

Entwicklung eines flexiblen Frameworks zur Generierung von Datenmodellen

Alexander Brand

17. März 2025

Begutachtung:
Prof. Dr. Bernhard Steffen
Dr. Oliver Rüthing

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 5
ps.cs.tu-dortmund.de

Inhaltsverzeichnis

1 Motivation

Die zentrale Modellierung von Domainmodellen ist sehr verbreitet in der Entwicklung von großen Software-Projekten und zentraler Bestandteil von Product Line Engineering. Sie fällt damit in den Bereich der **Model Driven Software Development (MDSD)**. Dabei stellt die Modellierung des Domainmodells 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.

1.0.1 Model Driven Software Development (MDSD)

Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen. [**modellbuch**]

Zu diesem großen Begriff gehören verschiedenste Ausprägungen. Komplett modellierte Projekte nutzen grafische Modellierung für Daten, Prozesse und Schnittstellen. Modelle für einzelne Komponenten stellen das Gegenteil dar. Sie basieren häufig auf einer Datei für jedes Modell. Diese Dateien können auch direkt im Texteditor bearbeitet werden.

1.0.2 Product Line Engineering (PLE)

PLE 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 **PLE** wichtige Assets dar. Einheitliche Modelle verhindern das Übersetzen zwischen verschiedenen Produkten.

1.0.3 EMF

Eclipse Modelling Framework (EMF) ist ein häufig eingesetztes 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.

1.0.4 Effekte eines unflexiblen Frameworks

Dieser fehlerhafte Ansatz schädigt das Projekt auf mehreren Ebenen:

1. **Konzentrierung von Wissen und Erfahrung**

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. **sinkende 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 einem Zero-Day-Exploit direkt mehrere Schichten im "Schweizer Käse Modell" wegfallen.

Ein Zero-Day-Exploit beschreibt einen Angriffsweg, bei dem den Betreibern eines Systems keine Reaktionszeit bleibt. [ibmZeroDay] Diese Angriffswege nehmen verschiedene Formen an. Die Unbekanntheit der Zero-Day-Exploits bis zur Ausnutzung, stellt besondere Schwierigkeiten bei der Vorhersage dar, weshalb neue Metriken entwickelt wurden. [wang2013k]

Das Schweizer Käse Modell wurde durch die amerikanische Behörde FAA für die Analyse von Verkehrsunfällen entwickelt. Es bildet die Redundanzen die bei einem Unfall fehlschlagen als Schichten, durch deren Löcher ein spezifischer (Un-)Fall passt, ab. Diese Schichten existieren auch in der Software Entwicklung.[bergeon2009swiss]

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.

1.1 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 Planung des DMF

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 verschiedenen Sprachen
5. Integration mit verschiedenen Build Tools

2.1.1 DSL-Frameworks

Die Entwicklung eigener DSLs kann durch Frameworks vereinfacht werden. Es müssen keine einzelnen Lösungen für die verschiedenen Probleme gefunden werden, da ein Framework schon eine komplette Lösung bereitstellt.

XText

XText ist ein Framework der Eclipse Foundation.

Es bietet die Möglichkeit eine Domain Specific Language (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 Language Server Protokoll-Protokolls unterstützt. Dateigeneration und die Verarbeitung des AST's müssen auch mit den Java-Interfaces von XText vorgenommen werden. Dies setzt immer die Verwendung von JVM basierten Sprachen voraus. 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 und die zwingende Entwicklung in Eclipse sehr abweisend. Ein Framework welche eine einfache und flexible Entwicklung ermöglichen soll, sollte nicht schwer und nur in einer IDE zu entwickeln sein.

Jetbrains MPS

Mit der MPS-IDE bietet JetBrains die Möglichkeit eigene DSLs zu entwickeln. Leider setzen die in MPS entwickelten Sprachen eine Entwicklung/Nutzung der Sprachen in MPS voraus. Die angestrebte Flexibilität des DMFs ist damit nicht gegeben.

Da kein Framework den Anforderungen des DMF genügt, müssen Technologien für einzelne Probleme ausgewählt werden.

2.1.2 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 DMF's verwendet werden. Deshalb ist hier die Einschränkung der Technologien anderer Komponenten unerwünscht.

Treesitter

Treesitter ist ein Open Source Framework zur Generierung von Parsern. Dabei wird die Grammatik mithilfe einer Javascript **Application Programming Interface** definiert. Mithilfe der Treesitter **Command Line Interface** wird aus der Javascript Datei der Parser generiert.

Bei LR-Parsern handelt es um sogenannte "Bottom-Up-Parser". Diese Parser bauen den AST von den Blättern auf. Sie zeichnen durch ihr deterministisches Parsen und die große Klasse an nutzbaren Grammatiken aus.

Für die Generierung von Parsern eignen sich LR-Parser, da die Aktions-Tabellen, auf denen das Parsen aufbaut, automatisch generiert werden kann und während des Parsens keine Rückverfolgung durchgeführt werden muss. [aho1974lr] Aus diesen Gründen generiert Treesitter LR(1)-Parser.

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 Par-sers 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 intuition; the size of the development record is decreased; and the performance of further analyses (such as semantics) improves.[twagner]

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 Durchlaufzeit des iterativen Parse-Durchlaufs 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 $O(t + \text{slg}N)$ time for t new terminal symbols and s modification sites in a tree containing N nodes [twagner]

ANother Tool for Language Recognition (ANTLR)

Das Ziel von **ANTLR** ist sehr ähnlich zu Treesitter: ein Framework für die einfache Entwicklung von Parsern zu bieten.

Doch es zeigen sich tiefere Unterschiede in den Strategien der Frameworks. **ANTLR** benutzt wie auch Treesitter ein eigenes **Application Programming Interface (API)** zum Schreiben der Grammatiken. Diese **API** nutzt jedoch keine weitverbreitete Sprache wie Javascript, sondern **ANTLR** setzt auf eine eigne **Domain Specific Language (DSL)**. Dies erhöht die Ausdruckskraft der **API**, jedoch auch den Einarbeitungsaufwand.

Das **ANTLR** Framework verzichtet auf die Möglichkeit iterativ zu Parsen und bietet stattdessen die Generation von **LL(*)**-Parsern.

LL-Parser sind im Gegensatz zu ?? handelt es sich bei **LL**-Parsern um 'top-down-parser'. Sie bauen damit den **AST** von der Wurzel aus auf. Dafür benötigen die Parser einen Look-Ahead. Bei den Look-Ahead handelt es sich um Eingabe Tokens welche für die Entscheidung zwischen verschiedenen Regeln der Grammatik benötigt werden. **ANTLR** generiert **LL(*)**-Parser, welche die Größe des Look-Ahead dynamisch anpassen können. So können die Parser deutlich mehr Grammatiken verarbeiten und produzieren bessere **ASTs** als **LL(k)**-Parser. [parr2011ll] Der Look-Ahead verhindert jedoch das iterative Parsen, da der Zustand des Look-Aheads sich bei jedem Schritt verändert und nicht aus dem **AST** wiederherstellbar ist.

Auswahl Parser

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

2.1.3 AST Verarbeiten

Bei der Verarbeitung des AST's müssen verschiedene Regeln abgearbeitet werden und der Inhalt des AST's in einem Modell vorbereitet werden. Essenziell 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.

2.1.4 Kommunikation mit verschiedenen IDE's

Damit ein Framework die Entwicklung nicht einschränkt muss es in verschiedenen **Integrated Development Environment (IDE)** 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 (LSP) Eine einfachere Möglichkeit bietet das **Language Server Protokoll (LSP)**. Dieses Protokoll bietet die Möglichkeit, dass viele verschiedene IDE's eine Serverimplementierung nutzen. Im Fall von Zed und Eclipse lassen sich **Language Server Protokoll**-Server sogar ohne jegliche Plugins einbinden. Wobei hier auf die schlechte Unterstützung des **Language Server Protokoll**-Protokolls in Eclipse hingewiesen werden muss. IntelliJ und NeoVim nutzen Plugins, um **Language Server Protokoll**-Server anzubinden. VSCode bietet eine **Application Programming Interface** und einen einfachen **Language Server Protokoll**-Client in ein kleines Plugin zu implementieren. Im **Language Server Protokoll**-Server können gebündelt Logik und Protokoll implementiert werden.

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

Typescript

Von der VSCode Dokumentation wird die Implementierung eines **Language Server Protokoll**-Servers in Typescript empfohlen. Dafür werden Bibliotheken bereitgestellt. 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 LLM's schnell 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 "lsp4j" eine Bibliothek zur einfachen Entwicklung. Bei der Einbindung des Parsers gestalten sich jedoch zusätzliche Herausforderungen 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 Language Server Protokoll-Server verwaltet werden, sondern benötigt zusätzlich Zeit zum Starten. So muss der Entwickler länger warten bis seine Entwicklungsumgebung bereitsteht.

