technische universität
dortmund

Entwicklung eines flexiblen Frameworks zur generierung von Datenmodellen

Alexander Brand

25. Februar 2025

Begutachtung:
Prof. Dr. Erich Schubert
Name des Zweitgutachters

Technische Universität Dortmund
Fakultät für Informatik
Data Mining Arbeitsgruppe
https://dm.cs.tu-dortmund.de/

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die zentrale Modellierung von Domainmodellen ist sehr verbreitet in der Entwicklung von großen Software Projekten und zentraler Bestandteil von Product Line Engineering. Dabei stellt die Modellierung des Domain Modells einen Kompromiss zwischen der kompletten Modellierung einer Software und der klassischen Entwicklung ohne Modelle dar.
Ziel dieses Kompromisses ist die Effizienz und Sicherheit der Codegenerierung für das Datenmodell einzusetzen, um die Entwicklung der restlichen Software zu vereinfachen.

1.1.1 Product Line Engineering

Product Line Engineering befasst sich mit der Entwicklung von mehreren verwandten Softwareprodukten. Dabei handelt es sich häufig um Software für Teilaufgaben und angepasste Kundenversionen der Standardsoftware.
Hierbei besteht für eine Organisation die Gefahr, viele Komponenten mehrfach zu entwickeln und zu verwalten. Durch gemeinsam genutzte Komponenten (Assets) wird die Entwicklung vereinfacht und die Software verhält sich beim Kunden einheitlich. Datenmodelle stellen im Product Line Engineering wichtige Assets dar. Einheitliche Modelle verhindern das Übersetzen zwischen verschiedenen Produkte.

1.1.2 EMF

EMF(Eclipse Modelling Framework) ist ein häufig Framework zur Modellierung von Modellen in Java. Es lassen sich große Modelle darstellen und mithilfe von Maven Workflows können diese durch das Build Tool übersetzt werden.
EMF bietet dabei jedoch keine Wahl bei der IDE oder der Programmiersprache. Dies führt dazu, dass Projekte und ganze Firmen bei ihren bisherigen Technologien stehen bleiben. Es wird bei Neuentwicklungen nicht mehr die gefragt, was wären die besten Technologien um das Problem zu lösen, sondern es wird gefragt, wie lösen wir das mit unserer bisherigen Architektur.

1.1.3 Effekte eines unflexiblen Frameworks

Dieser fehlerhafte Ansatz schädigt das Projekt auf mehreren Ebenen:

1. Wissen und Erfahrung Konzentrierung
   Da nur eine Architektur in Betracht gezogen wird, hat jedes Mitglied des Teams nur Erfahrung mit der aktuellen Architektur und jegliche Erfahrung mit anderen Technologien verfällt mit der Zeit. Dies schränkt die die Perspektiven auf Probleme sehr stark ein und macht einen Wechsel sehr aufwendig.
2. Schrinkender Bewerberzahl
   Da nur Bewerber für die gewählten Technologien in Betracht gezogen werden, verringert sich die Anzahl stark. Der Effekt wird verstärkt, wenn die Technologien als veraltet gelten. Eine kleinere Bewerberanzahl zwingt Unternehmen auch Bewerber, die andernfalls nicht beachtet worden wären, in Betracht zu ziehen. Dies führt zu weiteren Negativen Effekten, da einige schlechte Angestellte die Produktivität vieler guter Angestellter stark senken können.
3. Anfälligkeit gegenüber Sicherheitslücken
   Eine starke Festlegung auf Technologien führt dazu, dass Sicherheitslücken gleich jedes Projekt betreffen. So könnten bei einer Zero-Day-Lücke direkt mehere Schicht im SSchweizer Käse Modell"(Source suchen) wegfallen. Diese Anfälligkeit wird stark erhöht sobald eine Technologie nicht mehr aktiv weiterentwickelt wird. Dies führt häufig dazu, dass andere Updates auch nicht genutzt werden können.

1.2 Aufgabenstellung

Das Domain Modell Framework(DMF) soll es ermöglichen Datenmodelle zentral zu modellieren, sodass diese von verschiedenen Software Projekten genutzt werden können. Dabei soll die Flexibilität besonders beachtet werden, um die bisher bestehenden Nachteile zu vermeiden. Zur Flexibilität gehört die (möglichst) freie Wahl der Programmiersprache und die freie Wahl der Entwicklungsumgebung.
Ziel ist es das DMF für Java und Typescript zu implementieren. Es sollen primär Intellij und Visual Studio Code unterstützt werden.

2 Entwicklung

2.1 Auswahl der verwendeten Technologien

Ein zentraler Teil einer Architektur ist die Auswahl der verwendeten Technologien. Diese Technologien sollen die Lösung der Aufgaben einer Software vereinfachen.
Im DMF müssen folgende Aufgaben gelöst werden:

1. Modelldatei Parsen und AST generieren
2. AST auslesen und verarbeiten
3. Kommunikation mit verschiedenen IDE's
4. Generieren von Codedateien in verschieden Sprachen
5. Integration mit verschiedenen Build Tools

2.1.1 Parser

Der Parser für das DMF muss große Dateien wiederholt mit kleinen Änderungen parsen. Diese Anforderung stammt aus der Notwendigkeit des AST's um Syntaktische und Semantische Fehler, sowie die verschiedenen Tokens(siehe Abschnitt LSP) nach jeder Eingabe an die IDE zu übermitteln. Hierbei ist Latenz die höchste Priorität, denn die Reaktionsfähigkeit der IDE beeinflusst die Geschwindigkeit mit der Entwickelt werden kann.
Zusätzlich muss der Parser auch von jeder anderen Komponente des DMFs verwendet werden. Deshalb ist hier die Einschränkung der anderen Technologien unerwünscht.

XText

XText ist ein Framework der Eclipse Foundation.
Es bietet die Möglichkeit eine DSL mit verschiedenen Modellen zu modellieren und Regeln automatisch zu überprüfen. XText setzt auf Modellierung vieler Bestandteile und generiert andere Komponenten komplett. Dies ermöglicht eine schnelle Entwicklung wenn die Anforderungen perfekt zu XText passen. XText schränkt stark ein, wo Anpassungen vorgenommen werden können. So ist es nicht vorgesehen die LSP-Server Implementierung anzupassen, obwohl XText nicht alle Features des LSP-Protokolls unterstützt. Dateigeneration und die Verarbeitung des AST's müssen auch mit dem Java-Interfaces von XText vorgenommen werden. Dies setzt immer die Verwendung von JVM basierten Sprachen vorraus. Jede JVM-Implementierung benötigt beachtliche Zeit zum Starten weshalb Code Generation immer auf den Start Warten muss.
Abschließend waren an XText die nicht funktionierenden Beispiel Projekte sowie zwingende Entwicklung in Eclipse seht abweisend. Ein Framework welche eine einfache und

flexible Entwicklung ermöglichen soll, sollte nicht schwer und nur in einer IDE zu entwickeln sein.

Treesitter

Treesitter ist ein Open Source Framework zur Generierung von Parsern.
Dabei wird die Grammatik mithilfe einer Javascript API definiert. Mithilfe der Treesitter CLI wird aus der Javascript Datei der Parser generiert. Der generierte Parser nutzt C. C eignet sich hier sehr gut, da es die höchste Performance und die Möglichkeit es in jeder anderen Sprache zu nutzen bietet. Das Nutzen von C ist für jede Sprache eine Voraussetzung, um mit dem Betriebssystem zu kommunizieren. C's größter Nachteil, die manuelle Speicher Verwaltung, wird durch die Generation des Parsers gelöst. Die bereitgestellten Schnittstellen übergeben Strukturen welche vom Aufrufer verwaltet werden.

