JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Eva Mair

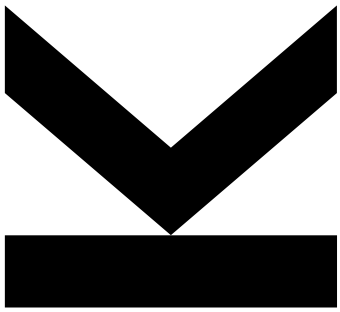
Angefertigt am
Institut für Wirtschaftsinformatik - Data & Knowledge Engineering

Beurteiler / Beurteilerin
Univ.-Prof. Dipl.-Ing. Dr. techn. Michael Schrefl

Mitbetreuung
Mag. Dr. Christoph Schütz

März 2018

PESTEL Modeler: Ein Werkzeug für die Strategische Umweltanalyse nach der PESTEL-Methode unter Verwendung von MetaEdit+, iStar 2.0 und Semantischen Technologien



Masterarbeit

zur Erlangung des akademischen Grades

Master of Science

im Masterstudium

Wirtschaftsinformatik

JOHANNES KEPLER UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Ort, Datum

Unterschrift

Kurzfassung

Viele Unternehmen und Organisationen führen strategische Analysen durch, um auf Basis der Ergebnisse eine Unternehmensstrategie zu entwickeln und weitere wichtige strategische Entscheidungen treffen zu können. Die Ergebnisse strategischer Analysen sind meistens in textueller und informaler Form, wodurch sich das Problem ergibt, dass diese schwer vergleichbar und kombinierbar sind. Als Lösung soll die Formalisierung des strategischen Analyseprozesses dienen, indem strategische Analysen mit Hilfe von Business-Model-Ontologien dargestellt werden. Dazu wird die strategische PESTEL- und SWOT-Analyse anhand des Frameworks iStar 2.0 als gerichtete Graphen dargestellt. Die Verwendung von semantischen Technologien ermöglicht die Formalisierung der strategischen Analysen. Zu semantischen Technologien zählen das Resource Description Framework (RDF), RDF-Schema (RDFS), sowie die Abfragesprache SPARQL. Modelle, welche auf Basis des iStar 2.0 Frameworks erstellt wurden, können mittels RDF-Graphen formalisiert und maschinenlesbar gemacht werden. Mit SPARQL können auf Basis des RDF-Graphen strategische Chancen und Risiken abgefragt werden, welche als Grundlage für strategische Entscheidungen dienen. Die Analysen werden somit maschinenauswertbar.

Das Ziel dieser Arbeit ist die Entwicklung des konzeptuellen Modellierungswerkzeugs PESTEL Modeler, mit dem unter der Verwendung von MetaEdit+, iStar 2.0 und semantischen Technologien strategische Analysen graphisch dargestellt und ausgewertet werden können.

Abstract

Many companies and organisations conduct strategic analyses in order to develop a business strategy and make other important strategic decisions based on their results. These results are usually in the form of informal text, which results in the problem that they are difficult to compare and combine. The solution is to formalize the strategic analysis process by conducting strategic analysis using business model ontologies. The strategic PESTEL and SWOT analysis will be illustrated as directed graph using the iStar 2.0 framework. The use of semantic technologies enables the formalization of strategic analysis. Semantic technologies include the Resource Description Framework (RDF), the RDF schema (RDFS), and the query language SPARQL. Models created based on the iStar 2.0 framework can be formalized and made machine readable by using RDF graphs. With SPARQL, strategic opportunities and risks can be queried based on the RDF graph. These derived opportunities and risks serve as a basis for strategic decisions.

The goal of this work is the development of the conceptual modeling tool *PESTEL Modeler*, which can be used to graphically display and evaluate strategic analysis using MetaEdit+, iStar 2.0 and semantic technologies.

Inhaltsverzeichnis

1 Einleitung

Dieses Kapitel leitet das Thema dieser Arbeit ein und gibt einen Überblick über den Aufbau der Arbeit. Dabei wird die Problemstellung geschildert und die Motivation zur Arbeit dargestellt. Abschließend wird die Gliederung der Arbeit beschrieben. Dazu wird für jedes Kapitel eine kurze Beschreibung angeführt.

1.1 Problemstellung und Motivation

Um dem Wettbewerb am Markt Stand halten zu können, ist strategisches Management für jedes Unternehmen und jede Organisation Voraussetzung. Dabei werden strategische Analysen durchgeführt, um Informationen zu sammeln, welche die Basis für die Unternehmensstrategie und die strategischen Entscheidungen darstellen. Dazu zählen unter anderem Analysen und Prognosen der Umwelt des Unternehmens oder der Organisation. Strategische Analysen liefern verschiedene Berichte, meistens in textueller oder informaler Form, was zu dem Problem führt, dass der Vergleich und die Kombination der Analyseergebnisse erheblich erschwert werden.

Die Lösung für das Problem ist die Darstellung der Analysen, sowie deren Ergebnisse unter Verwendung von Business-Model-Ontologien in OLAP-Cubes. Traditionelle OLAP-Systeme basieren normalerweise auf numerischen Daten. Schütz et al. (2013) beschreiben die Verwendung von OLAP-Cubes auf Basis von sogenannten *ontology-valued measures*, welche komplexe Zusammenhänge zwischen Geschäftsobjekten oder Handlungsträgern anhand von RDF-Graphen darstellen.

Schütz und Schrefl (2017) beschreiben eine Möglichkeit der Formalisierung des strategischen Analyseprozesses, indem strategische Analysen mit Hilfe von Business-Model-Ontologien dargestellt werden. Das bedeutet, dass man versucht, unterschiedliche Sachverhalte und Zusammenhänge aus der Betriebswirtschaftslehre mit Hilfe von gerichteten Graphen als verschiedene Statements zu erfassen und darzustellen.

Als Beispiel für die Formalisierung von strategischen Analysen wird im Rahmen dieser Arbeit die PESTEL-Analyse und ein Teil der SWOT-Analyse mit Hilfe von Business-Model-Ontologien dargestellt. Die PESTEL-Analyse beschreibt die politischen (political), wirtschaftlichen (economical), sozialen (social), technologischen (technological), rechtlichen (legal) und ökologischen (economical) Einflussfaktoren eines Unternehmens. Bei der SWOT-Analyse werden die Stärken (Strength), Schwächen (Weaknesses), Chancen (Opportunities) und Risiken (Threats) eines Unternehmens identifiziert. Im Rahmen dieser Arbeit werden nur die externen Chancen und Risiken einbezogen.

Goal Modeling wird seit Jahren erfolgreich in unterschiedlichen Bereichen, zum Beispiel im Requirements Engineering, angewandt (Fernandes et al. 2011). Der Vorteil dieser Methode ist, dass man bereits vor der Entscheidungsphase ein grundlegendes Verständnis zur Situation eines Unternehmens oder einer Organisation erhält. Es wird ein klarer Überblick über die Handlungsträger, auch Actors genannt, verschafft, welcher Fragen wie „Welche Ziele verfolgen welche Actors?“, „Welche Pläne und Resources werden zur Zielerreichung benötigt?“ oder „Wie hängen die Ziele, Resources und Pläne der Actors zusammen?“ (Fernandes et al. 2011). Aus den Antworten dieser Fragen lassen sich Handlungsmaßnahmen ableiten. Das Framework iStar 2.0 (Dalpiaz et al. 2016) unterstützt den Ansatz des Goal Modeling und ermöglicht es im Rahmen von strategischen Analysen

für bestimmte Actors, wie zum Beispiel Unternehmen oder politische Institutionen, dazugehörige Goals, Qualities, Tasks und Resources, sowie deren Zusammenhänge als gerichtete Graphen darzustellen.

Die Verwendung von semantischen Technologien ermöglicht die Formalisierung der strategischen Analysen (Schütz und Schrefl 2017). Dazu zählen das Resource Description Framework (RDF), das RDF-Schema (RDFS), sowie die Abfragesprache SPARQL. Modelle, welche auf Basis des iStar 2.0 Frameworks erstellt wurden, können mittels RDF-Graphen formalisiert und maschinenlesbar gemacht werden. Mit SPARQL können auf Basis des RDF-Graphen strategische Chancen und Risiken abgefragt werden, welche als Grundlage strategischer Entscheidungen dienen. Die Analysen werden somit maschinenauswertbar.

Das Ziel dieser Arbeit ist es, den von (Schütz und Schrefl 2017) beschriebenen Ansatz praktisch als kontextspezifisches Modellierungs- und Analysewerkzeug umzusetzen. Der Name des entwickelten Modells ist *PESTEL Modeler*. Des Weiteren wird ein unterstützendes metaCASE-Werkzeug, wie MetaEdit+ verwendet, welches erlaubt, ein Metamodell zu definieren und von dort aus die Analysen zu starten.

1.2 Aufbau der Arbeit

Diese Arbeit besteht neben der Einleitung aus folgenden weiteren fünf Kapiteln:

- Kapitel 2 beschäftigt sich mit den Grundlagen der verwendeten Analysen, Frameworks, Technologien und Werkzeuge. Dazu zählen die strategischen Analysen PESTEL-Analyse und SWOT-Analyse, das iStar 2.0 Frameworks, sowie das metaCASE-Werkzeug MetaEdit+.
- Kapitel 3 gibt einen Überblick über die Architektur des entwickelten Werkzeugs PESTEL Modeler. Dazu werden die Systeme, Komponenten und Artefakte, sowie ihre Zusammenhänge beschrieben.
- Im Kapitel 4 wird aufgezeigt, wie die einzelnen Funktionen des Werkzeugs PESTEL Modeler angewandt werden können.
- Kapitel 5 beschreibt die Details zur Implementierung der einzelnen Komponenten. Des Weiteren wird Aufschluss über Implementierungsentscheidungen gegeben.
- Kapitel 6 schließt die Arbeit mit einem Fazit, sowie mit einem kurzen Ausblick über mögliche zukünftige Forschungsprojekte ab.
- Appendix: Der Anhang enthält ein Installationshandbuch, welches die nötigen Artefakte und Schritte beschreibt, um das PESTEL-Modeler-Werkzeug benützen zu können.

2 Hintergrund

In diesem Kapitel werden ausgehend von zwei strategischen Analysen der Hintergrund und die Grundlagen der Arbeit näher gebracht. Dazu werden zuerst die PESTEL-Analyse und die SWOT-Analyse vorgestellt. Anschließend werden die Konzepte der Modellierungssprache iStar 2.0 aufgezeigt, welche zur Darstellung von Beziehungen und Objekten der strategischen Analysen dienen soll. Unter der Verwendung von semantischen Technologien, wie RDF, RDFS und SPARQL, sollen diese Analysen messbar und computerauswertbar gemacht werden. Dazu werden kurz die Grundlagen der semantischen Technologien erläutert. Abschließend wird erklärt, worum es sich bei MetaCASE-Werkzeugen, wie MetaEdit+, handelt und wofür MetaEdit+ für diese Arbeit relevant ist.

2.1 Strategische Umweltanalyse

Bei der strategischen Umweltanalyse werden sämtliche externe Faktoren und Bedingungen, welche von außen auf ein Unternehmen oder eine Organisation einwirken, analysiert (Rufo und Zerres 2017). Einerseits hat die Umwelt meistens erheblichen Einfluss auf ein Unternehmen, andererseits kann ein Unternehmen jedoch nur gering die einzelnen Umweltfaktoren beeinflussen oder verändern. Deshalb ist es wichtig, externe Chancen und Risiken zu erkennen, um auf deren Basis eine passende Strategie entwickeln zu können.

Im Rahmen dieser Arbeit sind zwei strategische Umweltanalysen relevant. Dazu zählen die PESTEL-Analyse und die SWOT-Analyse.

2.1.1 PESTEL-Analyse

Die PESTEL-Analyse ist eine strategische Umweltanalyse, mit der für ein Unternehmen oder eine Organisation, wie in Abbildung 1 ersichtlich, die politischen (political),

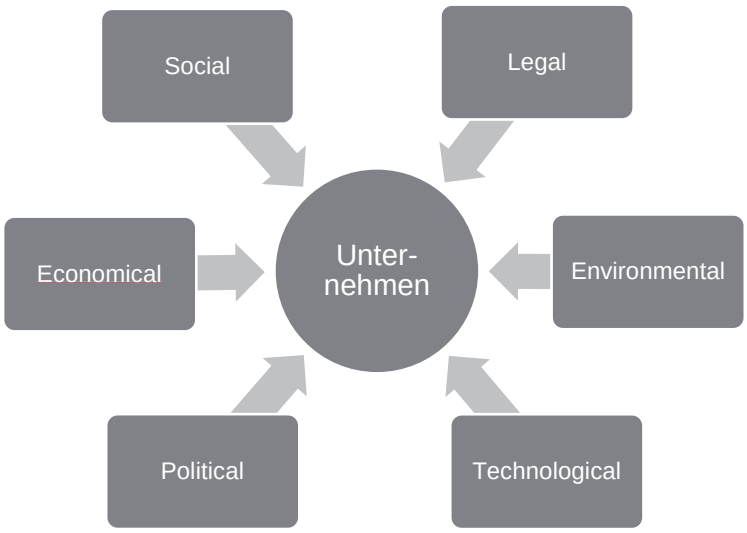


Abbildung 1: PESTEL Analyse

wirtschaftlichen (economical), sozialen (social), technologischen (technological), rechtlichen (legal) und ökologischen (economical) Einflussfaktoren analysiert werden (Johnson et al. 2008).

Bei der Analyse müssen nicht immer alle Faktoren berücksichtigt werden. Ganz im Gegenteil sollen nur jene Faktoren für die Analyse herangezogen werden, welche auch wirklich einen Einfluss darstellen (Rufo und Zerres 2017).

2.1.2 SWOT-Analyse

Die SWOT-Analyse ist eine strategische Analyse, bei der externe, sowie interne Faktoren eines Unternehmens oder einer Organisation untersucht werden (Rufo und Zerres 2017). Die SWOT-Analyse besteht somit zum einen Teil aus einer internen Unternehmensanalyse und zum anderen Teil aus einer externen Umweltanalyse. Bei der Unternehmensanalyse werden die Stärken und Schwächen analysiert. Die Stärken wirken sich positiv und die Schwächen wirken sich negativ auf den Erfolg des Unternehmens oder der Organisation aus (Srdjevic et al. 2012).

Bei der Umweltanalyse wird versucht, mögliche Chancen und Risiken zu identifizieren. Chancen sind externe Faktoren, welche von außen positiv das Unternehmen beeinflussen, hingegen können Risiken einen negativen Einfluss auf das Unternehmen haben(Srdjevic et al. 2012).

Die Abkürzung SWOT setzt sich durch die Anfangsbuchstaben der englischen Begriffe Strength, Weakness, Opportunity und Threat zusammen. Eine Übersicht ist in Abbildung 2 ersichtlich. Im Rahmen dieser Arbeit werden nur die beiden externen Faktoren Opportunity und Threat, also jene Teile, die die strategische Umwelt analysieren, betrachtet und angewandt.

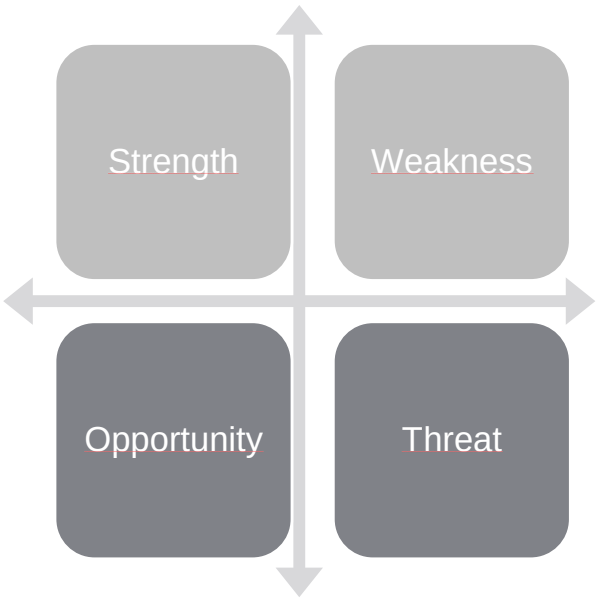


Abbildung 2: SWOT-Analyse

2.2 iStar 2.0

iStar ist eine konzeptuelle Modellierungssprache, welche den Fokus auf folgende drei Fragen setzt (Dalpiaz et al. 2016):

- Wer? – Wer sind die handelnden Entitäten?
- Warum? – Was sind die Absichten?
- Wie? – Was ist die Strategie?

iStar wurde Mitte der 90iger Jahre entwickelt und stellt ein Framework für die ziel- und actororientierte Modellierung dar (Dalpiaz et al. 2016). Im Jahre 2016 wurde das Grundkonzept von iStar im Rahmen mehrerer Diskussionsrunden mit verschiedenen Personen der iStar Community zu einem konsistenten und klar definierten Kernkonzept weiterentwickelt, welches nun als iStar 2.0 Framework bezeichnet wird.

Das iStar 2.0 Framework besteht aus verschiedenen Elementen und Beziehungen zwischen den Elementen. Im Rahmen dieser Arbeit wird nur ein Teil dieser Elemente und Beziehungen verwendet und umgesetzt. In den folgenden Kapiteln werden nur jene Elemente beschrieben, welche auch für die Arbeit relevant sind.

2.2.1 Elemente

Die Elemente *Actor* und *Agent* stellen den sozialen Aspekt der Modellierungssprache dar (Dalpiaz et al. 2016). Das Element *Agent* ist eine Unterkategorie des Elements *Actor*. Ein *Agent* ist ein konkretes, physisch existierendes Konstrukt, wie zum Beispiel eine Person oder eine Organisation. Beide Elemente stellen eine eigenständige Entität dar. Unter Berücksichtigung diverser anderer Faktoren und Abhängigkeiten ist die Absicht der Entitäten, ihre Ziele zu erreichen. In Abbildung 3 ist die graphische Notation der Elemente ersichtlich.

Abbildung 3: Elemente Actor und Agent

Die Absichten von *Actors* oder *Agents* werden innerhalb des Actor-Bereichs modelliert. Dieser Bereich wird mittels einer graphischen Umgrenzung, wie in Abbildung 4, dargestellt.

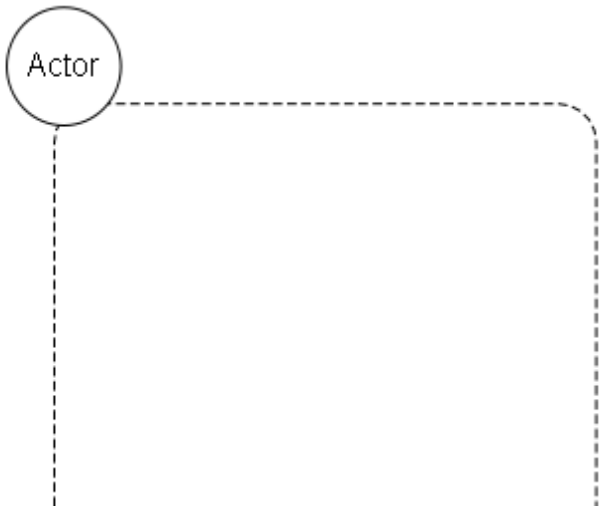


Abbildung 4: Actor mit Bereich

Zu den Absichten eines *Actors* oder *Agents* zählen folgende Elemente (Dalpiaz et al. 2016):

- **Goal**: Ein Goal ist ein klar definiertes Ziel, welches ein Actor erreichen möchte.
  Quality: Eine Quality stellt ein Attribut dar, für das der Actor einen bestimmten Grad an Leistung erreichen möchte. Mit Qualities wird versucht, verschiedene Lösungswege aufzuzeigen, damit das Ziel erreicht werden kann.
- **Task**: Ein Task stellt jene Aufgaben dar, welche ausgeführt werden müssen, um die Ziele erreichen zu können oder andere vorgegebene Regulierungen einhalten zu können.
- **Resource**: Resources sind ein physische oder informelle Mittel, welche zur Aufgabenerfüllung benötigt werden.

Abbildung 5: Intentional Elements

Diese vier Elemente werden nach iStar 2.0 als *Intentional Elements* bezeichnet (Dalpiaz et al. 2016). Die graphische Notation der Elemente ist in Abbildung 5 ersichtlich. Sie werden innerhalb des Bereichs eines *Actors* dargestellt, da diese die Anforderungen oder Wünsche eines *Actors* oder *Agents* darstellen.

2.2.2 Beziehungen

Um Zusammenhänge zwischen zwei *Actors* oder *Agents* darzustellen, sind nach iStar 2.0 die zwei folgenden Typen von Beziehungen definiert (Dalpiaz et al. 2016):

- **isA:** Mit der Beziehung *isA* kann eine Generalisierung beziehungsweise Spezialisierung von zwei *Actors* dargestellt werden. Dies ist jedoch nur zwischen zwei *Actors* möglich. *Agents* können keine Spezialisierung darstellen, da es sich dabei schon um konkrete Personen oder Organisationen handelt.
- **participatesIn:** Mit der Beziehung *participatesIn* können sämtliche andere Verbindungen zwischen *Actors* und *Agents* dargestellt werden, zum Beispiel, dass ein *Agent* einen Teil eines *Actors* abdeckt.

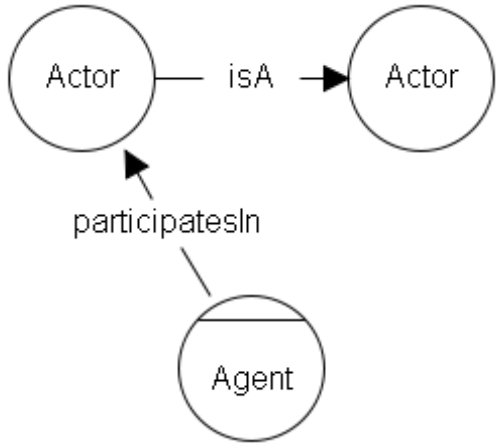


Abbildung 6: Beziehungen isA und participatesIn

Die graphische Notation für die Beziehungen *isA* und *participatesIn* ist in Abbildung 6 ersichtlich.

Es gibt vier verschiedene Beziehungen, um Zusammenhänge zwischen den Intentional Elements *Goal, Quality, Task* und *Resource* darzustellen (Dalpiaz et al. 2016). Diese sind die Beziehungen *refines, qualifies, helps* und *hurts*. Die graphische Notation dieser Beziehungen ist in Abbildung 7 ersichtlich. Welche Einschränkungen es bei den Beziehungen zwischen den Intentional Elements gibt, ist in Tabelle 1 dargestellt.

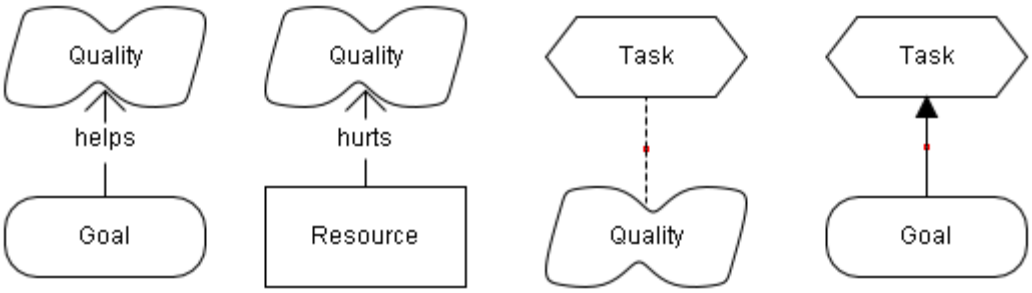


Abbildung 7: Beziehungen zwischen Intentional Elements

Beziehungen zwischen Intentional Elements

		Pfeilspitze geht zu			
		Goal	**Quality**	**Task**	**Resource**
Beziehung geht aus von	**Goal**	refines	helps/hurts	refines	-
	Quality	qualifies	helps/hurts	qualifies	qualifies
	Task	refines	helps/hurts	refines	-
	Resource	-	helps/hurts	-	-

Tabelle 1:Einschränkungen der Beziehungen zwischen Intentional Elements

Die letzte Beziehung, die nach iStar 2.0 modelliert werden kann, ist eine Abhängigkeit zwischen Intentional Elements, welche innerhalb des Bereichs zweier *Actors* oder *Agents* liegen (Dalpiaz et al. 2016).

Für solch eine Abhängigkeitsbeziehung, im Modell als dependerOf-Beziehung bezeichnet, gibt es folgende 5 Parameter:

- **Depender**: Der *Depender* stellt den *Actor* dar, in dessen Bereich sich das abhängige *Intentional Element* befindet.
- **Depender Element**: Das *Depender Element* stellt das *Intentional Element* dar, welches von einem anderen *Intentional Element* abhängig ist und sich im Bereich des *Depender Actors* befindet.
- **Dependum**: Das *Dependum* ist das *Intentional Element*, das das Objekt der Abhängigkeit darstellt.
- **Dependee**: Der *Dependee* stellt den Actor dar, der das Dependum bereitstellen muss.
- **Dependee Element**: Das Dependee Element ist das *Intentional Element*, welches sich im Bereich des *Dependee Actors* befindet.

