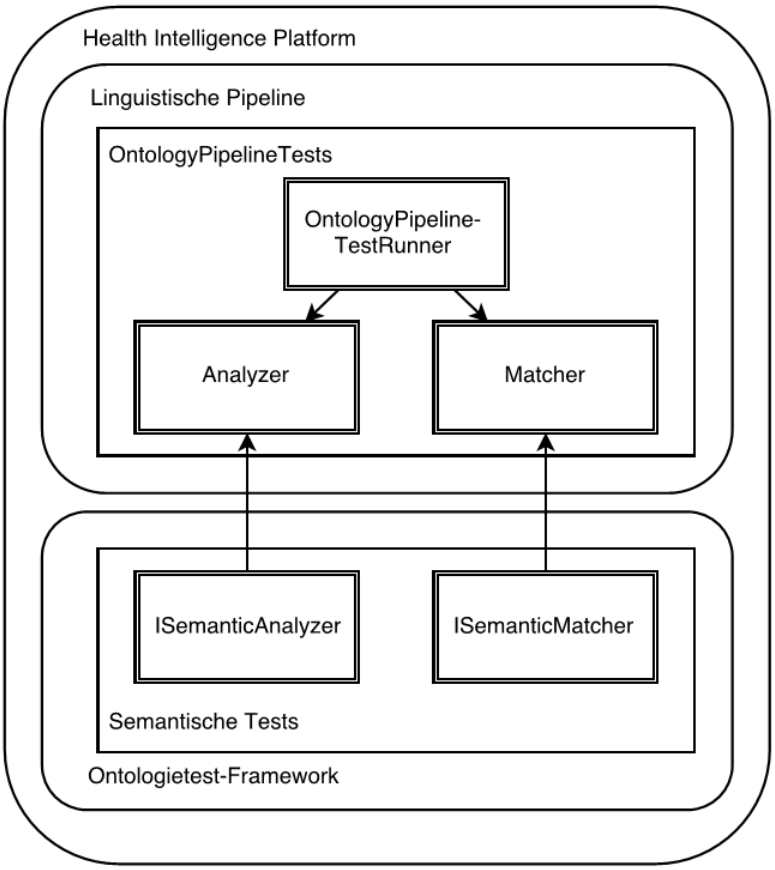

Abbildung 4.4.: Integration der Module der OntologyPipelineTests in das Test-Framework, Quelle: Eigene Abbildung

4.4.2. Aufbau des Semantischen Modells

Das Semantische Modell, welches dazu in der Lage sein muss, die Erwartungshaltungen des Nutzers darzustellen, lehnt sich an die Idee des HIP Resource Frameworks und seinem HIP Model an. Da sich dieses noch in der Entwicklung befindet und die Repräsentation der semantischen Informationen im Rahmen der Arbeit nicht die Komplexität des HRFWs benötigt, findet keine direkte Integration dessen statt.
Der Nutzer des Frameworks ist mit Hilfe des Semantischen Modells dazu in der Lage, auszudrücken, welche Fakten er im Rahmen der Texterkennung für einen bestimmten Satz erwartet. Aufgebaut wird diese Erwartungshaltung aus verschiedenen Elementen des Modells, die nach bestimmten Regeln ineinander geschachtelt werden können. Die nutzbaren Elemente sind dabei folgende:

- Konzepte,
- Relationen,
- Werte.

Einem konkreten Semantischen Modell wird immer als Erstes eine Relation als sogenannte Wurzel zugeordnet. Dieser kann ein zugehöriges Konzept oder ein Wert angehangen werden. Eine Verzweigung wird erreicht, in dem für ein Konzept Attribute definiert werden, die wiederum aus einem Relation-Konzept/Wert-Paar bestehen.
Zusätzlich wird ein Modul mit dem Namen ISemanticSerialization in das Framework eingebaut, um ein konkretes Semantisches Modell in andere RDF Serialisierungen umformen zu können und somit den Sinn des HRFWs zu unterstützen. Als Klasse wird dieses Modul im Rahmen der Arbeit jedoch nicht funktional implementiert und dient als Platzhalter für spätere Entwicklungen.

4.4.3. Nutzung der semantischen Tests

Um semantische Tests ausführen zu können, muss der Framework-Nutzer als erstes für jeden Anwendungsfall einen konkreten Test aufstellen, in dem er mit Hilfe der semantischen Teststruktur einen Testfall implementiert. Dieser umfasst unter anderem den Aufbau der Erwartungshaltung in Form eines Semantischen Modells, sowie die Angabe des Textes, der analysiert werden soll. Zusätzlich müssen an dieser Stelle der Aufruf des Analyzers und des Matchers gemockt werden. Das heißt, dass die Module festgelegte Werte zurückgeben und so die eigentlichen Funktionen nachahmen. Mit dieser Methode ist es möglich, den kompletten Einsatz der semantischen Tests im Framework darzustellen.
Die Ausführung eines solchen konkreten semantischen Tests wird ebenfalls über die Benutzung des JUnit-Frameworks ermöglicht und durch die Implementierung eines JUnit-Tests. Innerhalb der Test Suiten können die Tests aufgerufen und zusammen mit Weiteren (semantisch und strukturell) ausgeführt werden.

5. Entwicklung

Das folgende Kapitel befasst sich damit, wie das Ontologietest-Framework im Rahmen dieser Arbeit umgesetzt wurde. Es wird anhand von UML-Klassendiagrammen (Unified Modeling Language) beschrieben, welche Klassen implementiert wurden und wie sie miteinander arbeiten. Gleichzeitig soll die Nutzung und die Funktionalität des Frameworks veranschaulicht werden.
Alle Entwicklungen wurden mit Hilfe des Java SE Development Kit 8[1] und in der Entwicklungsumgebung IntelliJ IDEA 2017.1.2 der Firma Jetbrains[2] durchgeführt. Auch die UML-Klassendiagramme wurden innerhalb der IDE generiert. Zur Implementierung des Frameworks wurde ein Maven-Projekt aufgesetzt, in dem alle Entwicklungsschritte ausgeführt wurden.
Klassen, auf die sich der folgende Text näher bezieht, sind im Anhang in Kapitel A.3 vollständig dargelegt und kommentiert.

5.1. Entwicklung des Frameworks

Zunächst werden die Umsetzung der allgemeinen Struktur des Frameworks und die Klassen thematisiert, die zu dieser beitragen.

5.1.1. Struktur des Frameworks

Der erste Schritt in der Implementierung des Testrahmens war der Aufbau einer übersichtlichen und logischen Ordnerstruktur, die in Abbildung 5.1 dargestellt ist. Das eigentliche Framework wurde während seiner Entwicklung im src.main-Ordner zusammengefasst. Dort gibt es unter anderem einen test-Ordner, der wiederum in semantisch, strukturell und suite unterteilt ist. So können die abstrakten Klassen der unterschiedlichen Testarten und die strukturellen Testmuster einzeln voneinander betrachtet werden. Zusätzlich gibt es einzelne Unterordner

[1] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
[2] https://www.jetbrains.com/idea/?fromMenu

für das Semantische Modell (schema) und alle genutzten Ressourcen (tools). Dazu gehören Klassen für das Repository sowie für die Module ISemanticAnalyzer, ISemanticMatcher und ISemanticSerialization.

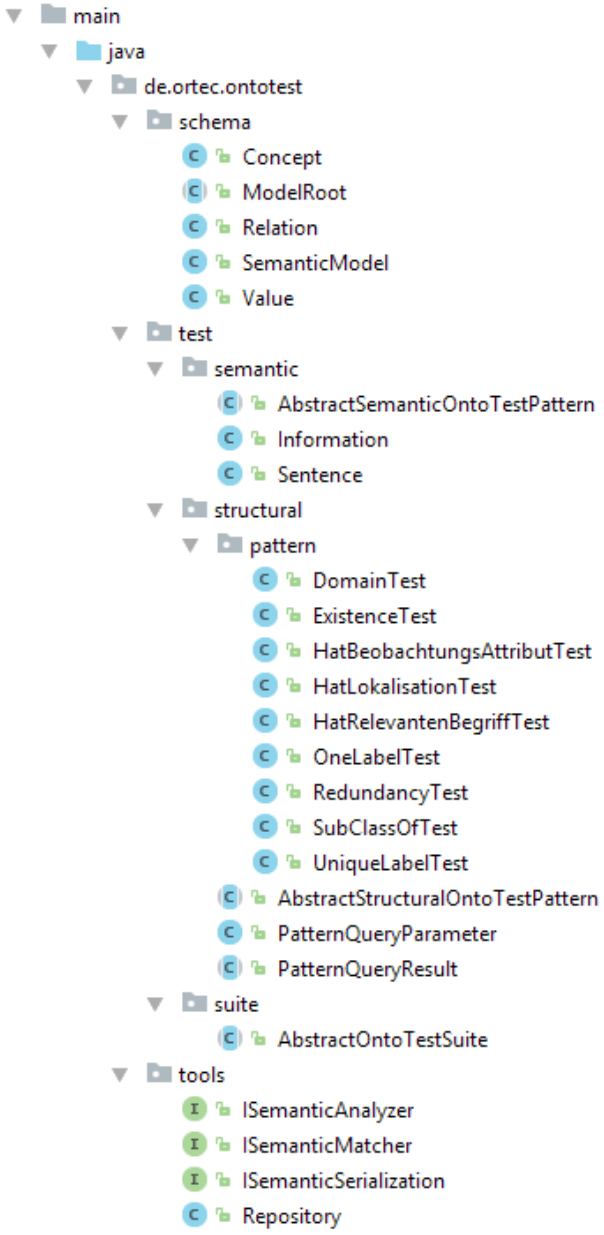

Abbildung 5.1.: Ordnerstruktur während der Entwicklung des Frameworks, Quelle: Eigene Abbildung

5.1.2. Implementierung des Repositories

Die *Repository*-Klasse wurde auf Basis der RDF4J-Bibliothek umgesetzt und bietet die Möglichkeit ein HTTP- oder In-Memory-Repository aufzusetzen und zu initialisieren. Das dazugehörige UML-Klassendiagramm ist in Abbildung 5.2 dargestellt. Es zeigt alle enthaltenen Variablen, Konstruktoren und Methoden.
Für die momentane Funktionalität des Frameworks ist einzig das In-Memory-Repository zu betrachten. Die Klasse stellt Methoden zur Verfügung, um solche Objekte zu erstellen und eine Verbindung zu ihnen zu öffnen (*initSail()*), sie mit Inhalt zu füllen (*addOntology()*) oder SPARQL-Anfragen zu stellen (*runQuery()*). Die direkte Kommunikation mit dem Repository geschieht jedoch auch dort nur über Methoden, die das RDF4J-Framework anbietet. Verwendung findet die *Repository*-Klasse erst, wenn konkrete Testfälle mit Hilfe des Frameworks erstellt und ausgeführt werden.

5.1.3. Implementierung der abstrakten Test Suite

Das Framework bietet eine abstrakte Klasse mit dem Namen *AbstractOntoTestSuite*, deren Klassendiagramm in Abbildung 5.3 zu erkennen ist. Sie wurde entwickelt, um die Implementierung konkreter Test Suiten für den Nutzer zu vereinheitlichen und zu vereinfachen. Die abstrakten Methoden sind für folgende Funktionen vorgesehen:

- *setup()*: Konfiguration der Testeigenschaften,
- *initSemanticTests()*: Initialisieren der semantischen Tests,
- *initStructuralTests()*: Initialisieren der strukturellen Tests,
- *initAll()*: Ausführen der Konfiguration und Initialisierungen.

Die *AbstractOntoTestSuite* besitzt keine Beziehungen zu anderen Klassen und ist somit unabhängig zu betrachten.