Iteratives Parsen Ein großes Unterscheidungsmerkmal von Treesitter ist die Möglichkeit iterativ zu parsen.

With intelligent [node] reuse, changes match the user's in tuition; the size of the development record is decreased; and the performance of further analyses (such as semantics) improves.[?]

Beim iterativen Parsen ist das Ziel den AST nicht bei jedem Parse Durchlauf neu zu erstellen, sondern möglichst viel des AST's wiederzuverwenden. Für das Iterative Parsen muss der AST sowie die bearbeiteten Textstellen an Treesitter übergeben werden. Die Durchlaufszeit des iterativen Parsedurchlaufs hängt nicht mehr der Länge der kompletten Modelldatei ab, sondern nur von den neuen Terminalsymbolen und Modifikationen im AST:

Our incremental parsing algorithm runs in O(t + slgN) time for t new terminal symbols and s modification sites in a tree containing N nodes [?]

ANTLR

ANTLR ist sehr ähnlich zu Treesitter. Die größten Unterschiede sind die API's zum Schreiben der Grammatiken und die Möglichkeit iterativ zu Parsen. Zusätzlich unterstützt ANTRL nur Java, C# und C++. Dies zwingt einen in der Wahl der Implementierungssprache ein.

Auswahl Parser

Für das DMF Framework wurde Treesitter verwendet. Die Exellente Performance sowie die Flexibilität bei der Implementierung der restlichen Komponenten hoben Treesitter von den restlichen Technologien ab.

2.1.2 AST Verarbeiten

Bei der Verarbeitung des AST's müssen verschiedene Regeln abgearbeitet werden und der Inhalt des AST's in einem Modell vorbereitet werden. Essentiell für die Verarbeitung ist die Zusammenarbeit mit den folgenden Komponenten.

Die Auswahl der Technologie für diesen Schritt basiert auf der Auswahl für die folgenden Schritte.

2.1.3 Kommunikation mit verschiedenen IDE's

Damit ein Framework die Entwicklung nicht einschränkt muss es in verschiedenen Ïntegrated Development Environments"(IDE's) genutzt werden können. Viele IDE's stellen Schnittstellen für Plugins bereit. Dazu zählen Intellij, Eclipse, NeoVim und VSCode. Jede Schnittstellen ist jedoch unterschiedlich, wodurch die Entwicklung von vielen Verschiedenen Plugins nötig wäre.

Language Server Protokoll Eine einfachere Möglichkeit bietet das "Language Server Protokoll"(LSP). Dieses Protokoll bietet die Möglichkeit, dass viele verschiedene IDE's eine Server Implementierung nutzen. Im Fall von Zed und Eclipse lassen sich LSP-Server sogar ohne jegliche Plugins einbinden. Wobei hier auf die schlechte Unterstützung des LSP-Protokolls in Eclipse hingewiesen werden muss. Intellij und NeoVim nutzen Plugins, um LSP-Server anzubinden. VSCode bietet eine API und einen einfachen LSP-Client in ein kleines Plugin zu implementieren. Im LSP-Server können gebündelt Logik und Prokoll implementiert werden.

LSP wird hauptsächlich über die Standard-Eingabe und -Ausgabe oder über einen Server Socket transportiert. Es wird ein JSON-RPC Format genutzt. Der LSP-Server muss somit JSON, Std-In und Std-Out, sowie Server Sockets unterstützen.

Typescript

Von der VSCode Dokumentation wird die Implementierung eines LSP-Servers in Typescript empfohlen. Dafür werden Bibliotheken bereit gestellt. Typescript eignet sich gut, für die JSON Parsing und für die Verwendung von Server Sockets. Probleme entstehen bei Typescript bei den Themen Performance, Anbindung an den Parser und bei der Fehlerbehandlung.

Golang

Golang ist eine Sprache welche für die Entwicklung von Backends ausgelegt wurde.
Es werden die Anforderungen für JSON-Parsing, Std-IO und Server Sockets erfüllt, durch

die große Standard Bibliothek erfüllt. Es gibt keine Bibliothek welche das komplette Protokoll beinhaltet. Dieses kann jedoch durch die Unterstützung von LLMs schnelle generiert werden.
Golang bietet zusätzlich eine simple Anbindung an den Parser und die Möglichkeit sehr einfach Parallelität einzubauen. Besonders erwähnenswert ist die Geschwindigkeit eines Golang Programmes und die Startgeschwindigkeit ohne auf Speichersicherheit zu verzichten.

Java

Java bietet eine bietet mit "lsp4jëine Bibliothek zur Einfachen Entwickung. Bei der Einbindung des Parsers gestallten sich jedoch zusätzliche Herrausforderungen da der Java Code Plattform unabhängig kompiliert wird und Plattform abhängigen Code aufrufen muss. Java benötigt für die Ausführung eine Installierte Instanz der JRE. Die JRE muss nicht nur zusätzlich zum LSP-Server verwaltet werden, sondern benötigt zusätzlich Zeit zum Starten. So muss der Entwickler länger warten bis seine Entwicklungsumgebung bereit steht.

2.1.4 Generation von Codedateien in verschiedenen Sprachen

Ziel des DMFs ist es große Mengen an Sourcecode zu Generieren. Dabei soll den Entwicklern die Wahl zwischen mehreren Zielsprachen gegeben werden. Diese Generation wird beim Build und damit sehr häufig ausgeführt. Eine Langsame Generation wird jeder Organisation viel Geld kosten.

Die Generation muss somit schnell und Zielsprachen unabhängig sein. Sie muss auch aus von den Build Tools gestartet werden.

Golang Templates

Golang Standardbibliothek bietet die Möglichkeit Templates zu definieren. Diese Templates werden hauptsächlich für die Generierung von HTML genutzt. Da sie Golang die Templates nicht nur für HTML sondern auch für generelle Texte anbietet, können diese auch für jede Zielsprache genutzt werden.
Die Anforderungen an einen Webserver (Geschwindigkeit, Resourcen schonend, Simpel) komplementieren die Anforderungen an einen Codegenerator sehr gut.
Golang Templates stechen besonders für ihre Integration in IDE's wie z.B. in Intellij heraus.

Java

Es gibt mehrere Template Engines für Java. Einige Beispiele wäre FreeMaker oder Apache Velocity. Beide sind gut unterstützt und bieten alle nötigen Features für die Generierung von Code.

Typescript

Für Typescript gibt es viele Template Engines. Zu den bekannten gehören Eta, liquidjs und squirrelly. Sie bieten alle die Möglichkeit verschiedene Zielsprachen zu generieren und können mit nodejs ausgeführt werden.

Auswahl

Da Golang eine exzellente Unterstützung in Intellij hat und keine Zusätzliche Installtion wie NodeJs oder JRE verwalten muss, fiehl meine Wahl auf Golang.
Mit der Wahl für Golang für den Generator, ist auch die Wahl für die Verarbeitung des AST's und für den LSP-Server.

2.1.5 Integration mit verschiedenen Build Tools

Damit eine Generation während des Buildvorgangs ist essentiell, um sicherzustellen das der generierte Code aktuell ist. Damit der Neugeneration werden auch alle eventuelle Anpassungen in den Dateien überschrieben, wodurch Fehler vermieden werden.