Im Rahmen dieser Arbeit wird der Parameter Dependum der Beziehung dependerOf als abstraktes Element dargestellt, da es für das Ergebnis der OT Analyse nicht relevant ist. Die graphische Notation dieser Beziehung ist in Abbildung 8 ersichtlich.

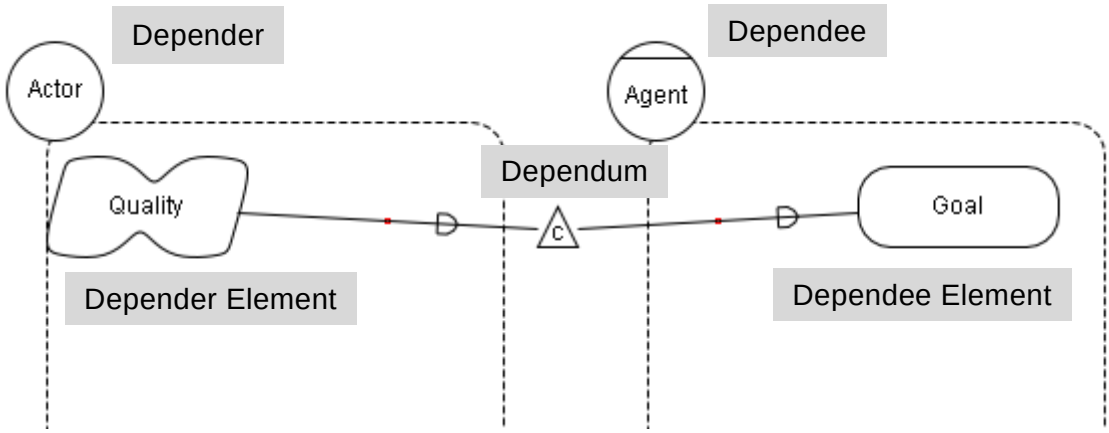


Abbildung 8: Beziehung dependerOf

2.3 Semantische Technologien

Unter semantischen Technologien versteht man Standards des World Wide Web Consortiums (W3C), welche verwendet werden, um Wissen repräsentieren zu können, sodass dieses Wissen für Menschen verständlich dargestellt wird und gleichzeitig von Maschinen gelesen und verarbeitet werden kann (Dengel 2012, S. 9).

In den folgenden Kapiteln werden die unterschiedlichen semantischen Technologien beschrieben, welche für diese Arbeit relevant sind. Dazu zählt das Resource Description Framework, das RDF-Schema, sowie die SPARQL-Abfragesprache.

2.3.1 Resource Description Framework

Das Resource Description Framework, kurz RDF, ist ein Framework zur Darstellung von Informationen aus unterschiedlichen Quellen und wurde ursprünglich zur Repräsentation von Resources des Word Wide Webs entwickelt (Schreiber et al. 2014). Die Informationen zu einer Resource werden als Menge an Aussagen dargestellt, welche als RDF-Tripel bezeichnet werden. Ein RDF-Tripel besteht aus drei Teilen: Subjekt, Prädikat und Objekt (Auer et al. 2013).

Durch die Darstellung der Resources in der Form von RDF-Tripel ergibt sich ein gerichteter Graph mit Knoten und Kanten (Auer et al. 2013). Jedes Subjekt und Objekt des Graphen wird als Knoten dargestellt. Die Prädikate werden als beschriftete Kanten dargestellt, welche die Knoten miteinander verbinden. Die Resources werden mit Eigenschaften und Werten beschrieben. Eigenschaftswerte können in der Position des Objekts in Form von Buchstabensymbolen (Literals) dargestellt werden (Auer et al. 2013).

Mit RDF sollen Resources auf Basis von IRIs (Internationalized Resource Identifiers) als eindeutig identifiziert werden können (Schreiber et al. 2014). Außerdem gibt es verschiedene RDF-Vokabulare, welche eine Sammlung von IRIs darstellen (Cyganiak et al. 2014). Diese IRIs sind als Namespace IRIs bekannt und besitzen meist eine Abkürzung, welche als Namespace Prefix bezeichnet wird. Ein Beispiel dafür ist der Namspace IRI `http://www.w3.org/1999/02/22-rdf-syntax-ns#` mit dem Namespace Prefix `rdf`.

Zur Darstellung von RDF-Graphen gibt es viele unterschiedliche Notationen (Schreiber et al. 2014). Das bedeutet, ein Graph kann in unterschiedlichen Formaten schriftlich erfasst werden und trotzdem dieselbe Logik darstellen. Im Rahmen dieser Arbeit wurde zur Darstellung von RDF-Triple die Notation `Turtle` verwendet, da diese eine gute Kombination aus guter Lesbarkeit und einfacher Maschinenverarbeitung bietet. Andere Notationen sind `RDF/XML`, `RDFa` oder `JSON-LD`.

2.3.2 RDF-Schema

RDF-Schema, kurz RDFS, stellt ein RDF-Vokabular für RDF-Graphen zur Verfügung. Damit wird ermöglicht, dass die Syntax für RDF-Modelle definiert werden kann. Ein RDF-Schema Dokument ist selbst als RDF-Dokument dargestellt (Brickley et al. 2014).

Im RDF-Schema Dokument werden Klassen und Properties definiert. Klassen werden mit der Resource `rdfs:class` und Properties mit der Resource `rdf:Property` dargestellt. Um Subklassen oder Subproperties zu definieren, gibt es die Properties `rdfs:isSubClassOf` und `rdfs:subPropertyOf`. Mit den Properties `rdfs:domain` und `rdfs:range` kann die Klasse für Subjekte oder Objekte in einzelnen Properties festgelegt werden.

Das RDF-Schema ermöglich es, eigene Vokabulare mit einer eigenen Syntax zu definieren (Brickley et al. 2014). Dies wird auch im Rahmen dieser Arbeit durchgeführt und wird benötigt, um von einem erstellten RDF-Modell neue Beziehungen, also RDF-Tripel, auf Basis eines definierten RDF-Schemas ableiten zu können (Decker et al. oJ). Das ursprüngliche RDF-Modell stellt gemeinsam mit den neuen abgeleiteten RDF-Tripel das sogenannte Inferenzmodell dar.

2.3.3 SPARQL

SPARQL steht für *SPARQL Protocol And RDF Query Language,* und ist eine Abfragesprache für RDF-Modelle (Dengel 2012, S. 162 f). Die Syntax von SPARQL ist ähnlich wie bei der Abfragesprache SQL, da auch Schlüsselwörter wie `SELECT` und `WHERE` verwendet werden. Im `SELECT`-Teil werden, wie auch bei SQL, jene Variablen angegeben, die im Ergebnis der Anfrage aufscheinen sollen. Der `WHERE`-Teil wird in Form von RDF-Tripel in der Turtle-Notation angegeben. Bei Variablen wird ein Fragezeichen (?) vorangestellt und sie können an jeder Position eines RDF-Tripels aufscheinen. Das Ergebnis einer SPARQL-Abfrage beinhaltet verschiedene Kombinationen aus den im `SELECT`-Teil angegebenen Variablen.

SPARQL stellt weitere Funktionen, welche anhand von verschiedenen Schlüsselwörtern verwendet werden können, zur Verfügung (Prud'hommeaux und Seaborne 2013). In diesem Kapitel werden nur jene Funktionen beschrieben, die für diese Arbeit relevant sind.

Mit dem Schlüsselwort `UNION` können verschiedene Variablenkombinationen vereint werden (Prud'hommeaux und Seaborne 2013). Das Schlüsselwort `BIND` ermöglicht es, auf Basis des RDF-Models neue Informationen abzuleiten oder zu ermitteln und diese an eine bestimmte Variable zu binden.

SPARQL erlaubt unterschiedliche Formen an Abfragen (Dengel 2012, S. 162 f). Neben der `SELECT`- Abfrage gibt es auch noch folgende andere: `CONSTRUCT`, `ASK` und `DESCRIBE`. In Rahmen dieser Arbeit wird nur die `SELECT` -Abfrage verwendet.

2.1 metaCASE-Werkzeuge

CASE-Werkzeuge (Computer-Aided Software Engineering) sind IT-gestützte Werkzeuge, die Entwickler bei der Konzeption oder dem Entwurf von Softwareprogrammen unterstützen. Da in diesem Bereich sehr viele unterschiedliche Methoden in stark heterogenen Umgebungen angewandt werden, welche spezifische und individuelle Kontexte darstellen, sind CASE-Werkzeuge nicht immer die beste Lösung für gewisse Aufgaben. MetaCase-Werkzeuge lösen dieses Problem, da man damit kontextspezifische CASE-Werkzeuge erstellen kann (Ebert et al. 1997).

Um das PESTEL-Modeler-Werkzeug, wie im Kapitel 1.1 beschrieben, implementieren zu können, wird ein metaCASE Werkzeug benötigt, welches die Erstellung und Verwendung von domainspezifischen Sprachen unterstützt. Es muss möglich sein, ein Metamodell zu erstellen, welches genau für die im Kapitel 1.1 beschriebene Problemstellung passend ist. Das bedeutet, das Werkzeug soll in der Lage sein, eine Sprache zu definieren, welche alle Sachverhalte der Domäne darstellen kann und keine zusätzlichen Darstellungen außerhalb der Domäne erlaubt.

Neben der Definition einer domainspezifischen Sprache muss das metaCASE-Werkzeug auch eine Möglichkeit aufweisen, erstellte Modelle auf das RDF-Format zu parsen und zu exportieren. Um externe RDF-Dokumente graphisch darstellen zu können, muss das metaCASE-Werkzeug auch einen Import ermöglichen.

Damit Modelle, die mit der erstellten domainspezifischen Sprache modelliert wurden, auch in Echtzeit manipuliert werden können, muss das metaCASE-Werkzeug eine passende API dafür bereitstellen, beziehungsweise darüber verfügen.

MetaEdit+ ist ein Beispiel für ein metaCASE-Werkzeug, welches alle oben beschriebenen Anforderungen erfüllt und wurde daher für diese Arbeit ausgewählt. Es bietet verschiedene User-Interfaces, um kontextspezifische Modellierungssprachen zu entwickeln, ohne dass fortgeschrittene Programmierkenntnisse vorhanden sein müssen (Pohjonen 2005). Dazu stellt MetaEdit+ verschiedene Werkzeuge zur Verfügung, mit denen Objekte, Beziehungen, Rollen, Eigenschaften und Graphen definiert werden können und so ein Metamodell erstellt werden kann. Im Diagramm Editor kann auf Basis des definierten Metamodells die Kontext-spezifische Modellierungssprache angewandt werden. Die erstellten Modelle werden im sogenannten *Graph Browser* gespeichert und können von dort aus aufgerufen und bearbeitet werden. Neben dem Diagramm Editor stellt MetaEdit+ auch eine Funktion der Code-Generation zur Verfügung. Dazu wird der sogenannte Generator Editor verwendet, der es ermöglicht, verschiedene Generatoren zu definieren, um erstellte Modelle in einer gewünschten vordefinierten Form exportieren zu können.

MetaEdit+ ermöglicht den Aufruf von Kommandozeilenbefehle, um zum Beispiel externe Java-Programme ausführen zu können. Des Weiteren wird eine API zur Verfügung gestellt, mit der über externe Programme die Modelle in Echtzeit manipuliert und sämtliche MetaEdit+-Funktionen ausgeführt werden können.

3 Architektur

Das Werkzeug PESTEL Modeler stellt eine Kombination der Methoden, Technologien und Frameworks aus dem Kapitel 2 dar. In diesem Kapitel wird beschrieben, aus welchen Komponenten und Artefakten das Werkzeug besteht und wie diese miteinander in Verbindung stehen und kommunizieren.

In Abbildung 9 ist die Architektur im groben Überblick in einer Kombination aus UML Use-Case- und UML Komponentendiagramm ersichtlich.

Die Hauptsysteme des PESTEL Modelers stellen die *MetaEdit+ Software*, das Java-Programm *RDF-Import,* sowie das Java-Programm *OT Analyse* dar. Innerhalb der MetaEdit+ Software gibt es die Komponenten *Diagramm Editor, Graph Browser, Metamodell Browser, MERL-Generator RDF-Export, MERL-Generator RDF-Import, MERL-Generator OT Analyse, MERL-Generator Reset OT Analyse,* sowie den *SOAP Server*.

Der User der Anwendung ist zugleich der Modellierer und der Analyst. Der User verwendet das User Interface des Diagramm-Editors, um Modelle auf Basis des definierten Metamodells zu erstellen. Das Artefakt *Metamodell* liegt in der Komponente *Metamodell Browser*. Die erstellten Modelle werden als Artefakt *Modell* in der Komponente *Graph Browser* gespeichert. Es besteht die Möglichkeit, mehrere Modelle zu erstellen, welche durch Auswahl im Diagramm Editor dargestellt werden.

Der User kann, durch die im Diagramm Editor bereitgestellten Buttons, die vier verschiedenen Funktionen zur Analyse ausführen. Den Mittelpunkt des Werkzeugs stellt also der Diagramm Editor dar, da von dort aus alles gesteuert werden kann.

Soll der RDF-Export ausgeführt werden, wird der MERL-Generator *RDF-Export* vom Editor aus gestartet. Dabei wird das Artefakt *RDF-Modell* erzeugt und extern gespeichert.

Beim RDF-Import wird der Generator *RDF-Import* gestartet. Von dort aus wird das Java-Programm *RDF-Import* ausgeführt, welches ein externes RDF-Modell einliest und daraus das Artefakt *XML Import Dokument* generiert. Anschließend wird das *XML Import Dokument* vom *RDF-Import Generator* importiert und in die Komponente *Graph Browser* geladen.

Damit die OT Analyse ausgeführt werden kann, muss, bevor diese gestartet wird, der SOAP Server über das für den User zur Verfügung gestellte MetaEdit+ API Tool gestartet werden. Über den Diagramm Editor wird zuerst der Generator RDF-Export ausgeführt und anschließend der Generator OT Analyse gestartet, welcher wiederum das gleichnamige Java-Programm ausführt. Innerhalb des Java-Programms wird die Abfrage auf das Artefakt *RDF-Modell* unter Verwendung der externen Artefakte *SPARQL Query, RDFS iStar* und *RDFS Pestel* ausgeführt. Auf Basis des Ergebnisses der Abfrage wird über SOAP Calls zum SOAP Server das im Diagramm Editor geöffnete Modell manipuliert.

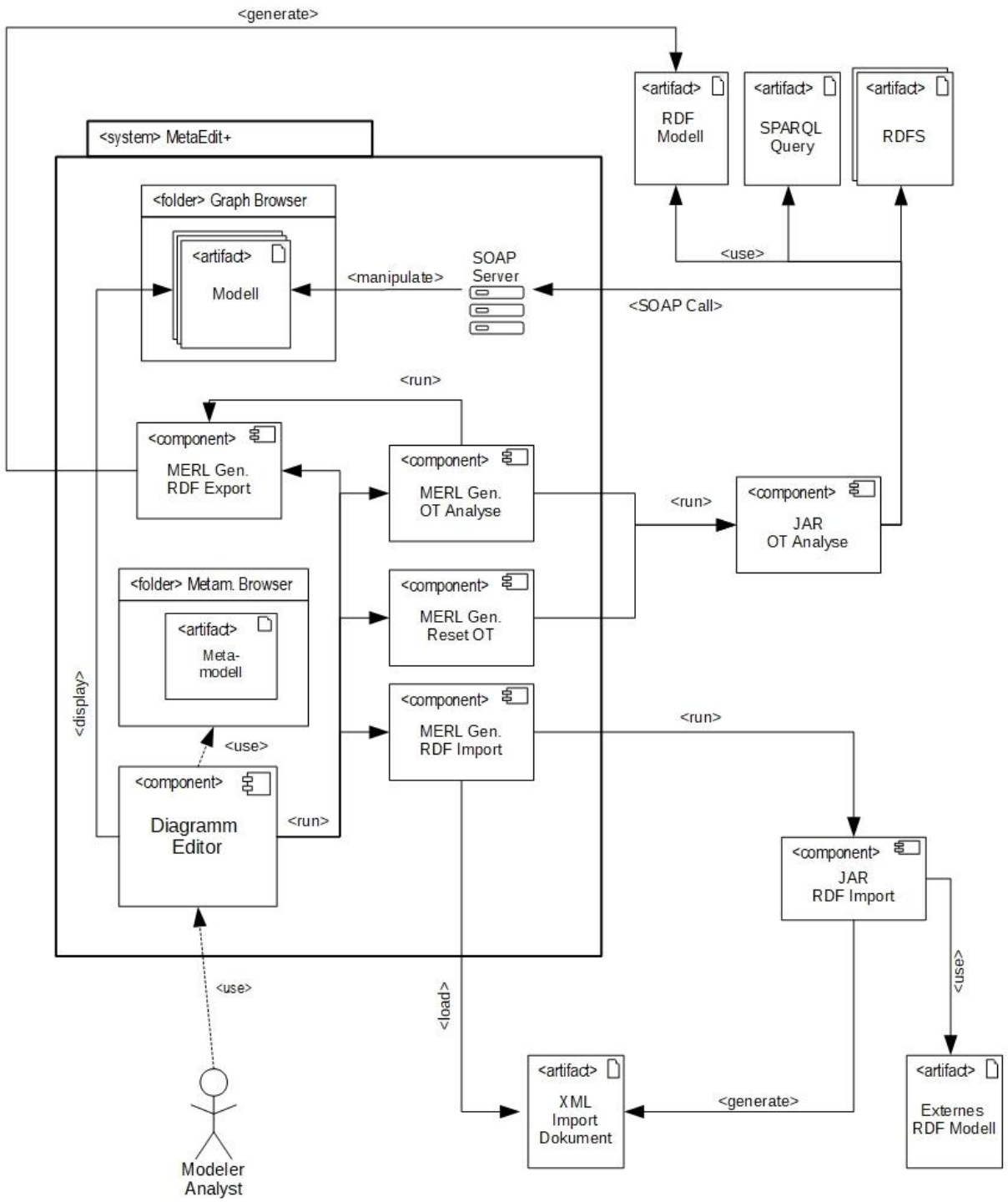


Abbildung 9: Architektur

Beim Zurücksetzen der OT Analyse wird ähnlich vorgegangen. Hier muss ebenso der SOAP Server zuerst vom User gestartet werden. Anschließend wird der Generator *Reset OT Analyse* gestartet, welcher wiederum das Java-Programm *OT Analyse* ausführt. Innerhalb des Java-Programms wird nun aber nur eine zuvor ausgeführte OT Analyse über SOAP Calls zurückgesetzt und das im Diagramm Editor geöffnete Modell manipuliert. Eine Ausführung der SPARQL-Abfrage ist hier nicht notwendig.

4 Anwendung

Wie bereits im Kapitel 3 erwähnt, stellt das metaCASE-Werkzeug MetaEdit+ gemeinsam mit dem integrierten Editor, sowie dem modellierten Metamodell das zentrale Element des Werkzeugs dar, da bei der Anwendung von dort aus alle implementierten Funktionen ausgeführt und gesteuert werden können. Zu den Funktionen zählen folgende:

- Erstellung eines Modells im Editor
- Export des erstellten Modells als RDF-Dokument
- Import eines externen RDF-Dokuments
- Chancen-Risiken-Analyse für einen bestimmten Actor
- Zurücksetzen der Chancen-Risiken-Analyse

In den folgenden Kapiteln wird aufgezeigt, wie die einzelnen Funktionen des Werkzeugs PESTEL Modeler angewandt werden können.

4.1 Modellerstellung

Um ein Modell im Editor erstellen zu können, muss zuerst das Werkzeug MetaEdit+ geöffnet, das richtige Repository ausgewählt und der Benutzername, sowie das Passwort eingegeben werden. Anschließend erhält man Zugang zum Graph Browser, wo man neue Graphen auf Basis des hinterlegten Metamodells erstellen kann. Dazu wählt man die Option „Create Graph“ aus und erstellt einen Graphen vom Typ PESTEL i-star, welcher im Diagramm Editor geöffnet werden soll.

Bei der Erstellung des Graphen werden die drei Eigenschaften Autor, Default Prefix und Company abgefragt. Der Autor ist jener User, der das Modell erstellt. Der Default Prefix, wird für den RDF-Export verwendet. Wenn diese Eigenschaft nicht angegeben wird, wird standardmäßig der Prefix `http://example.org/` verwendet. Bei der Eigenschaft Company kann man angeben, für welches Unternehmen oder welche Organisation die Analyse erstellt wird. Die Angabe aller Eigenschaften ist optional und kann auch später noch geändert oder angepasst werden.

Anschließend öffnet sich der Editor, mit dem nun das Modell erstellt werden kann. Es werden unterschiedliche Menüleisten zur Verfügung gestellt. Dazu zählen die Werkzeugleiste, sowie die Seitenleiste. Die Werkzeugleiste beinhaltet alle definierten Objekte und Beziehungen, mit denen das Modell im Zeichenbereich erstellt werden kann. In der Seitenleiste befindet sich eine Liste mit allen erstellten Objekten nach Typ geordnet. Durch Klicken auf ein Objekt oder eine Beziehung werden ebenso in der Seitenleiste die Eigenschaften des Objekts oder der Beziehung angezeigt.

Durch Klicken auf die Objekte und Beziehungen in der Werkzeugleiste können die einzelnen Objekte und Beziehungen erstellt werden. Dazu muss für jedes Objekt, ausgenommen für die PESTEL-Faktoren, sowie für die abstrakten Intentional Element-Objekte, ein Name angegeben werden.

Zur Modellierung wird als Beispiel eine Case-Study, welche von (Schütz und Schrefl 2017) dargestellt wird, herangezogen. In dieser Case-Study wird für die Fluglinie Ryan Air eine PESTEL-Analyse durchgeführt und Chancen und Risiken werden abgeleitet.

In Abbildung 10 ist ersichtlich, wie ein Modell am Beispiel des Unternehmens Ryan Air erstellt werden kann. Dazu werden zwei Actor-Objekte *Airline* und *Low-Fare-Airline* angelegt. Der Actor *Low-Fare-Airline* wird mit der Beziehung *isA* als Subklasse von *Airline*

definiert. Ryan Air wird als *Agent* dargestellt und mit der Beziehung *participatesIn* die Zugehörigkeit zum Actor *Low-Fare-Airline* definiert.

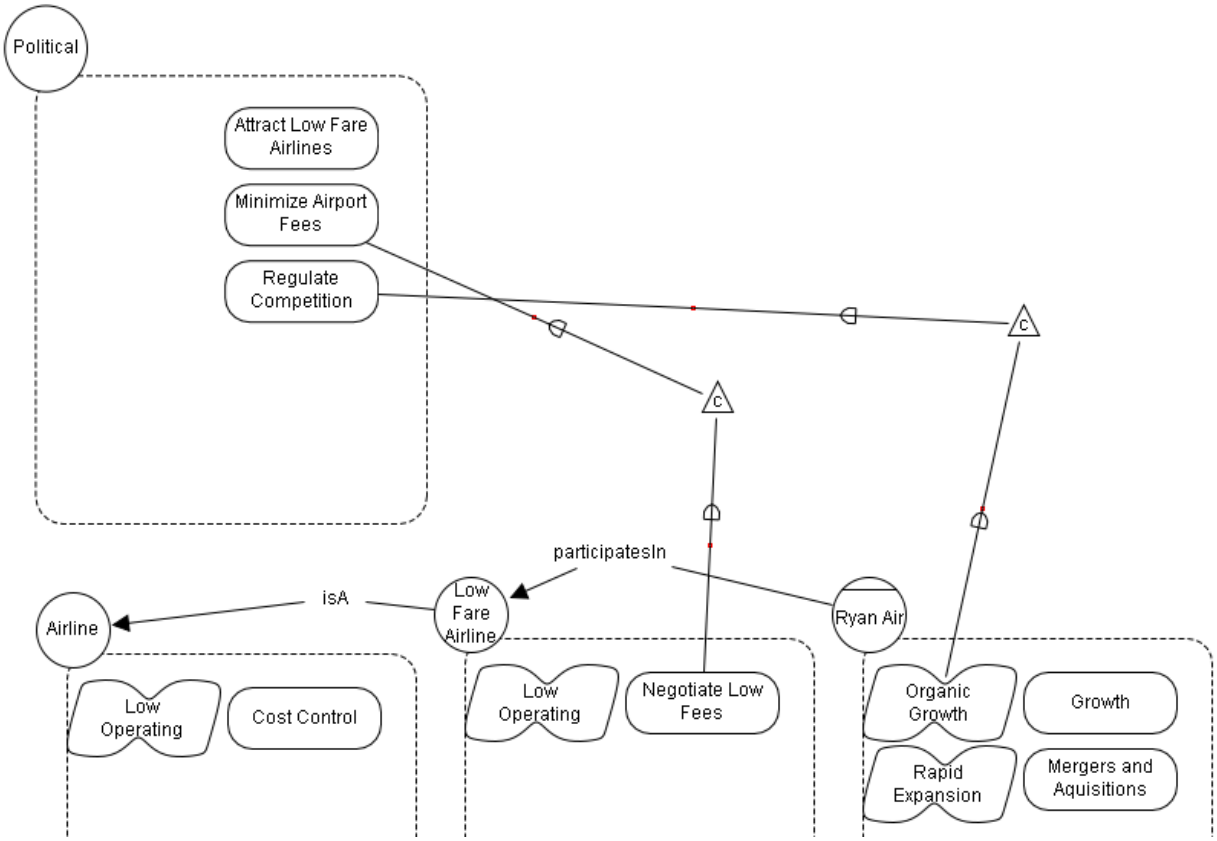


Abbildung 10: Aufbau Metamodell

Für die Analyse wird auf Basis des PESTEL-Faktors Politik durchgeführt. Dazu wird ein Political-Objekt erstellt.