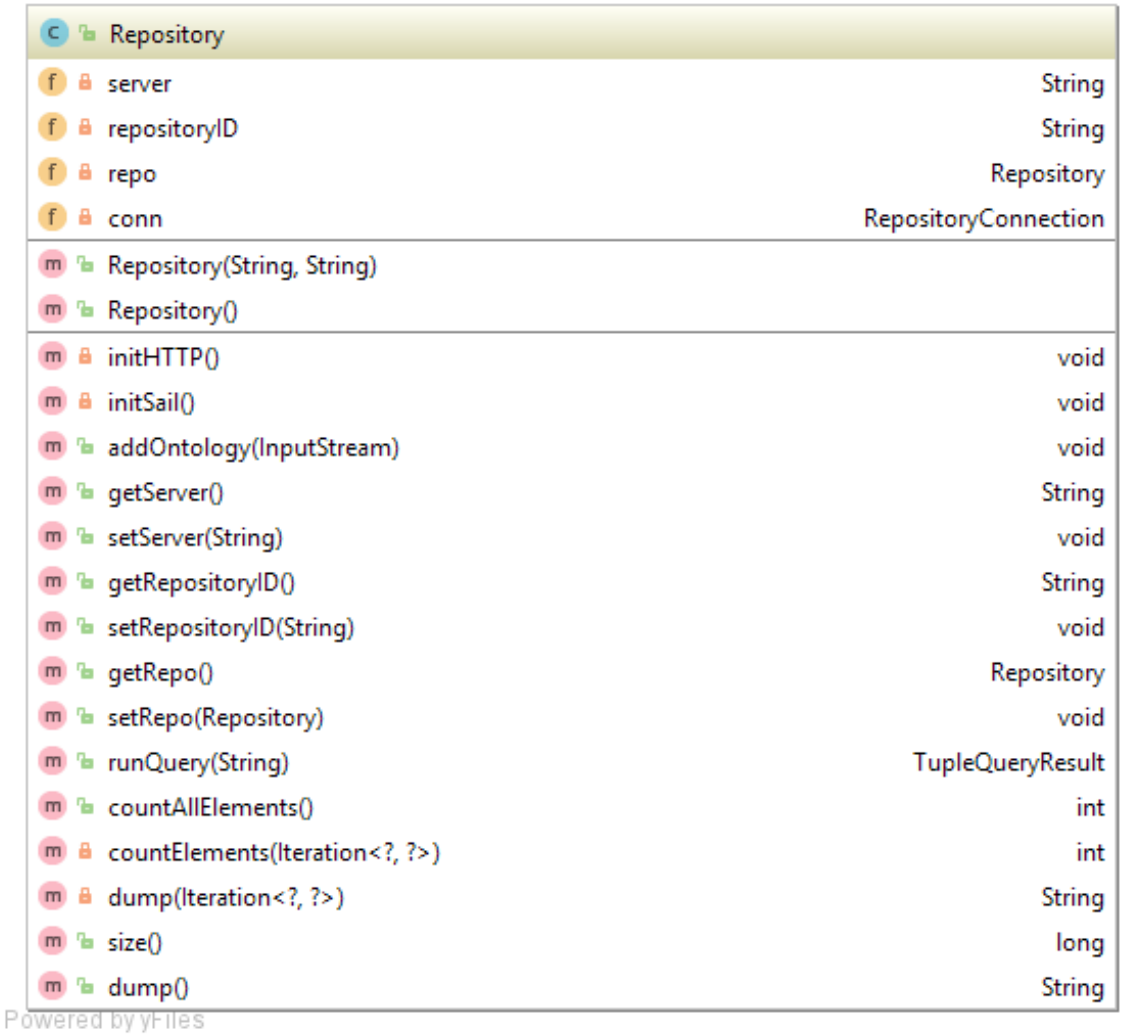

Abbildung 5.2.: Klassendiagramm der Klasse *Repository*, Quelle: Eigene Abbildung

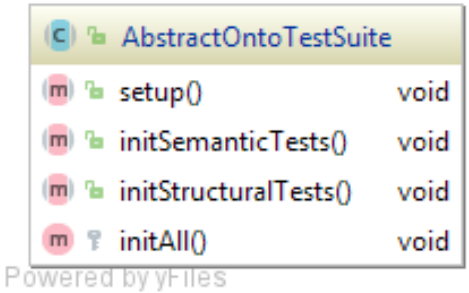

Abbildung 5.3.: Klassendiagramm der Klasse *AbstractOntoTestSuite*, Quelle: Eigene Abbildung

5.2. Entwicklung der strukturellen Tests

Einen großen Bestandteil des Frameworks nehmen die strukturellen Tests ein. Sie wurden in der Entwicklung inhaltlich, aber auch durch die Ordnerstruktur, nochmals unterteilt. Zum einen gibt es die allgemeinen strukturellen Testklassen, die den Rahmen zur Implementierung der Testmuster bieten. Zum anderen gehören auch die strukturellen Testmuster an sich zum Umfang des Frameworks.

5.2.1. Implementierung der allgemeinen strukturellen Klassen

Es wurden drei allgemeine Klassen eingeführt, um die Testmusterstruktur festzulegen und zu vereinheitlichen: *AbstractStructuralOntoTestPattern*, *PatternQueryParameter* und *PatternQueryResult*. Abbildung 5.4 zeigt die Klassen mit ihren Beziehungen untereinander. *PatternQueryParameter* ist durch eine Komposition von der Klasse *AbstractStructuralOntoTestPattern* abhängig und dient dem vereinfachten Umgang mit einer beliebigen Anzahl an Parametern, die in den Testmustern benötigt werden. Zur Umsetzung wurde ein String-Array genutzt, auf das mit verschiedenen Methoden zugegriffen werden kann. Eine Manipulation der darin enthaltenen Parameter ist nicht möglich und auch nicht erwünscht. Die abstrakte Klasse *AbstractStructuralOntoTestPattern* dient als Basis für alle Testmuster, die implementiert werden mussten und müssen. Die wichtigsten Methoden befassen sich mit der Erstellung und Ausführung der SPARQL-Anfrage (*buildQueryPatternString(), runQuery()*) und dem geordneten Ausführen dieser Schritte (*runTest()*).
PatternQueryResult ist ebenfalls abstrakt, spielt allerdings erst bei den strukturellen Mustern eine Rolle, in dem sie durch die Methode *isCorrect()* zurückgibt, ob das Ergebnis der SPARQL-Abfrage dem entspricht, was erwartet wird.

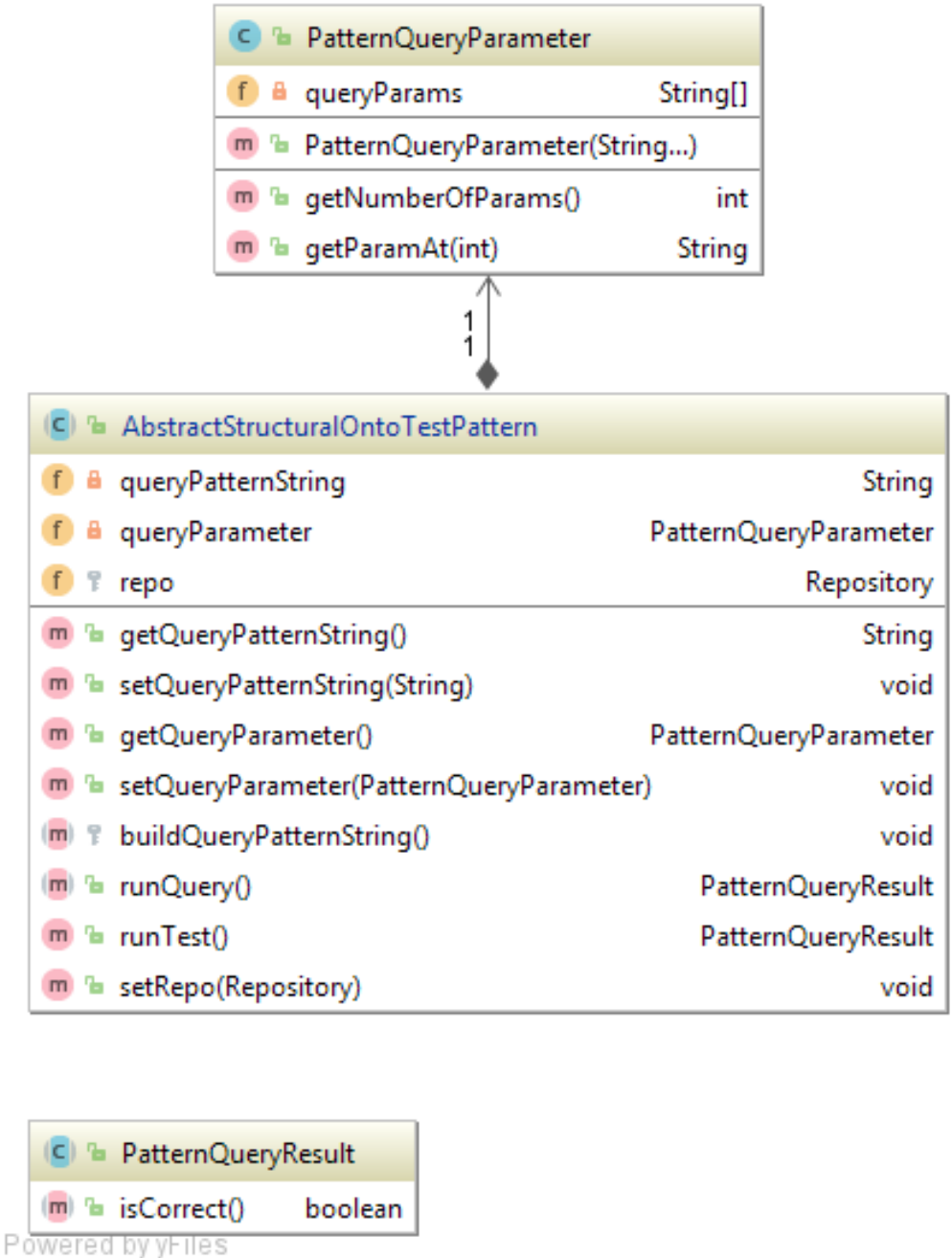

Abbildung 5.4.: Klassendiagramme der allgemeinen strukturellen Klassen, Quelle: Eigene Abbildung

5.2.2. Implementierung der Testmuster

Im Kapitel 4.3.2 wurden die Testmuster identifiziert, die im endgültigen einsetzbaren Test-Framework umgesetzt sein müssen. Im Zuge dieser Arbeit soll vor allem gezeigt werden, wie dieses Framework realisiert werden kann, weshalb auf Vollständigkeit in der Implementierung der Muster verzichtet wurde. Zur Demonstration der Umsetzung und Anwendung wurden folgende Testmuster in je einer Klasse implementiert:

- DomainTest,
- ExistenceTest,
- HatBeobachtungsAttributTest,
- HatLokalisationTest,
- HatRelevantenBegriffTest,
- OneLabelTest,
- RedundancyTest,
- SubClassOfTest,
- UniqueLabelTest.

Die Namen der Muster wurden dabei als Klassennamen übernommen. Die in Tabelle 4.1 angegebenen Parameter finden sich in den Konstruktoren der jeweiligen Muster-Klassen wieder. Nur bei Übergabe der benötigten Argumente kann ein Objekt einer dieser Klassen erstellt werden.
Wie bereits erwähnt, erben alle Testmuster-Klassen von *AbstractStructuralOntoTestPattern* und überschreiben somit die abstrakten Methoden. Die wichtigsten Unterscheidungen innerhalb der Muster-Klassen liegen in der Formulierung der SPARQL-Abfrage und der Verarbeitung des Anfrageergebnisses.
Nachfolgend wird anhand der Muster-Klasse *HatLokalisationTest* näher erläutert, wie ein struktureller Test aufgebaut ist. Abbildung 5.5 zeigt das Klassendiagramm des Musters, sowie die innere Klasse *HatLokalisationTestResult*, die von *PatternQueryResult* erbt.