Maven

Maven ist ein sehr verbreitetes Build Tool für Java. Maven unterstützt Plugins welche während des Builds ausgeführt werden und in der Maven Konfiguration konfiguriert werden können. Die API für Maven Plugins ist in Java geschrieben. Dieses Plugin muss den Generator aufrufen. Dies ist möglich indem die Datei des Generators ausgeführt wird.

NPM

NPM ist das führende Build Tool für Typescript Projekte. NPM unterstützt die Ausführung von Terminal Befehlen. Der Generator kann somit über das Terminal ausgeführt werden.

2.2 Abstraktion des DMF

Das DMF basiert auf einer Abstraktion der Datenstrukturen aus mehreren Sprachen. Diese Abstraktion wurde nach einer Analyse entwickelt.

2.2.1 Analyse

Für die Analyse wurden die Sprachen Java, Typescript,Python, Golang, Rust und C analysiert. Die Sprachen wurden spezifisch ausgewählt. Java ist weitverbreitet in Enterprise Software. Typescript ist die Standardsprache für jegliche Websiten und viele Backends. Python ist in der Datenanalyse weit verbreitet. Durch die Popularität in Umfragen wurde Python miteinbezogen. Golang ist eine moderne Alternative für Backends und die Implementierungsprache des DMFs. Rust ist die moderne Wahl für 'low level' Programmierung. C ist die Standardsprache für jede 'Foreign-Function-Interfaces' und ist weit verbreitet für ältere 'low level' Software.

Analyse der Typen

Es wurde analysiert, welche Typen als Referenz oder als Wert als Variablentyp genutzt werden können.

Typen	Java	Typescript	Python	Golang	Rust	C
Wert	Primitive Typen	Primitive Typen	Primitive Typen	Alle Typen	Alle Typen	Alle Typen
Referenz	Objekte	Objekte, Arrays, Funktionen, Klassen	Alles außer primitive Typen	Explizit	Explizit	Explizit

Bei den Sprachen Java, Typescript und Python werden nur Primitive Typen als Wert Variablen gespeichert. Deshalb wurden die Pritiven Typen dieser Sprachen genauer verglichen:

Primitive Typen	Java	Typescript	Python
	byte, short, int, long, float, double, char, boolean	number, bigint, string, boolean	int, float, bool, str

Auffällig sind hierbei die Zusammenfassung der Typen byte, short, int, long in Java in den Typen int in Python, sowie die Zusammenfassung aller Zahlentypen, bis auf long, in number in Typescript. Java besitzt als einzige Sprache String nicht als primitiven Datentyp.

Analyse von Nullwerten

Nullwerte sind besonders aus Java bekannt und stellen das Fehlen eines Wertes dar. Es zählt zu der Definition eines Types dazu, zu definieren, ob der Typ Nullwerte erlaubt. Dies muss auch für Werte und Referenzen evaluiert werden.

Nullwerte	Java	Typescript	Python	Golang	Rust	C
Wert	nein	nein	ja	nein	Explizit	nein
Referenz	ja	Explizit	ja	ja	Explizit	ja

Es ist klar zu erkennen, dass bis auf Python jede Sprache Wert-Variablen ohne Nullwerte darstellen kann. Referenzen können auch in jeder Sprache Nullwerte beinhalten. In Typescript und Rust muss dies bloß explizit definiert werden. Aus diesen Ergebnissen ergibt sich, dass die Unterteilung in Wert- und Referenz-Variablen auch die Unterteilung in Nullbare und nicht Nullbare Variablen abbildet.

Collectiontypen

Um 1:n- oder n:m-Beziehungen im Datenmodell modellieren zu können wurden drei Collection-Typen aus Java ausgewählt und passende Äquivalente zu finden.

Collectiontypes	Java	Typescript	Python	Golang	Rust	C
List	ja	ja (Array)	ja	ja (slice)	ja	ja (Array)
Set	ja	ja	ja	nein	ja	nein
Map	ja	ja	ja (dictionary)	ja	ja	nein

Die gewählten Typen sind die am häufigsten verwendeten Collection-Typen. Eine Map beinhaltet eine 1:n-Beziehung und ermöglicht einen schnellen Zugriff. Eine Liste bildet eine n:m-Beziehung zwischen den modelliertem Element und dem Inhalt der Liste. Ein Set bildet eine n:m.Beziehung mit der Garantie, dass jeder enthaltener Wert einzigartig ist.

In der Analyse der Liste gab es feine Unterschiede in der Implementierung. Typescript und C nutzen einen Array, jedoch verhält sich der Typescript Array wie eine Liste. In C sind Arrays in ihrer Größe bei ihrer Initialisierung festgelegt. Golang nutzt ein Konstrukt namens 'slice'. Es kommt mit bestimmten Eigenschaften, kann jedoch für eine Liste genutzt werden.

Ein Set findet sich nur in Golang und C nicht. Hier kann es durch eine Liste ersetzt werden. Die Garantien müssten selber verwaltet werden.

Bei der Analyse der Map wurde nur in C keine Implementierung gefunden. Python nutzt für die Map den Namen 'dictionary'.

2.2.2 Elemente eines Modells

Um mit dem DMF Daten in Strukturen verschiedener Programmiersprachen darstellen zu können, müssen auch diese Abstrahiert werden. Dieser Abschnitt beschriebt wie aus den Analysen der Programmiersprachen die Abstraktion des DMFs gebildet wurde.

Primitive Typen

Grundvoraussetzung sind die Primitiven Typen und Referenzen zu anderen Elementen. Bei der Analyse wurde ein unterschiedlichen Maß in der Feinheit der Zahlentypen festgestellt. Es gibt in SQL Datenbanksystem generell eine Unterscheidung zwischen ganzen Zahlen und rationalen Zahlen. Somit muss es eine Unterscheidung zwischen int und double geben. Es wird jedoch auch unterschieden wie groß ganze Zahlen werden, weshalb ein long Typ sinnvoll ist. Dieser kann auch mithilfe von bigint in Typescript abgebildet werden. Für die Verarbeitung von unbekannten Daten werden häufig Bytes genutzt. Von den drei verglichen Sprachen, beinhaltet nur Java den primitiven Typ. Die int-Typen der jeweiligen Sprachen ermöglichen jedoch ähnliche Operationen. Deshalb wurde auch Byte aufgenommen. Eine Unterscheidung zwischen float und double wurde nicht vorgenommen, da diese Unterscheidung in den Systemen die sie enthalten sehr wenig verwendet wird.
String ist vorallem in Scriptsprachen ein primitiver Typ und wird auch von Datenbanken unterstützt. Deshalb wurde auch String als primitiver Typ ins DMF aufgenommen.
Im Gegensatz zu allen verglichenen Programmiersprachen besitzen SQL-Datenbanken Unterstützung für Datum- und Zeitstempel-Werte. Damit die Generation diese Werte in das Datenbankmodell übernehmen kann, wurden 'date' und 'datetime' als primitive Typen hinzugefügt.
Abschließend gehört noch 'boolean' zu den primtiven Typen. Wahrheitswerte werden sowohl in allen Programmiersprachen als auch in allen Datenbankensystem unterstützt.

Somit beinhaltet das DMF die folgenden primitiven Typen:

Typ	ganze Zahlen	rationale Zahlen	Text	Zeit	Wahrheitswert
	byte, int, long	double	string	date, datetime	boolean

Diese Primitiven Typen werden im DMF in Argumenten abgebildet. Argumente bestehen aus einem Primitiven Typen und einem Namen. Alle anderen Datentypen werden als Referenzen abgebildet. Vorgesehen ist nur Referenzen explizit als Nullbar generiert werden.