2.1.5 Generation von Codedateien in verschiedenen Sprachen

Ziel des DMFs ist aus Modellen viel 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, Ressourcen 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 FreeMarker 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 Installation wie NodeJs oder JRE verwalten muss, fiel meine Wahl auf Golang. Die Verwaltung von zusätzlichen Runtimes stellt immer eine besondere Herausforderung dar, denn diese Runtimes werden häufig schon von den Entwicklern in einer bestimmten Version, die potenziell nicht kompatibel mit dem DMF wäre, genutzt. Es müssen auch der Pfad zur Installation verwaltet werden, welcher sich zwischen Betriebssystemen unterscheiden kann.

Mit der Wahl für Golang für den Generator ist auch die Wahl für die Verarbeitung des AST's und für den Language Server Protokoll-Server gefallen.

2.1.6 Integration mit verschiedenen Build Tools

Damit eine Generation während des Buildvorgangs ist essenziell, um sicherzustellen, dass 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 Application Programming Interfaces 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 folgender 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 Websites 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 Implementierungssprache des DMF. 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	alles außer primitive Typen	Explizit	Explizit	Explizit

Bei den Sprachen Java, Typescript und Python werden nur primitive Typen als Werte und nicht als Referenzen in Variablen gespeichert. Deshalb wurden die primitiven 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 beschreibt 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 unterschiedliches Maß in der Feinheit der Zahlentypen festgestellt. Es gibt in **Structured Query Language (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 die Typen enthalten, sehr wenig verwendet wird.

String ist vor allem 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 primitiven Typen. Wahrheitswerte werden sowohl in allen Programmiersprachen als auch in allen Datenbanksystemen 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 Rückgabewert eingeschränkt. Statt mehreren Werten wie z.B. in Golang kann im DMF nur ein einzelner Wert modelliert werden. Diese Einschränkung stammt aus vielen Sprachen, welche nur einen Wert unterstützen.

Komplexe Datentypen

In nahezu allen Programmiersprachen gibt es die Möglichkeit, mit sogenannten zusammengesetzten oder komplexen Datentypen zu arbeiten. Ihnen ist gemeinsam, dass wir mehrere Werte nebeneinander dort abspeichern können.[978-3-8348-9999-6.pdf]

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 essenziell 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 einer 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.

Die Vererbung bei Entities unterscheidet sich von Structs. 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.

Listing 2.1: Ein Enum in Rust aus **rustcookbook** [**rustcookbook**]

```

1  enum IpAddr {
2      V4(u8, u8, u8, u8),
3      V6(String),
4  }
```

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

Im DMF lassen sich diese Werte mithilfe von Argumenten modellieren. Bei der Definition eines Enum-Eintrags müssen die Konstanten mit angegeben werden.

Interfaces

Wichtig für die Abstraktion sind Interfaces. Sie stellen Funktionen bereit und können zusammen mit anderen Interfaces in Structs und Entities 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. Zusätzlich zu der Representation der Daten müssen die Funktionen separat implementiert werden können, damit die Implementierung nicht bei jeder Neugeneration des Datenmodells überschrieben werden. Dieses zusätzlich generierten Strukturen werden Delegates genannt.

Java

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

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. Um die Delegates aufgerufen zu können, wird eine statische Instanz der Delegate Klasse genutzt.

Datentyp	Java
byte	byte
int	int
long	long
double	double
boolean	boolean
string	java.lang.String
date	java.time.LocalDate
datetime	java.time.LocalDateTime
Map	java.util.Map(java.util.HashMap)
List	java.util.List(java.util.ArrayList)
Set	java.util.Set(java.util.HashSet)

Das DMF kann bei den Zahlen und Wahrheitswerten genau auf Java übersetzt werden. Für Text- und Zeitwerte werden Klassen der Standardbibliothek verwendet. Die Multireferenztypen nutzen Interfaces und Implementierungen aus der Standardbibliothek. Die Verwendung von allgemeinen Interfaces für diese Datentypen wird in Java erwartet.

Typescript

Element	Typescript
package	Ordner
struct	Typescript Klasse
entity	Typescript Klasse
interface	Typescript Interface
enum	Typescript Enum, Details Typescript Interface und Typescript Record
Delegate	Typescript Funktion

Die Übersetzung der DMF Elemente in Typescript stellt als ein wenig komplexer als die Übersetzung in Java dar. Structs, Entities und Interfaces können direkt in die nativen Elemente übersetzt werden.

Da Typescript Enums nur einen Wert pro Eintrag erlauben, müssen eine zusätzliche Struktur für zusätzlichen Daten und eine Struktur für Zuweisung zwischen Enum-Eintrag und Daten generiert werden. Das "Details"-Interface enthält die Definitionen der weiteren Argumente. Der "Info"-Record enthält die Zuweisung zwischen dem Enum-Eintrag und einer Instanz des Details-Interfaces. (Für Beispiel siehe ??)

Die Delegates können als Funktionen generiert werden, welche von der modellierten Klasse importiert werden können.

Datentyp	Typescript
byte	number
int	number
long	bigint
double	number
boolean	boolean
string	string
date	Date
datetime	Date
Map	Map
List	Array
Set	Set

Bei der Übersetzung in Typescript werden die Typen stark vereinfacht. Bis auf 'long' kann der Typ 'number' alle Zahlentypen darstellen. Auch die Zeittypen werden mit dem 'Date'-Typ zusammengefasst.

Die Multireferenztypen können auf Typen der Standardbibliothek zurückgreifen. Auch wenn in Typescript ein Array verwendet wird, enthält er alle Funktionen wie die Java-Liste. Der Unterschied liegt hier somit nur im Namen und der Syntax.

Datenbank

In der Datenbank müssen die Entitäten abgebildet werden.

Dafür wird für jede Entität eine Tabelle angelegt. Diese enthält alle Argumente der Entität. Der primäre Schlüssel wird durch die Spalten der Variablen, welche im Identifier-Statement spezifiziert werden, gebildet. Bei der Übersetzung der Referenzen wird zwischen Structs, Entitäten und Enums unterschieden:

Die Spalten eines Structs werden zur Tabelle der Entität hinzugefügt.

Bei einer Entität muss eine Referenz zu einer anderen Tabelle eingefügt werden. Dafür werden die Spalten des Primärschlüssels als Foreign Key zur Tabelle hinzugefügt.

Die Werte der Enum-Einträge müssen nicht in der Datenbank gespeichert werden. Deshalb reicht es den Index eines Enum-Eintrags zu speichern.

3 Die Implementierung der DMF-DSL

3.1 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 essenziell. DMF 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 das DMF mithilfe des Language Server Protokolls nur direkt Text-Editoren unterstützt. Text-Editoren bieten eine schnellere Entwicklung, da sie direkt auf der Textbearbeitung der jeweiligen IDE aufsetzen.

Somit wurde für das DMF eine Domain Specific Language (DSL) entwickelt.

3.1.1 Die DSL und die EBNF

Die DMF-DSL besteht aus vielen Segmenten, die in einer Erweiterte Backus-Naur-Form (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 Modellinhalts 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-Regel 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 3.1: 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_declaration' und 'model_declaration'. Beide bilden jeweils eine Zeile ab.

$\langle dmf_declaration \rangle ::= 'dmf' \langle version_number \rangle$

$\langle model_declaration \rangle ::= 'model' \langle string_value \rangle 'version' \langle version_number \rangle$

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 3.2: Import des Package de.base

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

Diese Importstatements werden alle in einem Importblock zusammengefasst. Sollten keine Importstatements genutzt werden, so wird der komplette Importblock aus dem AST entfernt, statt einen leeren Importblock zu enthalten.

$\langle import_block \rangle ::= \langle import_statement \rangle^+$

$\langle import_statement \rangle ::= 'import' \langle package_string \rangle 'from' \langle string_value \rangle$

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 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 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.

$\langle model_content \rangle ::= \langle package_content \rangle^+$

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 Section können Werte überschrieben werden und sprachspezifische Werte hinzugefügt werden.

$\langle override_block \rangle ::= 'override' \{ (\langle java_override \rangle | \langle typescript_override \rangle)^* \}$

$\langle java_override \rangle ::= 'java' \{ (\langle java_annotation \rangle | \langle java_extends \rangle | \langle java_implements \rangle | \langle java_class \rangle | \langle java_name \rangle | \langle java_type \rangle | \langle java_doc \rangle)^* \}$

$\langle java_annotation \rangle ::= 'annotations' \langle stringValue \rangle$

Java Override Option	Funktion
Annotations	Fügt den Text als Annotation zu dem jeweiligen Element hinzu. Java-Annotations gehören nicht zur <i>DMF</i> 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 weiteren als PackageElemente bezeichnet.