Die Absichten der einzelnen Actors und Agents werden mit den Objekten *Goal, Quality, Task und Resource* im Subgraphen der *Actors* und *Agents* dargestellt. Mit rechtem Mausklick auf die *Actors* oder *Agents* kann durch Auswahl der Option „Open Subgraph“ ein Subgraph für jeden *Actor* oder *Agent* erstellt werden. Für die Modellierung der Subgraphen öffnet sich ein neues Editor-Fenster, worin die *Goals, Qualites, Tasks* und *Resources* gemeinsam mit ihren Beziehungen modelliert werden können. In Abbildung 11 ist als Beispiel der Subgraph des Agents *Ryan Air* ersichtlich. Darin befindet sich das Goal *Growth*, welches von der Quality *Rapid Expansion* mit der Beziehung *qualifies* qualifiziert wird. Die Quality *Organic Growth* wird als negativer Einfluss auf die Quality *Rapid Expansion* durch die Beziehung *hurts* dargestellt. Im Gegensatz dazu wird das Goal *Mergers and Aquisitions* als positiver Einfluss durch die Beziehung *helps* dargestellt.

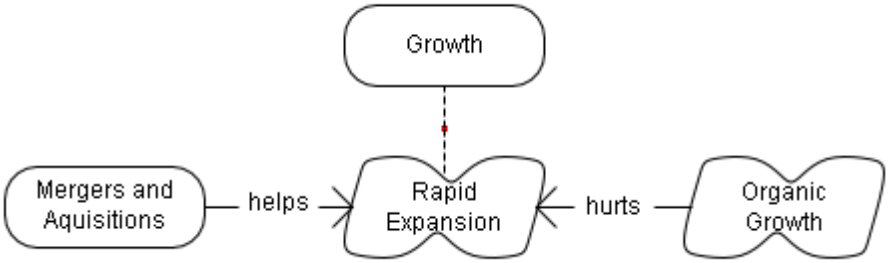


Abbildung 11: Aufbau Metamodell innerhalb des Subgraphen

Wie in Abbildung 10 ersichtlich, werden im erstellten Modell zwei Abhängigkeiten vom PESTEL-Faktor Political dargestellt. Die Quality *Organic Growth*, welche im Subgraphen des Agents *Ryan Air* liegt, ist abhängig vom Goal *Regulate Competition*, welche zum PESTEL-Faktor *Political* gehört. Des Weiteren ist das Goal Negotiate Low Fees des Actors *Low-Fare-Airline* abhängig vom Goal *Minimize Airport Fees* des PESTEL-Faktors *Political*. Um eine Abhängigkeitsbeziehung (*dependerOf*) erstellen zu können, muss zuerst das *Dependum* als abstraktes *Intentional Element* erstellt werden. Anschließend werden vom *Depender-Element* zum *Intentional Element,* sowie vom *Intentional Element* zum *Dependee-Element* dependerOf-Beziehungen erstellt.

4.2 RDF-Export

Zur Ausführung der Funktion *RDF-Export* wird im Diagramm Editor ein Button zur Verfügung gestellt. Durch Klick auf den Button wird auf Basis des erstellten Modells, wie zum Beispiel in Abbildung 10 ersichtlich, ein RDF-Dokument generiert. Dabei öffnet sich ein Fenster, in dem das generierte RDF-Dokument in einer Liste enthalten ist. Durch Doppelkick wird das Dokument geöffnet. Ebenso kann durch Klick auf den Button *Open Folder* der Ordner, in dem das Dokument gespeichert wurde, geöffnet werden.

In ist ein generiertes RDF-Dokument ersichtlich, welches auf Basis des in Abbildung 10 erstellten Modells generiert wurde. Als IRI wird für die Objekte der jeweilige Prefix inklusive der Name, welcher bei der Erstellung der Objekte angeben wird, verwendet. Dabei werden alle Leerzeichen im Namen mit einem Unterstrich ersetzt. Für die abstrakten *Intentional Elements* (Zeile 6 und 10 in Listing 1) wird eine zufällige UUID erstellt, damit auch diese Elemente eine eindeutige IRI im RDF aufweisen.

Listing 1: Generiertes RDF-Dokument

```
01  @prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
02  @prefix istar: <http://xmlns.com/istar/0.1/> .
03  @prefix pestel: <http://xmlns.com/pestel/0.1/> .
04  @prefix : <http://example.org/> .
05
06  :6d440acc-0557-4e14-945d-a3cac3895e92 rdf:type istar:IntentionalElement;
07     istar:dependerOf :Regulate_Competition;
08     istar:dependeeOf :Organic_Growth.
09
10  :176663e3-cda8-4d6b-ae67-e67388719902 rdf:type istar:IntentionalElement;
11     istar:dependerOf :Minimize_Airport_Fees;
12     istar:dependeeOf :Negotiate_Low_Fees.
13
```

```
14  :Airline rdf:type istar:Actor;
15     istar:wants :Cost_Control;
16     istar:wants :Low_Operating_Costs.
17
18  :Low_Fare_Airline rdf:type istar:Actor;
19     istar:isA :Airline;
20     istar:wants :Negotiate_Low_Fees;
21     istar:wants :Low_Operating_Costs.
22
23  :Ryan_Air rdf:type istar:Agent;
24     istar:participatesIn :Low_Fare_Airline;
25     istar:wants :Growth;
26     istar:wants :Mergers_and_Aquisitions;
27     istar:wants :Organic_Growth;
28     istar:wants :Rapid_Expansion.
29
30  pestel:Political rdf:type pestel:PESTELFactor;
31     istar:wants :Attract_Low_Fare_Airlines;
32     istar:wants :Minimize_Airport_Fees;
33     istar:wants :Regulate_Competition.
34
35  :Cost_Control rdf:type istar:Goal.
36
37  :Low_Operating_Costs rdf:type istar:Quality;
38     istar:qualifies :Cost_Control.
39
40  :Negotiate_Low_Fees rdf:type istar:Goal;
41     istar:helps :Low_Operating_Costs;
42     istar:dependerOf :176663e3-cda8-4d6b-ae67-e67388719902.
43
44  :Low_Operating_Costs rdf:type istar:Quality.
45
46  :Growth rdf:type istar:Goal.
47
48  :Mergers_and_Aquisitions rdf:type istar:Goal;
49     istar:helps :Rapid_Expansion.
50
51  :Organic_Growth rdf:type istar:Quality;
52     istar:hurts :Rapid_Expansion;
53     istar:dependerOf :6d440acc-0557-4e14-945d-a3cac3895e92.
54
55  :Rapid_Expansion rdf:type istar:Quality;
56     istar:qualifies :Growth.
57
58  :Attract_Low_Fare_Airlines rdf:type istar:Goal.
59
60  :Minimize_Airport_Fees rdf:type istar:Goal;
61     istar:refines :Attract_Low_Fare_Airlines;
62     istar:dependeeOf :176663e3-cda8-4d6b-ae67-e67388719902.
63
64  :Regulate_Competition rdf:type istar:Goal;
65     istar:dependeeOf :6d440acc-0557-4e14-945d-a3cac3895e92.
```

4.3 RDF-Import

Die Funktion RDF-Import ermöglicht es, ein externes RDF-Dokument in MetaEdit+ zu importieren und im Diagramm Editor graphisch anzuzeigen. Wenn man sich in MetaEdit+ im Graph Browser befindet, wählt man im Menü *Edit*>*Generate* aus. Daraufhin öffnet sich die

Generatorliste, bei der man *ImportRDF* selektiert. Mit Hilfe eines File-Dialogs kann man zum externen RDF-Dokument navigieren und dieses für den RDF-Import auswählen.

Das importierte Modell erscheint dann im Graph Browser. Die Anordnung der Objekte und Beziehungen ist direkt nach dem Import willkürlich. Im Diagramm Editor gibt es im Menü unter *Graph>Layout* die Möglichkeit, die Anordnung der Objekte auf Basis der Beziehungsrichtungen anzupassen.

Nach dem Import eines externen RDF-Dokuments kann das Modell im Diagramm Editor beliebig erweitert oder angepasst werden.

4.4 Chancen-Risiken-Analyse

Im Diagramm Editor wird ein Button *OTAnalyse* zur Verfügung gestellt, mit dem die Chancen-Risiken-Analyse, auch OT-Analyse genannt, gestartet werden kann. Dabei öffnet sich ein Dialog-Fenster, in dem der *Actor* oder *Agent*, für welchen die Analyse durchgeführt werden soll, angegeben werden muss. Bei Angabe eines nicht im Modell existierenden *Actors* oder *Agents* wird die Analyse abgebrochen und beendet.

Wird der Actor oder Agent richtig angegeben, so kommt es zur Ausführung der Analyse. Dabei wird das Ergebnis, wie in Abbildung 12 ersichtlich, graphisch dargestellt. Der Actor oder Agent, für welchen die Analyse ausgeführt werden soll, wird grau schattiert. Die Objekte, welche als Chancen identifiziert wurden, werden grün schattiert und die Objekte, welche als Risiken identifiziert wurden, werden rot schattiert. So werden in diesem Beispiel die Objekte *Attract Low Fare Airlines* und *Minimize Airport Fees* als Chancen und das Objekt *Regulate Competition* als Risiko erkannt.

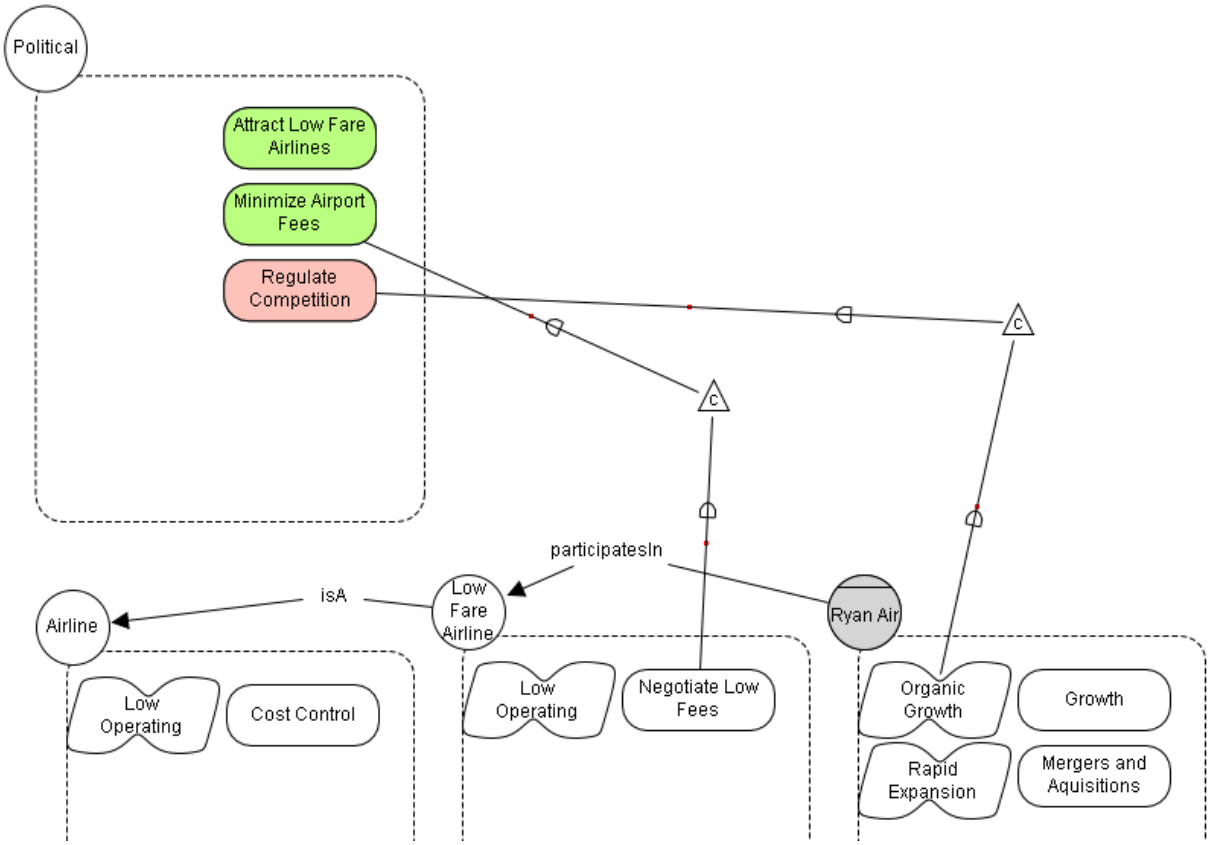


Abbildung 12: Ergebnis der Chancen-Risiken-Analyse

Neben der Ausführung der Chancen-Risiken-Analyse besteht auch die Möglichkeit, die Chancen-Risiken-Analyse zurückzusetzen. Dazu wird der Button *ResetOTAnalyse* zur Verfügung gestellt. Durch Klicken dieses Buttons wird die graphische Darstellung der Ergebnisse von zuvor ausgeführten Analysen wieder bereinigt.

5 Implementierung

In den folgenden Kapiteln werden nun die Implementierungsdetails der im Kapitel 3 erwähnten Komponenten und Artefakte beschrieben und aufgezeigt.
Zur Beschreibung der implementieren Java-Programme werden relevante Programm-Code-Teile aufgelistet und beschrieben. Der Programm-Code für das gesamte Java-Programm ist im GitHub-Repository unter `https://github.com/evamair/PESTEListarModeler` bereitgestellt.

5.1 Metamodell

MetaEdit+ stellt verschiedene Editoren zur Verfügung, mit denen ein Modell erstellt werden kann (MetaCase 2017b). Diese Arbeit berücksichtigt nur den Diagramm Editor. Damit können Graphen als Diagramme erstellt und verwaltet werden. Damit dies möglich ist, muss vorab ein Metamodell erstellt werden.

Mit dem Werkzeug MetaEdit+ kann eine Modellierungssprache, also ein Metamodell, erstellt werden (MetaCase 2017a). Dafür wird das Metamodellierungsframework GOPPRR verwendet. Die Abkürzung GOPPRR steht für all jene Elemente, die für das Metamodell definiert werden müssen und umfasst folgende:

- Graph
- Object
- Relationship
- Role
- Property

In weiterer Folge dieser Arbeit werden die deutschen Begriffe Graph, Objekt, Beziehung, Rolle und Eigenschaft für die GOPPRR Elemente verwendet.

In Abbildung 13 sieht man das definierte Metamodell mit all den definierten Objekten und deren Beziehungen zueinander. Die im Kapitel 2.1 beschriebene PESTEL Analyse, sowie

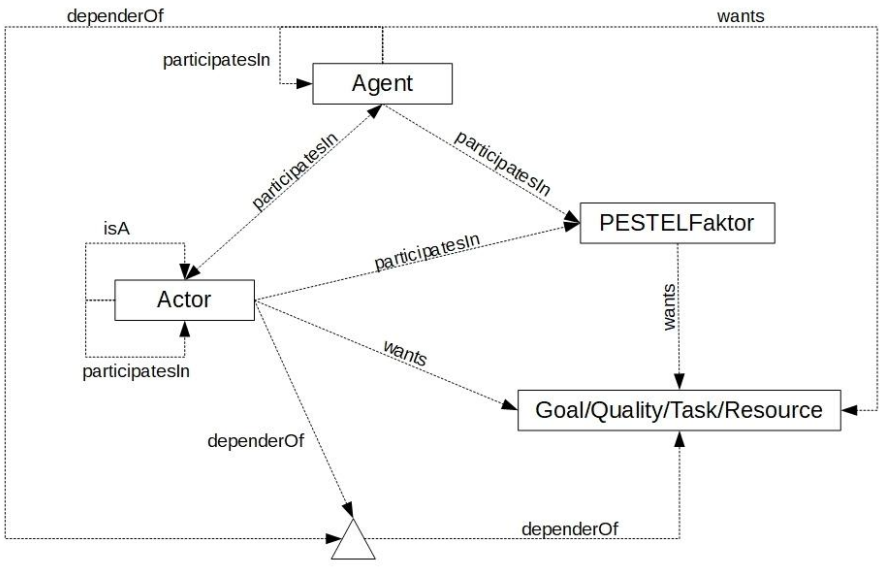


Abbildung 13: Aufbau Metamodell

das im Kapitel 2.2 beschriebene Framework iStar 2.0 werden nun im Metamodell miteinander kombiniert. Das bedeutet, es werden alle Elemente von iStar 2.0 als Objekte

oder Beziehungen inklusive Rollen und Eigenschaften definiert. Zusätzlich werden die einzelnen PESTEL Faktoren als zusätzliche fixe Actor-Objekte festgelegt.

Die Beziehung *istar:wants*, wie im Framework iStar 2.0 definiert, wird im Metamodell nicht als klassische Beziehung dargestellt. Stattdessen werden alle Objekte, die zu einem Actor oder Agent gehörenj in einem eigenen Subgraphen beziehungsweise in einer dargestellten Grenzlinie um den Actor oder Agent dargestellt. Nur die Objekte *Goal*, *Task*, *Resource* und *Quality* können innerhalb dieses Subgraphs definiert sein, da die Range der Beziehung *istar:wants* nur von einem dieser Elemente besetz ist. In Abbildung 14 sind alle Beziehungen, die nur innerhalb eines Subgraphen, also der Grenzlinie eines Actor- oder Agent-Objekt liegen, dargestellt.

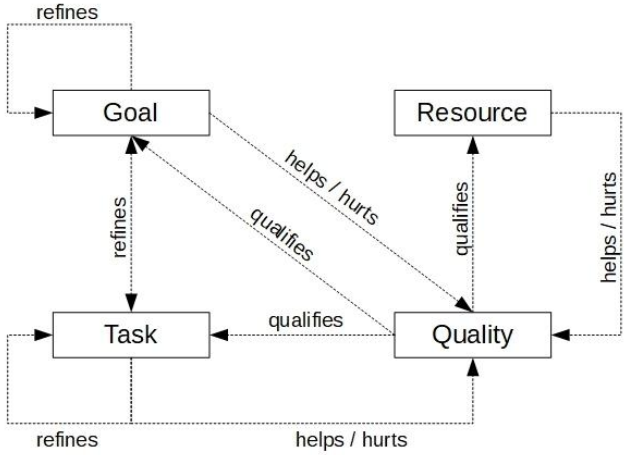


Abbildung 14: Aufbau Metamodell innerhalb des Subgraphen

Um in MetaEdit+ ein Metamodell zu definieren, wird die sogenannte MetaEdit+ Workbench angeboten (MetaCase 2017b). Dafür ist ein Set an verschiedenen Werkzeugen bereitgestellt, die die Definition der Graphen, Objekte, Beziehungen, Rollen und Eigenschaften ermöglichen.

Außerdem gibt es die Möglichkeit, mit weiteren Hilfswerkzeugen die Symbole, die Icons und den Eingabedialog für die GOPPRR-Elemente festzulegen (MetaCase 2017b). Mit dem Symbol Editor wird für die Elemente Objekt, Beziehung und Rolle ein grafisches Symbol definiert, mit dem diese Elemente im Diagramm Editor dargestellt werden. Die Symbole setzen sich aus verschiedenen Formen, Farben oder Textfeldern zusammen. Die Textfelder, Farben und Formen können statisch definiert sein, aber auch dynamisch von den Eigenschaften des Elements abhängig sein, indem verschiedene Konditionen für die einzelnen Formen und Felder spezifiziert sind. Zusätzlich gibt es noch eine sogenannte „Connectable“-Form, mit der man einen Bereich um das Symbol definieren kann, mit dem eine Rolle einer Beziehung verbunden wird. Mit dem Icon Editor kann man ebenso ein grafisches Symbol definieren, welches in der Werkzeugleiste des Diagramm Editors zur Darstellung herangezogen wird.

5.1.1 Objekte und Eigenschaften

Das sogenannte Object-Tool ermöglicht die Erstellung und Bearbeitung der einzelnen Objekte des zu definierenden Metamodells (MetaCase 2017b).

Wie in Abbildung 15 ersichtlich, benötigt man zur Definition des Objekts den Namen, den Vorfahren, sowie das dazugehörige Projekt. Im Feld Properties gibt man die dazugehörigen

Eigenschaften des Objekts an. Wie bereits im Kapitel 5.1 erwähnt, kann man über den Symbol und den Icon Editor für jedes Objekt ein Symbol und Icon definieren.

Wie bereits erwähnt, werden alle relevanten Elemente des iStar 2.0 Frameworks als Objekte für das Metamodell definiert. Zusätzlich werden die einzelnen PESTEL-Faktoren

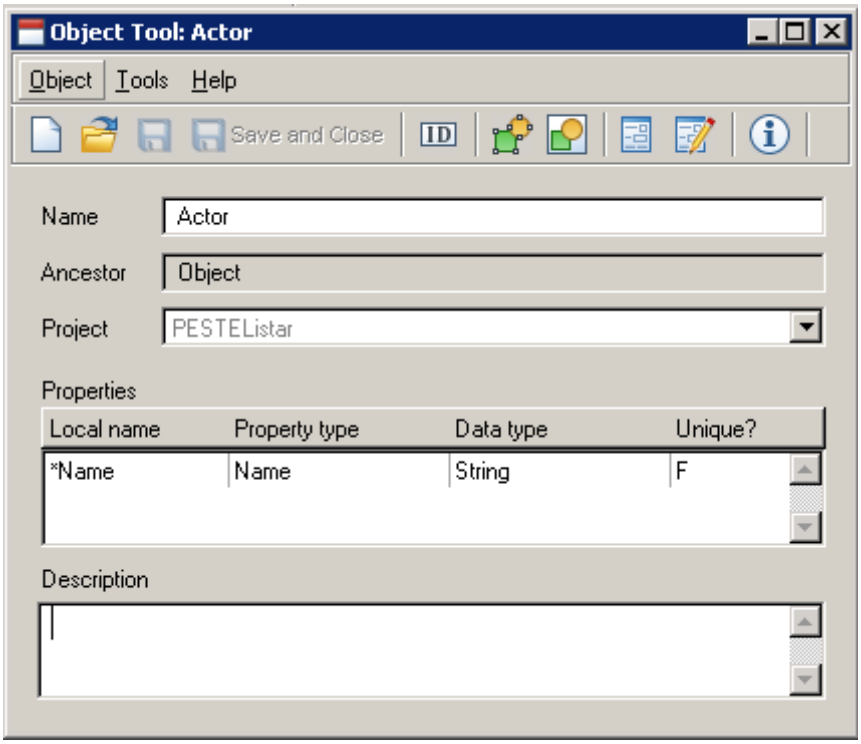


Abbildung 15: Object Tool

auch als eigene Objekte dargestellt. In Tabelle 3 sind alle Objekte mit Name, Vorfahre, Eigenschaften, Symbole und Icons aufgelistet. Die Objekte *Actor*, *ActorWithBoundary*, Agent, AgentWithBoundary, *IntentionalElement*, *Goal*, *Quality*, *Resource* und *Task* haben als Vorfahren die allgemeine Klasse Object zugeordnet. Anders ist das bei den einzelnen PESTEL-Factor-Objekten. Diese stellen eine Spezialisierung des Objekts PESTELFactor dar.

Eigenschaften der Objekte

Name	Data Type	Steuer-element	Default Wert	Regular Expression
Name	String	Input Field	-	[a-zA-Z\s]+
Analyse	Boolean	Option Field	False	
Opportunity	Boolean	Option Field	False	
Threat	Boolean	Option Field	False	

Tabelle 2: Eigenschaften der Objekte

Bis auf die PESTEL-Faktor-Objekte und das *IntentionalElement*-Objekt haben alle Objekte die Eigenschaft eines Namens. In Tabelle 2 sieht man die festgelegten Parameter für die Eigenschaften. Die Eigenschaft Name hat den Datentyp *String* und als Regular Expression ist *[a-zA-Z\s]+* definiert. Das bedeutet, dass der Name der Objekte nur aus Klein- und Großbuchstaben von A bis Z, sowie aus Leerzeichen bestehen darf. Die Objekte *ActorWithBoundary* und *AgentWithBoundary* weisen zusätzlich die Eigenschaft Analyse auf.