Funktionen

Funktionen gehören zu den Elementen die sich in jeder Programmiersprache wiederfinden. Im DMF werden Funktionen nur im Zurückgabewert eingeschränkt. Statt mehreren Werten wie z.B. in Golang kannn im DMF nur ein einzelner Wert modelliert werden. Diese Einschränkung stammt aus vielen Sprachen, welche nur einen Wert unterstützen.

Komplexe Datentypen

Structured constructors. (+), types can be built up from these basic types by means of type. The type constructors in our language include function spaces Cartesian products (x), record types (also called labeled Cartesian products), and variant types (also called labeled disjoint sums).[?]

In nahezu allen Programmiersprachen gibt es die Möglichkeit, mit so genannten zusammengesetzten oder komplexen Datentypen zu arbeiten. Ihnen ist gemeinsam, dass wir mehrere Werte nebeneinander dort abspeichern können.[?]

Das DMF muss diese Datentypen auch abbilden können. Deshalb beinhaltet es Structs. Der Name wurde von der Programmiersprache C übernommen, da diese Syntaktische Grundlage für fast alle Programmiersprachen dient.
Im DMF können Structs Argumente, Referenzen zu anderen Structs, Entities, Enums und Interfaces (siehe folgende Abschnitte) und Funktionen beinhalten. Funktionen gehören nicht zur Definition eines komplexen Datentyps, sondern stammen aus der Objekt Orientierten Programmierung. Da jedoch Funktionen auch ohne Objektorientierung für Datentypen generiert werden können, kann das DMF diese Abstraktion unterstützen.
Für die Modellierung wird auch die Abstraktion von Datentypen essentiell sein. Dafür müssen Structs von anderen Structs erben und Funktionen von Interfaces implementieren können.
Abstraktion funktioniert in jeder Sprache ein wenig unterschiedlich, weshalb das DMF nur garantieren kann, dass die Variablen und Funktionen die von einem Struct geerbt werden im Generat vorhanden sind. Zum Beispiel in C könnte ein DMF Generat keine Abstraktion generieren, sondern nur die Elemente kombinieren.

Identität einer Instanz in der Datenbank

Ein Modell im DMF Framework soll in einer Datenbank gespeichert werden können. Dafür müssen Datenbankschlüssel definiert werden. Ein Schlüssel definiert die Identität eines Zeile in einer Tabelle. Diese Identität muss auch im Modell abgebildet werden. Das DMF fügt deshalb den Typen 'Entity' hinzu, welcher eine Identität besitzt. Er basiert auf dem Struct und kann somit Argumente, Referenzen und Funktionen beinhalten. Eine Entity muss die Definition eines Identifiers beinhalten.

Ein weitere Besonderheit ist die Vererbung bei Entities. Eine Entity darf sowohl von einem Struct als auch von einer Entity erben. Ein Struct darf nur von einem Struct erben.

Aufzählungen

Aufzählungen sind Bestandteil vieler Programmiersprachen. Häufig existieren sie als reine Liste aus Codesymbolen. Aus Sprachen wie Rust sind jedoch auch Aufzählungstypen dessen Einträge konstante Werte beinhalten können bekannt. TODO: Beispiel Rust Enum Code

Diese Funktion kann auch in Sprachen dessen Enums diese Möglichkeit nicht beinhalten, durch Funktionen die für den Enumeintrag den modellierten Wert zurückgeben, emuliert werden.

Im DMF lassen sich diese Werte mithilfe von Argumenten modellieren. Bei der Definition eines Enumeintrags müssen die Konstanten mit angeben werden.

Interfaces

Wichtig für die Abstraktion sind Interfaces. Sie stellen Funktionen bereit und können zusammen mit anderen Interfaces in Structs und Enities implementiert werden.

Organisation der Elemente

In großen Softwareprojekten werden Datentypen generell in Gruppen organisiert. Diese Gruppierung erfolgt meistens über das Dateisystem. Dabei repräsentiert ein Ordner eine Gruppe. Diese Gruppe wird meistens 'Package' genannt.
Das DMF beinhaltet auch Packages. Diese werden jedoch nicht im Dateisystem modelliert, sondern sollen als Elemente im Modell enthalten sein.

2.2.3 Zuweisungen der Abstraktionen

Damit diese Abstraktion genutzt werden kann, müssen für jeden abstrakten Typen im DMF eine Zuweisung in jeder Sprache festgelegt werden.

Java

Element	Java
package	Java Package
struct	Java Klasse
entity	Java Klasse
interface	Java Interface
enum	Java Enum

Die DMF Elemente können sehr gut in Java übersetzt werden. Für die Entity kann sogar die Identität mithilfe der Implementation von der Methoden 'hashCode' und 'equals' übernommen werden. Die Enums unterstützen auch die zusätzlichen Argumente.

Datentyp	Java
byte	byte
int	int
long	long
double	double
boolean	boolean
string	java.lang.String
date	java.time.LocalDate
datetime	java.time.LocalDateTime

Das DMF kann bei den Zahlen und Wahrheitswerten genau auf Java übersetzt werden. Für Text- und Zeitwerte werden Klassen der Standardbibliothek verwendet.

Typescript

Element	Typescript
package	Ordner
struct	Typescript Klasse
entity	Typescript Klasse
interface	Typescript Interface
enum	Typescript Enum

TODO Nach Entwicklung des Typescript Generators

Datentyp	Typescript
byte	number
int	number
long	bigint
double	number
boolean	boolean
string	string
date	Date
datetime	Date

TODO Nach Entwicklung des Typescript Generators

2.3 Aufbau der DMF-DSL

Im Kern des DMFs sind die Modelldateien. Sie enthalten alle Informationen und sind die Schnittstelle zwischen DMF und Entwickler*innen. Deswegen ist das Format essentiell. EMF setzt auf XML als Format für die Modelldateien und erleichtert dadurch die Bearbeitung durch Grafische Editoren in Eclipse. Grafische Editoren funktionieren für das DMF

nicht, da DMF mithilfe des LSP-Protokolls nur direkt Text-Editoren unterstützt. Text-Editoren bieten auch eine schnellere Entwicklung, da sie direkt auf der Textbearbeitung der jeweiligen IDE aufsetzen.
Somit wurde für das DMF eine DSL entwickelt.

2.3.1 Die DSL und die EBNF

Die DMF-DSL besteht aus vielen Segmente die in einer EBNF dargestellt werden können. In diesem Abschnitt werden die verschiedenen Segmente und ihre Beziehung zur Abstraktion vorgestellt. Mithilfe ausgewählter EBNF-Regeln wird die Grammatik erläutert.

Die EBNF Grammatik nutzt '[]' für Optionale Elemente, '*' für optionale Wiederholungen (mindestens 0 Elemente) und '+' erforderliche Wiederholungen (mindestens 1 Element). Es können Elemente mit Klammern gruppiert werden. Für Tokens werden die Regulären Ausdrücke angegeben und mit einem 'R' markiert.

Die Grammatik beginnt mit der Regel 'source_file':

⟨*source_file*⟩ ::= ⟨*dmf_declaration*⟩ ⟨*new_line*⟩ ⟨*model_declaration*⟩ ⟨*new_line*⟩ [⟨*import_block*⟩] ⟨*model_content*⟩

In ihr sind die Regeln des Headers, des Importblocks und des Modelinhalts enthalten.