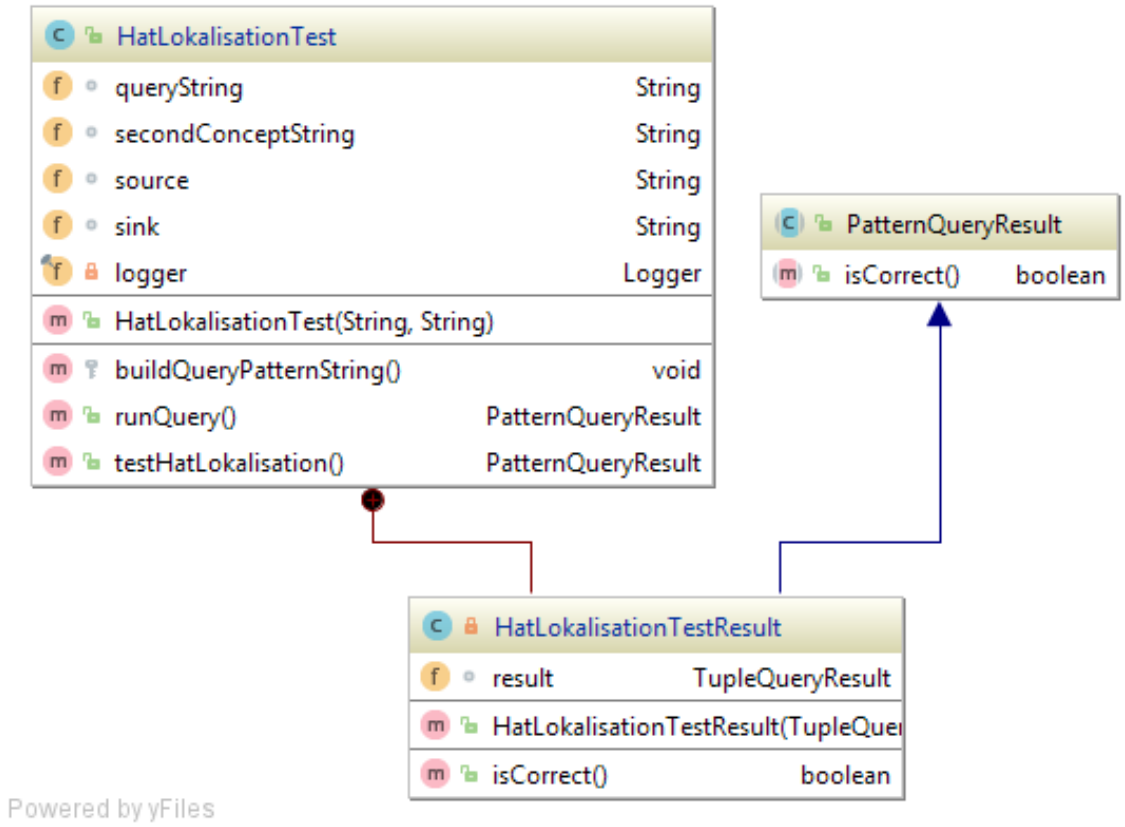

Abbildung 5.5.: Klassendiagramm der Klassen *HatLokalisationTest*, *HatLokalisationTestResult* und *PatternQueryResult*, Quelle: Eigene Abbildung

Die Muster-Klasse verfolgt die Aufgabe, zu überprüfen, ob zwischen zwei bestimmten Konzepten die HION-Relation :hatLokalisation besteht. Die beiden Konzepte werden über den Konstruktor übergeben und in Form eines *PatternQueryParameter*s in der Methode *buildQueryPatternString()* zur Generierung einer konkreten SPARQL-Anfrage genutzt. Diese wird bei jeder Testausführung aus einem parametrisierten String zusammengebaut. Im Falle der *HatLokalisationTest*-Klasse sieht dieser String folgendermaßen aus:

```
1  queryString = "
2  PREFIX x: <" + namespace + ">
3  PREFIX y: <http://ontology.hip-ontoTesting.de/core#>
4  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
5  SELECT ?a ?c
6   WHERE {
7    ?b y:hatLokalisation ?a.
8    OPTIONAL {?c rdfs:subClassOf+ ?a.}
9    FILTER(?b = x:" + concept + ")
10   }
11 ";
```

Die hier enthaltenen Variablen *namespace* (in Zeile 2) und *concept* (in Zeile 9) werden bei einer Testausführung durch String-Werte ersetzt.

5.3. Entwicklung der semantischen Tests

Der zweite große Teil des Test-Frameworks sind die semantischen Tests, deren Entwicklung in diesem Abschnitt näher betrachtet wird. Auch hier wurde nochmal eine Unterscheidung zwischen allgemeinen semantischen Klassen, den OntologyPipelineTest-Modulen und dem Semantischen Modell vorgenommen.

5.3.1. Implementierung der allgemeinen semantischen Klassen

In Abbildung 5.6 sind sowohl die Klassen der OntologyPipelineTest-Module als auch die allgemeinen semantischen Klassen dargestellt, um die Beziehungen untereinander zu verdeutlichen. Die abstrakte Klasse *AbstractSemanticOntoTestPattern* gibt den Rahmen für die spätere Implementierung konkreter semantischer Tests vor, während die Klassen *Sentence* und *Information* die Kommunikation mit *ISemanticMatcher* und *ISemanticAnalyzer* übernehmen.
Die abstrakte Oberklasse verbindet *Sentence*, *Information* und das Semantische Modell und besitzt eine abstrakte Methode *buildsemanticModel()*, in der dieses Modell in einem konkreten semantischen Testfall definiert werden muss. Eine weitere wichtige Methode ist *evaluate()*. In dieser wird entschieden, ob die von der Linguistischen Pipeline generierten Fakten mit der Erwartungshaltung aus dem Semantischen Modell übereinstimmen. Auch hier gibt es wieder eine Methode *runTest()*, die dafür sorgt, dass alle nötigen Schritte des semantischen Tests in der richtigen Reihenfolge ausgeführt werden.

5.3.2. Implementierung der OntologyPipelineTest-Module

Die Klassen, in denen die Module umgesetzt wurden, erhielten die gleichen Namen. Sowohl der *ISemanticAnalyzer* als auch der *ISemanticMatcher* implementieren eine Methode, die jedoch in beiden Fällen keine Funktion besitzt. Die Aufrufe genau dieser Methoden werden bei der Implementierung konkreter Semantiktests gemockt. Trotzdem nehmen sie wichtige Aufgaben ein. Die Methode *analyze()* des *ISemanticAnalyzers* erhält ein Objekt der Klasse *Sentence* und gibt eine *Information* als Ergebnis zurück, die die generierten Fakten enthält. Dieses *Information*-Objekt kann im Anschluss über den *ISemanticMatcher* mit einem Semantischen Modell verglichen werden.
Verwaltet wird die Ansteuerung der Klassen, wie schon erwähnt, von *AbstractSemanticOntoTestPattern*.

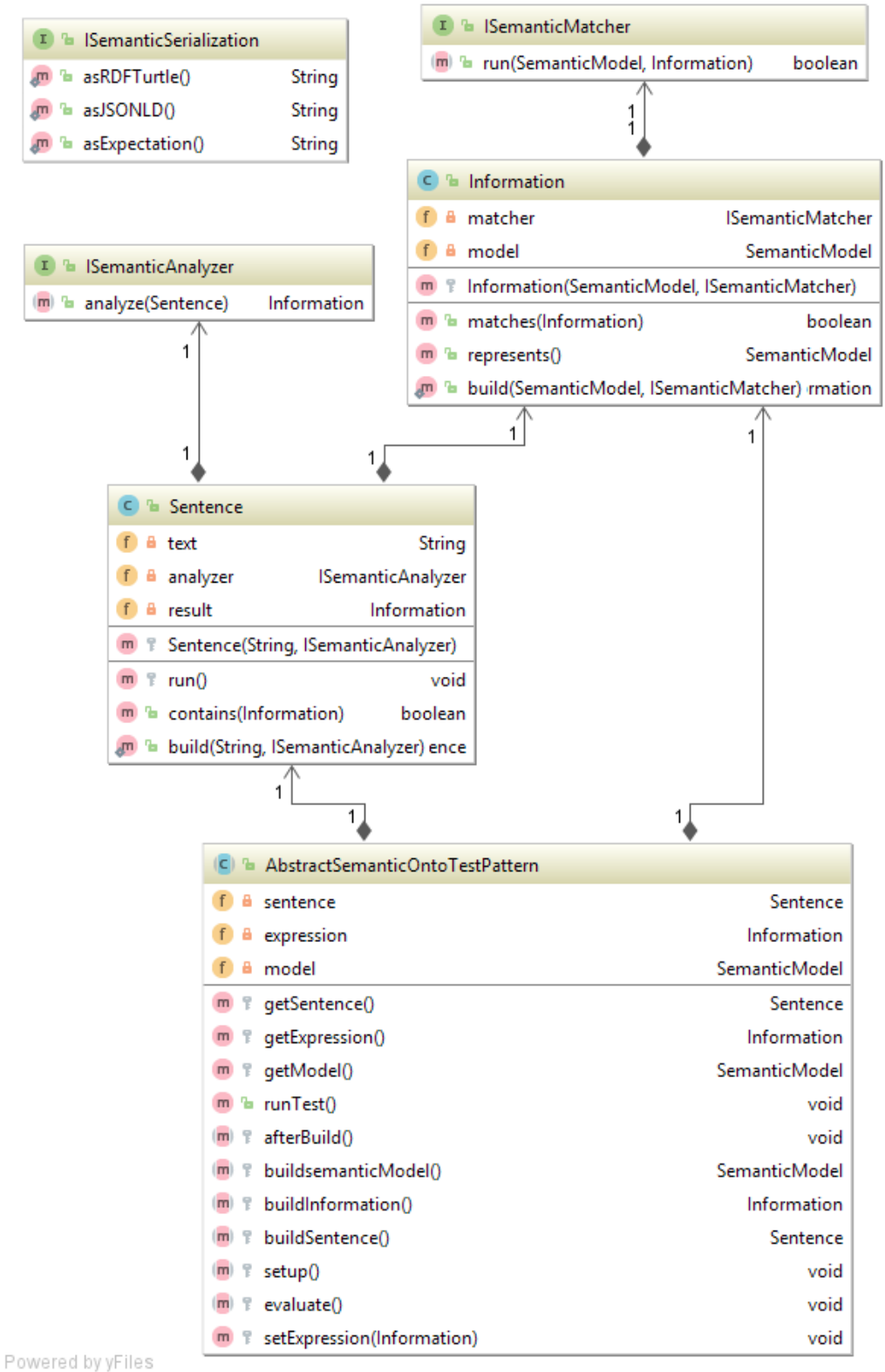

Abbildung 5.6.: Klassendiagramm der allgemeinen semantischen Klassen, Quelle: Eigene Abbildung

5.3.3. Implementierung des Semantischen Modells

Die Entwicklung des Semantischen Modells ist in Form von Klassendiagrammen in Abbildung 5.7 beschrieben und wurde durch fünf Klassen umgesetzt:

- *Concept*,
- *Relation*,
- *Value*,
- *ModelRoot*,
- *SemanticModel*.

Die ersten drei leiten sich von den konzeptionellen Elementen des Semantischen Modells ab. Die Klasse *ModelRoot* enthält die Wurzel des Modells und in *SemanticModel* wird die komplette Formulierung zusammengefasst.
Die Klasse *ISerialization* wird aufgrund ihrer Anlehnung an das HRFW ebenfalls in diesem Abschnitt betrachtet. Für sie sind Methoden vorgesehen, um ein Semantisches Modell in andere RDF-Serialisierungen, wie Turtle, JSON-LD[3] oder in das Erwartungshaltungsformat der Linguistischen Pipeline, zu überführen. Momentan sind diese Funktionen allerdings noch nicht implementiert.

[3] http://json-ld.org/

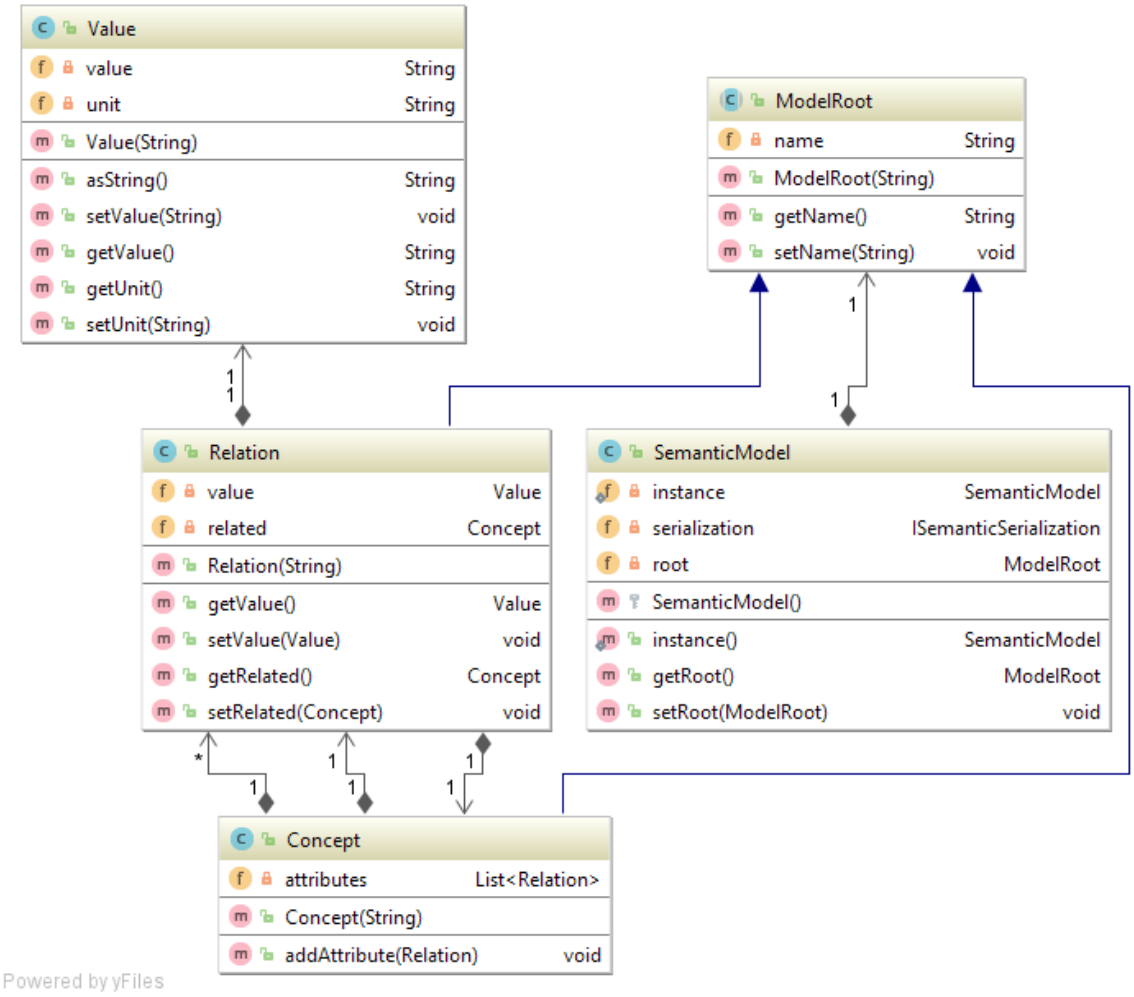

Abbildung 5.7.: Klassendiagramm des Semantischen Modells, Quelle: Eigene Abbildung

