Bachelor Thesis

Konzeption und Implementierung einer Model-to-Model Transformation zur Überführung unterspezifierter Domänenmodelle in Micoservices

An der Fachhochschule Dortmund
im Fachbereich Informatik
Studiengang Software und Systemtechnik
erstellte Bachelor Thesis
zur Erlangung des akademischen Grades
Bachelor of Science of Science

von
Till Paplowski
geb. am 09.03.1995
Matr.-Nr. 7090903

Betreuer:
Prof. Dr. Sabine Sachweh
Florian Rademacher, M. Sc.

Dortmund, 26.08.2018

Kurzfassung

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

1.2 Zielsetzung

1.3 Vorgehensweise

2 Grundlagen

2.1 Modellgetriebene Softwareentwicklung

Model Driven Software Development (MDSD)

2.1.1 Modellierungssprachen

- Unterscheidung zwischen GPL und DSML hier rein

2.1.2 Modelltransformation

Model To Model (M2M) *Query/View/Transformation* (QVT)

2.2 UML-Profil für das Domain-driven Design

2.2.1 Domain-driven Design

2.2.2 Eclipse Papyrus und UML-Profile

Object Constraint Language (OCL)

2.2.3 UML-Profil zu Erstellung von Domänenmodellen

2.3 AjiL Modeling Language

2.3.1 Der Microservice-Architekturstil

Hier rein erklärung von MSAs, aber auch grob Erkenntnisse aus der PA bezüflich der Verbindbarkeit von DSML und MSA

2.3.2 ECore

2.3.3 AjiL

3 Analyse der Anforderungen an die Transformation

In diesem Kapitel werden die Stakeholder an der Transformation (siehe Abschnitt 5.1) und die Problemdomäne (siehe Abschnitt 5.2) betrachtet und verschiedene Arten von Anforderungen (Siehe ABschnitt 5.3) definiert.

3.1 Stakeholder

Die üblichen Stakeholder für eine Transformation zwischen zwei *Domain Specific Modelling Language* (DSML)s lassen sich in drei Kategorien ähnlich [Pol06], welche die Stakeholder für *Domain Specific Language* (DSL)s beschreibt, einteilen:

Software- und System-Engineers: Sind verantwortlich für die Entwicklung und Auswahl der Modellierungssprachen, die transformiert werden. Zudem handelt es sich um Entwickler, die die Transformation weiterentwickeln.

Domänenentwickler: Nutzen *Ajil Modeling Language* (AjiL) zur Entwicklung von Modellen oder Quellcode.

Kunden: Fungieren als Domänenexperten bei dem Entwurf von Domänenmodellen nach *Domain-driven Design* (DDD), die die Problemdomäne des Kunden beschreiben.

3.2 Problemdomäne

Die Problemdomäne, in der die Transformation angewandt wird, ist die Planung und Modellierung von *Microservice Architecture* (MSA)s. Hierbei soll eine Möglichkeit geboten werden in Zusammenarbeit mit Domänenexperten entworfene DDD-*Unified Modeling Language* (UML) konforme Domänenmodelle in entsprechende MSA Modelle nach AjiL zu transformieren.

Es besteht kein Anspruch, dass alle Komponenten des Domänenmodells in ein MSA Modell übertragen werden, da nicht alle Konzepte von DDD Äquivalente in AjiL haben.

3.3 Anforderungen

In der folgenden Sektion werden die Anforderungen an die Modeltransformation zwischen dem formalen DDD-Profil und AjiL definiert. Gemäß ISO/IEC/IEEE 24765 [IEE10] besitzen Anforderungen drei grundlegende Eigenschaften:

- Ein Zustand oder Fähigkeit, die von einem Nutzer benötigt wird, um ein Problem zu lösen oder ein Ziel zu erreichen.
- Ein Zustand oder Fähigkeit, die von einem System oder einer Systemkomponente erfüllt oder besessen werden muss, um einem Vertrag, Standards, Spezifikationen oder anderen formal auferlegten Dokumenten nachzukommen.
- Eine dokumentierte Darstellung des Zustandes oder der Fähigkeit, die in (1) oder (2) beschrieben werden.

Hierbei wird zwischen drei Arten von Anforderungen unterschieden: Funktionale Anforderungen, Qualitätsanforderungen und Randbedingungen. Funktionale Anforderungen an die Transformationen beschreiben die eigentlichen Funktionalitäten der Software. Qualitätsanforderungen beschreiben die qualitiativen Eigenschaften der Software, und Randbedingungen geben Einschränkungen an die Lösung in einem technischen und organisatorischen Rahmen vor. [Poh08] Die Verwendung der Begriffe MUSS und SOLLTE ist angelehnt an ihre Verwendung in der 'MASTeR'-Schablonenreihe um die Verbindlichkeit von Anforderungen auszudrücken. Dementsprechend formulieren MUSS-Anfoderungen Funktionalitäten dar die in jedem Fall umgesetzt werden muss. SOLL-Anforderungen

stellen durch Stakeholder gewünschte Funktionalitäten dar, welche aber nicht zwingend umgesetzt werden müssen. [Rup]

Bei der Erstellung der Anforderungen wird davon ausgegangen, dass der Code, der die Transformation ausführt, in eine ausführbares Software integriert ist. Da in dieser Arbeit keine konkreten Technologien zur Umsetzung bestimmt werden, kann es sich bei dem ausführbaren Software je nach Mächtigkeit der Transformationssprache um die selbe Software handeln, die auch die Transformation beinhaltet, oder um eine Software, welche eine seperate Transformationsdatei ausführt.

3.3.1 Funktionale Anforderungen

/**F.01**/ Die Software MUSS aus einem Modell welches unter Anwendung des **DDD!** (**DDD!**)-UML-Profils erstellt und validiert wurde, eine AjiL konforme Datei erzeugen.

/**F.01.01**/ Die Software SOLL den Typen der Output Datei von .uml in eine .ajimlt Datei ändern.

/**F.02**/ Die Software MUSS die einzelnen DDD Komponenten gemäß der Tabelle 4.1 in entsprechende AjiL Komponenten transformieren.

/**F.03**/ Die Software SOLL durch Nutzer Eingaben Ergebnisse aus den in den Unterabschnitten 4.2.4 und 4.2.6 beschriebenen Verfahren validieren.

/**F.03**/ Die Software SOLL einen Modus ohne Nutzer Eingaben bieten.

3.3.2 Qualitätsanforderungen

/**Q.01**/ Die Software SOLL auf einem herkömmlichen Bürorechner (2 GHz, 2 GB Arbeitsspeicher) ausführbar sein.

/**Q.02**/ Die Transformation SOLL in einer einzelnen Transformationsdatei beschrieben werden.

/**Q.03**/ Die Software MUSS nur notwendige Interfaces und Abilities an Microservices modellieren

/**Q.04**/ Die Software SOLL durch Entwickler an Änderungen im UML-Profil oder AjiL anpassbar sein.

3.3.3 Randbedingungen

/**R.01**/ Die Software SOLL aus der Eclipse Entwicklungsumgebung heraus ausführbar sein.

4 Konzept für die Model-to-Model-Transformation

Im folgenden Kapitel wird konkret beschrieben und grafisch dargestellt wie Elemente des DDD-UML-Profils in Elemente von AjiL übersetzt werden. Hierzu werden die Elemente in semantisch zusammenhängende Gruppen eingeteilt.

4.1 Dezidierung von Microservices

In [New15] bezieht sich Newman direkt auf Evans DDD. Hiernach sollen Microservices exakt Bounded Contexts entsprechen. Für diese These gibt es zahlreiche Argumente.