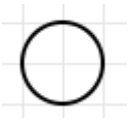
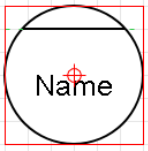
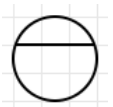
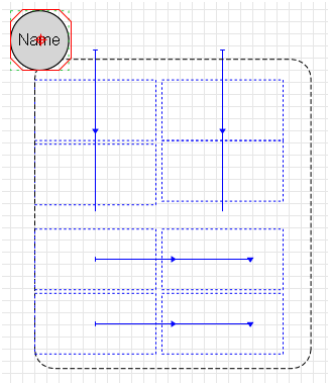
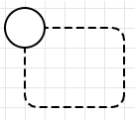
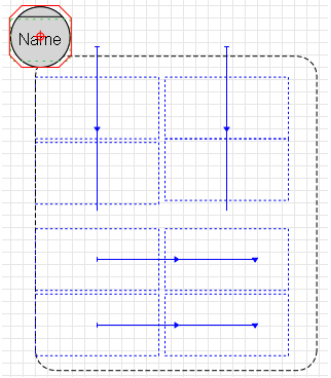
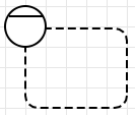
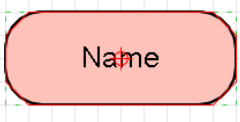
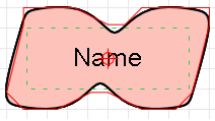
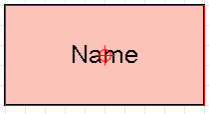
Wie man in Tabelle 2 ablesen kann, handelt es sich bei dieser Eigenschaft um einen Boolean-Wert. Dieser ist defaulmäßig auf *false* gesetzt und kann beim Erstellen des Objekts im Diagramm Editor auch nicht geändert werden. Dieser Wert wird nur über die MetaEdit+ API bei der OT-Analys auf *true* gesetzt, wenn die Analyse für diesen bestimmten *ActorWithBoundary* oder *AgentWithBoundary* ausgeführt wird. Ähnlich ist es bei den Eigenschaften Opportunity und Threat, welche bei den Objekten Goal, Quality, Resource und Task angeführt sind. Diese Eigenschaften sind ebenso vom Datentyp Boolean und werden nur über die MetaEdit+ API im Rahmen der OT-Analyse geändert. Werden diese Objekte für einen bestimmten Actor oder Agent als Chance oder Risiko identifiziert, so werden die Werte dieser Eigenschaften auf *true* gesetzt.

In Tabelle 3 sind für alle definierten Objekte die dazugehörigen Symbole und Icons abgebildet. Die rote Umrandung der einzelnen Objekte stellt, wie bereits im Kapitel 5.1 erwähnt, die „Connectable“-Form dar, also jenen Bereich, der bei einer Beziehung zu einem anderen Objekt verbunden wird. Bei all jenen Objekten, die die Eigenschaft eines Namens aufweisen, wird der konkrete Name des Objekts im Symbol angezeigt. Ausgenommen davon ist das IntentionalElement-Objekt. Bei den PESTEL-Faktor-Objekten wird als Text der jeweilige PESTEL-Factor angezeigt.

Bei den Objekten *ActorWithBoundary* und *AgentWithBoundary* ist eine Bedingung bei der Formatierung des Symbols definiert. Die Formatierung des Kreises ist abhängig von der Eigenschaft *Analyse*. Wird der Wert der Eigenschaft *Analyse* durch die MetaEdit+ API auf *true* gesetzt, so wird der Kreis des Symbols grau schattiert. Bei den Objekten *Goal*, *Quality*, *Resource* und *Task* ist dies ähnlich, denn auch hier hängt die Formatierung des Symbols von den Werten der Eigenschaften *Opportunity* und *Threat* ab. Wird der Wert der Eigenschaft *Opportunity* auf *true* gesetzt, wird das Symbol dieser Objekte grün schattiert. Wird der Wert der Eigenschaft *Threat* auf *true* gesetzt, wird es rot schattiert.

Wie bereits erwähnt, wird die *wants*-Beziehung nach dem Framework iStar 2.0 als Subgraph ausgehend von einem *Actor* oder *Agent* umgesetzt. Damit die Objekte des Subgraphen graphisch auch außerhalb dargestellt und über Beziehungen verbunden werden können, werden dynamische Ports benötigt. Seit der Version MetaEdit+ 5.0 gibt es das neue Symbolelement *Template*, welches erlaubt, Subobjekte, also jene des Subgraphen, dynamisch als Teil des Objektsymbols darzustellen (Kelly und Pohjonen 2013). Die Subobjekte agieren als dynamischer Port, das bedeutet, die Subobjekte können mit anderen Objekten über Beziehungen verbunden werden, obwohl sie eigentlich im Subgraphen des *Actors* oder *Agents* definiert sind (MetaCase 2017b). Wie in Tabelle 3 bei den Objekten ActorWithBoundary, AgentWithBoundary und PESTEL-Faktoren ersichtlich, wird für jedes der Objekte Goal, Quality, Resource und Task ein Template erstellt. Dargestellt wird das Template durch die blauen Rechtecke und Linien mit Pfeilen für die Anordnungsrichtung. Diese Darstellung ist nur im Symbol Editor und nicht im Diagramm Editor zu sehen. Unter den Einstellungen des Templates wird anschließend der Subgraph, sowie der Objekttyp des Subgraphen angeben, welcher in dem Template dargestellt werden soll (siehe Abbildung 16).

Objekte des Metamodells

Name	Vorfahren/ Superklasse	Eigen-schaften	Symbol	Icon
Actor	Object	Name		
Agent	Object	Name		
ActorWithBoundary	Object	Name Analyse		
AgentWithBoundary	Objekt	Name Analyse		
IntentionalElement	Object	-		
Goal	Object	Name Opportunity Threat		Wie Symbol
Quality	Object	Name Opportunity Threat		Wie Symbol
Resource	Object	Name Opportunity Threat		Wie Symbol

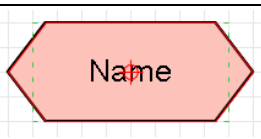
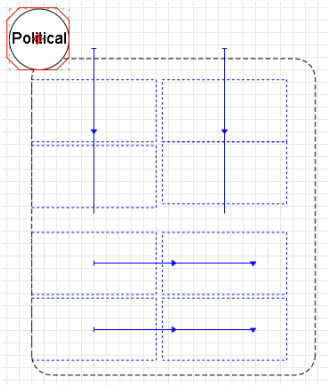
Task	Object	Name Opportunity Threat		Wie Symbol
PESTELFactor	Object	-	-	-
Political	PESTELFactor	-		
Economic	PESTELFactor	-	Wie bei Political	
Social	PESTELFactor	-	Wie bei Political	
Technological	PESTELFactor	-	Wie bei Political	
Environmental	PESTELFactor	-	Wie bei Political	
Legal	PESTELFactor	-	Wie bei Political	

Tabelle 3: Objekte des Metamodells

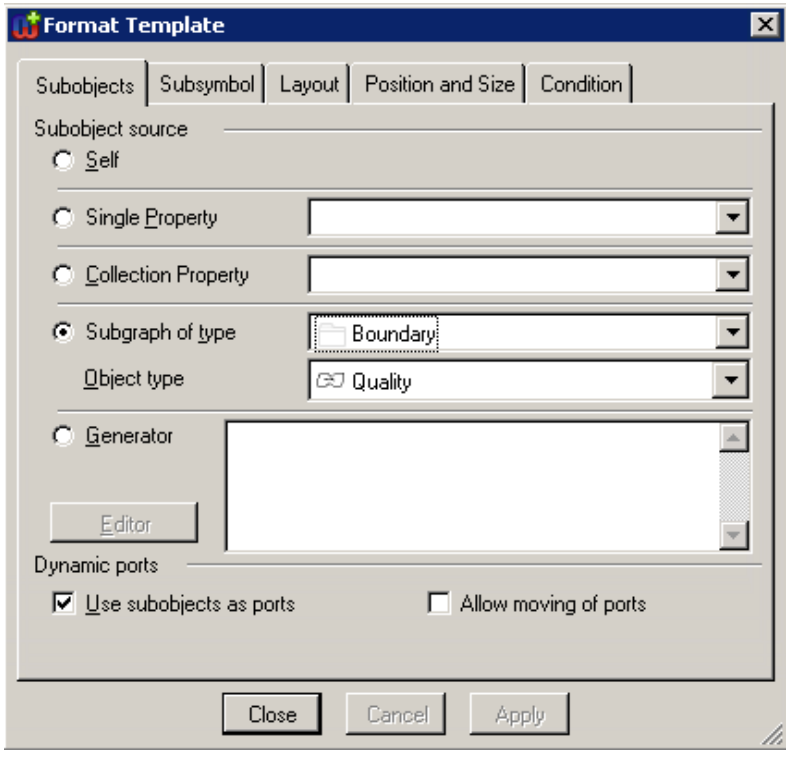


Abbildung 16: Template Einstellungen

5.1.2 Beziehungen

Das sogenannte Relationship-Tool ermöglicht die Erstellung und Bearbeitung der einzelnen Beziehungen des zu definierenden Metamodells (MetaCase 2017b). Bei der Darstellung, Funktionalität und Anwendung ist dieses Tool ident mit dem Object-Tool in Abbildung 15. Das bedeutet, man kann für jede Beziehung ebenso Name, Vorfahre, Symbol und Icon definieren.
In Tabelle 4 sind alle umgesetzten Beziehungen aufgelistet. Da der Vorfahre für alle Beziehungen gleich ist und keine der Beziehungen Eigenschaften aufweist, werden diese Werte nicht in der Tabelle angeführt.

Bei den Beziehungen *isA*, *participatesIn*, *helps* und *hurts* ist im Symbol Editor eine Darstellung für den Diagramm Editor definiert. Das hat zur Folge, dass im Diagramm Editor die Beschriftung der Beziehung hinzugefügt wird. Bei alle anderen Beziehungen wird keine Darstellung definiert, da auch laut dem Framework iStar 2.0 keine Beschriftung bei diesen Beziehungen existiert. Ebenso sind bei allen Beziehungen keine Icons definiert, somit wird in der Werkzeugleiste im Diagramm Editor kein Icon, sondern der Name der Beziehung angezeigt.

Wichtig zu erwähnen ist, dass die Regeln darüber, welche Objekte über welche Beziehungen verbunden werden dürfen, nicht in den Beziehungen selbst festgelegt wird, sondern bei der Definition der Graphen (MetaCase 2017b), wie im Kapitel 5.1.4 beschrieben.

Beziehungen des Metamodells

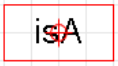
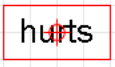
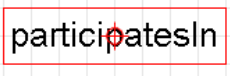
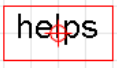
Name	Symbol	Name	Symbol
isA		hurts	
participatesIn		qualifies	-
dependerOf	-	refines	-
helps			

Tabelle 4: Beziehungen des Metamodells

5.1.3 Rollen

Das sogenannte Role-Tool ermöglicht die Erstellung und Bearbeitung der einzelnen Rollen des zu definierenden Metamodells (MetaCase 2017b). Bei der Darstellung, Funktionalität und Anwendung ist dieses Tool ident mit dem Object-Tool in Abbildung 15, sowie mit dem in Kapitel 5.1.2 beschriebene Relationship-Tool.

Auch hier werden für jede Rolle ein Name und ein Symbol definiert. Icons sind für Rollen nicht notwendig, da diese in der Werkzeugleiste des Diagramm Editors nicht aufscheinen. Rollen werden benötigt, um die Domain und die Range der Beziehungen festzulegen. Für jede Beziehung werden eine Domain-Rolle, sowie eine Range-Rolle definiert. Damit wird ermöglicht, dass genau festgelegt werden kann, welche Objekte welche Rollen einnehmen können. Ob eine Rolle die Domain oder die Range einer Beziehung darstellt, wird, wie im Kapitel 5.1.4. näher beschrieben, im Graph-Tool festgelegt.

In Tabelle 5 sind alle im Metamodell umgesetzten Rollen aufgelistet. Für die Rolle sind ebenso Symbole definiert. Diese spezifizieren die Stärke und die Darstellung der Linie, welche vom Objekt zur Beziehung beziehungsweise von der Beziehung zum Objekt führt.

Rollen des Metamodells

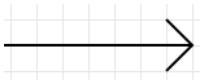
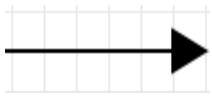
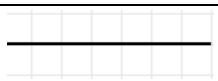
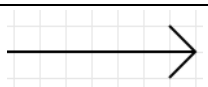
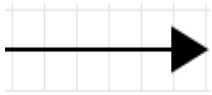
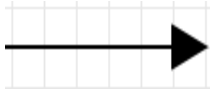
Name	Symbol	Name	Symbol
Subclass		Helpee	
Superclass		Injurer	
Member		Injuree	
Participation		Qualifier	
Depender		Qualifiee	
Dependee		Refiner	
Helper		Refinee	

Tabelle 5: Rollen des Metamodells

5.1.4 Graphen

Mit dem sogenannten Graph-Tool kann man verschiedene Typen von Graphen definieren (MetaCase 2017b). Der Unterschied vom Graph-Tool zu den anderen Tools (Object-, Role- und Relationship-Tool) ist, dass dieses nicht verwendet wird, um neue individuelle Elemente festzulegen, sondern alle bereits erstellten und definierten Elemente kombiniert, sowie weitere Einschränkungen und Einstellungen ermöglicht.

Beim PESTEL Modeler Werkzeug gibt es vier verschiedene Graphen, die definiert sind. Zum einen der Graph *PESTEL i-star* und zum anderen der Graph *Boundary.* Des Weiteren gibt es die Graphen *ActorBoundary* und *FactorBoundary,* für welche der Graph *Boundary* als Vorfahre definiert ist.

Das Graph-Tool besteht aus fünf Registerkarten, welche sich aus folgenden zusammensetzen: Basics, Types, Bindings, Subgraphs und Contraints.

Im Tab Basics kann man Namen, Vorfahren, Projekt, sowie die Eigenschaften des Graphen definieren. Der Graph *Boundary* verfügt über keine Eigenschaften. Ebenso verfügen die Graphen ActorBoundary und FactorBoundary über keine Eigenschaften, da diese alles vom Graphen Boundary erben. Wie in Tabelle 6 ersichtlich, verfügt der Graph *PESTEL i-star* über die Eigenschaften *Autor*, *Default Prefix* und *Company*. Alle drei Eigenschaften sind als Datentyp String definiert. Die Eigenschaft Default Prefix wird benötigt, damit der User für den RDF-Export auch einen eigenen Prefix angeben kann, welcher beim RDF-Export berücksichtigt wird. Die Eigenschaft Company stellt das Unternehmen oder die Organisation dar, für welche diese PESTEL Analyse durchgeführt werden soll.

Eigenschaften des Graphen PESTEL i-star

Eigenschaft	Datentyp
Autor	String
Default Prefix	String
Company	String

Tabelle 6: Eigenschaften des Graphen PESTEL i-star

Im Tab *Types* gibt man alle Objekte, Beziehungen und Rollen an, welche im definierten Graphen zur Verfügung stehen sollen. Wie in Abbildung 18 ersichtlich, sind alle Objekte, Beziehungen und Rollen angeführt, welche für die Erstellung des PESTEL i-star Graphen benötigt werden. Nicht darin enthalten sind jene Objekte, Beziehungen und Rollen die im Subgraphen eines Actors modelliert werden. Diese Objekte Beziehungen und Rollen werden als Typen für den Graphen *Boundary* definiert, wie in Abbildung 17 ersichtlich.

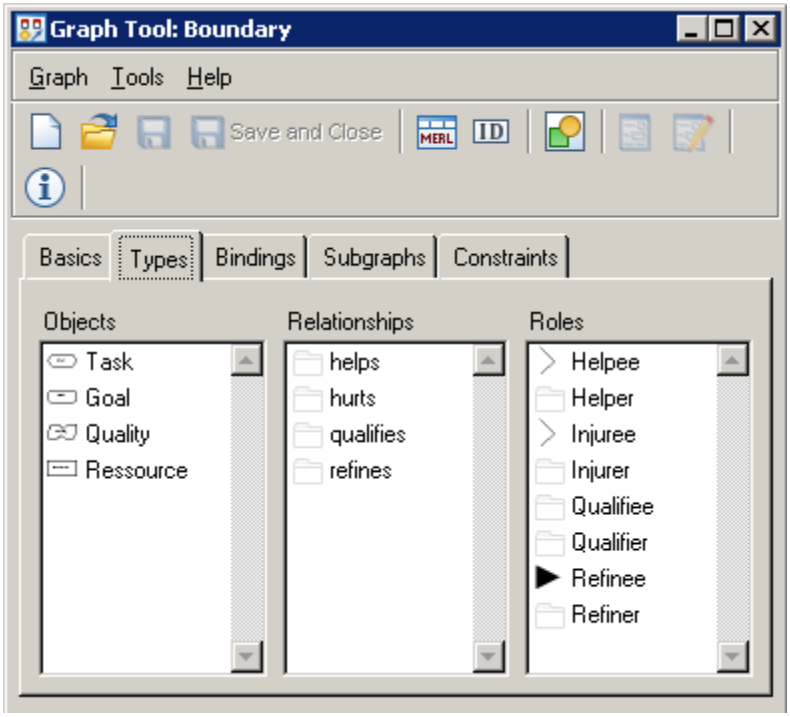


Abbildung 17: Types des Graphen Boundary

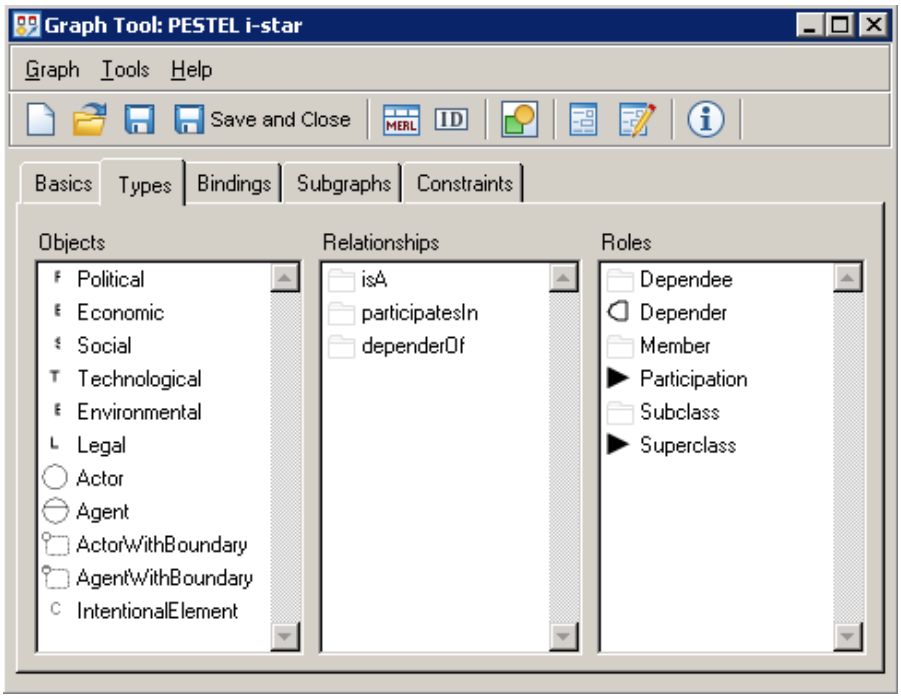


Abbildung 18: Types des Graphen PESTEL i-star

Im Tab *Bindings* wird festgelegt, wie die Objekte, Beziehungen und Rollen miteinander verbunden werden können. Dazu wird für jede Beziehung angegeben, welche Objekte als welche Rollen agieren können. In Tabelle 7 sind alle Beziehungen des Graphen PESTEL i-star aufgelistet und in Tabelle 8 jene des Graphen Boundary. Die Graphen *ActorBoundary* und *FactorBoundary* erben sämliche Types und Bindings des Vorfahren *Boundray.*

Die Beziehung dependerOf kann, wie in Tabelle 7 ersichtlich, in drei unterschiedlichen Konstellationen erstellt werden. Bei der ersten Möglichkeit kann die Beziehung von einem *Actor* zu einem *IntentionalElement* verlaufen.

Bei der zweiten Möglichkeit kann die Beziehung von einem *Goal-, Quality-, Taks-* oder *Resource*-Objekt, welches innerhalb des Subgraphen eines Actors oder Agents ist, zu einem *IntentionalElement* verlaufen. Die Depender-Rolle wird in diesem Fall als dynamischer Port definiert.

Die letzte Möglichkeit dieser Beziehung wäre ein Verlauf von einem IntentionalElement zu einem *Goal-, Quality-, Taks-* oder *Resource*-Objekt, welches innerhalb des Subgraphen eines PESTEL-Faktors ist. In diesem Fall wird die Dependee-Rolle über einen dynamischen Port definiert.

Somit können die Objekte im Subgraphen eines PESTEL-Faktors nur die Dependee-Rolle einnehmen. Das hat den Grund, dass für die Auswertung der Analyse nur die Abhängigkeiten von den Actors zu den verschiedenen PESTEL-Faktoren relevant sind und nicht umgekehrt.

Bei den Beziehungen *isA* und *participatesIn* hingegen kann ein Actor, sowie ein ActorWithBoundary beide Rollen annehmen. Zusätzlich ist es auch möglich, dass eine *participatesIn*-Beziehung von einem Actor zu einem PESTEL-Faktor verläuft.

Bindings des Graphen PESTEL i-star

Beziehung	Rolle	Dynamischer Port	Objekt
dependerOf	Depender		Actor
	Dependee		IntentionalElement
dependerOf	Depender	Goal Quality Resource Task	ActorWithBoundary AgentWithBoundary
	Dependee		IntentionalElement
dependerOf	Depender		IntentionalElement
	Dependee	Goal Quality Resource Task	PESTELFactor
isA	Subclass		Actor ActorWithBoundary
	Superclass		Actor ActorWithBoundary
participatesIn	Member		Actor Agent
	Participation		PESTELFactor
participatesIn	Member		Actor ActorWithBoundary Agent AgentWithBoundary
	Participation		Actor ActorWithBoundary Agent AgentWithBoundary

Tabelle 7: Bindings des Graphen PESTEL i-star

Wie die Objekte, Beziehungen und Rollen innerhalb des Graphen Boundary miteinander verbunden werden können, ist genau im Framework iStar 2.0 definiert (Dalpiaz et al. 2016).

So können die Rollen *Helper* und *Injurer* der Beziehungen *helps* und *hurts* von allen vier Objekten eingenommen werden, wobei die Rollen *Helpee* und *Injuree* nur vom Objekt *Quality* eingenommen werden können. Die Rolle *Qualifier* der Beziehung *qualifies* kann nur vom Objekt *Quality* ausgehen, aber zu allen anderen drei Objekten in der Rolle *Qualifiee* führen.

Bei der Beziehung refines können die Objekte *Goal* und *Task* sowohl die Rolle *Refiner,* als auch die Rolle *Refinee* einnehmen.

Bindings des Graphen Boundary

Beziehung	Rolle	Objekt
helps	Helper	Goal Quality Resource Task
	Helpee	Quality
hurts	Injurer	Goal Quality Resource Task
	Injuree	Quality
qualifies	Qualifier	Quality
	Qualifiee	Goal Resource Task
refines	Refiner	Goal Task
	Refinee	Goal Task

Tabelle 8: Bindings des Graphen Boundary

Im Tab Subgraphs kann in einer Liste definiert werden, für welche Objekte des Graphen Subgraphen erstellt werden können und von welchem Typ diese Subgraphen sind (MetaCase 2017b).

So können im Graphen PESTEL i-star für die Objekte *ActorWithBoundary*, *AgentWithBoundary* und für alle PESTEL-Faktor Objekte verschiedene Subgraphen erstellt werden. Für die Objekte *AgentWithBoundary* und *ActorWithBoundary* ist der Graphtyp *ActorBoundary*, für die PESTEL-Faktor Objekte der Graphtyp *FactorBoundary* definiert.

Außerdem sind im Graphen PESTEL i-star verschiedene weitere Einschränkungen festgelegt, welche das Vorkommen bestimmter Objekte reguliert. So ist definiert, dass jedes PESTEL-Element Objekt höchstens einmal vorkommen darf. Des Weiteren darf ein Objekt *IntentionalElement* nur jeweils einmal die Rolle *Depender,* sowie die Rolle *Dependee* einnehmen.

Für die Graphen *Boundary*, *FacorBoundary* und *ActorBoundary* gibt es keine Definition von Subgraphen oder sonstige Einschränkungen.

5.2 RDF-Export

In MetaEdit+ ist es möglich Generatoren zu schreiben, mit denen ein erstelltes Modell in einer gewünschten vordefinierten Form exportiert werden kann (MetaCase 2017b). Solche Generatoren sind mit der MetaEdit+ Reportling Language, kurz MERL, geschrieben, welche es ermöglicht, durch die Struktur des Modells zu navigieren und Informationen über die

einzelnen Elemente zu entnehmen. Diese Information kann anschließend in einem Dokument oder Fenster ausgegeben werden.

Der PESTEL Modeler stellt den Generator mit dem Namen *ExportRDF* zur Verfügung, welcher ein im Editor erstelltes Modell in eine RDF-Datei in Turtle Syntax konvertiert und zum weiteren externen Gebrauch zur Verfügung stellt. Dabei wird jedes modellierte Objekt als Resource in RDF dargestellt und als RDF-Typ wird der im Metamodell definierte Objekttyp verwendet. Außerdem wird jede modellierte Beziehung als Triple dargestellt, wobei die Beziehung das RDF-Prädikat darstellt.