5.4. Generierung des Frameworks zur Nutzung innerhalb der HIP

Um das entwickelte Framework auch als solches Nutzen zu können, musste es schlussendlich in eine Form gebracht werden, die einfach in andere Projekte integriert werden kann. Der Zweck besteht schließlich darin, den Code und somit die Möglichkeiten zu verwenden, die das Framework anbietet, ohne alle Klassen direkt in das Projekt einbinden zu müssen. Mit Hilfe von Maven wurde das Framework-Projekt gebaut und in eine JAR-Datei mit dem Namen *hip-ontologietests-1.0-SNAPSHOT.jar* überführt. Diese kann jederzeit in ein anderes Projekt, in dem es nötig ist, Ontologietests durchzuführen, eingebunden werden.
Dazu muss die Datei als Erstes in das lokale Maven-Repository des Projektes installiert werden. Um dies durchzuführen, wird über die Kommandozeile der folgende Maven-Befehl ausgeführt:

```
1 mvn org.apache.maven.plugins:maven-install-plugin:2.5:install-
    file -Dfile=hip-ontologietests-1.0-SNAPSHOT.jar
```

Anschließend kann die Datei über die Entwicklungsumgebung als Bibliothek eingebunden und verwendet werden. In Zukunft, wenn das Framework auch während der Laufzeit der HIP eingesetzt wird, befindet es sich frei zugänglich im dazugehörigen Repository und muss nicht mehr installiert werden.
Für den Fall, dass Änderungen am Framework nötig werden, muss im Anschluss eine neue JAR-Datei gebaut und in die abhängigen Projekte eingebunden werden.

6. Test des Frameworks

Während der Entwicklung des Frameworks wurden alle Schritte durch Tests überwacht. So entstanden zeitgleich zu den Framework-Klassen ausführbare Testklassen, mit denen zum Beispiel überprüft werden konnte, ob die strukturellen Testmuster eingebaute Fehler finden. Im Laufe der Entwicklung wurden deshalb immer wieder absichtlich strukturelle Fehler in die Testontologien eingebaut. Die endgültige Ordnerstruktur dieser Klassen ist in Abbildung 6.1 zu erkennen. Es wurden vier verschiedene semantische Tests instanziiert, die in drei Test Suiten zusammen mit strukturellen Tests über JUnit ausgeführt werden können. Innerhalb der Entwicklungsumgebung IntelliJ geschieht dies zum Beispiel über den Befehl *Run 'BrustengeSuiteTest'*.
Diese konkreten Tests sind nicht Teil des Frameworks und wurden aufgrund dessen beim Bau der JAR-Datei nicht miteinbezogen.

6.1. Implementierung einer Testontologie

Um überhaupt Tests ausführen zu können, wurde eine Ontologie gebraucht, die in das Repository geladen und überprüft werden kann. Da die momentan zugänglichen Ontologie-Versionen der HIP für diesen Test-Anwendungsfall zu umfangreich sind, ihre Eigenschaften allerdings trotzdem benötigt werden, wurde eine Testontologie aus einer Teilmenge einer HION zusammengestellt. Die wichtigsten Konzepte und Hierarchien, wie sie in Kapitel 3.1.1 beschrieben wurden, sind in der Testontologie zusammengefasst. Zusätzlich enthält sie noch einige andere Konzepte, die zur Abbildung der Tests nötig waren. Auch hier wurden mehrere Ontologiedateien angelegt, wie sie in Abbildung 6.1 im resources-Ordner zu erkennen sind.

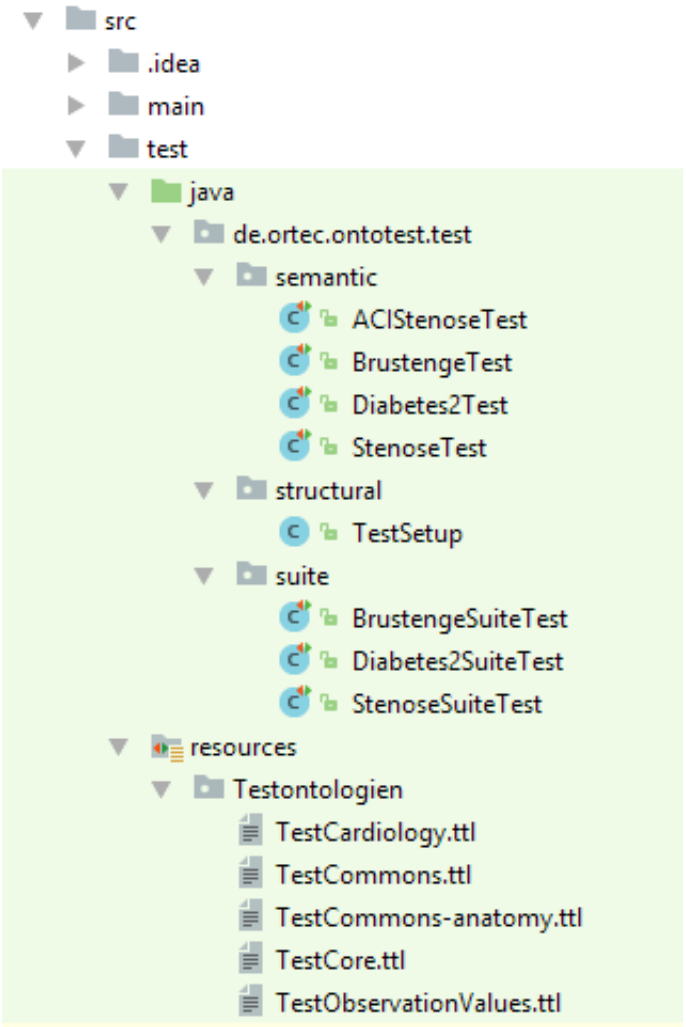

Abbildung 6.1.: Ordnerstruktur der Tests, Quelle: Eigene Abbildung

Um ein Objekt der Klasse *Repository* nutzen und darauf Tests laufen lassen zu können, muss es in einem konkreten Setup erzeugt werden. Diese Aufgabe übernimmt die Klasse *TestSetup*. Sie sorgt dafür, dass ein In-Memory-Repository erstellt und mit dem Inhalt der richtigen Ontologiedateien gefüllt wird.
In Zukunft und innerhalb der SWB wird es nicht mehr nötig sein, das Repository manuell über eine Java Klasse zu füllen. Für die eigentliche Entwicklung des Frameworks und seine vorübergehenden Einsatzmöglichkeiten ist es allerdings die beste Methode, ohne eine Abhängigkeit von der HIP zu schaffen.

6.2. Implementierung konkreter Testfälle

Jeder semantische Testfall erbt von der Klasse *AbstractSemanticOntoTestPatterns* und wird daher auf Grundlage der Framework-Klassen gleich aufgebaut. Abbildung 6.2 zeigt beispielhaft das Klassendiagramm der semantischen Testklasse *BrustengeTest*. Die Methode *testSemantic()* ist als JUnit-Test annotiert und spielt somit die entscheidende Rolle, da sie entweder direkt oder über eine Test Suite aufgerufen und somit ausgeführt werden kann. Anhand der Methode *afterBuild()* ist zu erkennen, dass das Mocken der OntologyPipelineTest-Module über das Framework Mockito[1] umgesetzt wurde. Die restlichen Methoden wurden aus der abstrakten Oberklasse überschrieben. An dieser Stelle ist auch die Nutzung der Module ISemanticAnalyzer und ISemanticMatcher deutlich zu erkennen.

[1] http://site.mockito.org/

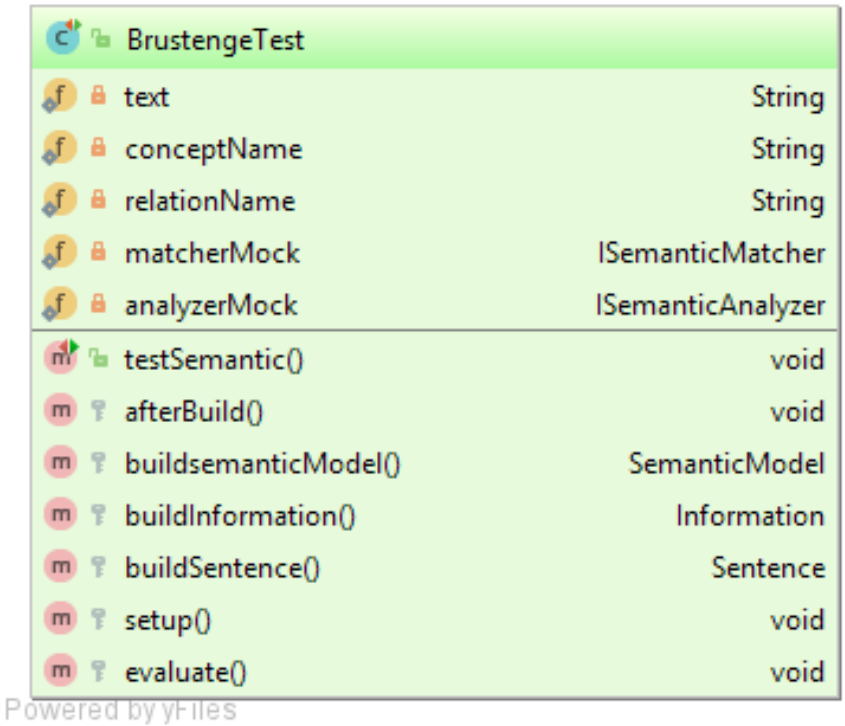

Abbildung 6.2.: Klassendiagramm des konkreten semantischen Testfalles *BrustengeTest*, Quelle: Eigene Abbildung

6.3. Implementierung von Test Suiten

Test Suiten bilden den Mittelpunkt der Testausführung, da sie die semantischen und strukturellen Tests sammeln und die gebündelte Ausführung ermöglichen. Während die semantischen Testklassen, wie beispielsweise *BrustengeTest*, fertig ausführbar außerhalb der Suiten liegen und nur aufgerufen werden müssen, ist die Einbindung der strukturellen Tests etwas umfangreicher. Als Erstes wird ein Objekt des gewünschten Testmusters mit Angabe der notwendigen Parameter erstellt, um anschließend über den Aufruf der Methode *isCorrect()* festzustellen, ob die Überprüfung des jeweiligen Musters erfolgreich war.
In Abbildung 6.3 ist als Beispiel die konkrete Test Suite *BrustengeSuiteTest* mit den Beziehungen zu anderen Klassen dargestellt. Die zu erkennenden create-Beziehungen zeigen an, welche strukturellen und semantischen Tests von dieser Suite ausgeführt werden.

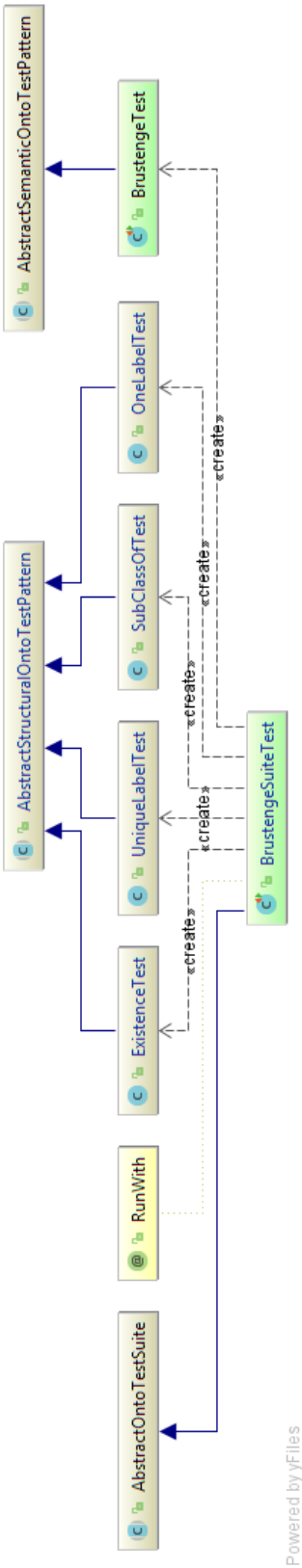

Abbildung 6.3.: Klassendiagramm der Test Suite *BrustengeSuiteTest*, Quelle: Eigene Abbildung

Die wichtigste Methode in der Klasse *BrustengeSuiteTest* ist *constructBrustengeSuite()*. Diese wird durch JUnit ausgeführt und sorgt dafür, dass alle definierten Tests abgearbeitet werden. Im Anschluss erscheint eine Ausgabe, die ebenfalls über JUnit dargestellt wird und die Status der einzelnen Tests anzeigt. In Abbildung 6.4 ist eine mögliche Ausgabe der *BrustengeSuiteTest* dargestellt, für den Fall, dass kein Test fehlschlägt. Andernfalls würde anstelle des grünen OK-Kreises ein orangener Kreis mit einem Ausrufezeichen zu erkennen sein.