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

Jedes PackageElement kann mit einem Kommentar, dem Keyword 'expand' und 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 Namenskonvention. Der 'package_string' beinhaltet Regeln für die Separation der Packages mit Punkten.

$$\begin{aligned}\langle package_content \rangle &::= [\langle comment_block \rangle] [\text{'expand'}] \langle package_block \rangle [\langle override_block \rangle] \\ &\quad | [\langle comment_block \rangle] [\text{'expand'}] \langle struct_block \rangle [\langle override_block \rangle] \\ &\quad | [\langle comment_block \rangle] [\text{'expand'}] \langle enum_block \rangle [\langle override_block \rangle] \\ &\quad | [\langle comment_block \rangle] [\text{'expand'}] \langle entity_block \rangle [\langle override_block \rangle] \\ &\quad | [\langle comment_block \rangle] [\text{'expand'}] \langle interface_block \rangle [\langle override_block \rangle] \\ \langle package_block \rangle &::= \text{'package'} \langle package_string \rangle \text{'{' } \langle package_content \rangle^* \text{'}'}\end{aligned}$$

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.

$$\begin{aligned}\langle struct_block \rangle &::= \text{'struct'} \langle identifier \rangle [\langle extends_block \rangle] [\langle implements_block \rangle] \text{'{' } \langle struct_content \rangle^* \\ &\quad \text{'}' } \\ \langle struct_content \rangle &::= [\langle comment_block \rangle] \langle arg_block \rangle [\langle override_block \rangle] \\ &\quad | [\langle comment_block \rangle] \langle ref_block \rangle [\langle override_block \rangle] \\ &\quad | [\langle comment_block \rangle] \langle multi_block \rangle [\langle override_block \rangle] \\ &\quad | [\langle comment_block \rangle] \langle func_block \rangle [\langle override_block \rangle]\end{aligned}$$

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 Regel zusammengefasst.

$$\langle arg_block \rangle ::= \text{'arg'} \langle primitive_type \rangle \langle identifier \rangle \text{';'}$$

Referenz

```
1 ref ..beispiel.Beispiel beispiel;
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 eines relativen Pfades folgen dabei den Regeln eines relativen Dateipfades.

$\langle \text{ref_block} \rangle ::= \text{'ref' } \langle \text{reftype} \rangle \langle \text{identifier} \rangle \text{' ;'}$
 $\langle \text{reftype} \rangle ::= \text{'.'}^* \langle \text{package_string} \rangle$

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.

$\langle \text{multi_block} \rangle ::= \text{'ref' } \langle \text{multi_name} \rangle \text{'<' } \langle \text{primitive_type} \rangle \text{' ,' } \langle \text{primitive_type} \rangle \text{'>' } \langle \text{identifier} \rangle \text{' ;'}$
 $\quad \quad \quad | \text{'ref' } \langle \text{multi_name} \rangle \text{'<' } \langle \text{reftype} \rangle \text{' ,' } \langle \text{primitive_type} \rangle \text{'>' } \langle \text{identifier} \rangle \text{' ;'}$
 $\quad \quad \quad | \text{'ref' } \langle \text{multi_name} \rangle \text{'<' } \langle \text{primitive_type} \rangle \text{' ,' } \langle \text{reftype} \rangle \text{'>' } \langle \text{identifier} \rangle \text{' ;'}$
 $\quad \quad \quad | \text{'ref' } \langle \text{multi_name} \rangle \text{'<' } \langle \text{reftype} \rangle \text{' ,' } \langle \text{reftype} \rangle \text{'>' } \langle \text{identifier} \rangle \text{' ;'}$

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 Rückgabetypp kann ein primitiver Typ, eine Referenz oder 'void' sein. 'void' bedeutet, dass kein Wert zurückgegeben wird. Die Parameter bestehen aus einem Typen, der entweder zu den primitiven Typen gehört oder eine Referenz ist, und einem Namen.

$\langle \text{func_block} \rangle ::= \text{'func' } \langle \text{reftype} \rangle \langle \text{identifier} \rangle \text{'(' } [\langle \text{param_definition} \rangle \text{' ,' } \langle \text{param_definition} \rangle]^* \text{')' ;'}$
 $\quad \quad \quad | \text{'func' } \langle \text{primitive_type} \rangle \langle \text{identifier} \rangle \text{'(' } [\langle \text{param_definition} \rangle \text{' ,' } \langle \text{param_definition} \rangle]^* \text{')' ;'}$
 $\quad \quad \quad | \text{'func' 'void' } \langle \text{identifier} \rangle \text{'(' } [\langle \text{param_definition} \rangle \text{' ,' } \langle \text{param_definition} \rangle]^* \text{')' ;'}$
 $\langle \text{param_definition} \rangle ::= \langle \text{reftype} \rangle \langle \text{identifier} \rangle$
 $\quad \quad \quad | \langle \text{primitive_type} \rangle \langle \text{identifier} \rangle$

Entity Entities sind im Aufbau sehr ähnlich zu Structs. Sie unterscheiden 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 ‘entity_block’-Regel ist, dass sie direkt die ‘struct_content’-Regel verwendet. Das Identifier-Statement schließt immer den Inhalt einer Entity ab. Hierbei ist zu beachten, dass ‘identifier’ als Keyword und als Regelname verwendet wird.

$$\langle \text{entity_block} \rangle ::= \text{'entity'} \langle \text{identifier} \rangle [\langle \text{extends_block} \rangle] [\langle \text{implements_block} \rangle] \text{'{' } \langle \text{struct_content} \rangle^* \langle \text{identifier_statement} \rangle \text{'}'}$$

$$\langle \text{identifier_statement} \rangle ::= \text{'identifier'} \text{'(' } \langle \text{identifier} \rangle \text{'(' } \langle \text{identifier} \rangle^* \text{'')'}$$

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 Variante.

$$\langle \text{interface_block} \rangle ::= \text{'interface'} \langle \text{identifier} \rangle [\langle \text{implements_block} \rangle] \text{'{' } \langle \text{interface_content} \rangle^* \text{'}'}$$

$$\langle \text{interface_content} \rangle ::= [\langle \text{comment_block} \rangle] \langle \text{func_block} \rangle [\langle \text{override_block} \rangle]$$

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.

$$\langle \text{enum_block} \rangle ::= \text{'enum'} \langle \text{identifier} \rangle \text{'{' } \langle \text{enum_content} \rangle^* \text{'}'}$$

$$\langle \text{enum_content} \rangle ::= [\langle \text{comment_block} \rangle] \langle \text{arg_block} \rangle [\langle \text{override_block} \rangle] \mid [\langle \text{comment_block} \rangle] \langle \text{enum_constant} \rangle [\langle \text{override_block} \rangle]$$

$$\langle \text{enum_constant} \rangle ::= \langle \text{identifier} \rangle \text{'(' } \langle \text{enum_index} \rangle \text{'(' } \langle \text{primitive_value} \rangle^* \text{'')'}$$

$$\langle \text{enum_index} \rangle ::= \text{'_'} \mid \langle \text{integerValue} \rangle$$

3.1.2 Beispieldatei

```

1  dmf 1.0.0
2  model "beispiel" version 0.0.1
3
4  import de.base from "../base.dmf"
5
6  expand package de.base {
7      expand interface IBeispiel {
8          func string printBeispielMarkdown();
9      }
10 }
11
12 package de.beispiel {
13     struct Beispiel implements ..base.IBeispiel {
14         arg int i;
15         ref .BeispielTyp typ;
16     }
17
18     entity Aufgabe {
19         ref .Beispiel beispiel;
20         arg string frage;
21         arg string antwort;
22         arg int id;
23         identifier(id);
24     }
25
26     enum BeispielTyp {
27         CODE(_);
28         TEXT(_);
29     }
30 }

```

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.

3.2 Semantische Verarbeitung des *Abstract Syntax Tree (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.

3.2.1 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 **NamedElemente**.

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     Cause  *FehlerStelle
6     rendered *string
7 }
8
9 type FehlerStelle struct {
10    ContextNode *tree_sitter.Node
11    ModelCode   string
12    Node        *tree_sitter.Node
13 }
```

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.

3.2.2 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.