Wie in Listing 2 aufgezeigt, wird der Generator mit dem Schlüsselwort *Report* definiert. Indem man vor dem Namen des Generators ein Rufzeichen einfügt, steht der Generator automatisch im Editor als Button zur Verfügung und die Ausführung des Generators kann durch Klick des Benutzers gestartet werden.

Listing 2: Definition Generator

```
01  Report '!ExportRDF'
02        .
03  endreport
```

Mit der Funktion `filename write`, wie in Listing 3 ersichtlich, wird das Dokument *RDFFile.ttl* erstellt und unter dem angegeben Pfad gespeichert. In Zeile 1 im Listing 3 ist der Speicherort als Pfad ersichtlich. Mit der Notation `<CurrentUserName>` ist der aktuell angemeldete User gemeint. Standardmäßig wird bei der Installation von MetaEdit+ im `Documents`-Ordner des Users ein Ordner `MetaEdit+ 5.5` angelegt. Als Unterordner wird zusätzlich ein Ordner `reports` angelegt. Innerhalb des Ordners `reports` wurde manuell ein weiterer Ordner `PESTEListar` angelegt, in welchen die RDF-Export-Dokumente gespeichert werden. Im Bereich zwischen dem Schlüsselwort *write* und *close* wird der Inhalt für das Dokument angegeben, welcher unter anderem durch das Iterieren über alle Objekte des Modells generiert wird.

Listing 3: Funktion filename write

```
01  filename 'C:\Users\<CurrentUserName>\Documents\MetaEdit+
    5.5\reports\PESTEListar\RDFFile.ttl' write
02        ...
03  close
```

Bei der Ausgabe in das RDF-Dokument werden als Erstes die Prefix-Statements abgearbeitet. Dazu wird der Subreport „_PrintPrefix“ erstellt. Der Unterstrich vor dem Report-Namen hat die Auswirkung, dass dieser Report weder im Editor noch in der Generatorliste aufscheint. Dieser Subreport gibt alle Prefix-Statments für das RDF-File aus. Außerdem gibt er den Default-Prefix, der bei der Erstellung eines Graphen angegeben werden kann, aus. Wird bei der Erstellung kein Default-Prefix festgelegt, so wird der Prefix `http://example.org/` verwendet.

Im nächsten Schritt (siehe Listing 4) wird mit einer `foreach`-Schleife über alle IntentionalElement-Objekte iteriert. Da diese Objekte abstrakte Objekte sind, wird bei der Erstellung auch kein Name beziehungsweise eine eindeutige UUID festgelegt. Daher muss

diese UUID für jedes dieser Objekte für den RDF-Export generiert werden. Dazu wird vorerst die Variable `conUUID` definiert.

Listing 4: Iteration über IntentionalElement

```
01  foreach .IntentionalElement {
02        variable 'conUUID' write
03           ''
04        close
05
06        subreport '_PrintRDFType' run
07        subreport '_PrintDependingRelationForConnectors' run
08
09        '.'
10        newline
11        newline
12  }
```

Der Subreport `_PrintRDFType` (Listing 5) wird nicht nur für die Objekte vom Typ IntentionalElement aufgerufen, sondern auch für Actors und Agents, über welche erst im nächsten Schritt iteriert wird. Dabei wird in Zeile 2 geprüft, ob es sich beim Typ des Objekts um ein *IntentionalElement* handelt. Anschließend wird die UUID für das *IntentionalElement* generiert, indem über die Funktion `external executeBlocking` in Zeile 3 ein externes Java-Programm ausgeführt wird. Dabei wird als Argument der Pfad zu einem Textdokument angegeben, in welches die generierte UUID gespeichert wird. Der Inhalt dieses Textdokuments wird in die Variable `conUUID` geschrieben. In der Zeile 9 wird nun der RDF-Typ der Objekte mit der generierten `conUUID` als URI ausgegeben. Handelt es sich um kein Objekt vom Typ IntentionalElement, so wird der Identifier des Objekts ausgegeben, indem wie in Zeile 11 das Schlüsselwort `id` angegeben wird. Dabei wird der Standard Translator `%var` verwendet, welcher Teil der MetaEdit+ Generator Development Environment ist (MetaCase 2017b). Dieser Translator übersetzt nicht-alphanumerische Zeichen in Unterstriche. Als RDF-Typ der Objekte wird der Prefix *istar* mit dem Schlüsselwort type kombiniert. Dieses Schlüsselwort gibt den Objekt-Typ des aktuellen Elements aus.

Listing 5: Subreport PrintRDFType

```
01  Report '_PrintRDFType'
02     if type = 'IntentionalElement' then
03        external 'java -jar C:\Users\<CurrentUserName>\UUIDGenerator.jar
    C:\Users\<CurrentUserName>\UUID.txt' executeBlocking
04
05        variable 'conUUID' append
06           filename 'C:\Users\<CurrentUserName>\UUID.txt' read
07        close
08
09         ':'$conUUID ' rdf:type ' 'istar:IntentionalElement'
10     else
11        ':'id%var ' rdf:type ' 'istar:' type
12     endif
13  endreport
```

In Zeile 7 in Listing 4 wird der Subreport `_PrintDependingRelationForConnectors` ausgeführt. Der Code zu diesem Subreport ist in Listing 6 ersichtlich. Mit dem Ausdruck in Zeile 2 und Zeile 6 wird überprüft, ob das IntentionalElement eine Beziehung *dependerOf* zu einem anderen Objekt aufweist. Außerdem muss das IntentionalElement-Objekt die Rolle *Depender* einnehmen, damit der Ausdruck wahr ist. In Zeile 2 ist am Ende des Ausdrucks ein Punkt angeführt, das bedeutet, es wird nach Objekten in der Rolle *Dependee* gesucht. In der Zeile 6 hingegen ist eine Raute am Ende des Ausdrucks. Das hat die Folge, dass hier nicht nach Objekten, sondern Ports gesucht wird. Das ist deshalb nötig, da bei dependerOf-Beziehungen von oder zu einem Objekt, welches zum Subgraphen eines Actors oder PESTEL-Faktors gehört, nicht das Objekt im Subgraphen angesprochen wird, sondern der Actor oder PESTEL-Faktor selbst, da das Objekt im Subgraphen nur einen Port darstellt. Unter der Verwendung einer `dowhile`-Schleife wird über alle vorhandenen Beziehungen des *IntentionalElements* iteriert und die Beziehung ins RDF-Dokument geschrieben. Zusätzlich wird in einer Variablen `factor` die inverse Beziehung *dependeeOf* dokumentiert, welche später bei den betroffenen Objekten abgefragt und ausgegeben wird.

Ab Zeile 29 wird wiederum die Beziehung *dependerOf* abgefragt. Hier werden jedoch die Rollen der Beziehung *dependerOf* vertauscht. Es wird also nach Beziehungen gesucht, in denen das *IntentionalElement* in der Rolle *Dependee* agiert. In Zeile 57 wird zusätzlich in der Variable dependingForActors mitdokumentiert, welche Actors oder Agents vom aktuellen IntentionalElement abhängig sind. Auf diese Variable wird bei der Iteration über die Actors und Agents zugegriffen und die Triple werden ausgegeben, damit später über alle Beziehungen nicht erneut iteriert werden muss.

Listing 6: Subreport PrintDependingRelationForConnectors

```
01  Report '_PrintDependingRelationForConnectors'
02     if ~Depender>dependerOf~Dependee.() <> '' then
03        ';'
04        newline
05
06        if ~Depender>dependerOf~Dependee#() <> '' then
07           $obj = $conUUID
08
09           dowhile ~Depender>dependerOf~Dependee#() {
10              '  istar:dependerOf '
11              ':'id%var
12
13              variable 'factor' append
14                 id%var
15                 newline
16                 'istar:dependeeOf :' $obj
17                 newline
18              close
19           }
20        else
21           dowhile ~Depender>dependerOf~Dependee.() {
22              '  istar:dependerOf '
23              ':'id%var ';'
24              newline
25           }
26        endif
27     endif
28
```

```
29    if ~Dependee>dependerOf~Depender.() <> '' then
30       ';'
31       newline
32       $obj = $conUUID
33
34       if ~Dependee>dependerOf~Depender#() <> '' then
35          dowhile ~Dependee>dependerOf~Depender#() {
36             '  istar:dependeeOf '
37             ':'id%var
38
39             variable 'factor' append
40                id%var
41                newline
42                'istar:dependerOf :' $obj
43                newline
44             close
45          }
46       else
47          dowhile ~Dependee>dependerOf~Depender.() {
48             '  istar:dependeeOf '
49             ':'id%var
50
51             variable 'dependingForActors' append
52                id%var
53                newline
54                'istar:dependerOf :' $obj
55                newline
56             close
57          }
58       endif
59    endif
60 endreport
```

Nachdem alle IntentionalElement-Objekte abgearbeitet sind, wird über alle Objekte vom Typ Actor und Agent iteriert (Listing 7). Dazu wird in Zeile 2 wieder der bereits beschriebene Subreport `_PrintRDFType` aufgerufen. Anschließend wird in Zeile 3 der Subreport `_PrintDependingRelation` ausgeführt, welcher über die Einträge der Variable `dependingForActors` iteriert und zugehörige Tripel ausgibt. Mit den definierten Subreports `_PrintIsARelation` und `_PrintParticipatesInRelation` in Zeile 4 und Zeile 5, werden alle Beziehungen vom Typ *isA* und *participatesIn*, bei denen der jeweilige Actor oder Agent die Rolle *Subclass* oder *Member* einnimmt, ausgegeben.

Listing 7: Iteration Actor/Agent

```
01 foreach .(Actor|Agent) {
02    subreport '_PrintRDFType' run
03    subreport '_PrintDependingRelation' run
04    subreport '_PrintIsARelation' run
05    subreport '_PrintParticipatesInRelation' run
06    '.'
07    newline
08    newline
09 }
```

Abschließend wird über jene Objekte iteriert, für welche ein Subgraph erstellt werden kann, also über alle PESTEL-Faktor-Objekte, sowie *ActorWithBoudary*- und

AgentWithBoundary-Objekte. Für diese Objekte gibt es einen eigenen Subreport zur Ausgabe der RDF-Typen, wie in Zeile 2 in Listing 8 ersichtlich. Grund dafür ist, dass bei diesen Objekten die Typdefinition des Metamodells von der Typ-Definition im RDF abweicht. Den PESTEL-Faktoren wird im RDF der Typ PESTELFaktor mit Prefix *pestel* zugewiesen. Der URI der PESTEL-Faktoren wird ebenso dem Prefix *pestel* zugewiesen. Bei den Actor- und Agent-Objekten wird bei der Typdefinition im Metamodell unterschieden, ob diese einen Subgraphen aufweisen oder nicht. Dies ist nötig, da beim Metamodell die Actors und Agents mit Subgraphen eine andere graphische Darstellung aufweisen, also jene Actors und Agents ohne Subgraphen. Im RDF hingegen ist es nicht nötig und auch nicht vorgesehen, hiervon eine Unterscheidung zu definieren. Deshalb wird den Objekten *ActorWithBoudary* und *AgentWithBoundary* der RDF-Typ *Actor* oder *Agent* mit Prefix *istar* zugewiesen. Die Subreports `_PrintIsARelation` und `_PrintParticipatesInRelation` werden wie bei den anderen Objekten ausgeführt.

Listing 8: Iteration PESTEL-Faktor und Actor/Agent mit Subgraphen

```
01  foreach .(Political|Economic|Social|Technological|Environmental|
    Legal|ActorWithBoundary|AgentWithBoundary) {
02       subreport '_PrintRDFTypesForBoundaryObjects' run
03       subreport '_PrintIsARelation' run
04       subreport '_PrintParticipatesInRelation' run
05       subreport '_DoDecompositions' run
06       '.'
07       newline
08       newline
09  }
10
11  variable 'subgraphs' read
12  close
```

In Zeile 5 in Listing 8 wird der Subreport `_DoDecompositions` ausgeführt. Dabei wird, wie in Zeile 2 in Listing 9 ersichtlich, mit Hilfe der Funktion `do decompositions` über alle Objekte des Subgraphen iteriert. Die Zugehörigkeit eines Objekts im Subgraphen zum Hauptobjekt wird im RDF-Dokument durch das RDF-Prädikat *wants* und dem Prefix *istar* ausgedrückt und ausgegeben.

Die RDF-Typdefinition der Objekte im Subgraphen, sowie alle bestehenden Beziehungen vom Typ helps, hurts, qualifies, refines, dependerOf und dependeeOf werden in die Variable `subgraphs` geschrieben (Zeile 10 in Listing 9). Am Ende des Generators (Zeile 11 und 12 in Listing 8) wird die Variable `subgraphs` ausgelesen und in das RDF-Dokument übertragen. Die Beziehungen *dependerOf* und *dependeeOf* wurden für die einzelnen Objekte im Subgraphen bereits im Subreport `_PrintDependingRelationForConnectors` in die Variable `factor` gespeichert. Über diese Variable wird nun iteriert und die jeweiligen Tripel werden in das RDF-Dokument übertragen. Diese Vorgehensweise ist notwendig, da es nicht möglich ist, ausgehend von einem Port über eine Beziehung zu einem Objekt zu navigieren, umgekehrt jedoch schon (MetaCase 2017b). Deshalb wird diese Information bereits bei der Iteration über die IntentionalElement-Objekte (Zeile 13 und 39 in Listing 6) in die Variable `factor` gespeichert.

Listing 9: Subreport DoDecompositions

```
01  Report '_DoDecompositions'
02    do decompositions {
03
04       foreach .(Task|Goal|Quality|Resource) {
05         ';'
06         newline
07         '  istar:wants '
08         ':'id%var
09
10         variable 'subgraphs' append
11           $sub = id%var
12           ':'id%var ' rdf:type ' 'istar:' type
13
14           if ~Helper>Helps~Helpee.() <> '' then
                  .
22           endif
23
24           if ~Injurer>Hurts~Injuree.() <> '' then
                  .
32           endif
33
34           if ~Refiner>Refines~Refinee.() <> '' then
                  .
42           endif
43
44           if ~Qualifier>Qualifies~Qualifiee.() <> '' then
                  .
52           endif
53
54           $next = 'f'
55           do $factor {
56              $line = id
57              if $next = 't' then
58                 ';'
59                 newline
60                 '   '$line
61                 $next = 'f'
62              endif
63
64              if $line = $sub then
65                 $next = 't'
66              endif
67           }
68           '.'
69           newline
70           newline
71         close
72       }
73    }
74  endreport
```

5.3 RDF-Import

Damit der Import eines RDF-Dokuments als Modell in MetaEdit+ funktioniert, ist es nötig, das RDF-Dokument in ein XML-Dokument mit der in Kapitel 5.3.1 beschriebenen Struktur zu transformieren (MetaCase 2017b). Dazu wird das RDF-Dokument von einem Java-Programm eingelesen und in ein XML-Dokument kompiliert. Um das Java-Programm auszuführen, wird, wie beim RDF-Export, auch ein MERL-Generator benötigt.

Der RDF-Import hätte, wie der RDF-Export, auch zur Gänze mit MERL implementiert werden können. Für den Import ist jedoch bei der Verwendung von Subgraphen im Metamodell die Generierung einer sehr komplexen und verschachtelten XML-Struktur Voraussetzung. MERL stellt eine reine Generatorsprache dar und ist deshalb für diese Aufgabe nicht geeignet, da die Auswahl der zur Verfügung gestellten Funktionen oder Bibliotheken begrenzt ist. Im Gegensatz dazu besteht bei der Verwendung der Programmiersprache Java eine große Auswahl an Bibliotheken, die hilfreiche Funktionen zur Verfügung stellt. Wie im Kapitel 5.3.3.2 beschrieben, wird bei der Implementierung des Java-Programms die Bibliothek Apache Jena[1] verwendet, was die Übertragung in die komplexen XML-Strukturen erheblich erleichtert. Zudem wird durch die Verwendung der Apache Jena Bibliothek die Abhängigkeit vom Format, in der das RDF definiert ist, umgangen. Würde man den RDF-Import mit MERL implementieren, so würde man für jedes gewünschte RDF-Format eine eigene Implementierung benötigen.

In den folgenden Kapiteln werden die XML-Struktur, der MERL-Generator und das Java-Programm beschrieben.

5.3.1 XML-Struktur

MetaEdit+ ermöglich es, ein externes Modell zu importieren, wenn es in einem bestimmten XML Format definiert ist (MetaCase 2017b). Das XML Format von MetaEdit+ ist nach den Strukturen der GOPPRR Metamodelling Sprache aufgebaut. Jedes XML Import Dokument beginnt mit einem `<gxl>`-Tag gefolgt von einem `<graph>`-Tag, wie in Zeile 1 und Zeile 2 in Listing 10 ersichtlich. Als Attribut des `<graph>`-Tags wird der Typ des Graphen angegeben. In Zeile 3 bis 7 wird demonstriert, wie Eigenschaften von GOPPRR Elementen im XML-Dokument dargestellt werden. Jede Eigenschaft wird mit einem `<slot>`-Tag eröffnet. Das Attribute `name` definiert den lokalen Namen der Eigenschaft. Innerhalb des `<slot>`-Tags wird mit zwei weiteren Tags `<value>` und `<string>` der Wert, sowie der Datentyp der Eigenschaft definiert. Diese Struktur der Eigenschaften wird auch bei Eigenschaften von Objekten, Beziehungen oder Rollen verwendet, wie zum Beispiel in Zeile 10 bis 14 ersichtlich.

Nachdem die Eigenschaften des Graphen definiert wurden, werden zuerst die Objekte und anschließend die Bindings im XML dargestellt (MetaCase 2017b). Wie in Zeile 9 zu sehen ist, werden bei den Objekten als Attribute die ID, sowie der Typ des Objekts angegeben.

Wie in Zeile 20 werden Bindings mit dem Tag `<binding>` begonnen (MetaCase 2017b). Innerhalb dieses Tags wird mit dem Tag `<relationship>` der Name der Beziehung definiert (Zeile 21). Jeder `<binding>`-Tag benötigt mindestens zwei `<connection>`-Tags. Innerhalb der Connection werden die zugehörigen Rollen, Ports und Objekte definiert (Zeile 23 bis 25), wobei Ports nur angegeben werden, wenn diese definiert sind. Bei Rollen wird

[1] https://jena.apache.org/

als Attribut `type` der Name der Rolle angegeben. Bei Ports und Objekten wird als Attribut `href` ein Raute-Symbol gefolgt von der ID der Objekte angegeben.

Wie in Zeile 15 stellen die Subgraphen eigene Graphen dar und werden als Teil des zugehörigen Objekts definiert (MetaCase 2017b). Die Objekte und Bindings des Subgraphen werden ebenso als Teil des Subgraphen dargestellt und innerhalb der Tags des Subgraphen definiert.

Listing 10: Struktur XML-Format (MetaCase 2017b)

```
01  <gxl>
02    <graph type="PESTEL i-star">
03      <slot name="Company">
04        <value>
05          <string>Ryan Air</string>
06        </value>
07      </slot>
08        .
09      <object id="Airline" type="Actor">
10        <slot name="Name">
11          <value>
12            <string>Airline</string>
13          </value>
14        </slot>
15        <graph id="AirlineBoundary" type="ActorBoundary">
16             .
17        </graph>
18      </object>
19        .
20      <binding>
21        <relationship>dependerOf</relationship>
22        <connection>
23          <role> type="Depender"</role>
24          <port> href="#Low Fare Airline"</port>
25          <object> href="#Negotiate Low Fees"</object>
26        </connection>
27        <connection>
28             .
29        </connection>
30      </binding>
31        .
32    </graph>
33  </gxl>
```

5.3.2 MERL-Generator

Wie auch beim RDF-Export, wird hier ein Generator benötigt, mit dem die Funktion in MetaEdit+ gestartet werden kann.

In Listing 11 ist der Code des Generators ersichtlich. In Zeile 1 wird der Name des Generators definiert. Dieser Generator benötigt im Gegensatz zum RDF-Export-Generator kein Rufzeichen als Prefix, da dieser nicht im Editor erscheinen soll. Dieser Generator kann nur über die Generatorliste aufgerufen werden.

In Zeile 3 bis 5 wird mit der Funktion `promt askFilename` vom User das gewünschte RDF-File mittels eines File-Dialogs abgefragt und in die Variable `rdfFileName` gespeichert.

Mit der Funktion `external executeBlocking` wird das Java-Programm ausgeführt (Zeile 7). Dazu wird zwischen den Schlüsselwörtern `external` und `executeBlocking` der Kommandozeilenbefehl zur Ausführung angegeben. Damit das Java-Programm ausgeführt werden kann, muss auch der Pfad zu den verwendeten Bibliotheken im Kommandozeilenbefehl angeführt werden. Dieser muss dem Format in Zeile 7 entsprechen. Es wird also nach dem Befehl `java –cp` der Name des JAR-Dokuments inklusive dem Pfad angegeben. Darauf folgen ein Semikolon, sowie der Pfad zu den im Java-Programm verwendeten Bibliotheken. Am Ende des Befehls müssen das Package und der Name der Main-Klasse angeführt werden. Außerdem erfordert die Ausführung des Programms ein Kommandozeilenargument mit dem Dokumentnamen und dem Pfad des zu transformierenden RDF-Dokuments.

Weitere Funktionen und Einzelheiten zu dem Java-Programm sind im Kapitel 5.3.3 beschrieben.

Nachdem das benötigte XML-Dokument erstellt und unter `C:\temp\output.mxm` gespeichert wurde, wird dieses mit der Funktion `internal execute,` sowie dem Kommandozeilenparameter `fileInPatch` in MetaEdit+ importiert (Zeile 9) (MetaCase 2017a).

Listing 11: MERL-Generator für RDF-Import

```
01  Report 'ImportRDF'
02
03     variable 'rdfFileName' write
04        prompt 'Select RDF-File for import!' askFilename
05     close
06
07     external 'java -cp "C:/temp/pestel-1.0.0-SNAPSHOT/bin/pestel-1.0.0-
    SNAPSHOT.jar;C:/temp/pestel-1.0.0-SNAPSHOT/lib/*" importMain.RDFImport '
    $rdfFileName executeBlocking
08
09     internal 'fileInPatch: "' 'C:\temp\output.mxm"' execute
10
11  endreport
```

5.3.3 Java-Programm

In den folgenden Kapiteln werden der Aufbau, der Ablauf und die Methoden des Java-Programms beschrieben. Zusätzlich wird im Kapitel 5.3.3.3 beschrieben, wie das Apache Maven Projekt konfiguriert ist.

5.3.3.1 Aufbau

Das Klassendiagramm des Java-Programms ist in Abbildung 19 ersichtlich und besteht aus drei Packages.

Das Package `at.jku.dke.importMain` beinhaltet die Klassen `RDFImport` und `Helper`, wobei sich die Main-Methode des Java-Programms in der Klasse `RDFImport` befindet.

Das Package `at.jku.dke.Visitor` beseht aus dem Interface `PESTEListarVisitor`, sowie den Klassen `PESTEListarElementVisitor` und `PESTEListarRelationVisitor`. Beide dieser Klassen implementieren das Interface `PESTEListarVisitor`.