Bei der Konzeptionierung von MSAs ist eine der zentralen Fragestellungen die nach der Größe einzelner Microservices. Als Anhaltspunkt wird hierzu das von Robert C. Martin definierte Single Responsibility Prinzip angeführt: [New15] *„Gather together the things that change for the same reasons. Separate those things that change for different reasons.“* [Mar03] Dies deckt sich mit Evans Definition eines Microservices als ein Gültigkeitsbereich, in dem lediglich eine gewisse Fachlogik umgesetzt wird und von anderen Logiken abgegrenzt wird [Eva03]. - Diese Philosophie entspricht dem Gedanken von MSAs, da hier ein Team für einen Microservice zuständig ist, der keine gemeinsame Implementierung mit anderen Services teilt. [Ric16]. Ein weiterer Grund für die Einteilung anhand von Bounded Contexts ist, dass Bounded Contexts durch ihre Eigenschaft nur ein innerhalb des Contexts konsistentes Modell zu verlangen, einem Team ermöglichen ohne genaue Kenntnis der Objekte und Vorgänge innerhalb anderer Bounded Contexts zu arbeiten. DDD [Eva03]. Auch in ihrer Funktionsweise weisen Bounded Contexts Ähnlichkeiten zu Microservices auf, da sie geregelte Zugriffspunkte auf die in ihnen enthaltenen Entitäten bieten [Eva03], was der Aufgabe von Interfaces von Microservices ähnelt. Aus diesen

Gründen sollen bei der Transformation Bounded Contexts in einzelne Microservices überführt werden.

Es sollen bei der Transformation eines `Package` Elements mit dem Stereotyp `Bounded Context` folgende Maßnahmen ergriffen werden (vgl. Abbildung 4.1): **1.)** Erzeugung eines `FunctionalServiceT` Elements mit dem Namen des `Bounded Context` und dem hinzugefügten Suffix *Service*. **2.)** Erzeugung eines `DataModelT` Elements mit dem Namen des `Bounded Context`, welches **3.** dem `FunctionalServiceT` Element als `domain` zugeordnet wird. Diese Zuordnung ist in AjiL stets 1 zu 1. **4.)** Evans beschreibt auch `Modules` als Konzept, diese haben jedoch die gleiche Funktionsweise wie `Packages` in UML, außer, dass Elemente in einem `Module` keine Beziehungen zu Elementen in anderen `Modules` haben sollen [Eva03]. Im DDD-Profil existiert für `Package` Elemente der Stereotyp `Module`, jedoch ist keine Beschränkung bezüglich der Beziehungen zwischen `Modules` implementiert. In AjiL existiert kein äquivalentes Konzept zu `Packages` oder `Modules`. Unter der Annahme, dass die Einschränkungen, die Evans für `Modules` formuliert in der UML Modellierung beachtet wird, verhalten sie sich wie `DataModels` in AjiL, und eine solche Transformation wäre sinnvoll. Da jedoch zum Zeitpunkt der Erstellung dieser Arbeit lediglich eine 1 zu 1 Beziehung zwischen `Service` und `DataModel` möglich ist, werden zunächst `Packages` mit dem `Module` Stereotyp nicht transformiert und lediglich eine entsprechende Log Ausgabe erzeugt.

4.2 Entities, Enumerations, Value Objects und SharedModels

Vergleiche Abbildung 4.2.: **1.)** Sowohl `ValueObjects`, `Enumerations` als auch `Entities` werden in `EntityT` Elemente aus AjiL transformiert. **2.)** Attribute von `Value Objects` und `Entities` werden zu dem entsprechenden primitiven Datentyp in AjiL. Hierbei ist zu beachten, dass DDD Modelle bewusst unterspezifiziert werden können (vgl. Abschnitt 2.2.1). Da es in AjiL nicht möglich ist ein Attribut ohne Datentypen zu erzeugen, wird in diesem Fall ein `StringT` Element erzeugt. Darüber hinaus ist es durch Zusammenarbeit mit den Entwicklern von AjiL auch in AjiL möglich komplexe Datentypen wie beispielsweise andere `EntityT` Elemente als Attribut zu verwenden. In diesem Fall soll ein Attribut einer UML `Entity` auch die entsprechende AjiL `EntityT` ein Attribut vom

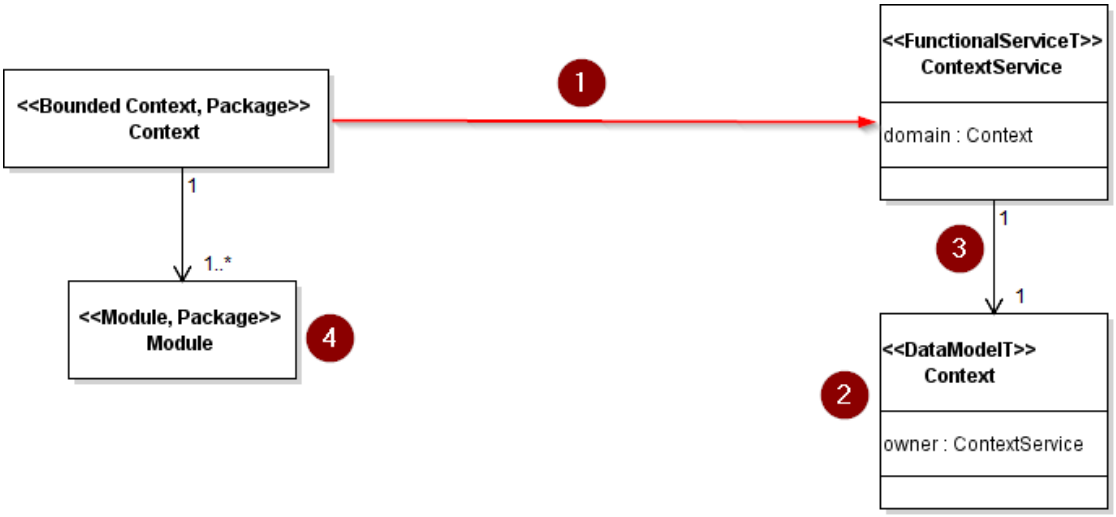

Abbildung 4.1: Transformation von Bounded Contexts

Typ `EntityT` mit dem gleichen Namen wie die referenzierte `Entity` erhalten. Dieses Attribut enthält jedoch aufgrund von Limitierung in AjiL keinen echten Zeiger auf das referenzierte `EntityT` Element.

3.) Während in DDD Operationen an `Entities` nicht ausgeschlossen werden, und im Fall von `Value Objects` sogar Teil des Konzepts sind, verfügt `EntityT` in AjiL über keine Operationen. Hier tritt ein Informationsverlust auf. Aus dem gleichen Grund werden `Specs`, welche sich nur durch ihre Validierungsoperationen definieren, nicht in AjiL überführt (**4**), sondern in der Logging-Datei der Transformation ausgegeben. **5.)** Die `EnumerationLiterals` von `Enumerations` werden standardmäßig in `StringT` Elemente umgewandelt. **6.)** Da in AjiL `Entities` einem `DataModel` zugeordnet sein müssen, ist es nicht möglich Shared Models außerhalb eines `Bounded Context` zu modellieren. Aus diesem Grund soll eine aus einem Shared Model erzeugte `EntityT` dem gleichen `DataModelT` wie der `supplier` ihrer `Abstraction` zugeordnet werden. Zudem soll der `supplier` als `parent` der `EntityT` gesetzt werden.

4.2.1 Abstrakte Klassen und Generalisierungen

In UML wird die Beziehung zwischen abstrakten Klassen und den jeweiligen spezifizierenden Klassen, die von der abstrakten Klasse erben, durch `Generalizations` modelliert [Obj15]. Abstrakte Klassen sollen nur in `EntityT` Elemente transformiert werden, wenn sie über den Stereotyp `Entity` verfügen. Die spezifizierenden Klassen müssen entweder selbst den Stereotyp `Entity` besitzen, oder die generalisierende Klasse muss ihn besitzen, um zu `EntityT` Elementen transformiert zu werden.

4.2.2 Assoziationen

Von den in UML modellierten Beziehungen, werden nur die Beziehungen die zwischen den in diesem Unterkapitel betrachteten Elementen als gesonderte `AssociationT` Elemente in AjiL erzeugt, da in AjiL nur `EntityT` Elemente Assoziationen besitzen können. AjiL ist in der Repräsentation von Assoziationen auf `ONE TO ONE` und `ONE TO MANY` Beziehungen beschränkt. In UML hingegen können zusätzlich `MANY TO ONE` und `MANY TO MANY` Beziehungen modelliert werden. Um dieses Problem zu umgehen, werden `MANY TO MANY` Assoziationen in jeweils eine `ONE TO MANY` Assoziation in beide Richtungen zwischen den Entitäten aufgeteilt. Für `MANY TO ONE` Beziehungen gibt es keine Möglichkeit zur Transformation, die auf die ursprngliche Semantik schließen lässt.