Namenskonvention in der DMF Grammatik Die Grammatik des DMFs nutzt eine Namenskonvention, um den Zweck einer Regel zu dokumentieren.
Ein Block beschreibt immer die Regeln die ein Element umschließen.
Eine Content-Regeln beschreibt den Inhalt in einem Block. Ein Block kann immer mehrere Content Elemente beinhalten. Ein Content-Element schreibt auch ein modelliertes Element, z.B. ein Argument.
Eine Value-Regel beschreibt eine Regel welche einen Wert welcher während der Semantischen Verarbeitung des ASTs eingelesen wird.

Header

Jede DMF Datei beginnt mit einem Header. Dieser Header wird genutzt die Version des Formats und Metadaten über das Modell zu dokumentieren.

Listing 2.1: Header einer DMF-Modelldatei

```
1 dmf 1.0.0
2 model "beispiel" version 0.0.1
```

Der Header besteht aus 2 wichtigen Grammatik-Regeln: "dmf_declarationünd "model_-declaration". Beide bilden jeweils eine Zeile ab.

Listing 2.1: Die EBNF Regeln

⟨*dmf_declaration*⟩ ::= 'dmf' ⟨*version_number*⟩

⟨*model_declaration*⟩ ::= 'model' ⟨*string_value*⟩ 'version' ⟨*version_number*⟩

Importblock

Es gibt zwei Gründe für den Import von vorhandenen Modellen:

1. Um die enthaltenen Klassen in das eigene Modell zu kopieren.
2. Um die enthaltenen Klassen zu erweitern.

Um ein DMF Modell zu Importieren wird die Datei und das Package das vom Modell übernommen werden soll. Es muss ein Package angegeben werden damit immer klar ist, welche Pfade schon belegt sind.

Listing 2.2: Import des Package de.base

```
1 import de.base from "./base.dmf"
```

Diese Importstatements werden alle im Import-Block zusammengefasst. Sollten keine Imports genutzt werden, so wird das komplette Import-Block aus dem AST entfernt, statt ein leeres Import-Block Element zu enthalten.

⟨*import_block*⟩ ::= ⟨*import_statement*⟩+

⟨*import_statement*⟩ ::= 'import' ⟨*package_string*⟩ 'from' ⟨*string_value*⟩

Modellelemente erweitern Das DMF beinhaltet die Möglichkeit Elemente zu erweitern. Dies ermöglicht Code für das Ursprüngliche Modell wiederzuverwenden und trotzdem Argumente, Referenzen und Funktionen hinzuzufügen.
Eine Erweiterung im DMF muss generell explizit gekennzeichnet sein. Dafür muss zuerst das Package, indem der Entwickler*in etwas erweitern erweitern will, importiert werden. Das importierte Package muss nun im Modell modelliert werden. Es muss bei jedem, schon im importierten Modell vorhandenen Element, das Keyword 'expand' verwendet werden. In den mit 'expand' gekennzeichneten Elementen können nun neue Inhalte modelliert werden.

Modellinhalt

Der Modellinhalt enthält alle Elemente aus der Abstraktion des DMFs. Die Elemente werden in den Packages organisiert. Deshalb enthält die Regeln für den Modellinhalt auch die Regeln für Packages.

⟨*model_content*⟩ ::= ⟨*package_content*⟩+

Override Die Abstraktion des DMFs kann bei manchen Sprachspezifischen Anforderungen an ihre Grenzen stoßen. Deshalb gibt es in der DMF DSL das Konzept des Overrides. An jedem PackageElement und jedem Element innerhalb des PackageElements kann ein Override-Block hinzugefügt werden.
Dieser Block beinhaltet eine Section für einen Generat. In dieser Sektion können Werte überschrieben werden und Sprachspezifische Werte hinzugefügt werden.

⟨*override_block*⟩ ::= 'override' '{' (⟨*java_override*⟩ | ⟨*typescript_override*⟩)* '}'

⟨*java_override*⟩ ::= 'java' '{' (⟨*java_annotaion*⟩ | ⟨*java_extends*⟩ | ⟨*java_implements*⟩ | ⟨*java_class*⟩ | ⟨*java_name*⟩ | ⟨*java_type*⟩ | ⟨*java_doc*⟩)* '}'

⟨*java_annotation*⟩ ::= 'annotations' ⟨*stringValue*⟩

Java Override Option	Funktion
Annotations	Fügt den Text als Annotation zu dem jeweiligen Element hinzu. Java-Annotations gehören nicht zur DMF Abstraktion, sind jedoch für viele Frameworks wichtig.
Extends	Überschreibt die Oberklasse.
Implements	Überschreibt die implementierten Interfaces.
Class	Überschreibt den Klassennamen.
Name	Überschreibt den Namen des Elements (z.B. Variablenname).
Type	Überschreibt den Typen des Elements (z.B. Variablentyp).
JavaDoc	Überschreibt den Kommentar.

Packages Innerhalb eines Package können weitere Packages, Structs, Entities, Enums und Interfaces enthalten sein. Diese verschiedenen Elemente werden im weiterem als PackageElemente bezeichnet.

```
1 package de.beispiel {
2
3 }
```

Jedes PackageElement kann mit einem Kommentar, dem Keyword ëxpandünd einem Override-Block versehen werden:

```
1 // Das ist ein Packages mit dem Pfad "de.beispiel"
2 expand package de.beispiel {
3
4 }
5 override {
6 }
```

Die Regeln für ein Package folgen der Namenkonvention. Der 'package_string' beinhaltet Regeln für die Seperation der Packages mit Punkten.

⟨*package_content*⟩ ::= [⟨*comment_block*⟩] ['expand'] ⟨*package_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ['expand'] ⟨*struct_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ['expand'] ⟨*enum_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ['expand'] ⟨*entity_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ['expand'] ⟨*interface_block*⟩ [⟨*override_block*⟩]

⟨*package_block*⟩ ::= 'package' ⟨*package_string*⟩ '{' ⟨*package_content*⟩* '}'

Structs Structs beginnen wie Packages mit ihrem Identifizierenden Keyword. Die Syntax für die Abstraktion folgt der Java-Syntax.

```
1 struct Beispiel extends SuperBeispiel implements IBeispiel, IABeispiel {
2
3 }
```

Die Grammatik-Regeln für ein Struct folgen der Namenskonvention. Der Inhalt eines Structs können Argumente, Referenzen, Multi-Referenzen (Collections) oder Funktionen sein.

⟨*struct_block*⟩ ::= 'struct' ⟨*identifier*⟩ [⟨*extends_block*⟩] [⟨*implements_block*⟩] '{' ⟨*struct_content*⟩* '}'

⟨*struct_content*⟩ ::= [⟨*comment_block*⟩] ⟨*arg_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ⟨*ref_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ⟨*multi_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ⟨*func_block*⟩ [⟨*override_block*⟩]

Inhalt-Elemente Die Regeln für Argumente, Referenzen, Multi-Referenzen (Collections) und Funktionen werden in den verschiedenen PackageElementen wiederverwendet. Deshalb werden sie hier zentral vorgestellt.

Argument

```
1 arg int i;
```

Beim Argument werden die verschiedenen primitiven Typen mit einer Regeln zusammengefasst.

⟨*arg_block*⟩ ::= 'arg' ⟨*primitive_type*⟩ ⟨*identifier*⟩ ';'

Referenz

```
1 ref ..beispiel.Beispiel beilspiel;
2 ref de.beispiel.Beispiel zweitesBeispiel;
```

Referenzen besitzen zwei verschiedene Arten einen Typen zu spezifizieren.
Entweder kann ein Typ anhand seines relativen Pfades oder seines absoluten Pfades angegeben werden. Die Punkte zu beginn einer relativen Pfades folgen dabei den Regeln eines relativen Dateipfades.