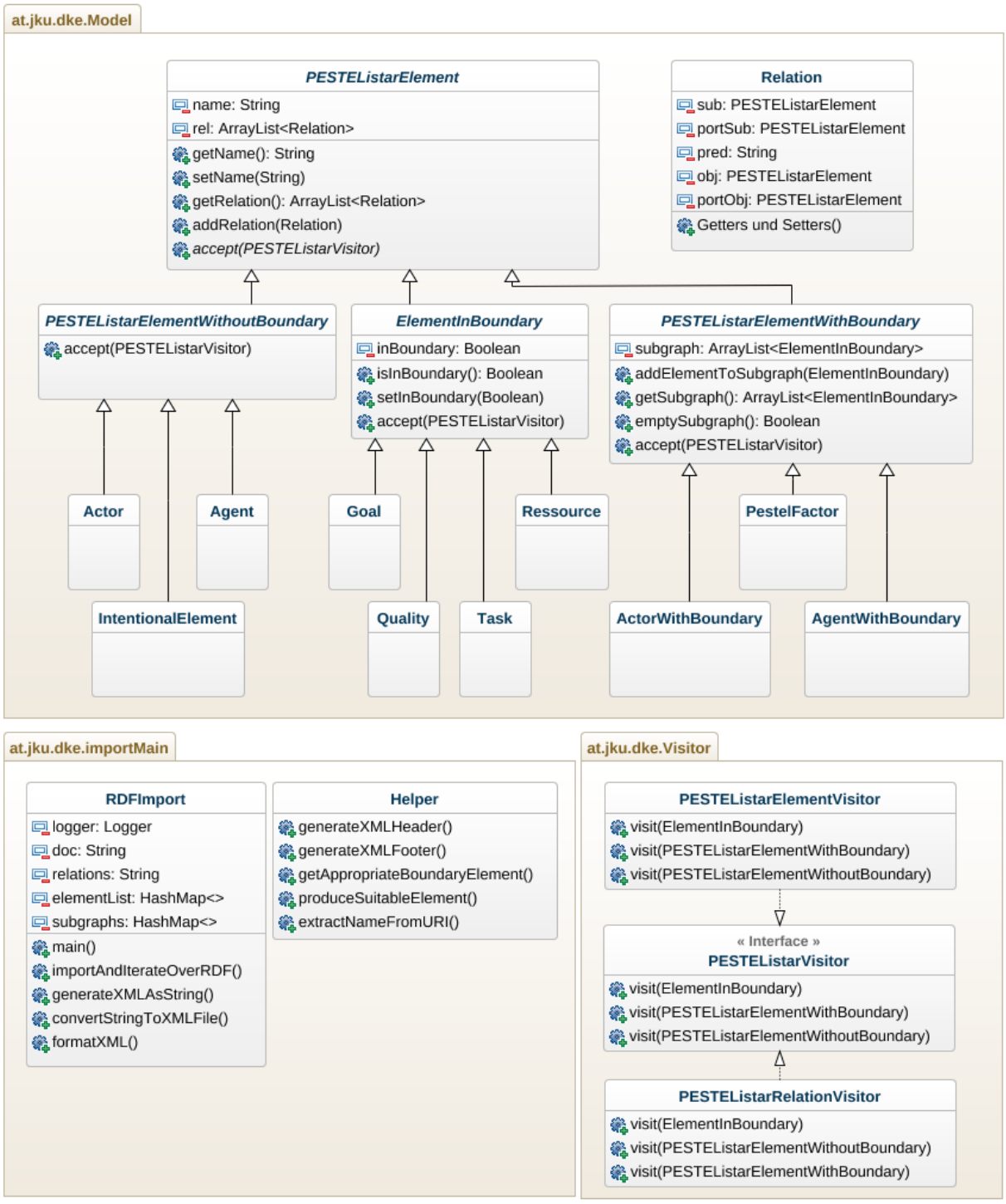


Abbildung 19: Klassendiagramm RDF-Import

Das Package at.jku.dke.Model beinhaltet unter anderem alle Objekte des Metamodells als eigene Klasse. In dieses Package gibt es die abstrakte Klasse PESTEListarElement, welche die Superklasse der abstrakten Klassen PESTEListarElementWithBoundary, PESTEListarElementWithoutBoundary und ElementInBoundary darstellt. Die Klasse PESTEListarElementWithoutBoundary stellt die Superklasse aller Klassen dar, bei denen es einen Subgraphen gibt, also für PestelFactor, ActorWithBoundary und AgentWithBoundary. Die Klasse PESTEListarElementWithoutBoundary ist die Superklasse der Klassen Actor, Agent und IntentionalElement, also jener Objekte, bei denen es keinen Subgraphen gibt und die nicht innerhalb eines Subgraphen modelliert werden können. Die Klassen Goal, Quality, Task und Resource erben von der abstrakten Klasse ElementInBoudary, da diese Elemente nur innerhalb eines Subgraphen modelliert werden können.

Bei der Kompilierung der Elemente in die bereits oben beschriebene notwendige XML-Syntax eignet sich die Anwendung des Visitor-Patterns besonders, da das Pattern typisch für den Compilerbau ist (Palsberg und Jay 1998). Unterschiedliche Objekte fordern unterschiedliche Ausgaben. Das Visitor-Pattern ermöglicht es, unterschiedliche Operationen auf Objekte durchzuführen, welche abhängig von der konkreten Klasse des Objekts sind (Springer et al. 2016). Bei der Kompilierung der Elemente gibt es folgende zwei Operationen: die Kompilierung der Elemente, sowie die Kompilierung der Beziehungen. Die Kategorisierung der PESTEL Elemente in die drei abstrakten Klassen Superklassen ist notwendig, da bei den Elementen innerhalb der drei einzelnen Klassen die Vorgangsweise bei der Kompilierung ins XML-Dokument identisch ist. So implementiert jede der abstrakten Klassen die accept-Methode des Visitor-Patterns.

Des Weiteren gibt es im Package at.jku.dke.Model die Klasse Relation, welche zur Darstellung einer Beziehung zwischen zwei Objekten benötigt wird. Für diese Klasse sind die fünf Variablen sub, portSub, pred, obj und portObj definiert. Die Variablen portSub und portObj werden nur bei der Beziehung *dependerOf* benötigt, da die Objekte innerhalb eines Subgraphen nur über den dynamischen Port erreicht werden und auch so in der Beziehung definiert werden müssen. Bei allen anderen Beziehungen haben diese Variablen den Wert null.

Jedes der PESTEL-Objekte verfügt über eine Variable name, sowie eine ArrayList rel, in welcher die Beziehungen zu anderen Elementen gespeichert werden. Objekte der Klasse ElementInBoundary verfügen zusätzlich über eine boolesche Variable inBoundary. Objekte der Klasse PESTEListarElementWithBoundary verfügen zusätzlich über eine ArrayList, in der alle Objekte des Subgraphen gespeichert sind.

Die Klasse RDFImport enthält die Variablen doc, relations, elementList und subgraphs. Die Variable doc speichert das kompilierte XML-Dokument als String. Die Variable relations speichert alle definierten Beziehungen als XML String. Die Variablen elementList und subgraphs sind vom Typ HashMap. Bei der Variable elementList ist der Key vom Typ String und der Value vom Typ PESTEListarElement. Nach dem Einlesen des RDF-Dokuments werden von allen PESTEL-Elementen Objekte erzeugt und als Value in diese HashMap gespeichert. Der Key stellt den Namen des Elements dar. Bei der Variable subgraphs sind sowohl Key, als auch Value vom Typ PESTEListarElement. Der Key-Wert stellt ein Objekt dar, welches innerhalb eines Subgraphen liegt. Als Value wird das Objekt mit dem Subgraphen gespeichert.

5.3.3.2 Ablauf und Methoden

Wie bereits im Kapitel 5.3.3.1 erwähnt, befindet sich die Main-Methode in der Klasse `ImportRDF`. Die Implementierung der Main-Methode ist in Listing 12 ersichtlich. Als Argument werden beim Ausführen des Java-Programms der Pfad, sowie der Dateiname des RDF-Dokuments angegeben.

Listing 12: Main-Methode RDF-Import

```
01  public static void main(String[] args) {
02       try{
03          final String filepath = args[0];
04
05          logger.info("Import and Iterate over RDF");
06          importAndIterateOverRDF(filepath);
07
08          logger.info("Build XML as String");
09          generateXMLAsString();
10
11          logger.info("Format XML String");
12          doc = formatXML(doc);
13
14          logger.info("Convert String to XML File");
15          convertStringToXMLFile();
16
17          System.exit(0);
18       }catch(IndexOutOfBoundsException e){
19          logger.info("No arguments defined!");
20       }
21    }
```

Mit der Methode `importAndIterateOverRDF` (Zeile 6 in Listing 12) wird das angegebene RDF-Dokument eingelesen und die Triple werden als Java-Objekte gespeichert. Zum Einlesen der RDF-Triple wird die Java-Bibliothek Apache Jena[2] verwendet. Diese stellt die sogenannte RIOT API zur Verfügung, welche es erlaubt, über Parser Output zu iterieren (Apache Software Foundation 2017a). Dazu werden, wie in Zeile 1 und 2 in Listing 13 ersichtlich, die Klassen `PipedRDFIterator` und `PipedRDFStream` benötigt. Der `PipedRDFIterator` und der `PipedRDFStream` müssen auf zwei verschiedenen Threads laufen, da sonst ein Deadlock entstehen könnte. Deshalb wird in Zeile 4 ein neuer Thread für den Parser erstellt, welcher in Zeile 13 gestartet wird. Am Haupt-Thread kann nun über die einzelnen RDF-Triple iteriert werden (Zeile 15 bis 18).

Listing 13: Iteration über Parser Output (Apache Software Foundation 2013)

```
01  PipedRDFIterator<Triple> iter = new PipedRDFIterator<>();
02  final PipedRDFStream<Triple> inputStream = new PipedTriplesStream(iter);
03
04  ExecutorService executor = Executors.newSingleThreadExecutor();
05
06  Runnable parser = new Runnable() {
07     @Override
```

[2] https://jena.apache.org/

```
08     public void run() {
09       RDFParser.source(filepath).parse(inputStream);
10     }
11  };
12
13  executor.submit(parser);
14
15  while (iter.hasNext()) {
16     Triple next = iter.next();
17        .
18  }
```

Es werden auf Basis des RDF-Prädikats `rdf:type` für alle PESEL Elemente Java-Objekte erzeugt und in die ArrayList `elementList` gespeichert. Dabei stellt das RDF-Subjekt den Namen des PESTEL Elements dar und das RDF-Objekt die Java-Klasse. Mit der Methode `produceSuitableElement` der Klasse `Helper` wird mittels einer `switch-case`-Anweisung das Objekt der richtigen Java-Klasse erzeugt.

Da im RDF bei der Typdefinition nicht konkret spezifiziert ist, ob es sich um ein Element mit oder ohne Subgraphen handelt, wird diese Information vom RDF-Prädikat `istar:wants` abgeleitet. So werden die RDF-Subjekte dieser Triple als Objekte der Subklassen der Klasse `PESTEListarElementWithBoundary` gespeichert. Dazu wird mit Hilfe der Methode `getAppropriateBoundaryElement` der Klasse Helper wiederum mittels einer `switch-case`-Anweisung die richtige Java-Klasse erzeugt. Die RDF-Objekte dieser Triple stellen die Objekte im Subgraphen dar und werden deshalb zur ArrayList `subgraph` des `PESTEListarElementWithBoundary`-Objekts hinzugefügt. Des Weiteren werden alle Objekte, die innerhalb eines Subgraphen sind, in die HashMap `subgraphs` der Klasse ImportRDF gespeichert.

Bei RDF-Tripel, die weder als RDF-Prädikat `istar:wants`, noch `rdf:type` enthalten, handelt es sich um die Darstellung der im Metamodell definierten Beziehungen. Bei diesen wird zum als Subjekt definierten Java Objekt ein Objekt der Klasse Relation angefügt. Dieses Objekt der Klasse Relation enthält in Variablen sämtliche Informationen der Beziehung, also Subjekt, Prädikat und Objekt. Wenn es sich um eine Beziehung *dependerOf* handelt, werden ebenso die Variablen für den Port des Objekts oder Subjekts definiert.

Wie im Kapitel 2.2 erwähnt, gibt es laut der Definition von iStar 2.0 bei der Beziehung dependerOf nicht nur einen Depender, sowie einen Dependee, sondern auch ein sogenanntes Dependum (Dalpiaz et al. 2016). Das Dependum kann vom Typ *Goal*, *Quality*, *Task* oder *Resource* sein und stellt das Objekt der Abhängigkeit dar. Dieses Dependum wird jedoch beim PESTEL Modeler als abstraktes Verbindungsobjekt dargestellt. Da es aber trotzdem möglich sein soll, ein RDF-Dokument zu importieren, bei dem das Dependum nicht abstrakt gehalten wird, muss dies beim RDF-Import berücksichtigt werden. Dazu wird mittels der Methode `isInBoundary` der Klasse `ElementInBoundary` überprüft, ob Objekte der Klasse Goal, *Quality*, *Task* existieren, welche nicht in einem Subgraphen liegen. Wenn dies der Fall ist, so werden diese Objekte in Objekte der Klasse `IntentionalElement` transformiert.

Nachdem nun alle `PESTEListarElement`-Objekte inklusive der Beziehungen und Subgraphen in der Variable `elementList` gespeichert sind, wird in der Main-Methode in Zeile 9 in Listing 12 die Methode `generateXMLAsString` aufgerufen, in der mit Hilfe der Methoden `generateXMLHeader` und `generateXMLFooter` der XML-Kopf, sowie XML-Fuß

erzeugt (Zeile 3 und 12 in Listing 14) und der Variable doc angehängt werden. Zwischen den beiden Methoden wird ein Objekt der Klasse PESTEListarElementVisitor erzeugt (Zeile 5). Wie in Zeile 6 bis 10 ersichtlich, wird anschließend über alle PESTEListarElement-Objekte, außer jener der Klasse ElementInBoundary, iteriert. Dabei werden die Objekte vom definierten PESTEListarElementVisitor besucht. Je nach Objektklasse wird die accept-Methode einer der zwei abstrakten Klassen PESTEListarElementWithBoundary oder PESTEListarElementWithoutBoundary aufgerufen. Die accept-Methode der Klasse ElementInBoundary wird erst besucht, wenn über den Subgraphen der PESTEListarElementWithBoundary-Objekte iteriert wird. Der PESTEListarElementVisitor hat die Aufgabe, je nach Objektklasse die XML-Tags inklusive Attribute und Eigenschaften zu erzeugen und an die statische Variable doc anzuhängen. Dabei werden die Methoden getClass und getSimpleName verwendet, um, wie in Listing 10, den Typ des Objektes im XML-Dokument anzugeben.

Innerhalb der visit-Methode des PESTEListarElementVisitors wird ein PESTEListarRelationVisitor erzeugt, welcher anschließend für jedes Objekt über die in der Variable rel gespeicherten Beziehungen iteriert und diese in der Form, wie in Zeile 22 bis 26 in Listing 10 dargestellt, in die statische Variable relations speichert.
In der accept-Methode für PESTEListarElementWithBoundary-Objekten wird, wie bereits erwähnt, mit Hilfe eines PESTEListarElementVisitors über die Objekte, welche in der Variable subgraph gespeichert sind, iteriert und wiederum der statischen Variable doc angehängt.

Zur Generierung der XML-Konstrukte stehen der Klasse PESTEListarElementVisitor die statischen Hilfsmethoden generateXMLObject, generateSubgraphOpenTag, generateSubgraphCloseTag und doSubgraph zur Verfügung. Auch die Klasse PESTEListarRelationVisitor verfügt über statische Hilfsmethoden wie generateXMLRelation und getRole.

Nachdem die Objekte und Beziehungen getrennt in den beiden String-Variablen doc und relations generiert wurden, wird die Variable relations an die Variable doc angefügt (Zeile 11 in Listing 14).

Listing 14: Methode generateXMLAsString

```
01  private static void generateXMLAsString() {
02
03     Helper.generateXMLHeader();
04
05     PESTEListarVisitor visitor = new PESTEListarElementVisitor();
06     for(PESTEListarElement e: elementList.values()){
07        if(!(e instanceof ElementInBoundary)){
08           e.accept(visitor);
09        }
10     }
11     doc = doc + relations;
12
13     Helper.generateXMLFooter();
14  }
```

Zur Verbesserung der Lesbarkeit des XML-Dokuments wird der generierte XML-String mit der statischen Methode formatXML formatiert (Zeile 12 in Listing 12). Ziel dabei ist es,

die XML-Elemente optimal einzurücken, um die Objekte, Beziehungen und Subgraphen besser voneinander abgrenzen zu können.

Abschließend wird mit der statischen Methode `convertStringToXMLFile` der XML-String in ein XML-Dokument konvertiert und im Ordner `C:/temp/output.mxm` gespeichert. Zur Codierung wird ein Objekt der Klasse `Charset` erzeugt, für welches das Format `UTF-8` ausgewählt wird.

5.3.3.3 Apache Maven

Das Java-Programm wurde als Apache Maven[3] Projekt erstellt, da durch die Verwendung von Apache Maven Java-Programme einfacher erstellt und die Abhängigkeiten zu anderen Bibliotheken besser verwaltet werden können (Miller et al. 2010).

Die Datei pom.xml des Maven-Projekts ist in Listing 15 ersichtlich. In Zeile 16 bis 35 sind alle Abhängigkeiten zu externen Java-Bibliotheken angegeben. Dazu gehören Apache Jena Bibliothek[4], Simple Logging Facade[5] for Java und Commons CLI Bibliothek[6].

In Zeile 34 bis 68 werden die Plug-Ins angegeben, welche für den Build-Prozess benötigt werden. Dazu zählt das Maven-Compiler-Plugin[7] (Zeile 58 bis 67), welches für die Kompilierung der Quellen zuständig ist (Apache Maven 2017b). Das zweite Plugin ist das Maven-Assembly-Plugin[8] (Zeile 36 bis 57). Dieses Plugin ermöglicht es, dass das Projekt inklusiver aller Abhängigkeiten zu anderen Bibliotheken aggregiert wird und in einem einzigen Archiv ausgegeben wird (Apache Maven 2017a). Dazu wird als Descriptor der Pfad zu einem externen XML-Dokument in Zeile 40 angegeben, welches definiert, in welchem Format und mit welcher Ordnerstruktur das Projekt als Archiv ausgegeben werden soll. Dabei werden sämtliche Abhängigkeiten als JAR-Dateien in den Ordner `lib` ausgegeben. Die JAR-Datei des erstellen Java-Programms wird inklusive aller Klassen in den Ordner `bin` gespeichert.

Beim Ausführen des Maven-Projekts werden als Lifecycle-Phasen `clean` und `package` angegeben. Mit der Phase `package` wird der Code kompiliert und in eine ausführbare JAR-Datei gepackt. Mit der Phase `clean` wird das Projekt von früheren Build-Prozessen bereinigt.

Listing 15: pom.xml RDF-Import

```
01  <project xmlns="http://maven.apache.org/POM/4.0.0"
02  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
04  http://maven.apache.org/xsd/maven-4.0.0.xsd">
05    <modelVersion>4.0.0</modelVersion>
06    <groupId>at.jku.dke</groupId>
07    <artifactId>pestel</artifactId>
08    <version>1.0.0-SNAPSHOT</version>
09    <properties>
10      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
11      <cli.version>1.3</cli.version>
```

[3] https://maven.apache.org/
[4] https://jena.apache.org/
[5] https://www.slf4j.org/
[6] https://commons.apache.org/proper/commons-cli/
[7] https://maven.apache.org/plugins/maven-compiler-plugin/
[8] http://maven.apache.org/plugins/maven-assembly-plugin/

```
12      <logback.version>1.0.13</logback.version>
13      <java.version>1.8</java.version>
14      <jena.version>3.3.0</jena.version>
15    </properties>
16    <dependencies>
17      <dependency>
18        <groupId>commons-cli</groupId>
19        <artifactId>commons-cli</artifactId>
20        <version>${cli.version}</version>
21      </dependency>
22      <dependency>
23        <groupId>ch.qos.logback</groupId>
24        <artifactId>logback-classic</artifactId>
25        <version>${logback.version}</version>
26      </dependency>
27      <dependency>
28        <groupId>org.apache.jena</groupId>
29        <artifactId>apache-jena-libs</artifactId>
30        <type>pom</type>
31        <version>${jena.version}</version>
32      </dependency>
33    </dependencies>
34    <build>
35      <plugins>
36        <plugin>
37          <groupId>org.apache.maven.plugins</groupId>
38          <artifactId>maven-assembly-plugin</artifactId>
39          <configuration>
40            <descriptor>assembly/bin.xml</descriptor>
41            <finalName>pestel-${project.version}</finalName>
42            <archive>
43            <manifest>
44               <addClasspath>true</addClasspath>
45                  <mainClass>importMain.RDFImport</mainClass>
46              </manifest>
47            </archive>
48          </configuration>
49          <executions>
50            <execution>
51              <phase>package</phase>
52              <goals>
53                <goal>single</goal>
54              </goals>
55            </execution>
56          </executions>
57        </plugin>
58        <plugin>
59          <groupId>org.apache.maven.plugins</groupId>
60          <artifactId>maven-compiler-plugin</artifactId>
61          <version>3.5.1</version>
62          <configuration >
63            <source>${java.version}</source>
64            <target>${java.version}</target>
65          </configuration>
66        </plugin>
67      </plugins>
68    </build>
69  </project>
```

5.4 Chancen-Risiken-Analyse

Wie in Kapitel 3 beschrieben, besteht die Chancen-Risiken-Analyse aus zwei Funktionen. Zum einen kann man mit einem Button im Editor von MetaEdit+ die Analyse starten und zum anderen die Analyse zurücksetzen. Dazu werden zwei MERL-Generatoren benötigt, welche im Kapitel 5.4.1 beschrieben werden.

Von diesen Generatoren aus wird das in Kapitel 5.4.2 beschriebene Java-Programm ausgeführt. Das Java-Programm greift für die Analyse auf das RDF-Schema, sowie die SPARQL-Abfrage zu, welche in Kapitel 5.4.4 und 5.4.5 beschrieben werden.

5.4.1 MERL-Generator

Wie bereits erwähnt, werden bei der Opportunity-Threat-Analyse, kurz OT Analyse, zwei MERL-Generatoren benötigt, da im Editor von MetaEdit+ zwei Buttons zur Verfügung stehen sollen. Einer davon ist zum Starten der OT Analyse und der zweite, um zuvor gestartete Analysen wieder zurückzusetzen.

Der MERL-Generator zum Starten der Analyse hat den Namen `!OTAnalyse`. Durch das Rufzeichen am Anfang des Namens wird zum Ausführen des Generators automatisch im Editor von MetaEdit+ ein Button zur Verfügung gestellt.

Im Code wird als Erstes mit einer `foreach`-Anweisung über alle `ActorWithBoundary`- und `AgentWithBoudnary`-Objekte iteriert und in eine Variable `actors` gespeichert. Anschließend wird mit der Funktion `promt ask` vom Benutzer der Actor, für welchen die Analyse ausgeführt werden soll, abgefragt. Der vom Benutzer angegebene Actor wird auf die Verfügbarkeit geprüft, indem über die zuvor gespeicherte Variable `actors` iteriert wird. Wenn der angeführte Actor nicht existiert, so wird der Benutzer darüber informiert und der Generator wird beendet. Ist der angegebene Actor in der Variable `actors` enthalten, werden die Funktionen in Listing 16 ausgeführt. Mit der Funktion `internal execute,` sowie dem Kommandozeilenparameter `forAll:run:` in Zeile 1 wird der MERL-Generator `!ExportRDF` ausgeführt. Die Implementierung dieses MERL-Generators wurde bereits im Kapitel 5.2 beschrieben. Dabei wird das erstellte Modell in MetaEdit+ als RDF-Exportiert und unter `C:\Users\<CurrentUserName>\Documents\MetaEdit+5.5\reports\PESTEListar\RDFFile.ttl` gespeichert.

In Zeile 3 wird mit der Funktion `external executeBlocking` das in Kapitel 5.4.2 beschriebene Java-Programm ausgeführt. Dazu wird der Kommandozeilenbefehl `java –cp` verwendet. Zuerst wird der Pfad zur JAR-Datei angegeben, gefolgt von einem Semikolon und dem Pfad zu den abhängigen Java-Bibliotheken. Zur Ausführung der JAR-Datei müssen drei Optionen als Argumente am Ende des Befehles angeführt werden. Die erste Option wird mit `–g` angeführt und gibt den Pfad und Dateinamen zum RDF-Dokument des erstellten Modells an. Die zweite Option wird mit `-a` angeführt und gibt den Namen des ausgewählten Actors an. Die letzte Option (`-d`) bestimmt mit einem booleschen Wert, ob es sich um die Ausführung der Analyse (`false`) oder das Zurücksetzen der Analyse (`true`) handelt.

Listing 16: Ausführung des OT Analyse-JARs

```
01  internal 'forAll:run: PESTEL* !ExportRDF' execute
02
03  external 'java -cp "C:/temp/OTanalyse-1.0.0-SNAPSHOT/bin/OTanalyse-1.0.0-
```

```
SNAPSHOT.jar;C:/temp/OTanalyse-1.0.0-SNAPSHOT/lib/*" at/jku/dke/OTanalyse.App
-g "C:\Users\<CurrentUserName>\Documents\MetaEdit+ 5.5\reports\PESTEListar
\RDFFile.ttl" -a ' $input%var ' -d false' executeBlocking
```

Der MERL-Generator zum Zurücksetzen der Analyse wird mit dem Namen `!ResetOTAnalyse` definiert. Auch für diesen Generator wird im Editor ein Button zum Starten zur Verfügung gestellt. Der Befehl für die Ausführung des Java-Programms ist bis auf zwei Unterschiede genau wie jener in Zeile 3 in Listing 16. Der erste Unterschied ist, dass bei der zweiten Option (`-a`) statt des Namens des Actors der boolesche Wert `false` angeführt ist. Der zweite Unterschied ist, dass bei der letzten Option (`-d`) der Wert `true` angegeben ist.

5.4.2 Java-Programm

In den folgenden Kapiteln werden der Aufbau, der Ablauf und die Methoden des Java-Programms, sowie die MetaEdit+ API genauer erläutert.

5.4.2.1 Aufbau

Das Klassendiagramm zum Java-Programm der OT Analyse ist in Abbildung 20 ersichtlich. Das Java-Programm besteht aus zwei Packages. Das erste Package hat den Namen `at.jku.dke.Model` und beinhaltet die Klasse `Factor`. Die Klasse wird verwendet, um identifizierte Chancen und Risiken darzustellen. Dafür werden der Name der Chance oder des Risikos, sowie die Klassifikation als Variable gespeichert.