4.3 Service und Repository Objekte als Schnittstellen

`Repository` und `Service` Objekte verwalten und manipulieren in DDD Entitäten innerhalb eines Bounded Contexts. In der Regel sind Zugriffe auf Entitäten aus anderen `Bounded Contexts` heraus nur über Objekte dieser Art möglich. [Eva03] Wenn ein `Bounded Context` einen Microservice darstellt, können dementsprechend `Repositories` und `Services` als ServiceInterfaces verwendet werden und Aufschluss über die Abhängigkeiten zwischen den verschiedenen Microservices geben. Damit ein Microservice nur notwendige Funktionen als Interfaces bereitstellt und keine geheimen Operationen freigegeben werden, sollen in der Transformation lediglich `Repository` und `Service` Objekte, die durch UML `Dependencies` aus anderen `Bounded Contexts` referenziert werden, mit ihren Operationen in ein Interface umgewandelt werden [RSZ17]. Der Abbildung 4.2

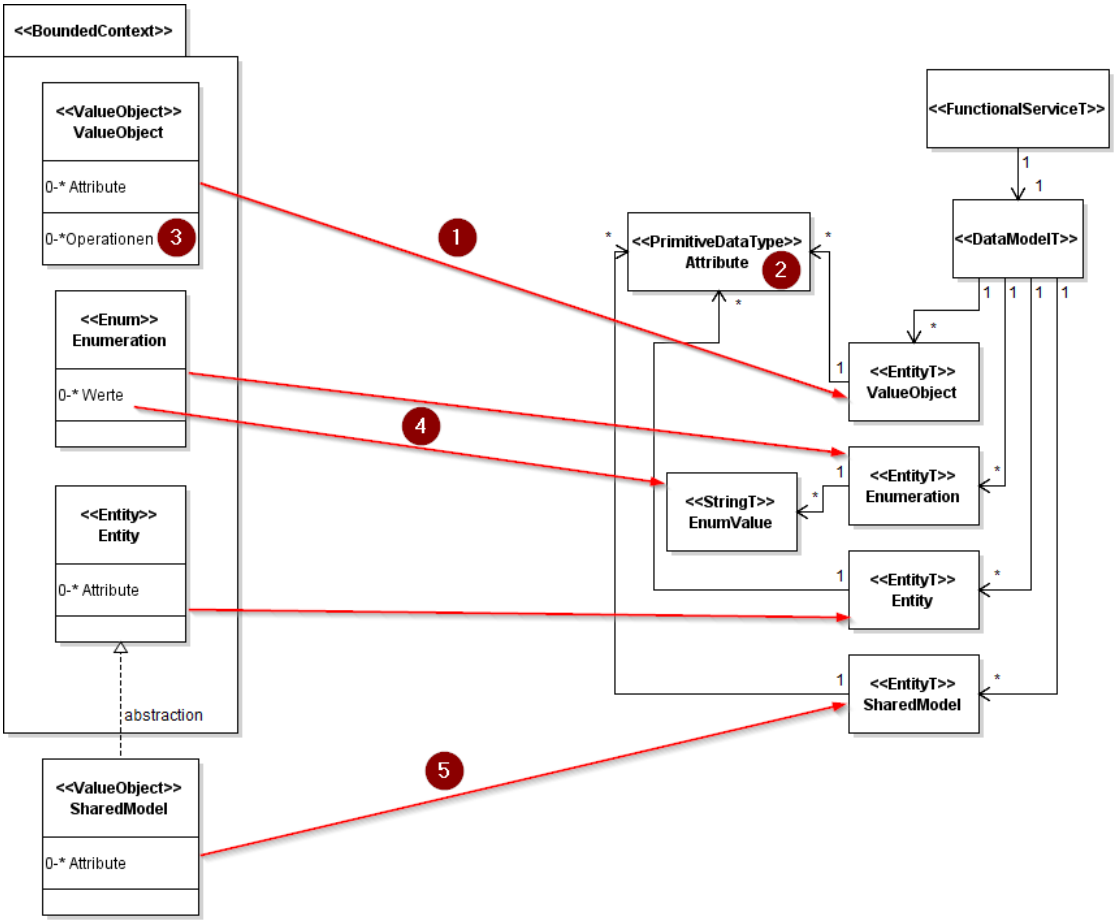

Abbildung 4.2: Transformation von ValueObjects, Specs, Enumerations und Entities

Ability	*Schlagwörter*
CreateT	*create*
ReadT	*find*, *read*, *get*
UpdateT	*update*
DeleteT	*delete*, *remove*
CustomT	keine

Tabelle 4.1: Schlagwörter zur Bestimmung des Ability Typen

sind die verschiedenen Aspekte der Transformation von `Repositories` und `Services` zu entnehmen.

1.) und 3.) Für Elemente vom UML Typ `Class` mit den Stereotypen `Service` oder `Repository` die als `supplier` einer `Dependency` fungieren, werden in AjiL `ServiceInterfaceT` Elemente mit dem alten Namen und *Interface* als Suffix erzeugt und diese an dem `FunctionalServiceT`, der dem ursprünglichen `Bounded Context` entspricht als `providedInterfaces` hinzugefügt. Bei beiden Stereotypen werden die Operationen in Abilities umgewandelt. **2.)** Bei Operationen von `Services` der Anfang des Methodennamens untersucht (Die Schlagwörter nach denen kategorisiert wird, sind in der Tabelle 4.1 zu finden), um festzustellen um welchen Typ von `Ability` es sich in AjiL handeln wird. Es wird nur auf den Anfang des Methodennamens betrachtet, um die Fehleinordnung von Operationen, welche zum Beispiel `deleteReader` heißen vorzubeugen. Bei den mit **2** und **4** markierten Transformationen ist ein weiterer Entscheidungspunkt zu erkennen. Bei Operationen, bei denen nicht durch `type` ein Rückgabetyp modelliert ist (**2**), wird als `subject` der `Ability` das transformierte Ziel der ersten `Association` des `Services` oder `Repositories` verwendet. Andernfalls wird das transformierte Rückgabe Objekt zum `subject`. **5.)** Der Microservice aus dem `Bounded Context`, der in UML die Abhängigkeiten zu den Objekten besitzt, speichert diese als `serviceDependencies`. Da abgesehen von Interfaces in AjiL kein Konzept existiert, welches `Services` oder `Repositories` innerhalb eines `DataModels` entspricht, entfällt die Transformation von Elementen, zu denen keine Abhängigkeiten aus anderen `Bounded Contexts` besteht, und es wird lediglich ein Vermerk auf die verlorenen Informationen in der Logging-Datei der Transformation erzeugt.