3.2.3 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 type walkRule struct {
3     lookup      *smodel.TypeLookup
4     elements    []errElement.ErrorElement
5     iWalkRule   iWalkRule
6 }

```

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 geerbten 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.

4 Programme des DMF

4.1 Der LSP-Server

Der LSP-Server besteht aus 3 Bereichen:

1. Dem Modell für das Language Server Protokoll.
2. Der Implementierung der Server-Schicht, welche die Kommunikation mit den Clients verwaltet und die Services aufruft.
3. Die Service-Schicht, welche Funktionen des LSPs in Services implementiert.

In den folgenden Abschnitten werden der Aufbau, die Aufrufe und die Funktionen dieser Schichten erläutert.

4.1.1 Das Language Server Protokoll (LSP)

Das LSP wurde von Microsoft für die Verwendung in Visual Studio Code entwickelt. Es ermöglicht die Funktionen von Plugins über ein Protokoll zu transportieren.

Das Protokoll nutzt JSON-RPC. Wie Nachrichten übermittelt werden, wird im Abschnitt ?? beschrieben.

Grundlegende Nachrichten

Im LSP basieren alle Nachrichten auf dem Message-Interface. Deshalb beinhaltet jede Nachricht die JSON-RPC-Version auf der sie basiert.

```
1 interface Message {  
2     jsonrpc: string;  
3 }
```

Die Nachrichten gehören zusätzlich zu einem der folgenden Typen:

Anfragen Request-Message

```
1 interface RequestMessage extends Message {  
2     id: integer | string;  
3     method: string;  
4     params?: array | object;  
5 }
```

Eine Anfrage wird genutzt, um das Ergebnis einer Methode anzufragen.

Der Type der Parameter wird von der Methode festgelegt. Die Antwort wird zu der Anfrage mithilfe der ID zugeordnet. Diese ID wird von jedem Teilnehmer hochgezählt.

Antwort Response-Message

```
1 interface ResponseMessage extends Message {
2     id: integer | string | null;
3     result?: LSPAny;
4     error?: ResponseError;
5 }
```

Antworten werden immer nach einer Anfrage geschickt, selbst bei einem Fehler.

Die meisten LSP-Clients geben dem Server für die Antwort 5 Sekunden Zeit.

Die ID beinhaltet die ID aus der Anfrage.

Das Ergebnis('result') existiert immer bei einer erfolgreichen Durchführung der Methode. Soll kein Wert zurückgegeben werden, so wird beim Ergebnis der Wert "null" gesetzt.

Bei einem Fehler während der Ausführung der Methode muss der Fehler('error') in der Antwort befüllt werden.

```
1 interface ResponseError {
2     code: integer;
3     message: string;
4     data?: LSPAny;
5 }
```

Der beinhaltet zwei wichtige Elemente: den Error-Code und die Error-Nachricht. Die Error-Codes sind vordefiniert vom Protokoll.[**response**] Die Nachricht ist frei wählbar, sollte jedoch dem Nutzer erklären, wodurch der Fehler entstand.

Es können zusätzliche Daten übermittelt werden. Jedoch ist das Format nicht definiert. Es bietet sich nur an dieses Element zu nutzen, wenn sowohl Server als auch Client selbst implementiert wurden.

Benachrichtigungen Notification-Message

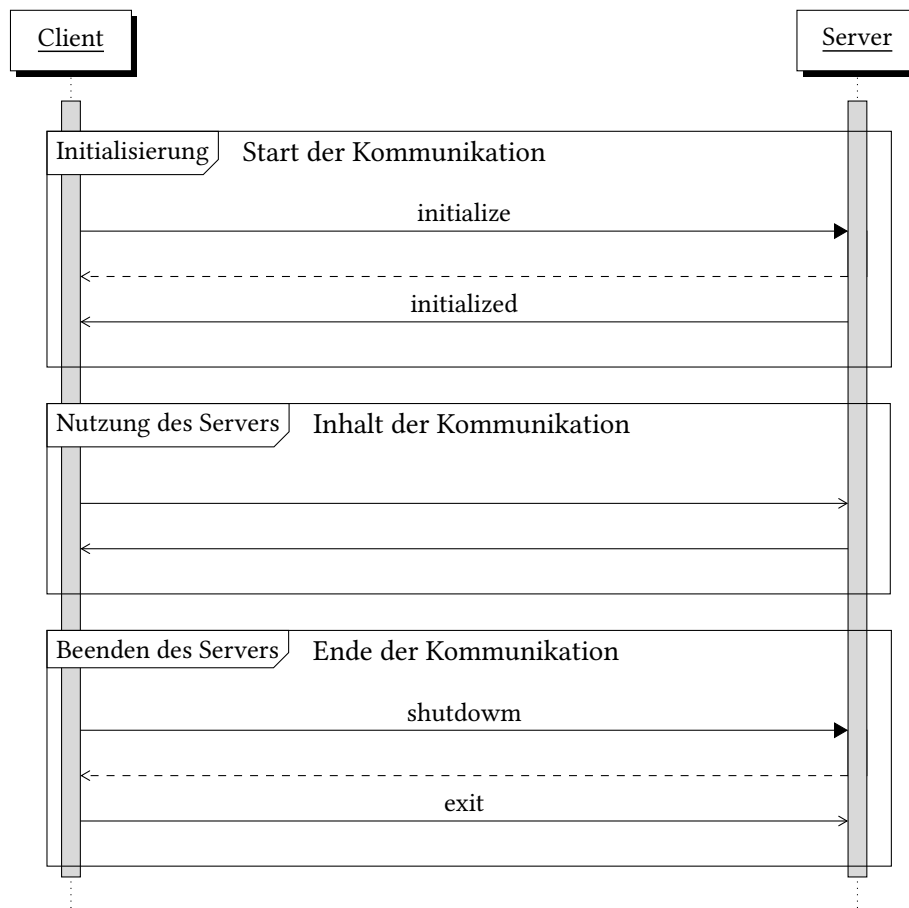
```
1 interface NotificationMessage extends Message {
2     method: string;
3     params?: array | object;
4 }
```

Benachrichtigungen übermitteln Daten, ohne eine Antwort zu erwarten. Sie werden für z.B. den Abbruch einer Anfrage oder dem Übermitteln der aktuellen Fehler genutzt.

Das Datenmodell einer Benachrichtigung unterscheidet sich von einer Anfrage nur durch die fehlende ID.

Der LSP-Kommunikation-Lebenszyklus

Die Kommunikation eines Clients mit einem Server folgt einem Ablauf, welcher den Anfang und das Ende der Kommunikation definiert. Der Start eines Vorgangs im Lebenszyklus wird immer vom Client gestartet.



Initialisierung

Die Initialisierung umfasst die ersten Nachrichten, nachdem die Transportschicht die Verbindung etabliert hat.

Sie beginnt mit dem Aufruf der Methode 'initialize' des Servers. In den Parametern des Aufrufs werden die vom Client unterstützten Funktionen des *LSP*s übermittelt. Der Server antwortet mit der Auswahl aus den vom Client gesendeten Funktionen, die auch der Server unterstützt. Die Unterstützung einer Funktion muss keine Entscheidungsfrage sein. Server und Client übermitteln auch wie sie Funktionen unterstützen. So können Client oder Server kommunizieren, dass sie z.B. die Veränderungen an einer Datei und/oder die vollständigen Dateien unterstützen. (Für genauere Informationen siehe Abschnitt ??)

Nachdem die Feinheiten der Kommunikation bekannt sind, beginnt der Server mit den Vorbereitungen, um alle vereinbarten Funktionen verarbeiten zu können. Ist diese Vorbereitung abgeschlossen, sendet der Server die 'initialized' Benachrichtigung. Nun ist der Start der Kommunikation ist erfolgreich abgeschlossen.

Beenden des Servers

Wird der *LSP*-Server nicht mehr vom Client benötigt, z.B. da keine relevanten Dateien mehr geöffnet sind oder die *IDE*/das Projekt geschlossen wird, so wird die Methode 'shutdown' aufgerufen. Die Methode erwartet leere Parameter und beendet alle Strukturen für die Verarbeitung der *LSP*-Funktionen.

Sobald der Client die Kommunikation beendet, sendet er eine 'exit'-Benachrichtigung. Hat der Server vorher eine 'shutdown'-Anfrage erfolgreich beantwortet, so soll er mit dem Exit-Code 0 beenden. Ansonsten soll er sich mit dem Exit-Code 1 beenden.

Der Exit-Code kann nur vom Client gelesen werden, wenn dieser den Prozess des Servers auch gestartet hat.

In den folgenden Abschnitten werden die Funktionen die der DMF-LSP-Server unterstützt vorgestellt.

Anfragen stornieren

Cancellation Support (\$/cancelRequest) [**cancellation**]

Der Client kann jederzeit entscheiden, dass das Ergebnis einer Anfrage nicht mehr benötigt wird.

Definition der Parameter [**cancellation**]