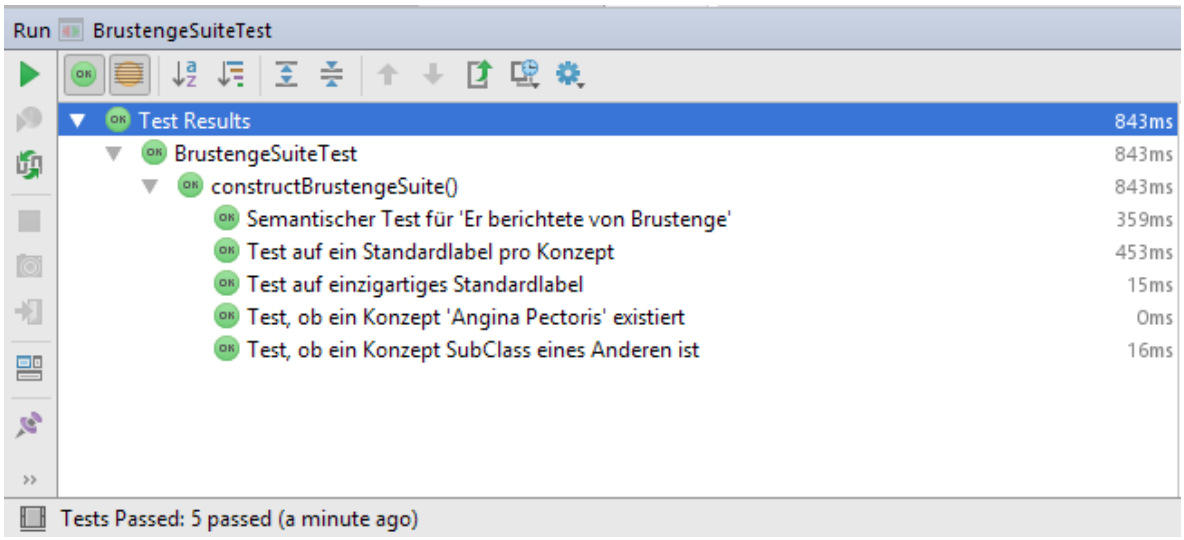

Abbildung 6.4.: Screenshot der Ausgabe der *BrustengeSuiteTest*, Quelle: Eigene Abbildung

7. Zusammenfassung

Ziel dieser Arbeit war es, eine Systematik zu erarbeiten, um im Entwicklungsprozess und während der Laufzeit der HIP möglichst einfach Ontologietests erstellen und ausführen zu können, und daraus ein Framework zu entwickeln.
Nachdem der Aufbau der HION analysiert sowie an sie gestellte Anforderungen zusammengetragen waren, konnte eine Liste von Ontologie-Qualitätskriterien aufgestellt werden, die auch durch andere Ansätze motiviert wurde. Strukturelle und funktionale/semantische Korrektheit wurden als Kriterien identifiziert, die regelmäßig durch Tests geprüft werden müssen. Auf dieser Grundlage wurde die Test-Driven Linked-Data Quality Methodology nach [Kon14] als Vorlage verwendet, um daraus die Testsystematik zur Überprüfung der strukturellen Korrektheit abzuleiten. Die Kernidee umfasst eine Testmuster-Bibliothek, die parametrisierte SPARQL-Anfragen enthalten und daher auf unterschiedliche konkrete Testfälle anwendbar sind. Für die semantischen Tests wurden in der HIP genutzte OntologyPipelineTests identifiziert, die anhand eines Erwartungswertes überprüfen, ob die Linguistische Pipeline bei der Textverarbeitung in Zusammenarbeit mit der Ontologie die richtigen Fakten erkennt. Die Integration dieser Tests wurde im Rahmen der Arbeit vorbereitet.
Auf Grundlage der entstandenen Systematik wurde das Ontologietest-Framework entwickelt und kann in Form einer JAR-Datei in andere Entwicklungsprojekte eingebunden werden. Die Nutzung während der Laufzeit soll in der Umgebung der SWB stattfinden, zu der es momentan noch keine ausreichende Planung gibt.
Das Framework stellt Klassen zur Verfügung, mit deren Hilfe konkrete Ontologietests zur strukturellen und semantischen Überprüfung implementiert und über JUnit ausgeführt werden können.

7.1. Ausblick

Das entstandene Test-Framework bildet einen ersten Ansatz zur Entstehung automatisierter oder halb-automatisierter Ontologietests in der HIP. Bis dahin gibt es allerdings noch einige Verbesserungen, die umgesetzt werden müssen. Auch im jetzigen Zustand bietet das Framework aufgrund seiner Strukturen und abstrakten Klassen eine Erleichterung in der Testgenerierung. Um die Verwendung innerhalb der HIP zu ermöglichen, muss als Erstes die Integration in den HIP-Entwicklungsprozess vollzogen und der Anschluss an die OntologyPipelineTests vorgenommen werden. Zusätzlich sollten Maßnahmen ergriffen werden, um das HRFW einschließlich des Serialisierungs-Moduls in den semantischen Tests endgültig umzusetzen.
Im Bereich der strukturellen Tests müssen die restlichen Testmuster aus Tabelle 4.1 implementiert werden, um eine vollständige Abdeckung nach momentanem Kenntnisstand zu gewährleisten.
Die Nutzung des Frameworks und vor allem der Prozess der Testgenerierung ist definitiv ausbaufähig. Im Umgang mit den strukturellen Tests hat der Nutzer mit der Auswahl der Testmuster und ihrem parametrisierten Aufruf zu viel Aufwand, als dass er diese Aufgaben nach jeder Ontologieänderung abarbeiten würde. Durch die Integration in die Entwicklungsumgebung soll zukünftig eine automatisierte Testgenerierung über die formulierten JBehave-Stories umgesetzt werden. Auch die Instanziierung der semantischen Tests soll über die Entwicklungsumgebung vereinfacht werden, zum Beispiel in dem aus Nutzeranforderungen automatisch Semantische Modelle generiert werden.
Für die Zukunft muss auch das Framework immer wieder an die System- und Ontologieanforderungen angepasst werden und selbst flexibel bleiben.

7.2. Schlusswort

Abschließend kann festgehalten werden, dass das Ziel dieser Arbeit, die Entwicklung eines Ontologietest-Frameworks, erreicht wurde. Die entstandene JAR-Datei kann in Entwicklungsprojekte eingebunden werden und vereinfacht die Implementierung von konkreten Ontologietests. Trotzdem besitzt das Framework noch viel Potenzial und muss an vielen Stellen weiterentwickelt werden, um als endgültige Lösung der Qualitätssicherungsproblematik von Ontologien in der HIP eingesetzt werden zu können.

A. Anhang

A.1. Tabellen

Kriterium	**Unterkriterium**	**Beschreibung**
Struktur	Formalisierung	Fähigkeit, Schlussfolgerungen zu unterstützen
	Unterstützung formaler Relationen	Einsatz von formalen Relationen
	Redundanz	Vermeidung von Wiederholungen
	Strukturelle Genauigkeit	Einhaltung von z.B. Domain und Range Beschränkungen
	Konsistenz (logische, strukturelle)	Grad der Konsistenz
	Tangledness	Maß für Polyhierarchie
	Zyklen	Existenz von Zyklen in semantischen Relationen
	Kohäsion	Maß für den Zusammenhang von Klassen
	Domänenabdeckung	Maß für die Domänenabdeckung
Funktionale Angemessenheit	Referenzontologie	Ob als Referenzontologie einsetzbar
	Kontrolliertes Vokabular	Fähigkeit, Heterogenität von Begriffen zu vermeiden

Kriterium	**Unterkriterium**	**Beschreibung**
	Schema- und Werteabstimmung	Präsenz eines Datenmodells zur Integration
	Wissensakquise und -repräsentation	Fähigkeit, das erworbene Wissen darzustellen
Kompatibilität	Ersetzbarkeit	Grad, in dem die Ontologie für eine andere zum gleichen Zweck eingesetzt werden kann
	Interoperabilität	Grad, in dem eine Ontologie mit einer anderen kombiniert genutzt werden kann
Übertragbarkeit	Portabilität	Grad, in dem eine Ontologie in einer anderen Hardware- oder Softwareumgebung genutzt werden kann
	Anpassungsfähigkeit	Grad, in dem eine Ontologie an eine andere Umgebung anpassen kann
Funktionsfähigkeit	Angemessenheit	Grad, in dem eine Ontologie es ihren Nutzern ermöglicht, zu erkennen, ob sie angemessen für ihre Bedürfnisse ist
Wartbarkeit	Modularität	Grad, in dem eine Ontologie in diskrete Komponenten eingeteilt ist
	Veränderbarkeit	Einfachheit der Änderungen an einer Ontologie
	Testbarkeit	Einfachheit der Validierung einer Ontologie
Nutzungsqualität	Verwendbarkeit	Effektivität und Effizienz bei der Nutzung einer Ontologie

Tabelle A.1.: Übersicht über ausgewählte Qualitätskriterien des OQuaRE Frameworks, Quelle: Angelehnt an [Duq11] und [Ram13]

A.2. Abbildungen

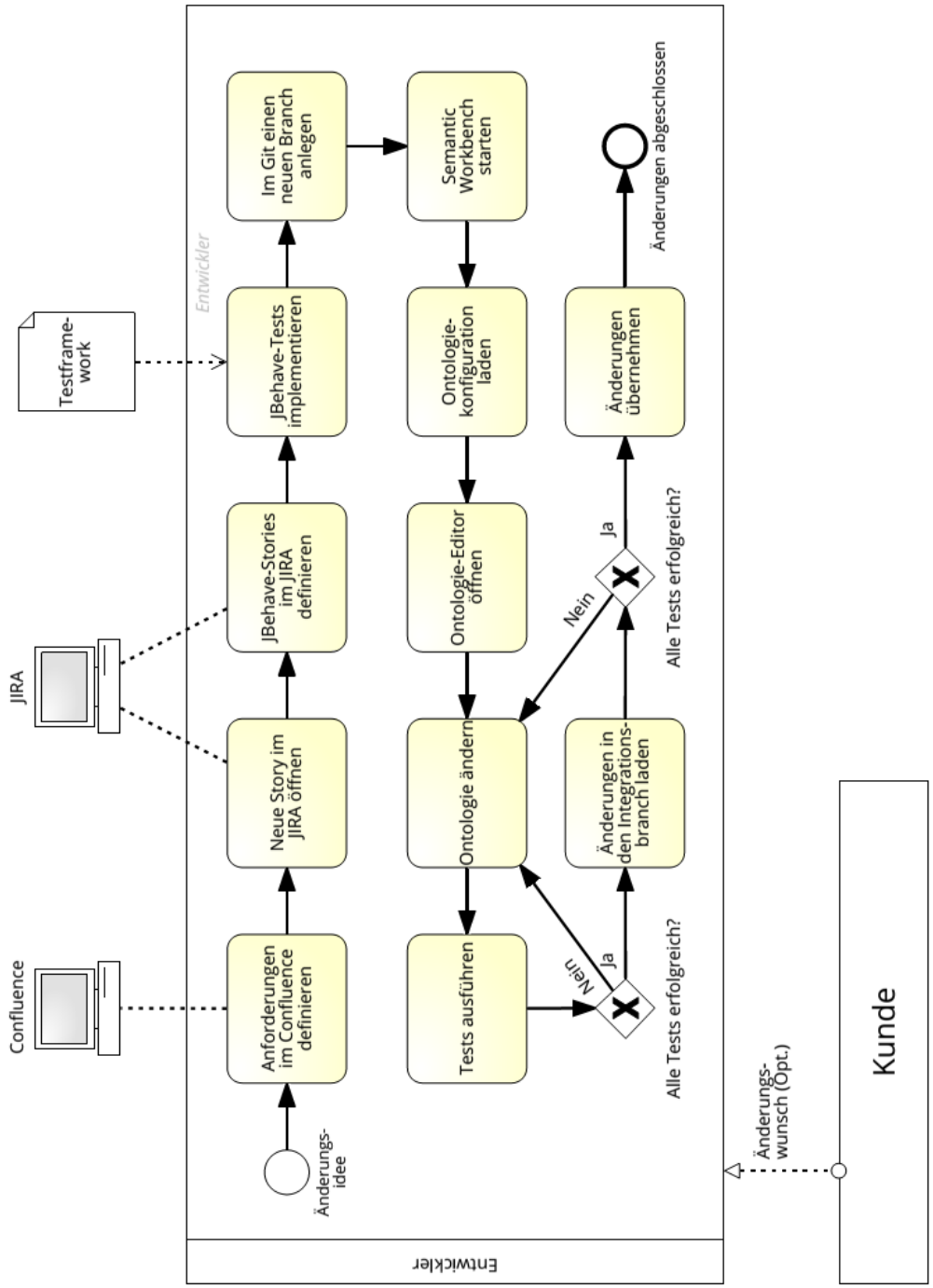

Abbildung A.1.: Prozessmodell für die Ontologieevolution im Entwicklungsprozess, Quelle: Eigene Abbildung