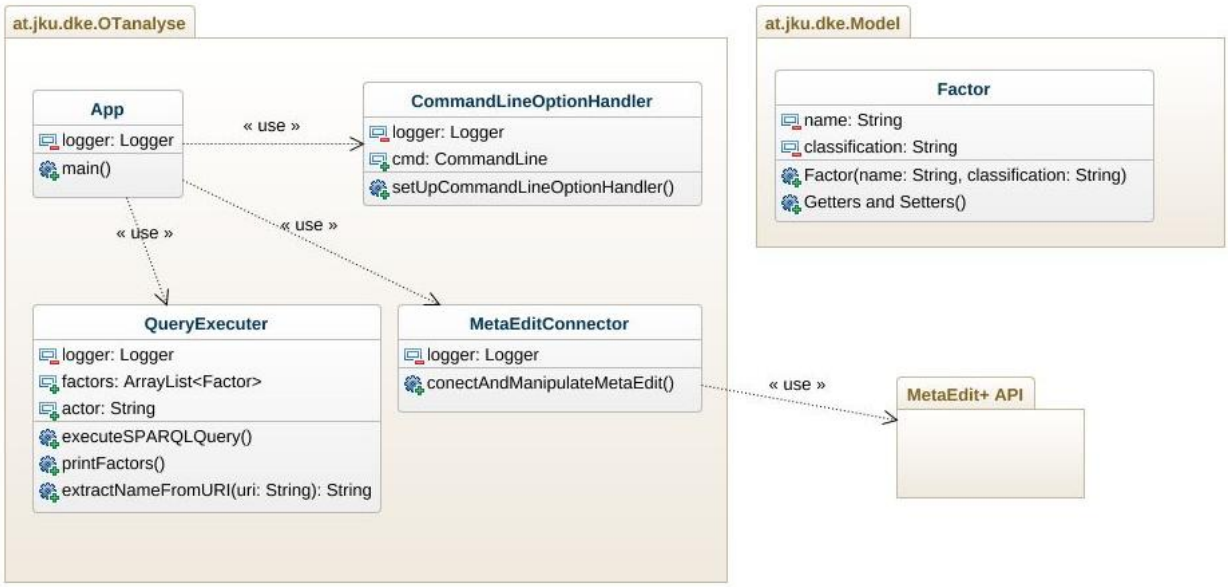


Abbildung 20: Klassendiagramm OT Analyse

Das zweite Package hat den Namen at.jku.dke.OTanalyse und beinhaltet die Klassen `App`, `CommandLineOptionHandler`, `QueryExecuter` und `MetaEditConnector`. Die Main-Methode des Programms befindet sich in der Klasse `App`. Die Klassen `CommandLineOptionHandler`, `QueryExecuter` und `MetaEditConnector` werden von der Klasse App über Aufrufe ihrer statischen Methoden `setUpCommandLineOptionHandler`, `executeSPARQLQuery` und `conectAndManipulateMetaEdit` und ihrer statischen Variable `cmd` verwendet.

Die Klasse `MetaEditConnector` greift über Schnittstellen, welche in einer WSDL-Datei definiert sind, auf die API von MetaEdit+ zu.

5.4.2.2 Ablauf und Methoden

Wie bereits erwähnt, befindet sich die Main-Methode in der Klasse `App` und ist in Listing 17 ersichtlich.

Damit das Logging der verwendeten Java-Bibliotheken nicht in der Konsole ausgegeben wird, wird in Zeile 7 in Listing 17 die Methode `write` der Klasse `OutputStream` überschrieben und so die Ausgabe gestoppt. Zusätzlich wird in Zeile 3 der Logging Level der selbst erstellten Logging Ausgaben auf `INFO` gesetzt.

Im ersten Schritt wird die statische Methode `setUpCommandLineOptionHandler` der Klasse `CommandLineOptionHandler` aufgerufen (Zeile 11 in Listing 17), um die Kommandozeilenoptionen zu parsen und zur weiteren Verwendung zur Verfügung zu stellen. Dazu wird die Java-Bibliothek Apache Commons CLI[9] verwendet. Als Eingabeparameter wird hier das String-Array `args` übergeben. Innerhalb der Methode werden die drei möglichen Options erzeugt und einer Variablen der Klasse Options hinzugefügt.

Die drei Options setzen sich zusammen aus dem Pfad zum RDF-Graphen (`graphPath`), dem Actor (`actor`), für den die Analyse durchgeführt werden soll, sowie einem boolschen Wert (`deleteOnly`), welcher angibt, ob es sich um die Ausführung der gesamten Analyse handelt (Wert ist `false`) oder nur um das Zurücksetzen der Analyse (Wert ist `true`). Für alle drei Options wird festgelegt, dass hier ein Argument benötigt wird und dass die Angabe der Options erforderlich ist, um das Programm auszuführen.

Anschließend werden die Options mit einem `CommandLineParser` geparst und in die statische Variable `cmd` der Klasse `CommandLine` gespeichert, um den Inhalt der Options für die weitere Analyse zur Verfügung zu stellen. Kommt es zu einer ParseException, wird diese mit einem `try-catch`-Block abgefangen und das Programm wird beendet.

Listing 17: Main-Methode OT Analyse

```
01  public static void main( String[] args ){
02     BasicConfigurator.configure();
03     Logger.getRootLogger().setLevel(Level.INFO);
04
05     PrintStream out = System.out;
06     System.setOut(new PrintStream(new OutputStream() {
07         @Override public void write(int b) throws IOException {}
08     }));
09     try {
```

[9] https://commons.apache.org/proper/commons-cli/

```
10        logger.info("Command line will be assessed and handeled");
11        CommandLineOptionHandler.setUpCommandLineOptionHandler(args);
12
13        boolean deleteOnly =
14  Boolean.valueOf(CommandLineOptionHandler.cmd.getOptionValue("deleteOnly"));
15
16        if(!deleteOnly){
17           logger.info("SPARQL Query will be executed");
18           QueryExecuter.executeSPARQLQuery();
19        }
20
21        logger.info("Connecting to MetaEdit and Manipulate Graph");
22        MetaEditConnector.connectAndManipulateMetaEdit(deleteOnly);
23     } finally {
24        System.setOut(out);
25     }
16  }
```

Da die SPARQL-Abfrage nur durchgeführt werden muss, wenn es sich um die Ausführung der Analyse und nicht um das Zurücksetzen der Analyse handelt, wird in Zeile 13 in Listing 17 der boolesche Wert der Option `deleteOnly` abgefragt. Die statische Methode `executeSPARQLQuery` der Klasse `QueryExecuter` in Zeile 18 wird nur ausgeführt, wenn der Wert der Variable `deleteOnly` `false` ist.

Kommt es zur Ausführung der Methode `executeSPARQLQuery` in Zeile 18 in Listing 17 wird innerhalb der Methode über die statische Variable `cmd` der Klasse `CommandLineOptionHandler` der Pfad zum RDF-Dokument, sowie der angegebene Actor abgefragt.

Für die weitere Abfolge der Methode wird die Java-Bibliothek Apache Jena[10] verwendet. Damit die SPARQL-Abfrage auf das RDF-Modell richtig durchgeführt werden kann, muss ein Inferenzmodell auf Basis zweier definierter RDF-Schema Modelle, welche in Kapitel 5.4.4 genauer erklärt sind, erstellt werden. Die beiden RDF-Schema Dokumente müssen unter dem Pfad, wie in Zeile 4 und 5 in Listing 18, gespeichert sein. In dem Inferenzmodell sind alle RDF-Tripel enthalten, welche von den beiden RDF-Schema Modellen abgeleitet werden können. Dazu werden zuerst für das RDF-Modell, sowie für die beiden RDF-Schema Modelle Objekte der Klasse `Model` erstellt (Zeile 8 bis 10 in Listing 18). Anschließend wird in Zeile 13 und 14 mit der Methode `createRDFSModel` aus dem RDF-Modell, sowie den beiden RDF-Schema Modellen ein Inferenzmodell erstellt, welches die Grundlage für die SPARQL-Abfrage darstellt.

Listing 18: Erstellung des RDF-Inferenzmodells

```
01   String graphPath = CommandLineOptionHandler.cmd.getOptionValue("graphPath");
02   actor = ":" + CommandLineOptionHandler.cmd.getOptionValue("actor");
03
04   String pestelSchemaPath = "C:\\Users\\metaedit\\Documents\\MetaEdit+
     5.5\\PESTEListar\\schema\\pestelSchema.ttl";
05   String iStarSchemaPath = "C:\\Users\\metaedit\\Documents\\MetaEdit+
     5.5\\PESTEListar\\schema\\iStarSchema.ttl";
06
07   //Load model (RDF-Graph and schema)
08   Model model = RDFDataMgr.loadModel(graphPath);
```

[10] https://jena.apache.org/

```
09   Model iStarSchema = RDFDataMgr.loadModel(iStarSchemaPath);
10   Model pestelSchema = RDFDataMgr.loadModel(pestelSchemaPath);
11
12   //Generate inference model
13   InfModel schema = ModelFactory.createRDFSModel(iStarSchema, pestelSchema);
14   InfModel infModel = ModelFactory.createRDFSModel(schema, model);
```

In Listing 19 ist die Vorgangsweise zum Ausführen der SPARQL-Abfrage auf das erstellte Inferenzmodell abgebildet. Dabei wird in Zeile 1 über die Methode `create` ein Objekt der Klasse `Query` erstellt, welche als Eingabeparameter die SPARQL-Abfrage als String benötigt. Die Funktionsweise und der Aufbau dieser SPARQL-Abfrage wird im Kapitel 5.4.5 genauer erläutert. In Zeile 2 wird ein Objekt der Klasse `QueryExecuter` auf Basis der definierten Abfrage und dem Inferenzmodell erstellt. Indem die Methode `execSelect` in Zeile 3 aufgerufen wird, wird die Abfrage ausgeführt und das Ergebnis wird als `ResultSet` zurückgegeben. Über dieses `ResultSet` wird anschließend iteriert und identifizierte Chancen und Risiken werden als Objekte der Klasse `Factor` in eine statisch definierte `ArrayList factors` gespeichert, damit später bei der Manipulation des MetaEdit+ Modells auf die Faktoren zugegriffen werden kann.

Listing 19: Ausführung der SPARQL-Abfrage

```
01  Query query = QueryFactory.create(queryString);
02  QueryExecution qexec = QueryExecutionFactory.create(query, infModel);
03  ResultSet results = qexec.execSelect();
```

In Zeile 22 der Main-Methode (Listing 17) wird die statische Methode `connectAndManipulateMetaEdit` der Klasse `MetaEditConnector` ausgeführt. Innerhalb dieser Methode wird mit einem Objekt der Klasse `MetaEditAPILocator`, sowie der Methode `getMetaEditAPIPort` dieser Klasse eine Verbindung zu MetaEdit+ hergestellt (MetaCase 2017b) (Zeile 5 und 6 in Listing 20). Über den hergestellten Port kann nun mit unterschiedlichsten Methoden der API das Modell in MetaEdit+ manipuliert werden. Kann keine Verbindung hergestellt werden, so wird das Programm beendet.

Listing 20: Verbindungsherstellung zu MetaEdit+

```
01  MetaEditAPI meServer = null;
02  MetaEditAPIPortType port = null;
03
04  try {
05     meServer = new MetaEditAPILocator();
06     port = meServer.getMetaEditAPIPort();
07  } catch (Exception ex) {
08     logger.info(ex.getMessage(), ex);
09     System.exit(1);
10  }
```

Für die Analyse werden zwei unterschiedliche Objektgruppen manipuliert. Als Erstes wird jener Actor gesucht, für den die Analyse durchgeführt werden soll, um diesen grafisch hervorzuheben. Dazu werden mit der Methode `allGoodInstances` der API alle Graphen vom Typ `PESTEL i-star` zurückgegeben und in einem Array gespeichert. Mit der Methode

`objectSet` werden die Objekte innerhalb des Graphen zurückgegeben und wiederum in ein Array gespeichert. Die grafische Hervorhebung von zuvor ausgeführten Analysen wird zurückgesetzt. Soll die Analyse nicht nur zum Zurücksetzen zuvor ausgeführter Analysen durchgeführt werden, so wird durch die Iteration über das Array der ausgewählte Actor ermittelt. Anschließend ermöglicht die Methode `setValueAt` der API den Wert der Eigenschaften von Objekten zu ändern. Dabei wird die Eigenschaft `Analyse` des Actors auf den Wert `true` gesetzt. Dadurch wird wie im Metamodell definiert, die grafische Darstellung des Actors verändert und das Symbol wird grau schattiert.

Als Zweites werden jene Tasks, Qualities, Goals und Resources gesucht, welche im Subgraphen eines PESTEL-Faktors liegen und bei der SPARQL-Abfrage als Chancen oder Risiken für den angegeben Actor identifiziert wurden. Dazu wird wiederum die Methode `allGoodInstances` benötigt. Jedoch wird in diesem Schritt nicht nach dem Graph-Typ `PESTEL i-star` gesucht, sondern nach dem Graph-Typ `FactorBoundary`. Mit der Methode `objectSet` werden die Objekte innerhalb des Graph-Typen `FactorBoundary` zurückgegeben. Sollte es sich nur um das Zurücksetzen der Analyse handeln, wird über alle diese Objekte iteriert und die Eigenschaften Opportunity und Threat werden auf `false` gesetzt. Ist dies nicht der Fall, werden zusätzlich die Eigenschaften der Objekte, welche von der SPARQL-Abfrage als Opportunities und Threats identifiziert wurden, so manipuliert, dass sie grafisch, wie im Metamodell definiert, als Opportunies und Threats erscheinen.

5.4.2.3 Apache Maven

Auch dieses Java-Programm wurde als Apache Maven[11] Projekt erstellt. Das Projekt-Object-Model-Dokument der OT Analyse ist grundsätzlich identisch im Inhalt und Aufbau wie jenes des RDF-Imports. Der einzige Unterschied ist, dass bei der OT Analyse zusätzlich zu den beim RDF-Import benötigten Abhängigkeiten drei weitere Java-Bibliotheken angegeben sind. Dazu gehören die Bibliotheken Apache Xerces[12], Apache Axis[13], sowie Apache Log4j[14], welche in Listing 21 ersichtlich sind.

Listing 21: pom.xml OT Analyse

```
01  .</dependencies>
02        ...
..        <dependency>
08              <groupId>xerces</groupId>
09              <artifactId>xercesImpl</artifactId>
10              <version>${xerces.version}</version>
11        </dependency>
12        <dependency>
13              <groupId>axis</groupId>
14              <artifactId>axis</artifactId>
15              <version>${axis.version}</version>
16              <exclusions>
17                    <exclusion>
18                          <groupId>commons-logging</groupId>
19                          <artifactId>commons-logging</artifactId>
20                    </exclusion>
```

[11] https://maven.apache.org/
[12] http://xerces.apache.org/
[13] http://axis.apache.org/
[14] https://logging.apache.org/log4j/2.x/

```
21              </exclusions>
22        </dependency>
23        <dependency>
24              <groupId>log4j</groupId>
25              <artifactId>log4j</artifactId>
26              <version>1.2.17</version>
27        </dependency>
    </dependencies>
```

5.4.3 Konfiguration MetaEdit+ API

Die Verwendung der MetaEdit+ API funktioniert über SOAP Aufrufe (MetaCase 2017b). Dazu stellt MetaEdit+ einen SOAP Server, sowie ein WSDL Dokument mit den API Schnittstellen zur Verfügung. Das MetaEdit+ API Tool, wie in Abbildung 21 ersichtlich, stellt dazu das User-Interface dar. Hier müssen die Servereinstellungen Hostname und Port angegeben werden. Nach dem Starten des Servers (Klick auf den Button „Start Server“) kann MetaEdit+ die API SOAP Aufrufe von externen Java-Programmen empfangen und verarbeiten.

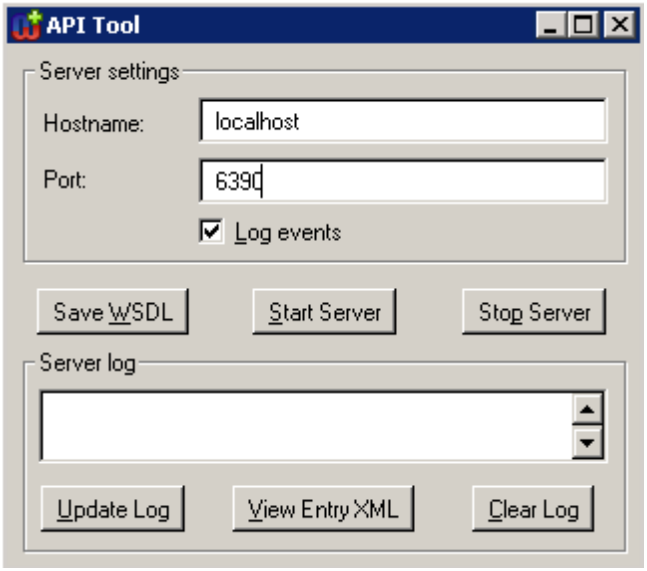


Abbildung 21: MetaEdit+ API Tool

Durch Klick auf den Button „Save WSDL“ wird ein WSDL Dokument erzeugt, welches für den SOAP Client verwendet wird. Die Umwandlung des WSDL Dokuments wurde im Eclipse durch die Erstellung eines Web Server Clients durchgeführt. Dabei wurde als Service Definition das erzeugte WSDL Dokument angegeben. Dadurch wurden die zwei Packages erzeugt. Das Package `com.metacase.type` enthält die Klassen `MEAny`, `MENull`, `MEOop` und `METype`. Das Package `com.metacase.wsdl` enthält die Klassen `MetaEditAPI`, `MetaEditAPILocator`, `MetaEditAPIPortType`, `MetaEditAPIPortTypeProxy` und `MetaEditAPISoapBindingStub`.

In der Klasse `MetaEditAPILocator` werden in der String-Variablen `MedaEditAPIPort_address` die Servereinstellungen angegeben (Zeile 10 in Listing 22).

Listing 22: Serviceeinstellungen MetaEditAPILocator (MetaCase 2017b)

```
01  public class MetaEditAPILocator ... {
..     ...
10     private String MetaEditAPIPort_address =
    "http://localhost:6390/MetaEditAPI";
..     ...
30  }
```

Damit das SOAP Framework funktioniert, werden zwei Java-Bibliotheken benötigt. Dazu zählt zum einen die Apache Xerces[15] Bibliothek (MetaCase 2017b), welche benötigt wird, um XML Dokumente zu parsen, validieren oder manipulieren (Apache Xerces 2016). Die zweite ist die Apache Axis[16] Bibliothek, welche für die SOAP Aufrufe verwendet wird (Apache Software Foundation 2017b).

5.4.4 RDF-Schema

Damit es möglich ist, die SPARQL Query auf das RDF-Modell korrekt ausführen zu können, ist es nötig, ein Inferenzmodell zu erstellen, bei dem das RDF-Modell um Fakten erweitert wird, welche von definierten RDF-Schema Modellen abgeleitet werden können.

Dazu wurden, wie in Listing 23 und Listing 24 ersichtlich, zwei RDF-Schema-Modelle erstellt. Alle Elemente des iStar 2.0 Frameworks werden im RDF-Schema Modell iStar in Listing 23 mit dem Prefix `istar` dargestellt (Dalpiaz et al. 2016). Da die Objekte *Actor*, *Agent, Goal, Quality, Task* und *Resource* in einem Triple mit dem Prädikat `rdfs:subClassOf` definiert werden, stellen sie daraus abgeleitet einen RDF-Typ rdfs:Class dar. Zusätzlich stellen die Objekte *Goal, Quality, Task* und *Resource* eine Subklasse von `istar:IntentionalElement` dar. Das Objekt Agent stellt eine Subklasse von `istar:Actor` dar.

Des Weiteren werden sämtliche iStar 2.0 Beziehungen als RDF-Typ `rdf:Property` definiert. Mit den RDF-Prädikaten `rdfs:domain` und `rdfs:range` werden die Domain und Range der Eigenschaften definiert. Die Beziehungen `istar:isA` und `istar:participatesIn` werden mit dem RDF-Prädikat `rdfs:subPropertyOf` als Subbeziehung von `istar:relatedWith` definiert. Die Beziehungen `istar:helps` und `istar:hurts` werden als Subbeziehung von `istar:contributesTo` definiert. Diese Definitionen ermöglichen, dass zum Beispiel bei dem RDF-Tripel "`:RyanAir istar:isA :Airline`" für das Inferenzmodell auch der RDF-Tripel "`:RyanAir istar:relatedWith :Airline`" abgeleitet wird.

Listing 23: RDF-Schema iStar (Schütz und Schrefl 2017)

```
01  @prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
02  @prefix istar: <http://xmlns.com/istar/0.1/> .
03  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
04  @prefix : <http://example.org/> .
05
06  istar:Agent rdfs:subClassOf istar:Actor.
07  istar:Goal rdfs:subClassOf istar:IntentionalElement.
```

[15] http://xerces.apache.org/
[16] http://axis.apache.org/

```
08  istar:Quality rdfs:subClassOf istar:IntentionalElement.
09  istar:Task rdfs:subClassOf istar:IntentionalElement.
10  istar:Resource rdfs:subClassOf istar:IntentionalElement.
11
12  istar:relatedWith rdf:type rdf:Property;
13     rdfs:domain istar:Actor;
14     rdfs:range istar:Actor.
15
16  istar:isA rdf:type rdf:Property;
17     rdfs:domain istar:Actor;
18     rdfs:range istar:Actor;
19     rdfs:subPropertyOf istar:relatedWith.
20
21  istar:participatesIn rdf:type rdf:Property;
22     rdfs:domain istar:Actor;
23     rdfs:range istar:Actor;
24     rdfs:subPropertyOf istar:relatedWith.
25
26  istar:wants rdf:type rdf:Property;
27     rdfs:domain istar:Actor;
28     rdfs:range istar:IntentionalElement.
29
30  istar:dependerOf rdf:type rdf:Property.
31
32  istar:contributesTo rdf:type rdf:Property;
33     rdfs:domain istar:IntentionalElement;
34     rdfs:range istar:Quality.
35
36  istar:helps rdf:type rdf:Property;
37     rdfs:domain istar:IntentionalElement;
38     rdfs:range istar:Quality;
39     rdfs:subPropertyOf istar:contributesTo.
40
41  istar:hurts rdf:type rdf:Property;
42     rdfs:domain istar:IntentionalElement;
43     rdfs:range istar:Quality;
44     rdfs:subPropertyOf istar:contributesTo.
45
46  istar:qualifies rdf:type rdf:Property;
47     rdfs:domain istar:IntentionalElement;
48     rdfs:range istar:Quality.
49
50  istar:refines rdf:type rdf:Property;
51     rdfs:domain istar:IntentionalElement.
```

Alle PESTEL-Faktoren werden mit dem Prefix `pestel` im RDF-Schema Pestel in Listing 24 als Subklassen der Klasse `pestel:PESTELFactor` definiert. Zusätzlich wird `pestel:PESTELFactor` als Subklasse von `istar:Actor` definiert. Das führt dazu, dass im Inferenzmodell für alle PESTEL-Faktoren auch die Klasse `istar:Actor` abgeleitet wird und diese als solche erkannt werden.

Listing 24: RDF-Schema PESTEL

```
01  @prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
02  @prefix istar: <http://xmlns.com/istar/0.1/> .
03  @prefix pestel: <http://xmlns.com/pestel/0.1/> .
04  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
05  @prefix : <http://example.org/> .
06
07  pestel:PESTELFactor rdfs:subClassOf istar:Actor.
08  pestel:Political rdfs:subClassOf pestel:PESTELFactor.
09  pestel:Economic rdfs:subClassOf pestel:PESTELFactor.
10  pestel:Social rdfs:subClassOf pestel:PESTELFactor.
11  pestel:Technological rdfs:subClassOf pestel:PESTELFactor.
12  pestel:Environmental rdfs:subClassOf pestel:PESTELFactor.
13  pestel:Legal rdfs:subClassOf pestel:PESTELFactor.
```

5.4.5 SPARQL-Abfrage

Mit der SPARQL-Abfrage werden für einen bestimmten Actor oder Agent die Chancen und Risiken, welche innerhalb des Subgraphen eines PESTEL-Faktors liegen, ermittelt (Schütz und Schrefl 2017). Die dazu verwendete SPARQL-Abfrage ist in Listing 25 ersichtlich. Die verwendete Variable `?elementInFactor` in Zeile 7 stellt den Namen der Chance oder des Risikos dar. Die Variable `?classification`, welche ebenso in Zeile 7 definiert ist, stellt die Klassifikation, also ob es sich um eine Chance oder um ein Risiko handelt, dar.
Eine Chance oder ein Risiko wird dann identifiziert, wenn folgende fünf Fakten im RDF-Modell zutreffen:

1. Es muss einen PESTEL-Faktor geben, welcher in seinem Subgraphen durch die Beziehung `istar:wants` die Chance bzw. das Risiko (Variable `?elementInFactor`) enthält. Dieser Fakt wird in Zeile 15 und 21 dargestellt.
2. Es muss ein Objekt (`?hurtingElems` / `?helpingElems`) geben, von dem eine Beziehung `istar:dependerOf`, sowie die Beziehung `istar:refines` zur Chance oder zum Risiko (`?elementInFactor`) verläuft. Dieser Fakt wird in Zeile 14 und 20 dargestellt. Dabei wird durch das Plus (+) nach der Beziehung `istar:dependerOf` ausgedrückt, dass es sich mindestens um eine, aber auch um mehrere aufeinander folgende Beziehungen handeln muss. Mit dem Stern (*) nach der Beziehung `istar:refines` wird ausgedrückt, dass diese Beziehung optional vorhanden sein kann, also gar nicht oder auch mehrmals hintereinander.
3. Es muss ein Objekt geben, das dasselbe Objekt von Fakt 2 darstellt, von dem mindestens eine oder mehrere aufeinanderfolgende Beziehungen `istar:hurts` oder `istar:helps` zu einem anderen Objekt (`?quality`) verlaufen. Da laut RDF-Schema istar (siehe Listing 23) die Range dieser Beziehungen vom RDF-Typ `istar:Quality` sein muss, muss das Objekt mit der Variable `?quality` auch ein Objekt vom RDF-Type `istar:Quality` darstellen. Dieser Fakt wird in Zeile 13 und 19 dargestellt, je nachdem, ob eine Chance oder ein Risiko identifiziert werden soll.
4. Vom Objekt, welches laut Fakt 3 eine `istar:Quality` darstellt, muss eine Beziehung `istar:qualifies` zu einem anderen Objekt (Variable `?goal`) verlaufen, wobei das andere Objekt (`?goal`) innerhalb des Subgraphen des gewünschten Actors/Agents oder innerhalb des Subgraphen eines Actors/Agents, zu welchem der gewünschte

Actor/Agent eine oder mehrere aufeinanderfolgende `istar:relatedWith`-Beziehungen aufweist, liegen muss. Dieser Fakt wird in Zeile 9 und 10 dargestellt.