```
1 interface CancelParams {
2     id: integer | string;
3 }
```

Die Parameter enthalten die ID der Anfrage, dessen Ergebnis nicht mehr benötigt wird. Der Server muss jedoch die Anfrage trotzdem noch beantworten, um dem Ablauf des Protokolls zu folgen.

Diese Funktion wird im subsubsec:cancel-service implementiert.

Dokumenten Synchronisation

Text Document Synchronization

(textDocument/didOpen, textDocument/didChange, textDocument/didClose) [**dokumente**]

Damit ein LSP-Server Aussagen über eine Datei oder ein ganzes Projekt treffen kann, muss der Server den aktuellen Stand der Dateien kennen.

Während der Initialisierung wird vom Server festgelegt, ob das Öffnen und Schließen von Dateien an den Server übertragen und wie die Bearbeitungen an den Server übertragen werden sollen (gar nicht/komplette Datei/inkrementell). Der DMF-LSP-Server nutzt sowohl das Öffnen und Schließen der Dateien als auch die inkrementelle Übertragung der Bearbeitungen.

Die Synchronisation beginnt, mit der Methode 'textDocument/didOpen', welche beim Öffnen einer Datei ausgeführt wird.

Definition der Parameter [**dokumente**]

```
1 interface DidOpenTextDocumentParams {
2     textDocument: TextDocumentItem;
3 }
4 interface TextDocumentItem {
5     uri: DocumentUri;
6     languageId: string;
7     version: integer;
8     text: string;
9 }
```

Die Parameter der Methode enthalten die Uniform Resource Identifier (URI), den Inhalt der Datei, die Sprache und die Version. Von diesen Parametern sind die URI und der Inhalt sehr bedeutend. Sie bestimmen welcher Inhalt unter welcher URI vom Server gespeichert und verarbeitet wird.

Nachdem ein Dokument geöffnet wurde, wird bei jeder Änderung die Methode 'textDocument/didChange' aufgerufen. Die Parameter enthalten neben der URI die Änderungen, welche die direkt in die Bearbeitungen der Treesitter-API übersetzt werden können.

Definition der Bearbeitungen (ohne irrelevante Elemente)[**dokumente**]

```
1 export type TextDocumentContentChangeEvent = {
2     range: Range;
3     text: string;
4 }
```

Wird eine Datei geschlossen, können im Server alle Ressourcen für die Datei freigegeben werden. Dafür ruft der Client die Methode ‘textDocument/didClose’ auf.

Die Funktionen für die Dokumenten Synchronization werden im ?? implementiert.

Referenzen bestimmen

Go to Declaration & Find References

(textDocument/declaration, textDocument/references) [**declaration**] [**references**]

Referenzen sind ein großer Teil des Typsystems des DMFs. Damit diese Referenzen auch nachvollziehbar für die Entwickler*innen sind, bietet das *LSP* mehrere Funktionen an.

Die Deklaration eines Typs kann mit der Methode ‘textDocument/declaration’ abfragt werden.

Listing 4.1: Definition der Deklarations-Parameter [**declaration**]

```

1 export interface DeclarationParams extends TextDocumentPositionParams,
2   WorkDoneProgressParams, PartialResultParams {
3 }
4 interface TextDocumentPositionParams {
5   textDocument: TextDocumentIdentifier;
6   position: Position;
7 }

```

Die Deklaration wird als LocationLink als Ergebnis bereitgestellt.

Definition des LocationLink [**declaration**]

```

1 interface LocationLink {
2   originSelectionRange?: Range;
3   targetUri: DocumentUri;
4   targetRange: Range;
5   targetSelectionRange: Range;
6 }

```

Ein LocationLink beschreibt einen Bereich in einem, auch vom aktuellem Dokument unterschiedlichem, Dokument. Dabei wird zwischen dem kompletten Bereich der Deklaration und dem Bereich welcher automatisch ausgewählt und in einer Auswahl angezeigt werden soll.

Alle Referenzen zu einem Typ können mit der Methode ‘textDocument/references’ abgefragt werden. Zu den Referenzen gehören die Deklarationen und die Verwendung des Typs in Referenzen, Multi-referenzen, Funktionen und Abstraktionen.

Die Parameter unterscheiden sich von den der Deklaration nur im ReferenceContext. Dieser beinhaltet die Information, ob die Deklaration in der Antwort enthalten sein soll.

Definition des ReferenceContext [**references**]

```

1 export interface ReferenceContext {
2   includeDeclaration: boolean;
3 }

```

Im Ergebnis werden die Referenzen nicht in einem LocationLink zurückgegeben, sondern nur in einer Location. Diese Location enthält nur die *URI* der Datei und den Bereich der Referenz.

Definition der Location [**references**]

```

1 interface Location {
2   uri: DocumentUri;
3   range: Range;
4 }

```

Die Beschreibung der Implementierung beider Methoden befindet sich im Abschnitt ??.

Hover-Effekt

Hover (textDocument/hover) [**hover**]

Das **LSP** bietet die Funktion Informationen über ein Element bereitzustellen, wenn die Entwickler*innen über den Text "hovern".

Während der Initialisierung gibt der Client die Formate an, die er für die Dokumentation unterstützt. Das **LSP** beinhaltet zwei Formate in der Spezifikation: normaler Text und Markdown.

Die Parameter der Anfrage erben von den TextDocumentPositionParams (siehe ??). Sie enthalten die **URI** der Datei und die Position.

Die Dokumentation wird zusammen mit einem optionalen Bereich übermittelt.

Listing 4.2: Definition des Hover Ergebnis [**hover**]

```
1 export interface Hover {  
2   contents: MarkedString | MarkedString[] | MarkupContent;  
3   range?: Range;  
4 }  
5 export interface MarkupContent {  
6   kind: MarkupKind;  
7   value: string;  
8 }  
9 export type MarkupKind = 'plaintext' | 'markdown';
```

Die Implementierung des Hover-Effekts wird im Abschnitt ?? beschrieben.

Faltbereich

Folding Ranges (textDocument/foldingRange) [**folding**]

Die Möglichkeit den Code in Abschnitte zu unterteilen und diese einfalten zu können, erleichtert die Übersicht in großen Dateien. Deshalb definiert das **LSP** eine Funktion, um diese Bereiche an die **IDE** zu übermitteln.

Während der Initialisierung kann der Client viele Vorgaben und Wünsche an den Server machen. Dazu zählen die gewünschte maximale Anzahl der Bereiche, ob nur komplette Zeilen gefaltet werden können, welche Faltpyten unterstützt werden und ob vom Server generierte Zusammenfassungen angezeigt werden können.

Die Anfrage an den Server beinhaltet nur die **URI** der Datei.

Die Antwort des Servers enthält eine Liste mit Faltbereichen. Die Faltbereiche decken die komplette Datei ab. Jeder Faltbereich enthält zusätzlich zur Startposition und Endposition auch den Faltpyten und optional auch eine Zusammenfassung.

Die Implementierung der Faltbereiche wird im Abschnitt ?? beschrieben.

Auswahlbereich

Selection Range (textDocument/selectionRange) [**selection**]

Durch die unterschiedlichen Grammatiken aller Programmiersprachen ist eine Verallgemeinerung der Auswahlbereiche in einem Dokument unmöglich. Deshalb bietet das **LSP** die Möglichkeit diese Bereiche vom Server abzufragen.

Die Anfrage beinhaltet ein Dokument und verschiedene Positionen, zu denen die Auswahlbereiche erfragt werden. Der Vorteil von mehreren Positionen ist die Bündlung der Anfragen für Editoren mit mehreren Eingabemarken(Cursor).

Auszug aus den Parametern

```
1 type SelectionRangeParams struct {
2     TextDocument protokolll.TextDocumentIdentifier `json:"textDocument"`
3     Positions []protokoll.Position `json:"positions"`
4 }
```

Auswahlbereiche bilden einen Auszug aus dem AST. Um eine Auswahl zu vergrößern oder zu verkleinern wird der höhere bzw. tiefere Auswahlbereich aus der Hierarchie des ASTs benötigt. Deshalb beinhaltet die Antwort im LSP auch die Möglichkeit pro Position eine Kette an Auswahlbereichen zu liefern. Diese Kette wird durch das Parent-Attribut gebildet.

Antwort des Servers