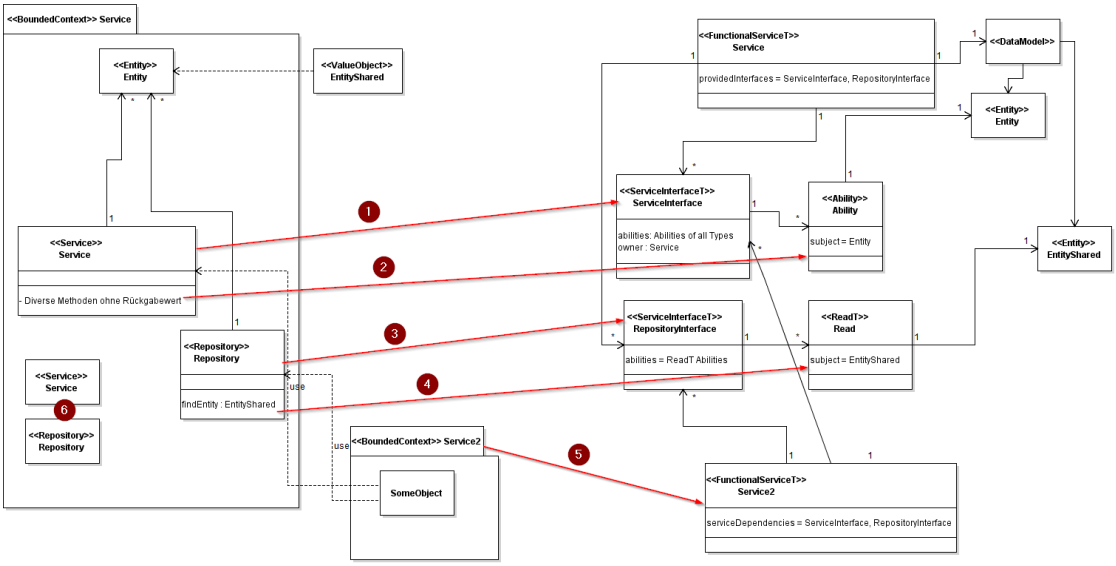

Abbildung 4.3: Transformation von Repositories und Services

4.4 Umgang mit Aggregaten

Wenn ein `Bounded Context`, der Aggregate beinhaltet in einen `FunctionalServiceT` transformiert wird, werden in seinem `DataModelT` bereits `EntityT` Elemente, welche die abstrakten Klassen AbstractAggregateRoot und AbstractAggregatePart repräsentieren sollen (**3 und 4**). Hierbei ist zu vermerken, dass AjiL keine abstrakten Klassen unterstützt, jedoch abstrakte Klassen in der späteren Implementierung der MSA nützlich sind, um von Evans formulierte Fähigkeiten von Aggregaten, wie zum Beispiel alle `AggregateParts` einer `AggregateRoot`erhalten zu können, allgemeingültig zu realisieren. **1.) und 2.)** `Entities`, die zusätzlich die Stereotypen `AggregateRoot` oder `AggregatePart` besitzen, werden in `EntityT` Elemente transformiert und um sie als `AggregateRoot` oder `AggregatePart` zu kennzeichnen, wird der Wert für `parent` auf die jeweilige abstrakte Klasse gesetzt, wodurch eine `extends` Beziehung entsteht.

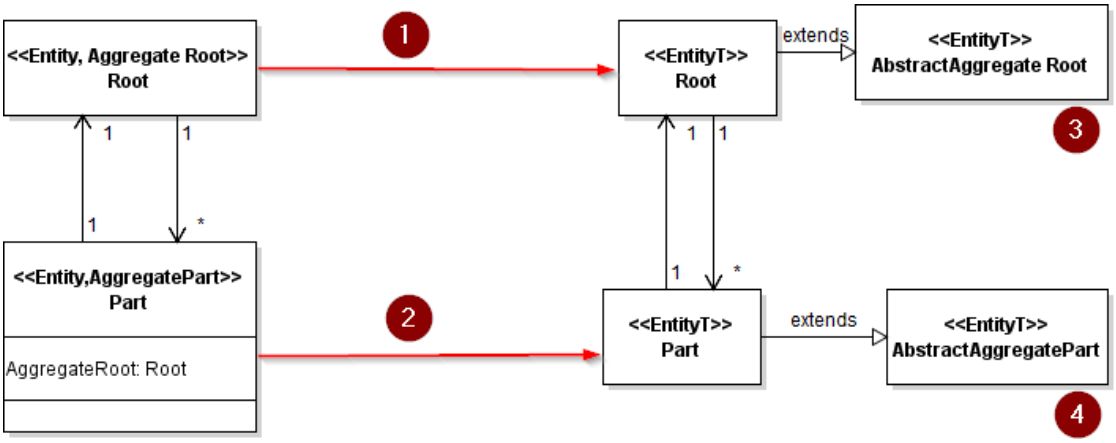

Abbildung 4.4: Transformation von Aggregaten

4.5 Sonstige Transformationen

In der Baumstruktur einer *.ajimlt* Datei stellt das `SystemT` Element die Wurzel dar. Da in UML dies dem Typen `Model` entspricht, wird ein `Model` in ein `SystemT` transformiert. Zudem werden die `Abstractions` von implementierenden `Entities` zu abstrakten `Entities` durch das Setzen des `parent` Attributs an der `Entity` in AjiL auf das Elternelement ausgedrückt. Da in AjiL keine entsprechenden Konzepte für die Stereotypen `Closure`, `SideEffectFree`,`definesIdentity` oder `validatesSpec` existieren, werden diese Stereotypen nicht nach gesonderten Regeln transformiert, sondern nach ihrem UML-Typ ausgewertet.

4.6 Tabellarischer Überblick über Regeln

Der Tabelle 4.2 sind die umzusetzenden Transformationsregeln anhand von UML-Typ und angewandtem UML-Stereotyp des Quellelements zum AjiL Zielelements mit Anmerkungen zu entnehmen.

UML-Typ	*UML-Stereotyp*	*AjiL-Elemente*
Model	-	SystemT
Package	Bounded Context	FunctionalServiceT + DataModelT
Package	Module	-
Class	Entity	EntityT
Class	Entity, AggregateRoot	EntityT (parent = AbstractAggregateRoot)
Class	Entity, AggregatePart	EntityT (parent = AbstractAggregatePart)
Class	Value Object	EntityT
Class	Spec	-
Class	Repository oder Service	ServiceInterfaceT (wenn Dependencies existieren)
Property	*irrelevant*	PrimitiveDataType (Default StringT)
Operation	*irrelevant*	Ability (nur Repositories, Services; Default CustomT)
Abstraction	-	supplier als parent an EntityT
Association	-	Association (Nur Entities, Enumerations, ValueObjects)
Dependency	-	ServiceDependency (wenn zwischen Bounded Contexts)

Tabelle 4.2: Transformationen Im Überblick

5 Implementierung des Konzepts

5.1 Auswahl eines geeigneten Transformationstools

5.1.1 Kriterien

In [Var18] werden verschiedene MDSD Tools zusammengetragen und kategorisiert. Hierbei werden verschiedene Kriterien zur Auswahl von MDSD Tools formuliert. Im Folgenden werden die Kriterien vorgestellt und beschrieben, inwieweit diese für das MDSD Tool, welches in dieser Arbeit zum Einsatz kommen soll, erfüllt werden sollen.

M2M oder M2T? So wie Transformationen an sich, lassen sich Transformationssprachen darin unterscheiden ob sie ein Model in ein anderes Model umwandeln, oder in Text. Da von einem UML Modell in ein AjiL Modell transformiert werden soll, muss die Sprache für M2M Transformationen geeignet sein.

Transformationsansatz Im Bereich der M2M Transformationen wird hinsichtlich der für Transformationen verwendeten Sprachkonstrukte und Mechanismen zwischen relational, imperativ, graphbasiert oder hybriden Ansätzen unterschieden. Da der Autor dieser Arbeit hauptsächlich Erfahrung mit imperativen Ansätzen hat, sollte die gewählte Sprache imperativ oder hybrid mit der Möglichkeit komplexe Transformationsoperationen imperativ zu beschreiben sein.

Generelle Bedingungen In dieser Kategorie befinden sich Kriterien, die mit nicht technischen Aspekten der verwendeten Technologie zusammenhängen, wie Lizensierung

und Support. Da die beiden Ansätze, die transformiert werden sollen noch in der Entwicklung befinden und eine Anforderung an die Transformation eine zukunftssichere Erweiterbarkeit darstellt, sollte die Technologie weiterhin vom Hersteller unterstützt werden. Zusätzlich kann bei Schwierigkeiten bezüglich der Entwicklung auf vom Hersteller zur Verfügung gestellte Dokumentation und Unterstützung zurückgegriffen werden. Zudem sollte es sich bei der Technologie um ein Open Source Produkt ohne Lizensierungskosten handeln, da keine monetären Ressourcen bestehen und um die Portabilität der Transformation einfacher zu gestalten.

Modellierungsaspekte Diese Kategorie umfasst sämtliche Eigenschaften bezüglich der Unterstützung verschiedener Aspekte der Modellierung, wie Kompatibilität mit bestimmten Modellierungssprachen, Metametamodellen oder bestehenden Standards zum Informationsaustausch auf Modellebene. Das verwendete Tool muss UML (*Meta Object Facility* (MOF))und Ecore unterstützen, um die den zu transformierenden Modellen zugrunde liegenden Metamodelle auswerten und anwenden zu können. Hieraus ergibt sich, dass das MOF Metametamodell unterstützt werden sollte.