⟨*ref_block*⟩ ::= 'ref' ⟨*reftype*⟩ ⟨*identifier*⟩ ';'

⟨*reftype*⟩ ::= '.'* ⟨*package_string*⟩

Multi-Referenz

```
1 ref Map<int, .Beispiel> beispielLookup;
2 ref Set<string> namen;
```

Multi-Referenzen werden für die Collection-Typen des DMFs genutzt und teilen sich das Keyword 'ref' mit den normalen Referenzen. Innerhalb von '<>' können bis zu zwei Typen angegeben werden. Nur die Map erlaubt zwei Typen. Liste und Set erlauben einen.
Die Typen können Referenzen oder primitive Typen sein.

⟨*multi_block*⟩ ::= 'ref' ⟨*multi_name*⟩ '<' ⟨*primitive_type*⟩ [',' ⟨*primitive_type*⟩] '>' ⟨*identifier*⟩ ';'
| 'ref' ⟨*multi_name*⟩ '<' ⟨*reftype*⟩ [',' ⟨*primitive_type*⟩] '>' ⟨*identifier*⟩ ';'
| 'ref' ⟨*multi_name*⟩ '<' ⟨*primitive_type*⟩ [',' ⟨*reftype*⟩] '>' ⟨*identifier*⟩ ';'
| 'ref' ⟨*multi_name*⟩ '<' ⟨*reftype*⟩ [',' ⟨*reftype*⟩] '>' ⟨*identifier*⟩ ';'

Funktionen

```
1 func void displayBeispiel(.Beispiel beispiel);
2 func int add(int a, int b);
3 func void execute();
```

Funktionen bestehen aus dem Keyword 'func' dem Rückgabetypen, dem Namen und den Parametern; Der Rückgabetyp kann ein primitiver Typ, eine Referenz oder 'void' sein. 'void' bedeutet, dass kein Wert zurückgegeben wurde. Die Parameter bestehen aus einem Typen, der entweder zu den primitiven Typen gehört oder eine Referenz ist, und einem Namen.

⟨*func_block*⟩ ::= 'func' ⟨*reftype*⟩ ⟨*identifier*⟩ '(' [⟨*param_definition*⟩ (',' ⟨*param_definition*⟩)*] ')' ';'
| 'func' ⟨*primitive_type*⟩ ⟨*identifier*⟩ '(' [⟨*param_definition*⟩ (',' ⟨*param_definition*⟩)* ')' ';'
| 'func' 'void' ⟨*identifier*⟩ '(' [⟨*param_definition*⟩ (',' ⟨*param_definition*⟩)*] ')' ';'

⟨*param_definition*⟩ ::= ⟨*reftype*⟩ ⟨*identifier*⟩
| ⟨*primitive_type*⟩ ⟨*identifier*⟩

Entity Entities sind im Aufbau sehr ähnlich zu Structs. Sie unterschieden sich im Keyword und im Identifier-Statement.

```
1 entity BeispielEntity extends AndereEntity implements IBeispiel, IABeispiel {
2     arg int i;
3     arg double d;
4     identifier(i,d);
5 }
```

Besonders bei der ëntity_blockRegel ist, dass sie direkt die ßtruct_contentRegel verwendet. Das Identifier-Statement schließt immer den Inhalt einer Entity ab. Hierbei ist zu beachten, dass ïdentifieräls Keyword und als Regelname verwendet wird.

⟨*entity_block*⟩ ::= 'entity' ⟨*identifier*⟩ [⟨*extends_block*⟩] [⟨*implements_block*⟩] '{' ⟨*struct_content*⟩* ⟨*identifier_statement*⟩ '}'

⟨*identifier_statement*⟩ ::= 'identifier' '(' ⟨*identifier*⟩ (',' ⟨*identifier*⟩)* ')' ';'

Interface Der Aufbau eines Interfaces ist simpler im Vergleich zu einem Struct. Es entfällt der Extends-Block und alle Inhalte bis auf Funktionen.

```
1 interface IBeispiel implements IAnderesBeispiel {
2
3 }
```

Da nur Funktionen in einem Interface vorkommen können, besitzt die Content-Regel nur eine Varainte.

⟨*interface_block*⟩ ::= 'interface' ⟨*identifier*⟩ [⟨*implements_block*⟩] '{' ⟨*interface_content*⟩* '}'

⟨*interface_content*⟩ ::= [⟨*comment_block*⟩] ⟨*func_block*⟩ [⟨*override_block*⟩]

Enum Enum beinhalten ein neues Element: Die Konstante.
Konstanten sind die Einträge eines Enums und bilden mit Argumenten den Inhalt eines Enums.

```
1 enum EBeispiel {
2     arg int i;
3
4     KONSTANTE(_,1);
5 }
```

Konstanten beinhalten immer einen Index. Dieser Index wird in Datenbanktabellen hinterlegt. Es müssen zusätzlich für jedes Argument ein passender Wert angegeben werden. Damit das DMF den Index automatisch berechnet kann ein '_' verwendet werden. Die Regel 'enum_index' bildet diese Besonderheit ab.
Die Regel 'primitive_value' vereinigt alle Value-Regeln.

⟨*enum_block*⟩ ::= 'enum' ⟨*identifier*⟩ '{' ⟨*enum_content*⟩* '}'

⟨*enum_content*⟩ ::= [⟨*comment_block*⟩] ⟨*arg_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ⟨*enum_constant*⟩ [⟨*override_block*⟩]

⟨*enum_constant*⟩ ::= ⟨*identifier*⟩ '(' ⟨*enum_index*⟩ (',' ⟨*primitive_value*⟩)* ')' ';'

⟨*enum_index*⟩ ::= '_' | ⟨*integerValue*⟩

2.3.2 Beispieldatei

```
dmf 1.0.0
model "beispiel" version 0.0.1

import de.base from "./base.dmf"

expand package de.base {
    expand interface IBeispiel {
        func string printBeispielMarkdown();
    }
}

package de.beispiel {
    struct Beispiel implements ..base.IBeispiel {
        arg int i;
        ref .BeispielTyp typ;
    }

    entity Aufgabe {
        ref .Beispiel beispiel;
        arg string frage;
        arg string antwort;
        arg int id;
        identifier(id);
    }

    enum BeispielTyp {
        CODE(_);
        TEXT(_);
    }
}
```

Dieses Beispiel zeigt wie man mithilfe des DMFs eine Struktur modelliert. Diese Struktur beinhaltet Aufgaben, die Benutzer*innen beantworten sollen. Als Hilfestellung gibt es ein Beispiel. Zur korrekten Darstellung wird am Beispiel der BeispielTyp referenziert. Ein Beispiel nutzt ein Interface aus einem anderen Modell. Dieses Interface wird mit einer neuen Methode erweitert, um das Beispiel auch in Markdown zu rendern.

2.4 Semantische Verarbeitung des AST

Für die korrekte Verwendung des ASTs benötigt das DMF auch eine Softwarekomponente, welche den AST durchläuft und ein semantisches Modell erstellt. Anhand dieses semantischen Modells werden semantische Regeln überprüft.

Diese Softwarekomponente ist die erste Komponente welche mithilfe von Golang implementiert wird.

2.4.1 Das semantische Modell

Das semantische Modell bildet alle Informationen, die aus dem AST entnommen werden können, ab. Dazu gehören Referenzen zum AST für Positionen in der Modelldatei, die verschiedenen PackageElemente und die NamendElemente.