```
1 type SelectionRange struct {
2     Range protokolll.Range `json:"range"`
3     Parent *SelectionRange `json:"parent,omitempty"`
4 }
5 type SelectionRangeResult []SelectionRange
```

Die Implementierung der Auswahlbereiche wird im Abschnitt ?? beschrieben.

Semantische Tokens

Semantic Tokens (textDocument/semanticTokens/full) [semantic]

Um ein schnelles Verständnis einer Datei zu ermöglichen, ist die Einfärbung der Syntax und Semantik wichtig. Dafür stellt das LSP die Möglichkeit bereit semantische Tokens/Symbole zu übermitteln.

Ein Token bezieht sich immer auf einen Bereich im Sourcecode und übermittelt einen Tokentyp und eine Auswahl der Tokenmodifikatoren. Die Tokentypen und Tokenmodifikatoren werden während der Initialisierung übermittelt.

Ein Client kann Anfragen für die semantischen Token an den Server stellen. Für das DMF wurde nur die Methode 'textDocument/semanticTokens/full' welche alle semantischen Token für eine Datei generiert beachtet.

Die Codierung des semantischen Token

Wenn die neuen semantischen Tokens übermittelt werden, werden die Tokens mithilfe einer Zahlenfolge codiert. Dies führt zu einer starken Komprimierung des Ergebnisses, welche besonders relevant für diese Methode des LSPs ist, da das Ergebnis besonders viele Daten beinhaltet.

Die Einträge der Zahlenfolge werden nach dem folgenden Schema codiert:

Index in der Zahlenfolge für Token mit Index i	Name	Erklärung
5*i	deltaLine	Die Zeilen zwischen dem letzten Token und diesem Token.
5*i+1	deltaStart	Die Zeichen zwischen dem letzten Token und diesem Token. Relativ zu 0, falls der aktuelle Token in einer neuen Zeile ist.
5*i+2	length	Die Länge des Tokens.
5*i+3	tokenType	Index des Typs des Tokens in der Semantik Token Typ Legende.
5*i+4	tokenModifiers	Zahl deren Bits als Wahrheitswerte für jeden Modifikator aus der Legende der Semantik Token Modifikatoren. Der erste Bit (0b00000001) steht dabei für den ersten Modifikator.

Die Implementierung der semantischen Tokens wird im Abschnitt ?? beschrieben.

Diagnosen

Publish Diagnostics (textDocument/publishDiagnostics) [**diagnostics**]

Um Fehler, Warnungen, Informationen und Hinweise den Entwickler*innen anzeigen zu können bietet das LSP die Möglichkeit Diagnosen zu übermitteln. Hierbei ist die Besonderheit, dass nur der Server weiß, ob und wann die Diagnosen einer Datei sich verändern. Deshalb bietet das LSP die Möglichkeit, dass der Server Benachrichtigungen an den Client mit den Diagnosen sendet.

Während der Initialisierung kann der Client (neben der Unterstützung von Feinheiten der Spezifikation) angeben, ob er Diagnosen-Tags (eng. diagnostic tag) unterstützt. Diese Tags differenzieren die Diagnosen, wie die semantischen Modifikatoren die semantisch Tokens weiter differenzieren.

Die Benachrichtigung beinhalten eine Liste mit Diagnosen. Jede Diagnose bezieht sich auf einen Bereich im Sourcecode. Eine Diagnose kann ein Fehler, eine Warnung, eine Information oder ein Hinweis sein. Der Unterschied zwischen einer Information und einem Hinweis liegt Bedeutsamkeit der enthaltenen Information. Eine Diagnose mit einer Information sollte beim Ermitteln eines Fehlers vor den Hinweisen beachtet werden.

Der Inhalt einer Diagnose setzt sich aus den Feldern 'Message' und 'Source' zusammen. 'Message' beschreibt den Fehler und 'Source' den Grund.

Es können auch zusätzliche Informationen in einer Diagnose enthalten sein. Diese könne andere Stellen im Code erwähnen, die für die Diagnose bedeutend sind.

Listing 4.3: Definition des Inhalts der Benachrichtigung

```
1 type Diagnostic struct {
2     Range Range `json:"range"`
3     Severity *DiagnosticSeverity `json:"severity,omitempty"`
4     Source string `json:"source,omitempty"`
5     Message string `json:"message"`
6     Tags []DiagnosticTag `json:"tags,omitempty"`
7     RelatedInformation []DiagnosticRelatedInformation `json:"relatedInformation,omitempty"`
8 }
9 type DiagnosticRelatedInformation struct {
10     Location Location `json:"location"`
11     Message string `json:"message"`
12 }
```

Die Implementierung für die Erstellung der Diagnosen wird im Abschnitt ?? beschrieben.

4.1.2 Server Implementierung

Die Serverschicht abstrahiert die Verbindung zum Client und das JSON-RPC Protokoll. So bekommen die Services direkt Nachrichten und können auch Nachrichten verschicken.

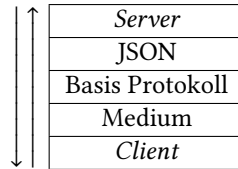
Abstraktion der Server-Client-Verbindung

Da das LSP Medium unabhängig ist, muss der Server eine Abstraktion für die Verbindung bereitstellen.

```
1 package connect
2 type Connection interface {
3     WriteMessage(message protokoll.Message)
4     WaitForMessage() (protokoll.Message, error)
5     BlockResponse(id json.RawMessage)
6     Close() error
7 }
```

Das Interface Connection stellt diese Abstraktion bereit. Es wird für jeden Verbindungstyp implementiert. Zwischen den Verbindungstypen unterscheiden sich nur die Implementierungen der Interfaces der Standardbibliothek.

Die Verbindung durchläuft immer die gleichen Schritte, nur die Richtung unterscheidet sich bei Lesen und Schreiben.



JSON

Damit die Nachrichten übermittelt werden können, müssen sie in JSON Text geparsed werden. Dafür wird die Standardbibliothek von Golang genutzt.

Listing 4.4: Umwandlung von und zu JSON

```
1 message := protokoll.Message{}
2 // Golang -> JSON
3 data, err := json.Marshal(message)
4 // JSON -> Golang
5 err = json.Unmarshal(data, &message)
```

Das Verhalten der verschiedenen Datenstrukturen beim Parsen wird direkt bei der Definition angegeben. Deshalb muss keine zusätzlich Logik für Werte die nicht übertragen werden soll, falls sie leer sind, oder für unterschiedliche Namen im JSON-Text implementiert werden.

Basis Protokoll

Das *LSP* besitzt nicht Spezifikationen für den Inhalt der Nachrichten, sondern auch für deren Übertragung. Vorausgesetzt wird nur eine bidirektionale Verbindung, welche parallel Text übertragen kann.

Jede Nachricht im Basis Protokoll besteht aus zwei Teilen: dem Header und dem Inhalt.

Der Header enthält Angabe zum Einlesen. Diese werden jeweils in eine eigene Zeile geschrieben. Die Zeilen werden mit den Kontrollzeichen '\r\n' beendet.

Variable	Beschreibung
Content-Length	Die Länge der übertragenen Nachricht (in Bytes).
Content-Type	Der -Typ der Nachricht. Wenn nicht angegeben: 'application/vscode-jsonrpc; charset=utf-8'

[header]

Der Header wird beendet einer leeren Zeile welche dennoch mit beiden Kontrollzeichen beendet wird. Zuletzt folgt der Inhalt der Nachricht. Dabei ist wichtig, dass die angegebene Länge genau der Länge der Nachricht entspricht.

Um das Basis Protokoll Lesen zu können, wird der 'BufReader' der Golang-Standardbibliothek verwendet. Beim Lesen aus dem 'BufReader' kann ein Zeichen bis zu dem gelesen werden soll, angegeben werden. So kann die Verbindung Zeilenweise ausgelesen werden.

Beginnt eine Zeile mit dem Identifier der Länge der Datei so wird der Rest der Zeile als Zahl interpretiert und zwischengespeichert.

Wird eine leere Zeile gelesen wird die Schleife zum Auslesen des Headers beendet.

Nun wird ein Buffer mit der gespeicherten Größe angelegt und mit dem Inhalt aus der Verbindung befüllt. Der Inhalt des Buffers kann nun als Text weiter verwendet werden.

Das Schreiben im Basis Protokoll beginnt mit Messung der Nachrichtenlänge. Nun kann der Header in die Verbindung geschrieben werden. Schließlich wird der Inhalt in die Verbindung geschrieben.

Medium

Der DMF-LSP-Server unterstützt zwei Medien: Standard-IO und TCP-Sockets.

Standard-IO bietet sich für die meisten Fälle an, da der Prozess häufig von der IDE verwaltet wird. Diese Übertragung enthält auch keine weiteren Schichten und ist somit sehr schnell. Sie vereinfacht auch den Server Prozess, da es nur einen Client geben kann.

Listing 4.5: Start eines StdIO-Servers

```
1 newServer := server.NewServer(connect.NewStdIOConnection())
2 newServer.MessageLoop()
```

Wenn der Prozess nicht von der ide verwaltet werden kann und der Server zentral gehostet werden soll, können TCP-Sockets genutzt werden. Sie ermöglichen mehreren Nutzern eine Server Instanz zu nutzen und die Nutzung eines Servers, welcher nicht auf dem gleichen Computer läuft.

Bei der Implementierung ist die Unterscheidung zwischen dem Begriff Server und der Datenstruktur 'Server' zu beachten. Für jede Verbindung wird eine eigene Server-Instanz in einer eigenen Routine erzeugt.

Listing 4.6: Verwaltung des TCP-Servers