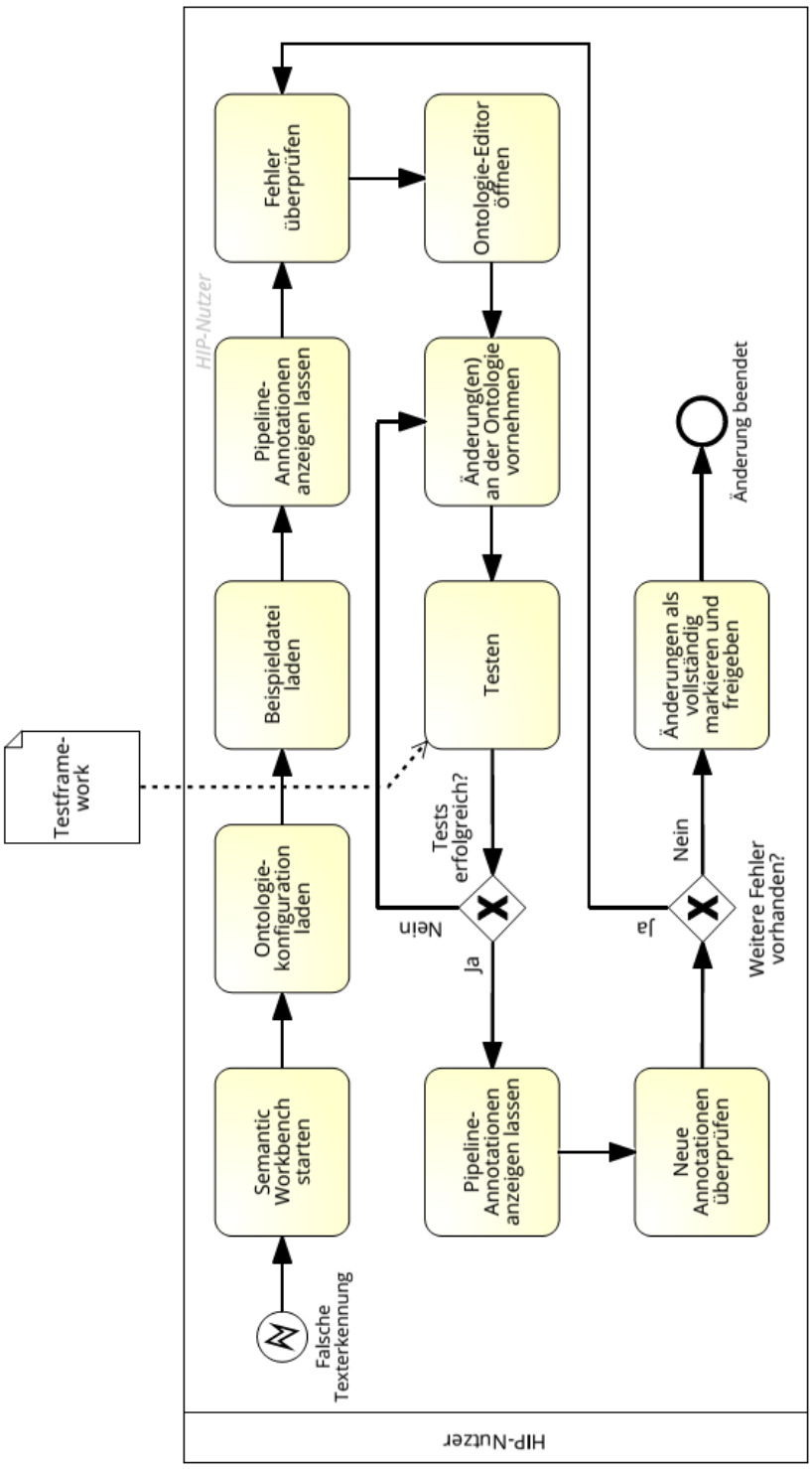

Abbildung A.2.: Prozessmodell für die Ontologieevolution im laufenden Betrieb durch Fehlerbeseitigung, Quelle: Eigene Abbildung

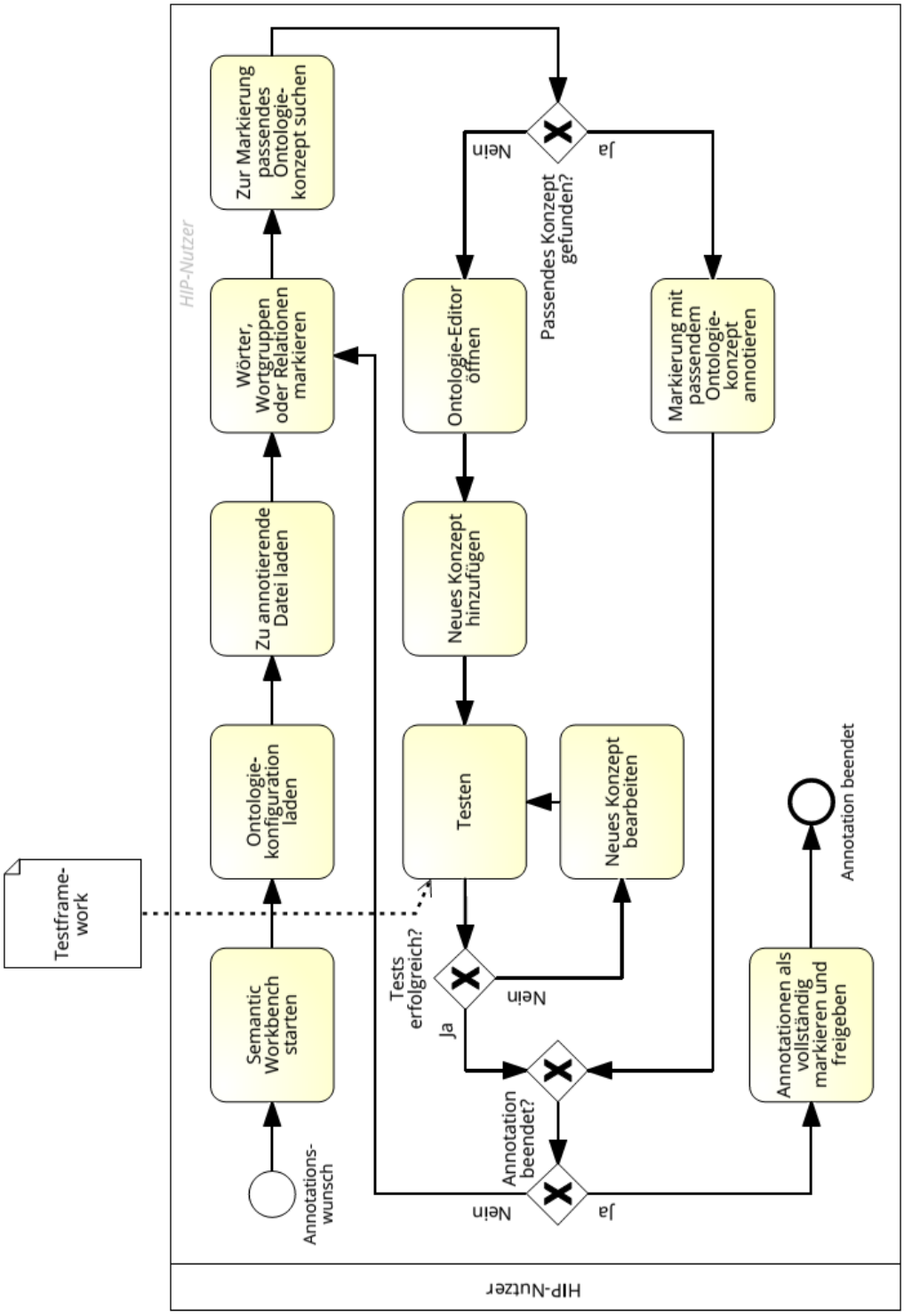

Abbildung A.3.: Prozessmodell für die Ontologieevolution im laufenden Betrieb durch Textannotation, Quelle: Eigene Abbildung

A.3. Programmiercode

Dieses Kapitel enthält einzelne Framework- und Testklassen, auf die im Text Bezug genommen wurden. Die Reihenfolge entspricht dem Vorkommen.

A.3.1. Repository.java

```
1  package de.ortec.ontotest.tools;
2  import org.eclipse.rdf4j.common.iteration.Iteration;
3  import org.eclipse.rdf4j.common.iteration.Iterations;
4  import org.eclipse.rdf4j.query.QueryLanguage;
5  import org.eclipse.rdf4j.query.TupleQuery;
6  import org.eclipse.rdf4j.query.TupleQueryResult;
7  import org.eclipse.rdf4j.repository.RepositoryConnection;
8  import org.eclipse.rdf4j.repository.RepositoryException;
9  import org.eclipse.rdf4j.repository.http.HTTPRepository;
10 import org.eclipse.rdf4j.repository.sail.SailRepository;
11 import org.eclipse.rdf4j.rio.RDFFormat;
12 import org.eclipse.rdf4j.rio.RDFParseException;
13 import org.eclipse.rdf4j.rio.UnsupportedRDFFormatException;
14 import org.eclipse.rdf4j.sail.SailException;
15 import org.eclipse.rdf4j.sail.memory.MemoryStore;
16 import java.io.IOException;
17 import java.io.InputStream;
18
19 public class Repository {
20     private String server =
          "http://localhost:8080/rdf4j-server/";
21     private String repositoryID = "repo";
22     private org.eclipse.rdf4j.repository.Repository repo;
23     private RepositoryConnection conn;
24
25     // Konstruktor für ein HTTP-Repository
26     public Repository(String server, String repositoryID) {
```

```
27          this.server = server;
28          this.repositoryID = repositoryID;
29          initHTTP(); }
30
31      // Konstruktor für ein In-Memory-Repository
32      public Repository() {
33          initSail(); }
34
35      // Methode zum Initialisieren eines HTTP-Repositories
36      private void initHTTP(){
37          repo = new HTTPRepository(getServer(),
                getRepositoryID());
38          repo.initialize();
39          conn = repo.getConnection(); }
40
41      //Methode zum Initialisieren eines In-Memory-Repositories
42      private void initSail(){
43          repo = new SailRepository(new MemoryStore());
44          repo.initialize();
45          conn = repo.getConnection(); }
46
47      // Methode, um eine Ontologie zu einer geöffneten
           Connection hinzuzufügen
48      public void addOntology(InputStream onto)
49          throws IOException, UnsupportedRDFormatException,
                RDFParseException, RepositoryException {
50          conn.add(onto, "", RDFFormat.TURTLE);
51          conn.commit(); }
52
53      public String getServer(){return server;}
54
55      public void setServer(String server){this.server = server;}
56
```

```
57    public String getRepositoryID(){return repositoryID;}
58
59    public void setRepositoryID(String
         repositoryID){this.repositoryID = repositoryID;}
60
61    public org.eclipse.rdf4j.repository.Repository
         getRepo(){return repo;}
62
63    public void setRepo
         (org.eclipse.rdf4j.repository.Repository repo){this.repo
         = repo;}
64
65    // Methode zur Ausführung der SPARQL-Abfrage
66    public TupleQueryResult runQuery(String query){
67            TupleQuery tupleQuery =
                conn.prepareTupleQuery(QueryLanguage.SPARQL,
                query);
68            TupleQueryResult result = tupleQuery.evaluate();
69            return result; }
70
71        // Zählt alle Elemente im Repository
72    public int countAllElements() throws Exception,
         SailException {
73        return countElements(repo.getConnection()
74        .getStatements(null, null, null, false)); }
75
76        //Zählt Elemente einer Eingabe
77    private int countElements(Iteration<?,?> iter) throws
         Exception {
78        int count = 0;
79        try{ while (iter.hasNext()){
80                iter.next();
81                count++;  } }
```

```
82          finally { Iterations.closeCloseable(iter); }
83          return count; }
84
85          // Dumpen der Daten einer Eingabe
86      private String dump(Iteration<?, ?> iter) throws Exception {
87          String dump = "";
88          try{ while (iter.hasNext()){
89                  String nextstr = String.valueOf(iter.next());
90                  dump += "\n" + nextstr; } }
91          finally { Iterations.closeCloseable(iter); }
92          return dump; }
93
94          // Gibt Größe des Repositories zurück
95      public long size() {return repo.getConnection().size();}
96
97          // Dumpen des gesamten Repositories
98      public String dump() throws Exception {
99          return dump(repo.getConnection().
100         getStatements(null, null, null, false)); }
101 }
```