ErrorElemente

Für eine annehme und effiziente Entwicklung ist die verständliche Kommunikation von Fehlern essenziell. Deshalb enthält das semantische Modell das ErrorElement.

```
1  package err_element
2  type ErrorElement struct {
3      Fehler FehlerStelle
4      Error  error
5      // Falls es einen Grund gibt, weshalb ein Fehler aufgetreten ist und dieser
6      Cause    *FehlerStelle
7      rendered *string
8  }
9
10 type FehlerStelle struct {
11     // Die Node von der, der ModelCode stammt, und die ein Parent von der eigent
12     // Ist nil, wenn der ModelCode von der eigentlichen Node kommt
13     ContextNode *tree_sitter.Node
14     ModelCode   string
15     Node        *tree_sitter.Node
16 }
```

Mithilfe dieser Strukturen lassen sich Fehler festhalten ohne die spätere Darstellung mit einzubeziehen zu müssen. Der Error dokumentiert um welchen Fehler es sich handelt.
Die FehlerStelle “Fehler” dokumentiert den Modellcode, der falsch ist.
Die FehlerStelle “Cause” dokumentiert den Modellcode, wodurch der Modellcode, welcher in der “Fehler”-FehlerStelle enthalten ist, falsch wurde.
Jede FehlerStelle beinhaltet zwei Nodes. Diese beinhalten die Positionen in der Modelldatei. Die ContextNode ist optional und wird gesetzt wenn für einen Fehler der umliegende Code wichtig. Es handelt sich dabei um eine der Parent-Nodes.
Der ModelCode wird erst beim Rendern der FehlerStelle genutzt. Die Variable dient als Zwischenspeicher für den Code den die Nodes referenzieren.

ErrorElemente werden in der gesamten Semantik-Komponente genutzt und erzeugt. Spätere Komponenten nutzen die ErrorElemente, um den Entwickler*innen die Fehler zu erläutern.

2.4.2 Übersetzen des ASTs

Die Übersetzung des ASTs beginnt mit dem Erstellen eines SemanticContext. Dieser Kontext beinhaltet die erkannten Fehler, das bisherige Modell, den Text der Modelldatei und die TreeCursor. Mithilfe des TreeCursors kann der AST durchlaufen werden. Der Kontext durchläuft den AST in der PreOrder-Reihenfolge. Für jedes Element des ASTs enthält der Kontext eine Methode zum Parsen.
Sollte der AST Import Statements beinhalten so werden zunächst die referenzierten Modelle verarbeitet. Die importierten Packages werden nun in das Modell übernommen. Das Laden der verschiedenen Modelle wird mithilfe einer Callbackstruktur außerhalb der Semantik-Komponente definiert. So können verschiedene Logiken genutzt werden.
Am Ende des Parsen ist Modell vollständig mit allen gültigen Elementen. Elemente die durch den Parser im AST als Fehlerhaft gekennzeichnet wurden, werden ignoriert.

Der Lookup

Um anhand des vollständigen Namens (Package Pfad + Name) ein Element schnell zu finden, wird nach dem semantischen Parsen ein Lookup erstellt. Dieser Lookup nutzt eine Map für den schnellen Zugriff. Zum Befüllen des Lookups wird das semantische Modell durchlaufen. Jedes erreichte Package Element wird dem Lookup hinzugefügt. Beim Packages werden auch alle enthaltenen Elemente durchlaufen.

2.4.3 Die semantischen Regeln

Die semantischen Regeln basieren alle auf dem Typen “walkRule”. Dieser Typ verallgemeinert das Iterieren über den Lookup und nutzt eine Instanz der “iWalkRule” um die Elemente zu verarbeiten.

```
1 package semantic_rules
2 // Supertyp alle Regeln, welche alle Typen durchlaufen.
3 type walkRule struct {
4     lookup    *smodel.TypeLookUp
5     elements  []errElement.ErrorElement
6     iWalkRule iWalkRule
7 }
```

Semantische Regeln implementieren das Interface “iWalkRule” und erweitern die “walkRule”. Durch das Nutzen der eignen Instanz werden die Methoden der Regel-Implementierung aufgerufen, um Elemente zu verarbeiten.

```
1 package semantic_rules
2 func newComputeSuperTypes(lookup *smodel.TypeLookUp) *computeSuperTypes {
3     types := computeSuperTypes{
4         walkRule: &walkRule{
5             lookup:   lookup,
6             elements: make([]errElement.ErrorElement, 0),
```

```
7           },
8       }
9       types.iWalkRule = &types
10      return &types
11  }
```

Die Reihenfolge der semantischen Regeln ist sehr relevant, denn sie überprüfen nicht nur das semantische Modell, sondern setzen auch Referenzen und befüllen Lookups innerhalb der Elemente.
Im Folgenden werden die Regeln in ihrer Ausführungsreihenfolge erläutert.

Compute Supertypes Regel

Die "Compute Supertypes Regel" ermittelt und überprüft die Supertypen der PackageElementen. Dazu gehören alle Referenzen in den Extends- und Implements-Blöcken. Dabei werden folgende Bedingungen überprüft:

1. Der referenzierte Typ muss im Lookup vorhanden sein.
2. Die Vererbung darf nicht rekursiv sein.
3. Structs dürfen nur von Structs erben.
4. Entities dürfen nur von Structs und Entities erben.
5. Es können nur Interfaces implementiert werden.
6. Ein Interface kann sich nicht selber implementieren.

Nachdem die referenzierten Type ermittelt wurden, werden Referenzen zu diesen Typen in die jeweiligen PackageElemente eingetragen. So kann später noch schneller auf diese Elemente zugegriffen werden.

Compute Elements Regel

Die "Compute Elements Regel" ermittelt alle Elemente innerhalb der PackageElemente. Diese werden als NamedElements in dem jeweiligen PackageElement eingetragen, sodass mithilfe des Namens schnell auf das Element zugegriffen werden kann. Zu den NamedElements gehören:

1. Argumente
2. Referenzen
3. Multireferenzen
4. Funktionen
5. Konstanten

Beim Hinzufügen eines Elementes wird überprüft, ob der Name schon von einem Element in der Map genutzt wird.

Die "Compute Elements Regel" einzigartig, denn sie überschreibt das Verhalten des Iterierens durch den Lookup mit einem zweifachen Durchlauf. Denn um alle geerbten Elemente

zu Bestimmen müssen die NamedElements-Maps der anderen PackageElemente schon die eigenen Elemente enthalten.

```
1  package semantic_rules
2  func (c *computeElements) walk() []errElement.ErrorElement {
3      c.walkRule.walk()
4
5      for _, element := range *c.lookup {
6          c.handleAbstraction(element)
7      }
8
9      return c.elements
10 }
```

Das Bestimmen der gerbten Elemente (handleAbstraction) ist rekursiv implementiert. So werden auch alle Elemente, die das erweiterte Element erbt mit eingeschlossen.

Check Entity Identifier Regel

Die “Check Entity Identifier Regel” überprüft, ob alle im Entity-Identifier referenzierten Variablen enthalten sind.

Check Enum Constants Regel

Die Konstanten eines Enums müssen semantische Regeln folgen. Deshalb überprüft diese Regel folgende Bedingungen für jede Konstante:

1. Der Name jeder Konstante muss innerhalb eines Enums einzigartig sein.
2. Der erste Wert muss ein Integer sein (Index).
3. Die Anzahl der Werte muss mit der Anzahl der Argumente im Enum übereinstimmen.
4. Der Typ der Werte muss mit den entsprechenden Argumenten übereinstimmen. Für die Reihenfolge der Argumente wird die Reihenfolge in der Modelldatei genutzt.