```
1 listener, err := net.Listen("tcp", args.Port)
2 for {
3     conn, err := listener.Accept()
4
5     // Server starten
6     go func(conn net.Conn) {
7         newServer := server.NewServer(
8             connect.NewSocketConnection(conn))
9         newServer.MessageLoop()
10    }(conn)
11 }
```

Abstraktion des JSON-RPC-Protokolls

Die Abstraktion des JSON-RPC-Protokolls verbindet die Services und die Verbindung zum Client. Sie beinhaltet zwei Phasen: Intialisierung und Verarbeitung.

Die Intialisierung erstellt alle Services und trägt diese eine Map ein. Die Map enthält die Referenzen zwischen den Methodennamen und den Services. Die Methodennamen liefern die Services, so kann ein Service mit nur zweilen hinzugefügt werden:

```
1 newCancelService := cancelService.NewCancelService(con)
2 s.addHandler(newCancelService)
```

Die Verarbeitung beginnt mit Lesen der 'initialize'-Methode. Diese wird an die Services weitergereicht, welche währenddessen die Antwort befüllen.

Nachdem die Kommunikation Intialisiert wurde, beginnt der Nachrichtenkreislauf.

Listing 4.7: Nachrichtenkreislauf

```
1 MessageLoop:
2     for {
3         if s.nextMethod(logger) {
4             break MessageLoop
5         }
6     }
```

Zuerst wird der blockierende Aufruf der Kommunikation durchgeführt.

Die Verarbeitung wird mit Zuordnung der Methode fortgeführt. Vor den Services muss auf die 'shutdown'-Methode überprüft werden. Die Verarbeitung der LSP-Lifecycle-Nachrichten fällt in den Aufgabenbereich

des Servers. Sollte die 'shutdown'-Anfrage vorliegen, wird der Message-Loop beendet. Somit wird schließlich der Server herunter gefahren.

Für alle anderen Methoden wird die Map der Services genutzt, um den richtigen Service zu finden. Die Verarbeitung wird im Service fortgeführt. Dafür wird eine neue Routine gestartet, so kann der Server schon während der Verarbeitung die nächste Nachricht einlesen.

Listing 4.8: Verarbeitung einer Nachricht

```

1 package server
2 func (s *Server) nextMethod(logger *log.Logger) bool {
3     message, err := s.con.WaitForMessage()
4
5     if err != nil {
6         logger.Printf("%sError While Reading Message: %e\n", logService.ERROR, err)
7         return false
8     } else {
9         // Lifecycle Methods
10        switch message.Method {
11        case "shutdown":
12            logger.Printf("%sReceived Shutdown Request\n", logService.INFO)
13            s.con.WriteMessage(protokoll.Message{
14                JsonRPC: "2.0",
15                ID:      message.ID,
16                Method: "",
17                Params: nil,
18                Result: nil,
19                Error:  nil,
20            })
21            return true
22        }
23        handler, found := s.methodMap[message.Method]
24        if !found {
25            logger.Printf("%sMethod not Found: %v\n", logService.ERROR, message)
26            connectUtils.WriteErrorToCon(s.con, message.ID, errors.New("method not found"), protokoll.MethodNotFound)
27        } else {
28            logger.Printf("%sHandle Method: %s\n", logService.INFO, message.Method)
29            go handler.HandleMethod(message)
30        }
31    }
32    return false
33 }

```

4.1.3 Die LSP-Services

Die Schnittstelle für alle Services zu der Server-Schicht bildet das MethodHandler-Interface:

```

1 package service
2 type MethodHandler interface {
3     Initialize(params *initialize.InitializeParams,
4               result *initialize.InitializeResult)
5     GetMethods() []string
6     HandleMethod(message protokoll.Message)
7 }

```

Jeder Service implementiert die drei Methoden.

1. **Initialize**

Der Service liest die Fähigkeiten des Clients und konfiguriert sich selbständig. Sollte der Client den Service nicht unterstützen, muss er sich deaktivieren. Der Service schreibt seine Fähigkeiten in die Antwort des Servers.

2. **GetMethods**

Gibt die Methoden aus dem LSP zurück für die der Service Meldungen verarbeitet.

3. **HandleMethod**

Verarbeite die Nachricht.

FileService

4 Programme des DMF

Protokoll in Abschnitt ??

Der FileService ist die Schnittstelle zwischen den Dateien, den Parsern und den restlichen Services. Wenn ein anderer Service auf Dateien, semantische Modell oder den Lookup zugreifen möchte, werden Methoden des FileServices genutzt.

```
1 package fileService
2 type FileService struct {
3     handleMap map[string]*fileHandle
4     listeners []FileChangeListener
5     con       connect.Connection
6 }
```

In der “handleMap” werden “fileHandles” gespeichert. Ein FileHandle speichert alle Daten zu einer Datei und wird mit jeder Veränderung aktualisiert.

```
1 package fileService
2 type fileHandle struct {
3     FileContent string
4     Ast          *tree_sitter.Tree
5     Model        *smodel.Model
6     LookUp       *smodel.TypeLookUp
7     Version      int32
8 }
```

Um einen FileHandle zu erzeugen, wird der Inhalt der Datei mithilfe Semantik-Schicht geparkt. Wird der Dateiinhalt geändert, so werden die Änderungen an die Semantik-Schicht übergeben. Dort werden die Änderungen zum iterativen Parsen des neuen Dateiinhalts genutzt. Abschließend werden der Lookup erzeugt und die semantischen Regeln durchlaufen.

FileChangeListener Es gibt Funktionen im LSP die nicht durch eingehende Nachrichten ausgelöst werden, sondern nach Dateiänderungen automatisch an den Client übermittelt werden. Dafür gibt es im DMF-LSP-Server die FileChangeListener.

```
1 package fileService
2 type FileChangeListener interface {
3     HandleFileChange(file protokoll.DocumentURI, fileContent string,
4                     ast *tree_sitter.Tree, model *smodel.Model,
5                     lookup smodel.TypeLookUp,
6                     errorElements []errElement.ErrorElement,
7                     version int32)
8 }
```

Der FileService enthält Referenzen zu allen aktiven Listener. Nachdem ein FileHandle erstellt oder bearbeitet wurde, werden alle Listener durchlaufen.

Im DMF-LSP-Server ist nur ein FileChangeListener implementiert:

Der DiagnosticsService (Protokoll im Abschnitt ??) übermittelt die aktuellen Fehler in der Modelldatei an den Client. Dafür werden alle ErrorElemente in die Diagnostic Strukturen des LSPs übersetzt. Schließlich werden die Daten mithilfe einer Request-Nachricht für die Methode “textDocument/publishDiagnostics” an den Client übermittelt.

SemanticTokensService

Protokoll in Abschnitt ??

Der SemanticTokensService implementiert die Methode für die semantischen Tokens. Diese Tokens werden für die Einfärbung des Textes einer Datei nach der Syntax und Semantik genutzt. Mit dieser Einfärbung des Sourcecodes können Entwickler*innen schnell ein Verständnis der Datei entwickeln.

Ein Token bezieht sich immer auf einen Bereich im Sourcecode und übermittelt einen Tokentyp und eine

Auswahl der Tokenmodifikatoren. Die Tokentypen und Tokenmodifikatoren werden während der Initialisierung übermittelt. Dabei ist für die Codierung (siehe ??) der Index in den Listen entscheidend. Für das *DMF* wird nur die folgende Auswahl der Tokenmodifikatoren genutzt.

1	declaration
2	definition

Die Tokentypen und ihre Verwendung zusammen mit den Tokenmodifikatoren

Index	Token Typ	Verwendung im DMF	
		Tokenmodifikator(-en)	Verwendung
0	namespace	declaration	Für den Namen eines Packages.
1	type	-	Für das AST-Element "reftype". Für den Override Wert des Java Typens.
2	class	-	Für den Override Wert der Java Klasse, der Oberklasse und der implementierten Interfaces.
3	enum	declaration	Für den Namen eines Enums.
4	interface	declaration	Für den Namen eines Interfaces.
5	struct	declaration	Für den Namen eines Structs oder einer Entity.
6	parameter	declaration	Für die Parameter einer Funktion.
7	variable	-	Für die Namen der Variablen im Entity Identifier. Für den Override Wert des Java Namens.
		declaration	Für die Namen von Argumenten, Referenzen und Multi-Referenzen.
8	property	Platzhalter	
9	number	definition	Für alle Zahlenwerte in einer Enum-Konstante.
10	enumMember	declaration	Für den Namen einer Enum-Konstante.
11	function	declaration	Für den Namen einer Funktion.
12	comment	-	Für alle Kommentare.
13	keyword	-	Für alle Keywords.
14	string	-	Für alle Strings außer den Werten im Override.
15	modifier	Platzhalter	
16	decorator	-	Für den Override Wert der Java Annotations.

Generierung der Semantik Tokens

Die Semantik Tokens werden mithilfe zweier Algorithmen generiert.

Mithilfe des *ASTs* werden alle *AST*-Elemente durchlaufen. Werden Elemente erreicht, deren semantischer Token allein am *AST* Element bestimmt werden kann, so werden die Token generiert. Dazu gehören die meisten semantischen Token.

Mithilfe des semantischen Modells werden die restlichen Token bestimmt. Dies ist möglich da die Elemente im semantischen Modell Referenzen zum *AST* beinhalten. Dazu gehören die Namen der verschiedenen Inhalte, wie z.B. von Argumenten. Bei diesen Namen wird das gleiche *AST*-Element genutzt, wodurch sie nur das semantische Parsen unterscheidbar sind.

4 Programme des DMF

Die semantischen Tokens befinden sich nun in einer Liste mit semantischen Elementen. Diese müssen für die Antwort codiert werden.

Listing 4.9: semantisches Element