5. Das Objekt mit der Variable `?hurtingElems` / `?helpingElems` muss innerhalb des Subgraphen des gewünschten Actors/Agents oder innerhalb des Subgraphen eines Actors/Agents, zu welchem der gewünschte Actor/Agent eine oder mehrere aufeinanderfolgende `istar:relatedWith`-Beziehungen aufweist, liegen. In Listing 25 wird als Beispiel der Actor `:RyanAir` verwendet. Dieser Fakt wird in Zeile 12 und 18 dargestellt. Die `istar:relatedWith`-Beziehungen werden für das Inferenzmodell auf Basis des definierten RDF-Schemas iStar (siehe Listing 23) abgeleitet.

Handelt es sich beim Fakt 3 um die Beziehung `istar:helps`, wird das identifizierte Element in Zeile 22 mit dem Schlüsselwort `BIND` als `swot:Opportunity`, also Chance, klassifiziert. Handelt es sich um die Beziehung `istar:hurts`, wird es in Zeile 16 als `swot:Threat`, also Risiko, klassifiziert. Mit dem Schlüsselwort `UNION` werden die identifizierten Chancen und Risiken für das Ergebnis vereint.

Listing 25: SPARQL-Abfrage (Schütz und Schrefl 2017)

```
01  PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
02  PREFIX istar: <http://xmlns.com/istar/0.1/>
03  PREFIX swot: <http://xmlns.com/swot/0.1/>
04  PREFIX pestel: <http://xmlns.com/pestel/0.1/>
05  PREFIX : <http://example.org/>
06
07  SELECT ?elementInFactor ?classification
08     WHERE {
09        :RyanAir istar:relatedWith*/istar:wants ?goal.
10        ?quality istar:qualifies ?goal.
11        {
12        :RyanAir istar:relatedWith*/istar:wants ?hurtingElems.
13        ?hurtingElems istar:hurts+ ?quality.
14        ?hurtingElems istar:dependerOf+/istar:refines* ?elementInFactor.
15        ?factor istar:wants ?elementInFactor.
16        BIND(swot:Threat AS ?classification)
17        } UNION {
18        :RyanAir istar:relatedWith*/istar:wants ?helpingElems.
19        ?helpingElems istar:helps+ ?quality.
20        ?helpingElems istar:dependerOf+/istar:refines* ?elementInFactor.
21        ?factor istar:wants ?elementInFactor.
22        BIND(swot:Opportunity AS ?classification)
23        }
24     }
```

6 Fazit und Ausblick

Diese Arbeit beschreibt die Anwendung, Architektur und Implementierung des Werkzeugs PESTEL Modeler. Dieses Werkzeug ermöglicht es, strategische Analysen, wie die PESTEL-Analyse oder die SWOT-Analyse, auf Basis der konzeptuellen Modellierungssprache iStar 2.0 graphisch darzustellen und mit semantischen Technologien – wie RDF, RDFS und SPARQL – maschinenlesbar und -auswertbar zu machen.

Zur PESTEL-Analyse zählen die politischen, wirtschaftlichen, sozialen, technologischen, rechtlichen und ökologischen Umweltfaktoren eines Unternehmens oder einer Organisation. Für diese Arbeit sind nur die externen Umweltfaktoren der SWOT-Analyse, also die Chancen (Opportunities) und Risiken (Threats), relevant.

Den Mittelpunkt des PESTEL Modelers stellt das metaCASE-Werkzeug MetaEdit+ dar, mit dessen Hilfe Metamodelle erstellt und in einem Editor angewandt werden können. Das Metamodell setzt sich aus den einzelnen Faktoren der PESTEL-Analyse, sowie den Objekten und Beziehungen, welche Teil des iStar 2.0 Frameworks sind, zusammen. Erstellte Modelle können als RDF-Modelle exportiert werden, ebenso können externe RDF-Modelle importiert und graphisch im Editor dargestellt und bearbeitet werden. Für erstellte Modelle kann mit der Abfragesprache SPARQL auf Basis des im Hintergrund erzeugten RDF-Modells eine Analyse ausgeführt werden, mit der die Chancen (Opportunities) und Risiken (Threats) abgeleitet und graphisch dargestellt werden.

Zur Anwendung des PESTEL Modelers stehen dem User also die folgenden Funktionen zur Verfügung: Modellerstellung, RDF-Export, RDF-Import, Chancen-Risiken-Analyse und Zurücksetzen der Chancen-Risiken-Analyse.

Das PESTEL-Modeler-Werkzeug besteht aus verschiedenen Komponenten, Artefakten und Systemen, welche über unterschiedliche Schnittstellen miteinander verbunden sind und kommunizieren. Die Hauptsysteme des PESTEL Modelers stellen die *MetaEdit+ Software*, das Java-Programm für den *RDF-Import,* sowie das Java-Programm für die Chancen-Risiken-Analyse dar. Innerhalb der MetaEdit+ Software wird ein Metamodell definiert, bei dem für jedes Objekt und jede Beziehung Name, Eigenschaften und Notation festgelegt wird. Im Mittelpunkt des Werkzeugs steht der Diagramm Editor, mit dem Modelle auf Basis des Metamodells erstellt und die Analysen gestartet werden können. Der Diagramm Editor ist ebenso Teil der MetaEdit+ Software. Zur Ausführung der oben genannten Java-Programme, sowie für den RDF-Export, gibt es drei implementierte MERL-Generatoren. Ebenso stellt der SOAP Server eine wichtige Komponente des MetaEdit+ Systems dar, welcher über SOAP Calls erreicht werden kann. Die Generatoren können im Diagramm Editor vom User gestartet werden. Beim Starten des Generators für den RDF-Export wird aus dem im Editor erstellten Modell ein RDF generiert und extern gespeichert. Wird der Generator für den RDF-Import gestartet, wird vom Generator das gleichnamige Java-Programm ausgeführt. Dabei wird ein externes RDF-Dokument in eine bestimmte XML-Struktur geparst, sodass es anschließend in MetaEdit+ importiert werden kann.

Durch das Starten des Generators der Chancen-Risiken-Analyse wird das dazugehörige Java-Programm ausgeführt, welches auf die externen Artifakte RDF-Modell, RDFS und SPARQL-Abfrage zugreift. Beim Starten der Chancen-Risiken-Analyse wird auf Basis der RDFS Dokumente aus dem RDF-Modell ein Inferenzmodell erstellt. Mit einer SPARQL-Abfrage werden die Chancen (Opportunities) und Risiken (Threats) aus dem Inferenzmodells abgeleitet und über SOAP Calls zum SOAP Server von MetaEdit+ graphisch im Diagramm Editor dargestellt.

Der PESTEL Modeler stellt somit eine Möglichkeit der Formalisierung von strategischen Analysen als Ergänzung zum intuitiven und kreativen strategischen Analyseprozess dar. Eine Modellierung und Analyse unter Verwendung des PESTEL Modelers fordert nicht nur die Kenntnisse des iStar 2.0 Frameworks, sowie der PESTEL-Analyse, sondern auch die Erhebung der Wissensfakten. Textuelle Berichte und Analysen müssen in Ontologien übersetzt werden können. Dafür müssen zuerst wichtige Elemente und Handlungsträger erkannt und ihre Zusammenhänge festgestellt werden.

Schütz und Schrefl (2017) beschreiben in ihrem Paper weiterführende Themen, die unter anderem Analysen enthalten, die den organisatorischen Umstrukturierungsaufwand untersuchen, welcher nötig ist, um herkömmliche textuelle Analysen mit maschinenlesbaren- und auswertbaren Analysesysteme zu erweitern. Sie beschreiben auch die Möglichkeit Benutzerstudien mit Fachexperten aus dem Bereich des strategischen Managements durchzuführen, um das erstellte PESTEL-Modeler-Werkzeug zu testen und zu bewerten.

Neben der Entwicklung des PESTEL-Modeler-Werkzeugs können zusätzliche graphische Modellierungswerkzeuge entwickelt werden, welche weitere strategische Analysen unter der Anwendung anderer Ontologien unterstützen. Zu diesem Thema beschreibt Helmberger (2017) in seiner Masterarbeit wie durch die Verwendung der Ontologien Resource-Event-Agent und e3forces für weitere strategische Analysen, wie zum Beispiel für die Wertketten- und Branchenstrukturanalyse, eine graphische Darstellung und Auswertung gewährleistet wird. Es besteht die Möglichkeit auch für diese Ansätze einen Modellierungssupport zu liefern beziehungsweise zu entwickeln. Auf Basis dieser weiteren graphischen Modellierungswerkzeuge können ebenso Benutzerstudien durchgeführt werden und mit den Benutzerstudien anderer graphischer Modellierungswerkzeuge, wie mit dem PESTEL-Modeler-Werkzeug, verglichen werden.

Abbildungsverzeichnis

Listingverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

Apache Maven (2017a): Apache Maven Assembly Plugin. Online verfügbar unter http://maven.apache.org/plugins/maven-assembly-plugin/, zuletzt aktualisiert am 13.18.2017.

Apache Maven (2017b): Apache Maven Compiler Plugin. Online verfügbar unter https://maven.apache.org/plugins/maven-compiler-plugin/, zuletzt aktualisiert am 01.09.2017.

Apache Software Foundation (2013): ExRIOT_6.java. Example of using RIOT : iterate over output of parser run. Hg. v. GitHub. Online. Online verfügbar unter https://github.com/apache/jena/blob/master/jena-arq/src-examples/arq/examples/riot/ExRIOT_6.java, zuletzt geprüft am 25.10.2017.

Apache Software Foundation (2017a): Reading RDF in Apache Jena. Iterating over parser output. Online verfügbar unter https://jena.apache.org/documentation/io/rdf-input.html, zuletzt geprüft am 20.10.2017.

Apache Software Foundation (2017b): Welcome to Apache Axis2/Java. Online verfügbar unter http://axis.apache.org/axis2/java/core/, zuletzt aktualisiert am 30.07.2017, zuletzt geprüft am 20.10.2017.

Apache Xerces (2016): The Apache Xerces™ Project. Apache Xerces2 Java. Online verfügbar unter http://xerces.apache.org/#xerces2-j, zuletzt aktualisiert am 17.08.2016, zuletzt geprüft am 25.10.2017.

Auer, Sören; Lehmann, Jens; Ngomo, Axel-Cyrille Ngonga; Zaveri, Amrapali (2013): Introduction to Linked Data and Its Lifecycle on the Web. In: Reasoning Web, S. 1–90. Online verfügbar unter http://jens-lehmann.org/files/2013/reasoning_web_linked_data.pdf.

Brickley, Dan; Guha, R.V.; McBride, Brian (2014): RDF Schema 1.1. W3C Recommendation. Online verfügbar unter https://www.w3.org/TR/rdf-schema/, zuletzt aktualisiert am 25.02.2014, zuletzt geprüft am 16.11.2017.

Cyganiak, Richard; Wood, David; Lanthaler, Markus; Klyne, Graham; Carroll, Jeremy J.; McBride, Brian (2014): RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation. Online verfügbar unter https://www.w3.org/TR/rdf11-concepts/, zuletzt aktualisiert am 25.02.2014, zuletzt geprüft am 16.11.2017.

Dalpiaz, Fabiano; Franch, Xavier; Horkoff, Jennifer (2016): iStar 2.0 language guide: arXiv:1605.07767 [cs.SE]. Online verfügbar unter dalp-fran-hork-16-istar.pdf.

Decker, Stefan; Brickley, Dan; Saarela, Janne; Angele, Jürgen (oJ): A Query and Inference Service for RDF. W3C Technology & Society Domain. Online verfügbar unter https://www.w3.org/TandS/QL/QL98/pp/queryservice.html, zuletzt geprüft am 16.11.2017.

Dengel, Andreas (Hg.) (2012): Semantische Technologien: Grundlagen - Konzepte - Anwendungen. Heidelberg: Spektrum Akademischer Verlag.

Ebert, Jürgen; Süttenbach, Roger; Uhe, Ingar (1997): Meta-CASE in practice: A CASE for KOGGE. In: Antoni Olivé und Joan Antoni Pastor (Hg.): Advanced Information

Systems Engineering: 9th International Conference, CAiSE'97 Barcelona, Catalonia, Spain, June 16-20, 1997 Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 203–216. Online verfügbar unter https://doi.org/10.1007/3-540-63107-0_15.

Fernandes, Paulo C. Barbosa; Guizzardi, Renata S. S.; Guizzardi, Giancarlo (2011): Using Goal Modeling to Capture Competency Questions in Ontology-based Systems. In: *JIDM* 2 (3), S. 527–540. Online verfügbar unter http://seer.lcc.ufmg.br/index.php/jidm/article/view/145.

Helmberger, Peter (2017): Einsatz von RDF und SPARQL zur strategischen Analyse mit Geschäftsmodellontologien am Beispiel ausgewählter Fallstudien.

Johnson, Gerry; Scholes, Kevan; Whittington, Richard (2008): Exploring Corporate Strategy. 8th. Upper Saddle River, NJ, USA: Prentice Hall Press.

Kelly, Steven; Pohjonen, Risto (2013): Dynamic Symbol Templates and Ports in MetaEdit+. In: Proceedings of the 2013 ACM Workshop on Domain-specific Modeling. New York, NY, USA: ACM (DSM '13), S. 19–20. Online verfügbar unter http://doi.acm.org/10.1145/2541928.2541932.

MetaCase (2017a): MetaEdit+ 5.5 User's Guide. MetaCase. Jyväskylä Finland. Online verfügbar unter https://www.metacase.com/support/55/manuals/meplus/Mp.html.

MetaCase (2017b): MetaEdit+ 5.5 Workbench User's Guide. MetaCase. Jyväskylä Finland. Online verfügbar unter https://www.metacase.com/support/55/manuals/mwb/Mw.html, zuletzt geprüft am 25.11.2017.

Miller, Frederic P.; Vandome, Agnes F.; McBrewster, John (2010): Apache Maven: Alpha Press.

Palsberg, Jens; Jay, C. Barry (1998): The Essence of the Visitor Pattern. In: Proceedings of the 22Nd International Computer Software and Applications Conference. Washington, DC, USA: IEEE Computer Society (COMPSAC '98), S. 9–15. Online verfügbar unter http://dl.acm.org/citation.cfm?id=645980.674267.

Pohjonen, Risto (2005): Metamodeling Made Easy - MetaEdit+ (Tool Demonstration). In: Robert Glück und Michael Lowry (Hg.): Generative Programming and Component Engineering: 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005. Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 442–446. Online verfügbar unter https://doi.org/10.1007/11561347_30.

Prud'hommeaux, Eric; Seaborne, Andy (2013): SPARQL 1.1 Query Language. W3C Recommendation. Online verfügbar unter https://www.w3.org/TR/rdf-sparql-query/, zuletzt aktualisiert am 21.03.2013, zuletzt geprüft am 16.11.2017.

Rufo, Marc; Zerres, Christopher (2017): Strategische Analysetechniken. In: Christopher Zerres (Hg.): Handbuch Marketing-Controlling: Grundlagen - Methoden - Umsetzung. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 69–90. Online verfügbar unter https://doi.org/10.1007/978-3-662-50406-2_4.

Schreiber, Guus; Raimond, Yves; Manola, Frank; Miller, Eric; McBride, Brian (2014): RDF 1.1 Primer. W3C Working Group. Online verfügbar unter https://www.w3.org/TR/rdf11-primer/, zuletzt aktualisiert am 24.06.2014, zuletzt geprüft am 16.11.2017.

Schütz, Christoph; Neumayr, Bernd; Schrefl, Michael (2013): Business Model Ontologies in OLAP Cubes. In: Camille Salinesi, Moira C. Norrie und Óscar Pastor (Hg.): Advanced Information Systems Engineering: 25th International Conference, CAiSE 2013, Valencia, Spain, June 17-21, 2013. Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 514–529. Online verfügbar unter https://doi.org/10.1007/978-3-642-38709-8_33.

Schütz, Christoph G.; Schrefl, Michael (2017): Towards Formal Strategy Analysis with Goal Models and Semantic Web Technologies. In: Sergio de Cesare und Ulrich Frank (Hg.): Advances in Conceptual Modeling: ER 2017 Workshops AHA, MoBiD, MREBA, OntoCom, and QMMQ, Valencia, Spain, November 6-9, 2017, Proceedings. Cham: Springer International Publishing, S. 144–153. Online verfügbar unter https://doi.org/10.1007/978-3-319-70625-2_14.

Springer, Matthias; Masuhara, Hidehiko; Hirschfeld, Robert (2016): Classes As Layers: Rewriting Design Patterns with COP: Alternative Implementations of Decorator, Observer, and Visitor. In: Proceedings of the 8th International Workshop on Context-Oriented Programming. New York, NY, USA: ACM (COP'16), S. 21–26. Online verfügbar unter http://doi.acm.org/10.1145/2951965.2951968.

Srdjevic, Zorica; Bajcetic, Ratko; Srdjevic, Bojan (2012): Identifying the Criteria Set for Multicriteria Decision Making Based on SWOT/PESTLE Analysis: A Case Study of Reconstructing A Water Intake Structure. In: *Water Resources Management* 26 (12), S. 3379–3393. DOI: 10.1007/s11269-012-0077-2.

A. Installationshandbuch

In diesem Appendix-Kapitel wird erklärt, welche Schritte notwendig sind, um das PESTEL-Modeler-Werkzeug zu installieren und in Betrieb zunehmen.

Des Weiteren werden folgende fünf Dateien beziehungsweise Artefakte benötigt, um das PESTEL-Modeler-Werkzeug verwenden zu können:

- **Metamodell.mxt**
  Bei dem File Metamodell.mxt handelt es sich um eine Datei vom Typ MXT, welche ein XML-Dokument darstellt. Diese Datei enthält alle Informationen zum Metamodell. Dazu zählen: Graphen, Objekte, Beziehungen, Rollen, Ports, Eigenschaftsfelder, Modell-Einschränkungen, Symbole, Icons, MERL-Generatoren
- **iStarSchema.ttl**
  Bei dem File iStarSchema.ttl handelt es sich um eine Datei vom Typ TTL. Diese Datei enthält das RDF-Schema des iStar-Namespace in der Turtle-Syntax.
- **pestelSchema.ttl**
  Bei dem File pestelSchema.ttl handelt es sich ebenso um eine Datei vom Typ TTL. Diese Datei enthält das RDF-Schema des pestel-Namespace in der Turtle-Syntax.
- **pestel-1.0.0-SNAPSHOT**
  Hier handelt es sich um einen Ordner, in dem das Java-Programm für den RDF-Import, sowie die abhängigen Bibliotheken enthalten sind. Sowohl das Java-Programm als auch die Bibliotheken sind vom Typ JAR.
- **OTanalyse-1.0.0-SNAPSHOT**
  Hier handelt es sich um einen Ordner, in dem das ausführbare Java-Programm für die Chancen-Risiken-Analyse, sowie die abhängigen Bibliotheken enthalten sind. Sowohl das Java-Programm als auch die Bibliotheken sind vom Typ JAR.
- **UUIDGenerator.jar**
  Der UUID-Generator ist vom Dateityp JAR. Es handelt sich also um ein ausführbares Java-Programm. Dieses wird benötigt, um für das abstrakte Verbindungsobjekt bei den dependerOf-Beziehungen eine UUID generieren zu können.

Diese Dateien und Artefakte sind im GitHub-Repository unter `https://github.com/evamair/PESTEListarModeler` bereitgestellt.

In den folgenden Punkten werden übersichtsmäßig die notwendigen Schritte dargestellt, welche später genauer erklärt werden:

1. Erstellung eines Repository
2. Anlegen eines Projektes
3. Import des Metamodells als MXT-Datei
4. Ablegen der Schema-Dateien am richtigen Speicherort
5. Speicherpfade anpassen
6. Ablegen der Java-Programme am richtigen Speicherort

Erstellung eines Repository

Nach dem Starten von MetaEdit+, gelangt man zur Übersicht über sämtliche Repositories. Hier muss nun ein neues Repository unter `File>Create Repositor` erstellt werden. Dazu muss ein Name sowie das Verzeichnis angegeben werden. Das Verzeichnis des Repository wird standardmäßig unter den `Pfad C:\Users\<CurrentUser>\Documents\MetaEdit+ 5.5` gespeichert. Wie in Abbildung 22 ersichtlich, muss ein Username, sowie ein Passwort bei der Erstellung des Repositories angegeben werden. Anschließend öffnet sich das Hauptfenster von MetaEdit+.

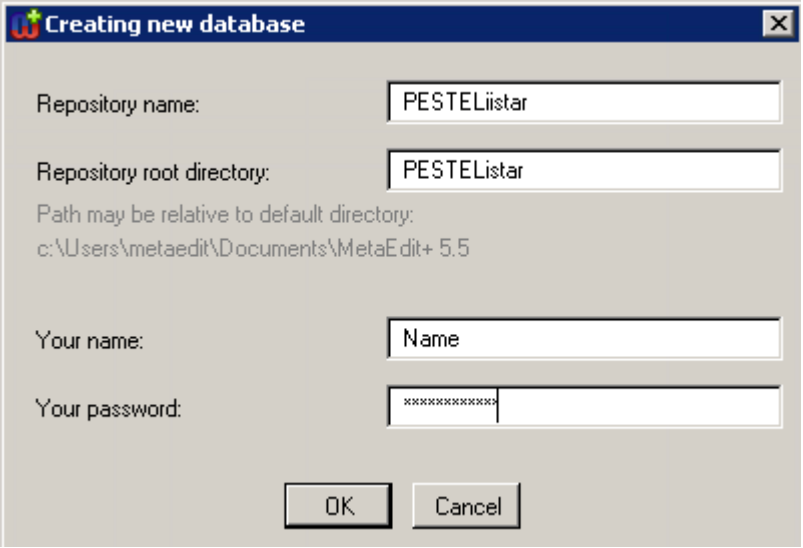


Abbildung 22: Erstellung Repository

Anlegen eines Projekts

Als nächsten Schritt muss ein Projekt angelegt werden. Dazu muss vorerst das Projekt durch Klicken des grünen Häkchens commited werden. Anschließend kann unter `Repository>New Projekt` ein Projekt erstellt werden, für welches ein Name festgelegt werden muss.

Import des Metamodell

Unter `Repository>Import` kann die oben beschriebene MXT-Datei über einen File-Dialog ausgewählt und das Metamodell in das Projekt importiert werden.

Schema-Dateien ablegen

Anschließend muss im Repository des Projekts ein Unterordner mit der Bezeichnung „schema“ angelegt werden. Die beiden oben beschriebenen Schema-Dateien für den iStar- und pestel-Namespace müssen in diesem Unterordner gespeichert werden.
Der Speicherort ist also folgender:

```
C:\Users\<CurrentUser>\Documents\MetaEdit+ 5.5\PESTEListar\schema
```

Speicherpfade anpassen

Damit die MERL-Generatoren funktionieren, müssen im Code sämtliche Speicherpfade an den aktuellen User angepasst werden!

Java-Programme ablegen

Damit das PESTEL-Modeler-Werkzeug funktioniert, werden die drei Java-Programme, die oben aufgelistet sind, benötigt. Das JAR-File für den UUID-Generator muss unter dem Pfad `C:\Users\<CurrentUser>\Documents\MetaEdit+ 5.5` abgelegt werden.
Die beiden anderen oben beschriebenen Java-Programme (für RDF-Import und Chancen-Risiken-Analyse), welche als Ordnerstruktur vorhanden sind, müssen unter dem Pfad `C:\temp` abgelegt werden.