Transformationsaspekte Diese Facette umfasst alle Anforderungen an die durchgeführten Transformationen, bezüglich der Art der Syntax, Kardinalität etc.. Bezüglich der Syntax ist eine textuelle Darstellung ausreichend, da die implementierte Transformation nur von Entwicklern erweitert werden soll. Da ein Modell in ein anderes überführt wird, ist eine Kardinalität von eins zu eins notwendig und nicht die Erzeugung oder Eingabe von mehreren verschiedenen Modellen. Der Output des Modells kann konservativ erfolgen, das heißt, dass ein neues Modell erzeugt wird und kein bestehendes verändert, da eine neue Datei unter einem anderen Metamodell erstellt werden muss. Regelapplikationskontrolle muss unterstützt werden, da gleiche Elemente unter verschiedenen Bedingungen unterschiedlich transformiert werden, wie zum Beispiel beim unterschiedlichen Umgang mit `ValueObjects` und Shared Models. Das Tool muss exogene Transformationen unterstützen, da beiden Modellen verschiedene Metamodelle zugrunde liegen. Die Fähigkeit zur Durchführung von Unidirektionalen Transformationen genügt.

User Experience Aspekte An dieser Stelle wird Nutzerfreundlichkeit des Tools betrachtet, diese Aspekte stellen für die Evaluation an dieser Stelle eine untergeordnete

Rolle. Für eine effektive Entwicklung sollten jedoch bestimmte Faktoren gegeben sein: Das Tool sollte über einen Syntax und Semantik sensitiven Editor mit Error Warnungen, Syntax Higlighting, Autovervollständigung etc. verfügen. Darüber hinaus sollte durch komplexere Transformationsbedingungen und den Hintegrund des Autors dieser Arbeit der Programmierstil Programmiersprachenähnlich sein.

Kollaborationsaspekte Diese Facette beschreibt wie gut sich das Tool in Projekte einbinden und für diese erweitern lässt. Um Transformation und Entwicklungsfortschritte zu sichern, sollten die vom Tools verwendeten Dateien kompatibel mit Versionskontrolltools sein.

Laufzeitsaspekte Zuletzt wird betrachtet unter welchen Umständen das Tool ausgeführt werden soll. Enweder soll die implementierte Transformation als standalone Applikation ohne viel Konfigurationsaufwand oder über Eclipse, da sowohl UML Modelle, als auch AjiL in Eclipse Umgebung integriert sind, ausführbar sein. Zudem sollte sowohl die Entwicklung, als auch die Ausführung der Transformation betriebssystemunabhängig möglich sein.

5.1.2 Entscheidung

[Var18] stellt zusätzlich zu der Veröffentlichung eine Website[1] zur Verfügung, auf der der Klassifizierung der Tools entsprechendete Filter verwendet werden können. Diese wurde verwendet, um eine Vorauswahl an Tools zu treffen. Viele Tools entfallen, weil sie durch einen Fokus auf DSMLs lediglich Ecore verwenden. Ebenfalls sind viele Tools im Bereich MDSD zum Zeitpunkt der Erstellung dieser Thesis nicht mehr unterstützt.

Für die Implementierung der Transformation soll die *Atlas Transformation Language* (ATL) verwendet werden. Es handelt sich um eine Transformationssprache, welche auf M2M Transformationen ausgelegt ist und eine Komponente der M2M Implementierung des *Object Management group* (OMG) QVT Standards im *Eclipse Modeling Project* (EMP) darstellt, womit sie unter die kostenfreie Eclipse Lizenz fällt. Dadurch ist die Transformation mit der Eclipse *Integrated Development Environment* (IDE) in der

[1] www.mdetools.com

selben Umgebung programmier- und ausführbar wie die beiden zu transformierenden Tools. Mit der Arbeit in der Eclipse IDE gehen alle Eigenschaften und Vorteile bezüglich der Laufzeit und Benutzerfreundlichkeit des Editors, die Eclipse bietet mit der Verwendung von ATL einher. Darüber hinaus kann durch die Verwendung der vom *Eclipse Modeling Framework* (EMF) zur Verfügung gestellten *EMF Transformation Virtual Machine* (EMFTVM) theoretisch eine IDE unabhängige Ausführung implementiert werden. ATL verfügt über die Möglichkeit exogene M2M Transformationen durchzuführen, wobei nicht nur Metamodelle des EMF wie Ecore verwendet werden können, deonrn auch UML. Der Transformationsansatz ist hierbei hybrid, wobei Elemente grundsätzlich nach deklarativ beschriebenen Regeln transformiert werden, jedoch auch imperativer Code verwendet werden kann, um komplexe Transformationsbedingungen zu erfüllen. [Waa]

Für den imperativen Code wird ähnlich dem UML-Profils für DDD und der Validierung in AjiL Ausdrücke der Sprache OCL verwendet, wodurch Entwickler mit Expertise in diesem Bereich eine Grundlage für die Arbeit mit ATL besitzen. Um die Einarbeitung in die Sprache zu erleichtern, existieren zu ATL zahlreiche Fallbeispiele [2] und ein reges Subforum im Eclipse Forum [3].

5.2 Atlas Transformation Language

5.2.1 Funktionsweise

Konfiguration und Funktionsweise der Transformation ist demnachin der Abbildung 5.1 gemäß [A.L06] zu erkennen. Hier folgt noch eine ausführlichere Beschreibung.

```
1 module ddd2ajimlt;
2 create OUT : AJI from IN : DDD;
```

Quellcode 5.1: Kopf des ATL Moduls

[2] www.eclipse.org/atl/atlTransformations
[3] www.eclipse.org/forums/index.php/f/241

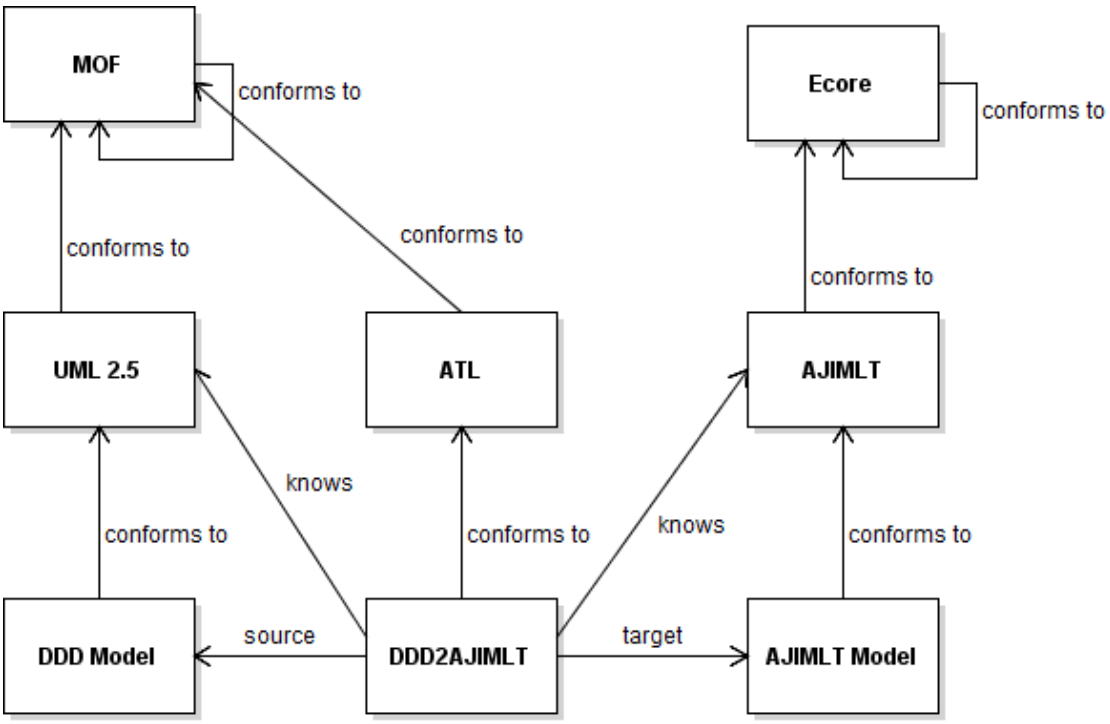

Abbildung 5.1: Transformationsmodellübersicht

5.2.2 Aufbau von Regeln

In der Implementierung werden die von ATL verwendeten **Matched Rules** verwendet. Sie erzeugen für bestimmte Eingabe Elemente eine bestimmte Menge an Ausgabe Elementen.