Check Referenzen Regel

Die “Check Referenzen Regel” überprüft, dass alle Referenzen existieren. In folgenden Elementen werden Referenzen überprüft:

1. Referenzen in Structs und Entities.
2. Return Typ und Parameter von Funktionen in Structs, Entities und Interfaces.
3. Generische Typen der Multireferenzen in Structs und Entities.

2.5 Der LSP-Server

2.5.1 Das LSP-Protokoll

2.5.2 Die Server Implementierung

2.5.3 Die LSP-Services

2.6 Der Java-Generator

2.6.1 Der Aufbau

2.6.2 Die Templates

2.6.3 Maven Plugin

3 Anhang

3.1 EBNF Grammatik für DMF

⟨*source_file*⟩ ::= ⟨*dmf_declaration*⟩ ⟨*new_line*⟩ ⟨*model_declaration*⟩ ⟨*new_line*⟩
[⟨*import_block*⟩] ⟨*model_content*⟩

⟨*dmf_declaration*⟩ ::= 'dmf' ⟨*version_number*⟩

⟨*model_declaration*⟩ ::= 'model' ⟨*string_value*⟩ 'version' ⟨*version_number*⟩

⟨*import_block*⟩ ::= ⟨*import_statement*⟩+

⟨*import_statement*⟩ ::= 'import' ⟨*package_string*⟩ 'from' ⟨*string_value*⟩ ⟨*new_line*⟩

⟨*model_content*⟩ ::= ⟨*package_content*⟩+

⟨*package_content*⟩ ::= [⟨*comment_block*⟩] ['expand'] ⟨*package_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ['expand'] ⟨*struct_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ['expand'] ⟨*enum_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ['expand'] ⟨*entity_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ['expand'] ⟨*interface_block*⟩ [⟨*override_block*⟩]

⟨*comment_block*⟩ ::= ⟨*comment*⟩+

⟨*comment*⟩ ::= R'//.*\n'

⟨*override_block*⟩ ::= 'override' '{' (⟨*java_override*⟩ | ⟨*typescript_override*⟩)* '}'

⟨*java_override*⟩ ::= 'java' '{' (⟨*java_annotaion*⟩ | ⟨*java_extends*⟩ | ⟨*java_implements*⟩
| ⟨*java_class*⟩ | ⟨*java_name*⟩ | ⟨*java_type*⟩ | ⟨*java_doc*⟩)* '}'

⟨*java_annotation*⟩ ::= 'annotations' ⟨*stringValue*⟩

⟨*java_doc*⟩ ::= 'javaDoc' ⟨*stringValue*⟩

⟨*java_extends*⟩ ::= 'extends' ⟨*stringValue*⟩

⟨*java_implements*⟩ ::= 'implements' ⟨*stringValue*⟩

⟨*java_class*⟩ ::= 'class' ⟨*stringValue*⟩

⟨*java_name*⟩ ::= 'name' ⟨*stringValue*⟩

⟨*java_type*⟩ ::= 'type' ⟨*stringValue*⟩

⟨*package_block*⟩ ::= 'package' '{' ⟨*package_content*⟩* '}'

⟨*struct_block*⟩ ::= 'struct' ⟨*identifier*⟩ [⟨*extends_block*⟩] [⟨*implements_block*⟩] '{' ⟨*struct_content*⟩* '}'

⟨*extends_block*⟩ ::= 'extends' ⟨*reftype*⟩

⟨*implements_block*⟩ ::= 'implements' ⟨*reftype*⟩ (',' ⟨*reftype*⟩)+

⟨*struct_content*⟩ ::= [⟨*comment_block*⟩] ⟨*arg_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ⟨*ref_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ⟨*multi_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ⟨*func_block*⟩ [⟨*override_block*⟩]

⟨*arg_block*⟩ ::= 'arg' ⟨*primitive_type*⟩ ⟨*identifier*⟩ ';'

⟨*ref_block*⟩ ::= 'ref' ⟨*reftype*⟩ ⟨*identifier*⟩ ';'

⟨*multi_block*⟩ ::= 'ref' ⟨*multi_name*⟩ '<' ⟨*primitive_type*⟩ [',' ⟨*primitive_type*⟩] '>' ⟨*identifier*⟩ ';'
| 'ref' ⟨*multi_name*⟩ '<' ⟨*reftype*⟩ [',' ⟨*primitive_type*⟩] '>' ⟨*identifier*⟩ ';'
| 'ref' ⟨*multi_name*⟩ '<' ⟨*primitive_type*⟩ [',' ⟨*reftype*⟩] '>' ⟨*identifier*⟩ ';'
| 'ref' ⟨*multi_name*⟩ '<' ⟨*reftype*⟩ [',' ⟨*reftype*⟩] '>' ⟨*identifier*⟩ ';'

⟨*func_block*⟩ ::= 'func' ⟨*reftype*⟩ ⟨*identifier*⟩ '(' [⟨*param_definition*⟩ (',' ⟨*param_definition*⟩)*] ')' ';'
| 'func' ⟨*primitive_type*⟩ ⟨*identifier*⟩ '(' [⟨*param_definition*⟩ (',' ⟨*param_definition*⟩)*] ')' ';'

| ’func’ ’void’ ⟨*identifier*⟩ ’(’ [⟨*param_definition*⟩ (’,’ ⟨*param_definition*⟩)*] ’)’ ’;’

⟨*param_definition*⟩ ::= ⟨*reftype*⟩ ⟨*identifier*⟩
| ⟨*primitive_type*⟩ ⟨*identifier*⟩

⟨*enum_block*⟩ ::= ’enum’ ⟨*identifier*⟩ ’{’ ⟨*enum_content*⟩* ’}’

⟨*enum_content*⟩ ::= [⟨*comment_block*⟩] ⟨*arg_block*⟩ [⟨*override_block*⟩]
| [⟨*comment_block*⟩] ⟨*enum_constant*⟩ [⟨*override_block*⟩]

⟨*enum_constant*⟩ ::= ⟨*identifier*⟩ ’(’ ⟨*enum_index*⟩ (’,’ ⟨*primitive_value*⟩)* ’)’ ’;’

⟨*enum_index*⟩ ::= ’_’ | ⟨*integerValue*⟩

⟨*entity_block*⟩ ::= ’entity’ ⟨*identifier*⟩ [⟨*extends_block*⟩] [⟨*implements_block*⟩] ’{’ ⟨*struct_content*⟩* ⟨*identifier_statement*⟩ ’}’

⟨*identifier_statement*⟩ ::= ’identifier’ ’(’ ⟨*identifier*⟩ (’,’ ⟨*identifier*⟩)* ’)’ ’;’

⟨*interface_block*⟩ ::= ’interface’ ⟨*identifier*⟩ [⟨*implements_block*⟩] ’{’ ⟨*interface_content*⟩* ’}’

⟨*interface_content*⟩ ::= [⟨*comment_block*⟩] ⟨*func_block*⟩ [⟨*override_block*⟩]

⟨*reftype*⟩ ::= ’.’* ⟨*package_string*⟩

3.2 Beispiel

⟨*statement*⟩ ::= ⟨*ident*⟩ ‘=’ ⟨*expr*⟩
| ‘for’ ⟨*ident*⟩ ‘=’ ⟨*expr*⟩ ‘to’ ⟨*expr*⟩ ‘do’ ⟨*statement*⟩
| ‘{’ ⟨*stat-list*⟩ ‘}’
| ⟨*empty*⟩

⟨*stat-list*⟩ ::= ⟨*statement*⟩ ‘;’ ⟨*stat-list*⟩ | ⟨*statement*⟩