A.3.2. AbstractOntoTestSuite.java

```
1  package de.ortec.ontotest.test.suite;
2
3  public abstract class AbstractOntoTestSuite {
4      // abstrakte Klasse für Test Suiten
5
6      // Konfiguration der Testeigenschaften
7      public abstract void setup();
8      // Initialisieren der semantischen Tests
9      public abstract void initSemanticTests();
10     // Initialisieren der strukturellen Tests
```

```
11      public abstract void initStructuralTests();
12
13      // Ausführen der Konfiguration und Initialisierungen
14      protected void initAll(){
15          setup();
16          initSemanticTests();
17          initStructuralTests();   }
18  }
```

A.3.3. AbstractStructuralOntoTestPattern.java

```
1  package de.ortec.ontotest.test.structural;
2  import de.ortec.ontotest.tools.Repository;
3
4  public abstract class AbstractStructuralOntoTestPattern {
5      // abstrakte Klasse für strukturelle Tests
6
7      private String queryPatternString;
8      private PatternQueryParameter queryParameter;
9      protected Repository repo;
10
11     public String getQueryPatternString() {return
           queryPatternString;}
12
13     public void setQueryPatternString(String
           queryPatternString){this.queryPatternString =
           queryPatternString; }
14
15     public PatternQueryParameter getQueryParameter(){return
           queryParameter;}
16
17     public void setQueryParameter(PatternQueryParameter
           queryParameter){this.queryParameter = queryParameter;}
```

```
18
19      public void setRepo(Repository repo){this.repo = repo;}
20
21      // abstrakte Methode zur Generierung von queryPatternString
22      protected abstract void buildQueryPatternString();
23
24      // abstrakte Methode zum Ausführen der SPARQL-Abfrage, gibt
            Ergebnis der Abfrage zurück
25      public abstract PatternQueryResult runQuery();
26
27      // Methode zur Ausführung eines Testes
28      public PatternQueryResult runTest(){
29          buildQueryPatternString();
30          PatternQueryResult result = runQuery();
31          return result; }
32  }
```

A.3.4. HatLokalisationTest.java

```
1  package de.ortec.ontotest.test.structural.pattern;
2  import de.ortec.ontotest.test.structural.
3          AbstractStructuralOntoTestPattern;
4  import de.ortec.ontotest.test.structural.PatternQueryParameter;
5  import de.ortec.ontotest.test.structural.PatternQueryResult;
6  import org.eclipse.rdf4j.model.Value;
7  import org.eclipse.rdf4j.query.BindingSet;
8  import org.eclipse.rdf4j.query.TupleQueryResult;
9  import org.slf4j.Logger;
10 import org.slf4j.LoggerFactory;
11
12 public class HatLokalisationTest extends
       AbstractStructuralOntoTestPattern {
13
```

```
14      String queryString = "";
15      String secondConceptString = "";
16      String source = "";
17      String sink = "";
18      private final Logger logger =
           LoggerFactory.getLogger(getClass());
19
20      public HatLokalisationTest(String source, String sink){ //
           Konstruktor erhält zwei Parameter
21          this.source = source;
22          this.sink = sink; }
23
24      // innere Klasse zur Verarbeitung eines
           SPARQL-Anfrageergebnisses
25      private class HatLokalisationTestResult extends
           PatternQueryResult {
26
27          TupleQueryResult result;
28
29          public HatLokalisationTestResult(TupleQueryResult
               result){this.result = result;}
30
31          @Override // Methode zur Überprüfung, ob
               Anfrageergebnis erwarteten Wert enthält
32          public boolean isCorrect() {
33              boolean foundViolation = true;
34
35              while (result.hasNext()) { // Iteration durch
                   Ergebnisse + Vergleich mit erwartetem String
36                  BindingSet bindingSet = result.next();
37                  Value valueOfA = bindingSet.getValue("a");
38                  System.out.println(valueOfA);
39                  if (valueOfA.toString().
```

```
40                  equals(secondConceptString)) { foundViolation =
                       false; }
41                  Value valueOfC = bindingSet.getValue("c");
42                  if (valueOfC != null) {
43                      System.out.println(valueOfC);
44                      if (valueOfC.toString().
45                      equals(secondConceptString)) {
                           foundViolation = false; }
46                  }}
47              return foundViolation == false; // gibt zurück, ob
                   Verletzung des Testpatterns gefunden wurde
48              }
49      }
50
51      @Override // Aufbau des SPARQL-Anfrage-Strings
52      protected void buildQueryPatternString() {
53          // Verarbeitung der übergebenen Parameter über ein
               PatternQueryParameter-Objekt
54          PatternQueryParameter param = new
               PatternQueryParameter(source, sink);
55          int firstIndex = param.getParamAt(0).indexOf('#');
56          String firstNamespace =
               param.getParamAt(0).substring(0, firstIndex + 1);
57          String firstConcept =
               param.getParamAt(0).substring(firstIndex + 1);
58          secondConceptString = param.getParamAt(1);
59          queryString = "PREFIX x: <" + firstNamespace + ">
               PREFIX y: <http://ontology.hip-ontoTesting.de/core#>
               PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
               SELECT ?a ?c WHERE {?b y:hatLokalisation ?a.
               OPTIONAL {?c rdfs:subClassOf+ ?a.}
               FILTER(?b = x:" + firstConcept + ")}";
60          logger.debug("queryString:   {}   ",queryString );
61
```

```
62          // Protokollierung in der Log-Datei }
63
64      @Override  // Ausführung der SPARQL-Abfrage
65      public PatternQueryResult runQuery() {
66          TupleQueryResult result = repo.runQuery(queryString);
67          HatLokalisationTestResult hatLokalisationResult = new
               HatLokalisationTestResult(result);
68          return hatLokalisationResult; }
69
70      // Methode, die die Abarbeitung der Testschritte initiiert
71      public PatternQueryResult testHatLokalisation(){
72          return this.runTest(); }
73  }
```

A.3.5. AbstractSemanticOntoTestPattern.java

```
1   package de.ortec.ontotest.test.semantic;
2   import de.ortec.ontotest.schema.SemanticModel;
3
4   public abstract class AbstractSemanticOntoTestPattern {
5       // abstrakte Klasse für semantische Tests
6
7       private Sentence sentence;
8       private Information expression;
9       private SemanticModel model;
10
11      protected Sentence getSentence(){ return sentence; }
12
13      protected Information getExpression(){ return expression; }
14
15      protected SemanticModel getModel(){ return model; }
16
```

```
17      protected void setExpression(Information expression) {
           this.expression = expression; }
18
19      // Methode zur Ausführung eines Testes
20      public void runTest(){
21          setup();
22          this.model = buildsemanticModel();
23          this.expression = buildInformation();
24          this.sentence = buildSentence();
25          afterBuild();
26          evaluate();  }
27
28      // Mocken der OntologyPipelineTest-Module und ihrer
           Funktionen
29      protected abstract void afterBuild();
30
31      // Semantischen Modell aufbauen
32      protected abstract SemanticModel buildsemanticModel();
33
34      // neue Information bauen
35      protected abstract Information buildInformation();
36
37      //neuen Sentence bauen
38      protected abstract Sentence buildSentence();
39
40       // OntologyPipelineTest-Objekte erzeugen
41      protected abstract void setup();
42
43      // entscheidet, ob vom Analyzer generierte Fakten dem
           Semantischen Modell entsprechen
44      protected abstract void evaluate();
45  }
```

A.3.6. TestSetup.java

```
1  package de.ortec.ontotest.test.structural;
2  import de.ortec.ontotest.tools.Repository;
3  import org.slf4j.Logger;
4  import org.slf4j.LoggerFactory;
5  import java.io.IOException;
6  import java.io.InputStream;
7
8  public class TestSetup {
9      private final Logger logger =
           LoggerFactory.getLogger(getClass());
10     private Repository repo;
11
12     // Repository wird erzeugt und mit Testontologien gefüllt
13     public void setup() {
14         repo = new Repository();
15         try { // Protokollierung in der Log-Datei
16             logger.debug("repo contains {} statements ",
                   repo.countAllElements());
17         } catch (Exception e){ e.printStackTrace(); }
18         logger.debug("repo size {} ", repo.size());
19
20         // Definition der Ontologiedateien als Inputstream
21         InputStream onto = getClass().getResourceAsStream(
22         "/Testontologien/TestCardiology.ttl");
23         InputStream onto2 = getClass().
24         getResourceAsStream("/Testontologien/TestCommons.ttl");
25         InputStream onto3 = getClass().getResourceAsStream(
26         "/Testontologien/TestCommons-anatomy.ttl");
27         InputStream onto4 = getClass().getResourceAsStream(
28         "/Testontologien/TestObservationValues.ttl");
29         InputStream onto5 = getClass().
30         getResourceAsStream("/Testontologien/TestCore.ttl");
```

```
31
32          try{ // Füllen des Repositories
33              repo.addOntology(onto);
34              repo.addOntology(onto2);
35              repo.addOntology(onto3);
36              repo.addOntology(onto4);
37              repo.addOntology(onto5);
38
39              // Protokollierung
40              logger.debug("repo contains  {} statements
                   ",repo.countAllElements() );
41              logger.debug("repo size  {}   ",repo.size() );
42              logger.debug("dump   {}   ",repo.dump() );
43
44          } catch (IOException e) {  e.printStackTrace();
45          } catch (Exception e){  e.printStackTrace();
46          } finally {
47              try {
48                  onto.close();
49                  onto2.close();
50                  onto3.close();
51                  onto4.close();
52                  onto5.close();
53              } catch (IOException e){ e.printStackTrace(); }
54          } }
55
56      public Repository getRepo() {return repo;}
57  }
```

A.3.7. BrustengeTest.java

```
1  package de.ortec.ontotest.test.semantic;
2  import de.ortec.ontotest.schema.Concept;
3  import de.ortec.ontotest.schema.Relation;
4  import de.ortec.ontotest.schema.SemanticModel;
5  import de.ortec.ontotest.tools.ISemanticAnalyzer;
6  import de.ortec.ontotest.tools.ISemanticMatcher;
7  import org.junit.jupiter.api.Test;
8  import org.junit.jupiter.api.DisplayName;
9  import org.mockito.Mockito;
10 import static org.hamcrest.MatcherAssert.assertThat;
11 import static org.hamcrest.core.Is.is;
12
13 public class BrustengeTest extends
      AbstractSemanticOntoTestPattern{
14    // Er berichtete von
         Brustenge.=>[hatBeobachtung:Brustenge[]]
15
16    private static String text = "Er berichtete von Brustenge.";
17    private static String conceptName = "AnginaPectoris";
18    private static String relationName = "hatBeobachtung";
19    private static ISemanticMatcher matcherMock;
20    private static ISemanticAnalyzer analyzerMock;
21
22    @Test // JUnit-Test
23    @DisplayName("Test für 'Er berichtete von Brustenge'")
24    public void testSemantic(){
25        runTest(); }
26
27    @Override // Mocken der Funktionen der
         OntologyPipelineTest-Module
28    protected void afterBuild(){
29        Mockito.when(analyzerMock.analyze(getSentence())).
```

```
30          thenReturn(getExpression());
31          Mockito.when(matcherMock.
32          run(getModel(),getExpression())).thenReturn(true); }
33
34      @Override // Aufbau des Semantischen Modells
35      protected SemanticModel buildsemanticModel() {
36          Concept brustenge = new Concept(conceptName);
37          Relation hatBeobachtung = new Relation(relationName);
38          hatBeobachtung.setRelated(brustenge);
39          SemanticModel model = SemanticModel.instance();
40          model.setRoot(hatBeobachtung);
41          return model; }
42
43      @Override // Erzeugen einer Information
44      protected Information buildInformation() {
45          return Information.build(getModel(),matcherMock);  }
46
47      @Override // Erzeugen eines Sentence
48      protected Sentence buildSentence() {
49          return Sentence.build(text,analyzerMock);  }
50
51      @Override // OntologyPipelineTest-Objekte erzeugen
52      protected void setup() {
53          matcherMock = Mockito.mock(ISemanticMatcher.class);
54          analyzerMock = Mockito.mock(ISemanticAnalyzer.class); }
55
56      @Override // Evaluation, ob generierte Fakten mit
           Erwartungshaltung übereinstimmen
57      protected void evaluate() {
58          assertThat(getSentence().
59          contains(getExpression()),is(true)); }
60  }
```