Die Auswahl der durch eine Regel zu transformierenden Elemente geschieht über das **Source Pattern** jeder Regel. Dieser Teil wird durch ein `from` deklariert und spezifiziert zunächst einen Elementtyp im Eingabemodell und den Variablennamen für das Element, welcher in der Regel verwendet wird um das Element zu referenzieren. Für jedes Element dieses Typen werden anschließend mithilfe von Aussagenlogik verschiedene Kriterien geprüft, die das Element erfüllen muss. Zu beachten ist hier, dass bei Ausdrücken in ATL generell der komplette Ausdruck in der Klammer ausgewertet wird, wodurch die Gefahr von Exceptions (z.B. Null-Pointern) bei der Auswertung der Eigenschaften eines Elements bedacht werden muss.

Wenn das Element im Eingabemodell die Bedingungen der Regel erfüllt, wird der **Target Pattern** Teil der Regel ausgeführt (durch die Verwendung von `to` eingeleitet), in welchem die Elemente des Ausgabemodells, welche durch die Regel erzeugt werden beschrieben werden. Hierbei wird zunächst ähnlich der Syntax im Source Pattern ein Variablenname und ein Elementtyp festgelegt. In den darauf folgenden Klammern können die Eigenschaften des zu erzeugenden Elements gesetzt werden, wobei auch auf Eigenschaften des Eingabeelements ausgewertet werden können. Darüber hinaus kann eine Regel über einen Teil zur Deklaration von weiteren im Target Pattern verwendeten Variablen verfügen, dieser wird über `using` eingeleitet, und imperativ implementiert. [Waa]

5.2.3 Aufbau von Helpern

ATL bietet Entwicklern durch die Definition sogenannter **Helper** imperativen Code zu schreiben, der durch die Funktionalität bestimmter Datentypen erweitern kann. Somit sind Helper an jeder Stelle im ATL Modul anwendbar und können bei der Validierung im Source oder der Beschaffung der gewünschten Werte im Target Pattern verwendet werden, obwohl innerhalb dieser selbst kein imperativer Code verwendet wird. Durch

`Context context_type` kann angegeben werden, von welchem Datentyp der Helper angewendet werden kann. Das aufrufende Objekt selbst kann anschließend über `self` ausgewertet werden. Wird dieser Wert nicht angegeben, ist der Helper dem globalen Kontext des ATL Moduls zugeordnet, und wird durch das Objekt `thisModule`, welches an jedem Punkt des Moduls verfügbar ist, aufgerufen. Darüber hinaus können neben dem Namen beliebig viele durch Komma separierte Parameter angegeben werden und ein Return Type angegeben werden. Der Körper `exp` eines Helpers besteht aus einem OCL Ausdruck, welcher den Rückgabewert des Helpers liefert. [Waa]

```
1 helper[context context_type]? def :helper_name(parameters) :return_type =exp;
```

Quellcode 5.2: Schema zur Deklaration von Helpern [Waa]

5.3 Implementierung der Regeln

5.3.1 Traversierung des Eingabemodells

Da bei der Konzeptionierung der Transformation zwischen den Modellen komplexe Zusammenhänge und Informationen betrachtet werden, welche an in der Baumstruktur des UML Modells teils nicht beieinander liegenden Stellen gepflegt sind (vgl. Abschnitt X.X) ist es notwendig das UML Modell effektiv traversieren zu können. In den Elementen eines DDD-uml Modells, die es zu transformieren gilt, kann die Eigenschaft `namespace` verwendet werden, um Informationen über das Elternelement eines Elements zu erhalten. Da jedoch ein wiederholtes manuelles Aufrufen des `namespace` eines Elements, z.B. von einer `Entity` bis zu dem enthaltenden `Bounded Context` weder leserlichen Code, noch eine dynamische und fehlerresistente Methode der Traversierung darstellt, werden im ATL Modul zwei verschiedene rekursive Helper verwendet, um ein UML Modell "nach obenßum Kern hin zu traversieren: `goUpToElementWithStereotype` für angewendete Stereotypen und `goUpToElementWithType` für UML Typen.

Die Methoden überprüfen ob das aufrufende Element selbst den gesuchten Stereotyp/UML-Typ hat. `GoUpToElementWithStereotype` verwendet an dieser Stelle einen selbst implementierten Helper `hasStereotype`. Falls das Element den Typ hat, wird es zurückgeliefert, ansonsten, falls der `namespace` des Elements nicht leer (`oclUndefined`) ist, soll der selbe Helper für das Elternelement aufgerufen werden. Damit dies möglich ist, sind

die Helper für jeden Elementtyp in UML als Context definiert. Falls der `namespace` des Elements leer ist, bedeutet dies, dass ohne Ergebnis bis zum Kern des UML Modells navigiert wurde In diesem Fall soll OclUndefined zurückgeliefert werden.

```
1  helper context UML!Element def: goUpToElementWithStereotype (stereotype :
       String) : DDD!Element =
2    if(self.hasStereotype(stereotype))
3      then
4        self
5      else
6        if(not self.namespace.oclIsUndefined())
7          then
8            self.namespace.goUpToElementWithStereotype(stereotype)
9          else
10           OclUndefined
11         endif
12     endif;
```

Quellcode 5.3: Helper zur rekursiven Suche nach Elternelementen

Zusätzlich sind besondere Helper für das effektive Sammeln von Elementen in `Bounded Contexts` notwendig, denn obwohl über die Eigenschaft `packagedElement` von `Packages` sämtliche vom `Package` beinhalteten Elemente ausgewertet werden können, kann es sein, dass innerhalb eines `Bounded Context` Elemente durch verschiedene `Modules` organisiert sind. Aus diesem Grund werden im Modul für das Sammeln von Elementen mit bestimmten Stereotypen oder UML-Typen durch mehrere `Packages` rekursive Helper zur Verfügung gestellt. Hierbei wird über `packagedElement` des aufrufenden Elements iteriert, und falls das beinhaltete Element den gesuchten Typ besitzt wird es einer `Collection` hinzugefügt. Falls es sich um ein weiteres `Package` handelt wird von dem Element aus der Helper erneut aufgerufen und die Ergebnis `Collection` dieses Aufrufs mit der bestehenden vereint.

```
1  helper context DDD!Element def: getPackagedElementsByType(type : String) :
       Sequence(DDD!Class) =
2      self.packagedElement ->
3        iterate(iter; col: Sequence(DDD!Element) = Sequence{}|
4      if(iter.oclIsTypeOf(type))
5        then
6          col.append(iter)
7      else
8        if(iter.oclIsTypeOf(DDD!Package))
```

```
9         then
10        col.union(iter.getPackagedElementsByType(type))
11      else
12        col
13      endif
14    endif);
```

Quellcode 5.4: Helper zur rekursiven Suche nach Kindelementen

5.3.2 Nutzung von Traceability Links

Bei der Transformation eines Elements erstellt ATL Traceability Links, um das Element aus dem Eingabemodell mit den im Target Pattern erzeugten Elementen des Ausgabemodells zu verknüpfen. Diese Links können verwendet werden, um Verbindungen zwischen Elementen im Eingabemodell bei der Transformation beizubehalten. Der Helper `getPackagedElementsByStereoType` liefert sämtliche UML Elemente, die das Eingabeelements. Wenn zum Beispiel bei dem aus einem `BoundedContext` Element in AjiL erzeugten `DataModelT` die Eigenschaft `entities` mit dem Ergebnis des Helpers befüllt wird (vgl. Quellcode 5.4, Zeile 16), kann ATL für jede UML `Entity` bestimmen, welches Element durch eine Regel (z.B Quellcode 5.5), die die `Entity` im Source Pattern akzeptiert hat erzeugt wurde und diese einsetzen.

```
1  rule BoundedContext2MicroServiceAndDataModelWithoutAggregate {
2    from
3      p : DDD!Package (p.oclIsTypeOf(DDD!Package) and p.hasStereotype('
           BoundedContext') and p.getPackagedElementsByStereoType('AggregateRoot
           ').size() = 0)
4    to
5
6      fs: AJI!FunctionalServiceT (
7        name <- p.name + 'Service',
8        domain <- dm,
9        providedInterfaces <- p.getPackagedElementsByStereoType('Repository'),
10       providedInterfaces <- p.getPackagedElementsByStereoType('Service'),
11       serviceDependencies <- p.getDependenciesToOtherContexts()
12     ),
13
14     dm: AJI!DataModelT (
15       name <- p.name + 'Model',
```

```
16      entities <- p.getPackagedElementsByStereoType('Entity'),
17      entities <- p.getPackagedElementsByStereoType('ValueObject'),
18      entities <- p.getPackagedElementsByType(DDD!Enumeration)
19    )
20  }
```

Quellcode 5.5: Regel zur Erzeugung eines Microservices

Wenn aus einem Eingabeelement durch eine Regel mehrere Ausgabeelemente erzeugt wurden, kann der Helper `resolveTemp` genutzt werden, um anhand des Variablennamens zu bestimmen, welcher Traceability Link verwendet werden soll (vgl. Quellcode 5.5, Zeile 11). Würde ein Element von mehreren Regeln akzeptiert werden und dadurch verschiedene Links entstehen, kann ebenfalls `resolveTemp` verwendet werden, mit dem Namen der Regel als Parameter. Während `Entity`, `ValueObject` und `Enumeration` Elemente durch Traceability Links dem entsprechenden `DataModelT` hinzugefügt werden, wird bei Klassen, die abstrakte Klassen spezifizieren das DataModel schon bei der Erzeugung der `EntityT` auf das DataModel der generalisierenden Klasse gesetzt. Dies wird an dieser Stelle implementiert, da der Filterungsaufwand vom `BoundedContext` aus größer wäre.

```
1  rule Entity2Entity_NoAggregate{
2    from
3      old : DDD!Class ((old.hasStereotype('Entity') or old.
           generalizationHasStereotype('Entity'))
4        and not (old.hasStereotype('AggregateRoot') or old.hasStereotype('
             AggregatePart')))
5    to
6      new : AJI!EntityT(
7          name <- old.name,
8          attributes <- old.attribute,
9          relations <- old.getAllAssocationsFromProperties(),
10         parent <- old.getGeneralization(),
11         domainModel <- thisModule.resolveTemp(old.
               generalizationBoundedContext(), 'dm')
12       )
13 }
```

Quellcode 5.6: Regel zur Erzeugung einer einfachen Entity

5.3.3 Transformation von Shared Models

Die Unterscheidung von normalen `ValueObjects` und Shared Models ist notwendig, da Shared Models speziellen Transformationregeln unterliegen (vgl. Abschnitt 4.2, Punkt 6). Ein Shared Model zeichnet sich nicht durch einen gesonderten Stereotypen aus, spondern dadurch, dass es als `ValueObject` zwischen `Bounded Contexts` besteht [Eva03]. Diese Eigenschaft wird genutzt, um aus `ValueObject` Elementen Shared Models zu identifizieren, da der Helper `goUpToElementWithStereotype`, wenn kein Elternelement mit dem Stereotyp `BoundedContext` gefunden wird, `OclUndefined` zurückgibt (vgl. Quellcode 5.6, Zeile 3).

Als für die Regel gültige Variable wird mit `abstraction` der `supplier` eines `Abstraction` Elements gesetzt, welches das Shared Model als `client` definiert (vgl. Zeile 5) Hierzu wird im zur Initialisierung der Variable verwendeten Helper bis zum Element mit dem UML-Typen `Model` gegangen, und von hier aus der rekursive Helper `getPackagedElementsByType` aufgerufen um alle `Abstractions` des Modells zu erhalten. Die entstandene `Collection` wird zuletzt hinsichtlich des `client` gefiltert. Es gilt zu beachten, dass es im Falle von mehreren Treffern keine Möglichkeit gibt zu bestimmen, welche `Abstraction` im weiteren Verlauf der Regel verwendet werden soll, weswegen in diesem Fall nur das erste Element zurückgegeben wird.

Bei der Erzeugung des Ausgabeelements wird anschließend `parent` auf die Klasse von der das SharedModel abgeleitet ist und `domainModel` auf das `domainModel` gesetzt, welches aus dem `BoundedContext` Element, dem sie zugeordnet ist,erzeugt wird. Da ATL bei der Transformation zunächst Platzhalter Elemente für jede ausgeführte Regel anlegt, und in einem weiteren Schritt die Eigenschaften dieser setzt [Waa], musste das `domainModel` einer `Entity` in AjiL auf `changeable = true` geändert werden, da ansonsten eine Änderung des `domainModel` nach der Initialisierung nicht möglich wäre.

```
1 rule SharedModel2Entity{
2   from
3     old : DDD!Class (old.hasStereotype('ValueObject') and old.
          goUpToElementWithStereotype('BoundedContext').oclIsUndefined())
4   using{
5     abstraction : DDD!Abstraction = thisModule.getAbstractFromSharedModel(old
          ).supplier.get(0);
6   }
7   to
```

```
8     new : AJI!EntityT(
9         name <- old.name,
10        attributes <- old.attribute,
11        domainModel <- thisModule.resolveTemp(abstraction.
              goUpToElementWithStereotype('BoundedContext'), 'dm'),
12        parent <- abstraction
13      )
14 }
```

Quellcode 5.7: Regel zur Erzeugung einer Entity aus einem Shared Model

5.3.4 Erzeugung von Interfaces und Abilities

Bei der Erzeugung von `ServiceInterfaces` aus `Repository` oder `Service` Klassen wird über den Helper `suitableForInterface` gefiltert (vgl. Quellcode 5.7, Zeile 3). Dieser prüft zunächst, ob das Element einen der beiden Stereotypen besitzt und ruft in diesem Fall durch das Element den Helper `countDependenciesOnThis` (Quelllcode 5.8) auf . Dieser sammelt zunächst vom kompletten `Model`, in welchem sich das Element befindet, `Dependencies`. Die `Dependencies` werden anschließend danach gefiltert, ob das aufrufende Element als `supplier` gesetzt ist, wodurch es eine `Dependency` auf das aufrufende Element ist, und ob `supplier` und `client` verschiedenen `Bounded Contexts` zugeordnet sind, wodurch es sich um eine Service Dependency handelt. Wenn die dadurch entstandene `Collection` nicht leer ist, wird aus dem Element ein `ServiceInterfaceT` erzeugt, andernfalls wird eine LOG Ausgabe erzeugt, die über den Informationsverlust des Elements informiert. Die `abilities` des `ServiceInterfaceT` werden durch Traceability Links zu den `Operations`, die das UML Element besitzt, zugeordnet.

```
1 rule RepositoryOrService2ServiceInterface{
2   from
3     old : DDD!Class (old.suitableForInterface())
4   to
5     new : AJI!ServiceInterfaceT(
6         name <- old.name + 'Interface',
7         abilities <- old.ownedOperation
8       )
9 }
```

Quellcode 5.8: Regel zur Erzeugung eines Service Interface

```
1  helper context DDD!Class def: countDependenciesOnThis() : Boolean =
2    self.goUpToElementWithType(DDD!Model).getPackagedElementsByType(DDD!
         Dependency) ->
3    iterate(iter; col: Sequence(DDD!Class) = Sequence{}|
4      if(iter.supplier.get(0).name = self.name)
5        then
6          if(thisModule.sameBoundedContextNames(iter.supplier.get(0), iter.
               client.get(0)))
7            then
8              col.append(iter)
9            else
10             col
11           endif
12       else
13         col
14       endif);
```

Quellcode 5.9: Helper zum Zählen von Interface Dependencies

Stellvertretend für die verschiedenen Regeln zur Transformation von `Operations` in die entsprechenden `Abilities` wird an dieser Stelle die Regel `Operation2CreateAbility` erläutert. Bei der Auswertung des die jeweilige `Operation` besitzende Element, wird direkt auf `namespace` der `Operation` zugegriffen, da in UML keine Elemente zwischen `Operations` und `Classes` modelliert werden, die die Auswertung verfälschen könnten. Zusätzlich wird der Anfang des Namens der `Operation` gemäß Tabelle 4.1 gefiltert, um in AjiL den entsprechenden `Ability` Typen zu erzeugen. Das letzte Filterkriterium ist mit Aufruf des Helpers `countDependenciesOnThis()` das gleiche wie für die Erzeugung von `Interfaces` auf dem `namespace` des Elements, um keine `Operations` zu transformieren, deren Elternelemente nicht transformiert werden. Hier wird nicht der Helper `suitableForInterface` verwendet, da dies duplizierte LOG Ausgaben erzeugen würde.

```
1  rule Operation2CreateAbility{
2    from
3      old: DDD!Operation ((old.namespace.hasStereotype('Repository') or old.
           namespace.hasStereotype('Service'))
4        and old.name.substring(1,6) = 'create'
5        and old.namespace.countDependenciesOnThis().size() > 0)
6    to
7      new : AJI!CreateT(
```

```
8         name <- old.name,
9         subject <- old.getCorrectSubject()
10      )
11  }
```

Quellcode 5.10: Regel zur Erzeugung einer Ability

```
1  helper context DDD!Operation def: getCorrectSubject() : DDD!Element =
2    if(self.type.oclIsUndefined())
3      then
4        if not self.namespace.getAllAssocationsFromProperties().first().
             oclIsUndefined()
5          then
6            self.namespace.getAllAssocationsFromProperties().first().memberEnd.
                 first().type
7          else
8            self.namespace.goUpToElementWithType(DDD!Model).
                 collectAssociationsWithMemberEnd(self.namespace).first().
                 memberEnd.get(0).type
9        endif
10     else
11       self.type
12   endif;
```

Quellcode 5.11: Helper für das korrekte Subject einer Ability

5.4 Ausführung

5.4.1 Ausführung als Standalone Jar

An Florian und Jonas: An dieser Stelle bin ich gescheitert es standalone auszuführen und konnte keine Hilfe aus dem Eclipse Forum bekommen, ist ein negatives Ergebnis in diesem Fall auch ein Ergebnis?