A.3.8. BrustengeSuiteTest.java

```
1  package de.ortec.ontotest.test.suite;
2  import de.ortec.ontotest.test.semantic.BrustengeTest;
3  import de.ortec.ontotest.test.structural.TestSetup;
4  import de.ortec.ontotest.test.structural.pattern.ExistenceTest;
5  import de.ortec.ontotest.test.structural.pattern.OneLabelTest;
6  import de.ortec.ontotest.test.structural.pattern.SubClassOfTest;
7  import de.ortec.ontotest.test.structural.
8          pattern.UniqueLabelTest;
9  import de.ortec.ontotest.tools.Repository;
10 import org.junit.jupiter.api.DynamicTest;
11 import org.junit.jupiter.api.function.Executable;
12 import org.junit.jupiter.api.TestFactory;
13 import org.junit.platform.runner.JUnitPlatform;
14 import org.junit.runner.RunWith;
15 import org.slf4j.Logger;
16 import org.slf4j.LoggerFactory;
17 import java.util.ArrayList;
18 import java.util.Collection;
19 import static org.hamcrest.MatcherAssert.assertThat;
20 import static org.hamcrest.core.Is.is;
21
22 @RunWith(JUnitPlatform.class)
23 public class BrustengeSuiteTest extends AbstractOntoTestSuite {
24
25     Collection<DynamicTest> dynamicTests = new ArrayList<>();
26     TestSetup setup;
27     Repository repo;
28     private static final boolean OK = true;
29     private final Logger logger =
          LoggerFactory.getLogger(getClass());
30
31
```

```
32      @Override // Initialisieren der semantischen Tests
33      public void initSemanticTests() {
34          BrustengeTest brustenge = new BrustengeTest();
35          Executable exec = () -> brustenge.testSemantic(); //
               Erzeugen einer Testausführung
36          String testName = "Semantischer Test für 'Er berichtete
               von Brustenge'";
37          DynamicTest dTest = DynamicTest.dynamicTest(testName,
               exec); // Erzeugen eines dynamischen Tests
38          dynamicTests.add(dTest); // Hinzufügen des dynamischen
               Tests zur Collection }
39
40      @Override // Initialisieren der strukturellen Tests
41      public void initStructuralTests() {
42          OneLabelTest oneLabelTest = new OneLabelTest();
43          oneLabelTest.setRepo(repo);
44          Executable exec = () -> assertThat(oneLabelTest.
45          testForOneLabel().isCorrect(),is(OK));
46          String testName = "Test auf ein Standardlabel pro
               Konzept";
47          DynamicTest dTest = DynamicTest.dynamicTest(testName,
               exec);
48          dynamicTests.add(dTest);
49
50          UniqueLabelTest uniqueLabelTest = new UniqueLabelTest();
51          uniqueLabelTest.setRepo(repo);
52          exec = () -> assertThat(uniqueLabelTest.
53          testForUniqueLabel().isCorrect(), is(OK));
54          testName = "Test auf einzigartiges Standardlabel";
55          dTest = DynamicTest.dynamicTest(testName, exec);
56          dynamicTests.add(dTest);
57
58
```

```
59          ExistenceTest existenceTest = new ExistenceTest(
60          "http://ontology.hip-ontoTesting.de/
61          cartos#AnginaPectoris");
62          existenceTest.setRepo(repo);
63          exec = () -> assertThat(existenceTest.
64          testExistence().isCorrect(), is(OK));
65          testName = "Test, ob ein Konzept 'Angina Pectoris'
               existiert";
66          dTest = DynamicTest.dynamicTest(testName, exec);
67          dynamicTests.add(dTest);
68
69          SubClassOfTest subClassTest = new SubClassOfTest(
70          "http://ontology.hip-ontoTesting.de/
71          cartos#AnginaPectoris", "http://ontology.
72          hip-ontoTesting.de/core#BaseConcept");
73          subClassTest.setRepo(repo);
74          exec = () -> assertThat(subClassTest.
75          testSubClassOf().isCorrect(), is(OK));
76          testName = "Test, ob ein Konzept SubClass eines Anderen
               ist";
77          dTest = DynamicTest.dynamicTest(testName, exec);
78          dynamicTests.add(dTest);
79
80          logger.debug("size   {}", dynamicTests.size() ); //
               Protokollierung }
81
82      @Override // Testkonfiguration
83      public void setup(){
84          setup = new TestSetup();
85          setup.setup();
86          repo = setup.getRepo(); }
87
```

```
88      @TestFactory // Ausführen aller in der Collection
           gesammelten Tests
89      public Collection<DynamicTest> constructBrustengeSuite(){
90          initAll();
91          return dynamicTests; }
92  }
```

Literaturverzeichnis

[Bec12] A. Becker, D. Naber, M. Frick, M. Trautwein: *Technischer Report: Concept-Mapping*. ORTEC medical GmbH. 2012. unveröffentlicht

[Bei04] H. Beier: *Vom Wort zum Wissen*. Information, Wissenschaft und Praxis (IWP), Nr.2, S.73-79. 2011

[Bru05] J. de Bruijn, R. Lara, A. Polleres, D. Fensel: *OWL DL vs. OWL flight: conceptual modeling and reasoning for the semantic Web*. Proceeding WWW '05 Proceedings of the 14th international conference on World Wide Web, S.623-632 . 2005

[BP06] A. Blumauer, T. Pellegrini: *Semantic Web und semantische Technologien: Zentrale Begriffe und Unterscheidungen*. Semantic Web - Wege zur vernetzten Wissensgesellschaft, S.9-25. Springer Berlin Heidelberg, 2006

[Ditt07] L. U. Dittmann: *OntoFMEA - Ontologiebasierte Fehlermöglichkeits- und Einflussanalyse*. Deutscher Universitätsverlag, 2007

[Dud14] Dudenredaktion, Wissenschaftlicher Rat: *Duden - Die deutsche Rechtschreibung. Bd. 1*. Dudenverlag, Bibliographisches Institut & F.A. Brockhaus, 2014

[Duq11] A. Duque-Ramos, J.T. Fernandez-Breis, R. Stevens, N. Aussenac-Gilles: *OQuaRE: a square-based approach for evaluating the quality of ontologies* . Journal of Research and Practice in Information Technology 43(2011)159-73. 2011

[FGJ97] M. Ferndandez, A. Gomez-Perez, N. Juristo : *METHONTOLOGY: From Ontological Art Towards Ontological Engineering*. AAAI-97 Spring Symposium Series, Stanford University, 24-26 March. 1997

[Gei12] P. Geibel, C. Nolte, T. Usnich, S. Krüger, A. Becker, M. Frick: *Technischer Report: Modellierung von medizinischen Wissen*. ORTEC medical GmbH. 2012. unveröffentlicht

[GOF09] S. García-Ramos, A. Otero, M. Fernández-López: *OntologyTest: A Tool to Evaluate Ontologies through Tests Defined by the User*. Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living. IWANN 2009. Lecture Notes in Computer Science, vol 5518. Springer-Verlag Berlin Heidelberg, 2009

[Goo15] B. M. Good, G. Ha, Ch. K. Ho, M. D. Wilkinson: *OntoLoki: an automatic, instance-based method for the evaluation of biological ontologies on the Semantic Web*. ArXiv e-prints, arXiv:1502.06025 [q-bio.QM]. 2015

[GP09] A. Gangemi , V. Presutti: *Ontology Design Patterns*. Handbook on Ontologies, Series International Handbooks on Information Systems, S.221-243. Springer Berlin Heidelberg, 2009

[Gri16] M. Griese: *Entwicklungsprozess: Entwurf Gesamtprozess*. ORTEC medical GmbH. 2016. unveröffentlicht

[Iqb13] R. Iqbal, M. A. A. Murad, A. Mustapha, N. M. Sharef: *An Analysis of Ontology Engineering Methodologies: A Literature Review*. Research Journal of Applied Sciences, Engineering and Technology, Volume 6, S.2993-3000. 2013

[KA16] S. Khwaja, M. Alshayeb: *Survey On Software Design-Pattern Specification Languages*. ACM Computing Surveys (CSUR), Volume 49, Issue 1, S.1-35. 2016

[Kar16] F. Karakol: *Konzeption einer ontologiebasierten Schnittstelle zur Integration von verteilt vorliegenden Informationsquellen*. disserta Verlag, 2016

[Kon14] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, R. Cornelissen: *Test-Driven Evaluation of Linked Data Quality*. WWW '14 Proceedings of the 23rd international conference on World wide web, S. 747-758. 2014

[Ram13] A. D. Ramos, J. T. Fernandez-Breis, M. Iniesta, St. Schulz, M. Dumontier, M. E. Aranguren, N. Aussenac-Gilles, R. Stevens: *Evaluation of the OQuaRE framework for ontology quality*. Expert Systems with Applications, Volume 40, Issue 7, S.2696–2703. 2013

[Rei10] K. Reichenberger: *Kompendium semantische Netze: Konzepte, Technologie, Modellierung*. Springer-Verlag, 2010

[Rei91] U. Reimer: *Einführung in die Wissensrepräsentation - Netzartige und schemabasierte Repräsentationsformate*. B. G. Teubner Stuttgart, 1991

[Reu13] S. Reuter, D. Naber, J.Setz, M. Trautwein: *Technischer Report: Dokumentation der Pipeline*. ORTEC medical GmbH. 2013. unveröffentlicht

[Rou11] C. Roussey, F. Pinet, M. Ah Kang, O. Corcho: *An Introduction to Ontologies and Ontology Engineering*. Ontologies in Urban Development Projects, Volume 1 of the series Advanced Information and Knowledge Processing, S.9-38. 2011

[Rub14] K. S. Rubin: *Essential Scrum: Umfassendes Scrum-Wissen aus der Praxis*. mitp-Verlag, 2014

[Ruy15] F. B. Ruy , C. C. Reginato, V. A. Santos, R. A. Falbo, G. Guizzardi : *Ontology Engineering by Combining Ontology Patterns*. Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings. 2015

[SBF98] R. Studer, R. Benjamins, D. Fensel: *Knowledge engineering: Principles and methods*. Data & Knowledge Engineering, Volume 25, Issue 1-2, S.161-197. 1998

[Sch10] A. Schatten, St. Biffl, M. Demolsky, E. Gostischa-Franta, Th. Östreicher, D. Winkler: *Best Practice Software-Engineering: Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen*. Spektrum Akademischer Verlag, 2010

[SS09] C. Spreckelsen, K. Spitzer: *Wissensbasen und Expertensysteme in der Medizin - KI-Ansätze zwischen klinischer Entscheidungsunterstützung und medizinischem Wissensmanagement*. Vieweg+Teubner Verlag, 2009

[Stu11] H. Stuckenschmidt: *Ontologien: Konzepte, Technologien und Anwendungen*. 2. Auflage, S.124-126. Springer Verlag Berlin-Heidelberg, 2011

[Tar05] S. Tartir, I. B. Arpinar, M. Moore, A. P. Sheth, B. Aleman-Meza: *OntoQA: Metric-Based Ontology Quality Analysis*. Proceedings of IEEE Workshop on Knowledge Acquisition from Distributed, Autonomous, Semantically Heterogeneous Data and Knowledge Sources, Houston. 2005

[Tru15a] J. Truber: *Projektbericht: Semantische Terminologien*. Fachhochschule Brandenburg - Fachbereich Informatik und Medien, Februar 2015. unveröffentlicht

[Tru15b] J. Truber: *Projektbericht: Semantische Terminologien - 2. Semester*. Fachhochschule Brandenburg - Fachbereich Informatik und Medien, August 2015. unveröffentlicht

[Tru16] J. Truber: *Seminararbeit: Anforderungen an die Semantic Workbench, um Inkonsistenzen durch Ontologieänderungen zu vermeiden*. Fachhochschule Brandenburg - Fachbereich Informatik und Medien, 2016. unveröffentlicht

[Ull14] C. Ullenboom: *Java ist auch eine Insel - Einführung, Ausbildung, Praxis*. Galileo Computing, 2014

[VBH17] B. Vogel-Heuser, T. Bauernhansl, M. ten Hompel: *Handbuch Industrie 4.0 Bd.2*. Springer Vieweg, 2017

Abbildungsverzeichnis

Tabellenverzeichnis