5.4.2 Einrichtung und Ausführung in Eclipse

Vorbedingungen und Konfiguration beschreiben

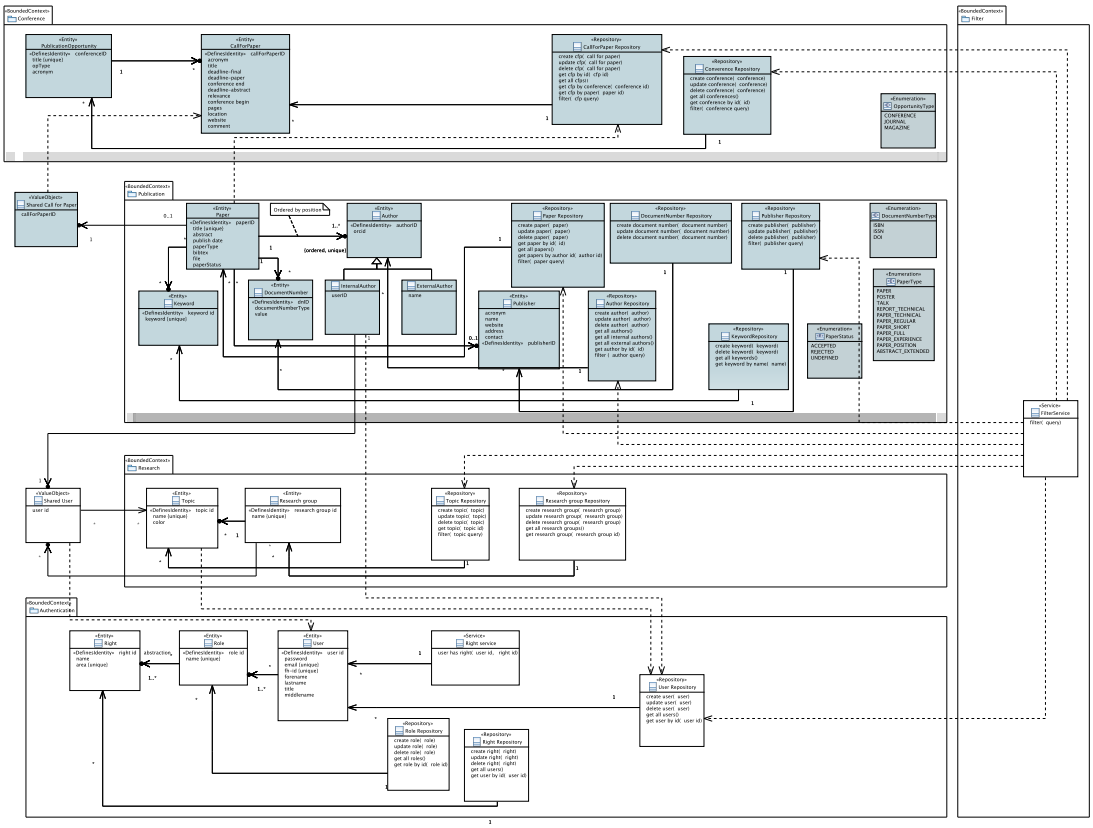

Abbildung 5.2: Research Management System als UML Modell

5.5 Evaluation anhand eines Domänenmodells aus der Praxis

Hier wird beschrieben, wie evaluiert wurde, vielleicht auch noch das Testdiagramm für Aggregate rein packen + Ausführungsdauer messen.

6 Abschluss

6.1 Zusammenfassung

6.2 Fazit

Anregung von Änderungen in ajil (Änderbarkeit von source und dataModel + Komplexe Datentypen für Entitäten.)

6.3 Ausblick

Ausführlichere null Checks und Überarbeitung des Moduls mit OCL erfahrenem Entwickler, da Autor dieser Arbeit nur rudimentäre OCL Kenntnisse besitzt

Weitere Anpassungen an AjiL können akkuratere Transformation ermöglichen (z.B. mehrere DataModels)

Abkürzungsverzeichnis

AjiL *Ajil Modeling Language*
ATL *Atlas Transformation Language*
DDD *Domain-driven Design*
DSL *Domain Specific Language*
DSML *Domain Specific Modelling Language*
EDM *Example-driven Modelling*
EMP *Eclipse Modeling Project*
EMF *Eclipse Modeling Framework*
EMFTVM *EMF Transformation Virtual Machine*
GPL *General Purpose Language*
IDiAL *Institut für die Digitalisierung von Arbeits- und Lebenswelten*
IDE *Integrated Development Environment*
MDSD *Model Driven Software Development*
M2M *Model To Model*
M2T *Model To Text*
MOF *Meta Object Facility*
MSA *Microservice Architecture*
OMG *Object Management group*
OCL *Object Constraint Language*
REST *Representational State Transfer*
SOA *Service Oriented Architecture*
SuS *System Under Study*
UML *Unified Modeling Language*
QVT *Query/View/Transformation*

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[A.L06] A.Lindow, J. Bezivin1; F. Büttner; M. Gogolla; F. Jouault; I. K.: Model Transformations? Transformation Models! In: *MoDELS* (2006)

[Eva03] Evans, E.: *Domain-Driven Design.* Addison-Wesley, 2003

[IEE10] *IEEE: Systems and software engineering – Vocabulary.* Piscataway NJ, USA, 2010

[Mar03] Martin, R. C.: Agile Software Development: Principles, Patterns, and Practices. (2003)

[New15] Newman, S.: *Building Microservices.* O'Reilly Media, 2015

[Obj15] Object Management Group: OMG Unified Modeling Language (OMG UML). Version 2.5 (2015), Nr. formal/2015-03-01

[Poh08] Pohl, K.: *Requirements engineering: Grundlagen, Prinzipien, Techniken.* Heidelberg, dpunkt-Verl., 2008

[Pol06] Polack, D. Kolovos; R. Paige; T. Kelly; F.: Requirements for Domain-Specific Languages. (2006)

[Ric16] Richards, M.: *Microservices vs. Service-Oriented Architecture.* O'Reilly Media, 2016

[RSZ17] Rademacher, F. ; Sachweh, S. ; Zündorf, A.: Differences between Model-Driven Development of Service-Oriented and Microservice Architecture. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, S. 38–45

[Rup] Rupp, R. Joppich; A. Pflüger; S. Queins; C.: *MASTeR - Schablonen für alle Fälle.* `https://www.sophist.de/publikationen/wissen-for-free/`, . – Abgerufen: 2018-03-16

[Var18] Varro, N. Kahani; M. Bagherzadeh; J. R. Cordy; J. Dingel; D.: Survey and classification of model transformation tools. In: *MoDELS* (2018)

[Waa] Waagelar, D.: ATL/User Guide - The ATL Language.

Eidesstattliche Erklärung

ACHTUNG: Korrekten Text bitte unbedingt vorab mit Studienbüro klären!
Hiermit versichere ich gemäß § 18 Abs. 5 der Bachelor-Prüfungsordnung des Studiengangs Informatik aus dem Jahr 2013, dass ich die vorliegende Arbeit selbstständig angefertigt und mich keiner fremden Hilfe bedient, sowie keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Dortmund, 26.08.2018 Till Paplowski