```
1 type semanticElement struct {  
2     line      uint32  
3     start     uint32  
4     length    uint32  
5     tokenType uint32  
6     tokenModifiers uint32  
7 }
```

Die Codierung der semantischen Tokens muss zunächst die generierten Tokens sortieren, da keine Garantie für die richtige Reihenfolge durch die beiden Algorithmen existiert.

```
1 slices.SortFunc(semanticElements, func(a, b *semanticElement) int {  
2     if a.line == b.line {  
3         return cmp.Compare(a.start, b.start)  
4     }  
5     return cmp.Compare(a.line, b.line)  
6 })
```

Nach der Sortierung können die Tokens durchlaufen werden.

```
1 data := make([]uint32, len(semanticElements)*5)  
2  
3 lastLine := uint32(0)  
4 lastStart := uint32(0)  
5  
6 for i, element := range semanticElements {  
7     line := element.line  
8     start := element.start  
9     length := element.length  
10    tokenType := element.tokenType  
11    tokenModifiers := element.tokenModifiers  
12    if line < lastLine {  
13        continue  
14    }  
15  
16    if line == lastLine && start < lastStart {  
17        continue  
18    }  
19  
20    // Calculate delta encoding  
21    deltaLine := line - lastLine  
22    deltaStart := uint32(0)  
23    if line == lastLine {  
24        deltaStart = start - lastStart  
25    } else {  
26        deltaStart = start  
27    }  
28  
29    data[i*5] = deltaLine  
30    data[i*5+1] = deltaStart  
31    data[i*5+2] = length  
32    data[i*5+3] = tokenType  
33    data[i*5+4] = tokenModifiers  
34  
35    lastLine = line  
36    lastStart = start  
37 }
```

Beim Durchlaufen wird ein Integer-Slice erstellt.

Für jeden Token werden die 5 Zahlen nach dem Protokoll hinzugefügt.

Überschneiden sich Token oder sind nicht in der richtigen Reihenfolge werden die Token ignoriert.

Die aktuelle Zeile und Spalte im Text wird nach jedem Token aktualisiert.

SelectionRangeService

Protokoll in Abschnitt ??

Der SelectionRangeService implementiert die LSP-Methode "textDocument/selectionRange". Diese wird dafür genutzt, dass bei der Auswahl immer der richtige Text ausgewählt wird.

Die Bereiche werden im Service mithilfe des ASTs berechnet.

```

1 package selectionRangeService
2 func (s *SelectionRangeService) computeSelectionRangeForPosition(
3     content fileService.FileContent,
4     position protokoll.Position) *selectionRange.SelectionRange {
5     finder := util.NewNodeFinder([]byte(content.Content))
6     node := finder.FindSmallestNodeAroundPosition(content.Ast, position)
7     var currentRange *selectionRange.SelectionRange
8     for node != nil {
9         newRange := &selectionRange.SelectionRange{
10             Range: protokoll.ToRange(node.Range()),
11         }
12         newRange.Parent = currentRange
13         currentRange = newRange
14
15         node = node.Parent()
16     }
17     return currentRange
18 }

```

Es wird zunächst das kleinste Element (das “Blatt” des Baums) des *ASTs* für die Position Code bestimmt. Nun wird durch die Schichten des *ASTs* bis zur Wurzel iteriert. Bei jedem Element eine neue “SelectionRange” Instanz angelegt. Dieses “SelectionRange” enthält immer den Bereich und eine Referenz zu der vorherigen “SelectionRange”. Durch die Referenz wird die Baum-Struktur auch im Ergebnis übermittelt und es müssen weniger Anfragen an den Server gesendet werden.

Ein Request kann mehrere Positionen enthalten. Deshalb wird der Algorithmus für jede Position ausgeführt.

FoldingService

Protokoll in Abschnitt ??

Der FoldingService stellt die Methode ‘textDocument/foldingRange’, für die Übermittlung der faltbaren Bereiche, bereit. Es werden drei verschiedene Bereicharten berechnet: PackageElemente, Kommentare und der Import-Block.

Für die PackageElemente werden die *AST*-Nodes des Elements durchlaufen. Es wird immer der Bereich, welcher von den geschweiften Klammern umschlossen wird, gefaltet. Deshalb werden die auch diese Nodes beim Durchlaufen gesucht.

Listing 4.10: Suche nach Start- und Ende-Node des Faltbereichs eines PackageElements

```

1 cursor.GotoFirstChild()
2 for {
3     if cursor.Node().GrammarName() == "{" {
4         startNode = cursor.Node()
5     }
6     if cursor.Node().GrammarName() == "}" {
7         endNode = cursor.Node()
8     }
9     if !cursor.GotoNextSibling() {
10         break
11     }
12 }

```

Mithilfe diese Nodes kann auf die Zeilen und Spalten beider Positionen zugegriffen werden. Damit die Klammern auch noch in der *IDE* angezeigt werden, wird der Startzeichenindex (Start-Spalte) um 1 erhöht und der Endzeichenindex (End-Spalte) um 1 gesenkt.

Für die Kommentare wird die Query-Funktion von Treesitter genutzt.

```

1 (comment_block) @cb

```

Die Ergebnisse einer Query werden in der Treesitter-API QueryMatches genannt. Für diese Query enthalten sie alle 'comment_block' Nodes im AST. Durch die Ergebnisse wird iteriert und für jeden QueryMatch ein Faltbereich erstellt. Dabei wird zwischen einzeiligen und mehrzeiligen Kommentaren unterschieden, die Position unterschiedlich interpretiert werden müssen. Einzeilige Kommentare werden nicht in jeder IDE gut verarbeitet, deshalb können sie über die Argumente ausgeschaltet werden.

Listing 4.11: Auszug aus der Bestimmung der Faltbereiche für die Kommentare

```
1 queryCursor := tree_sitter.NewQueryCursor()
2 captures := queryCursor.Captures(f.kommentarQuery,
3     root.RootNode(), nil)
4 for match, index := captures.Next(); match != nil;
5     match, index = captures.Next() {
6     node := match.Captures[index].Node
7     startPos := node.StartPosition()
8     endPos := node.EndPosition()
9
10    if startPos.Row != endPos.Row {
11        // Mehrzeilige Kommentare
12    } else if !args.DisableSingleLineCommentsFolding {
13        // Einzeilige Kommentare
14    }
15 }
```

Bei allen Kommentaren liegt das Ende im Ergebnis der Query in der nächsten Zeile. Der Faltbereich soll jeden am Ende der Kommentarzeile enden. Deshalb wird die Endposition verschoben.

Bei mehrzeiligen Kommentaren müssen Startzeichenindex und Endzeichenindex anhand von unterschiedlichen Zeilen bestimmt werden. Dafür muss der Inhalt des Kommentars in Zeilen aufgeteilt werden. Die erste und letzte Zeile werden für die Indexe genutzt. Für die Zusammenfassung wird die

Bei einzeiligen Kommentaren können die Indexe anhand der einen Zeile bestimmt werden.

Für den Import-Block wird die Node des Import-Blocks aus dem AST genutzt. Diese enthält jedoch alle Zeilen, ab dem ersten Import-Statement bis zum ersten PackageElement. Deshalb werden die Leerzeilen am Ende des Import-Blocks herausgefiltert.

Listing 4.12: Filterung der leeren Zeilen

```
1 var endPos tree_sitter.Point
2 for {
3     if cursor.Node().GrammarName() == "import_block" {
4         importBlock = cursor.Node()
5         cursor.GotoLastChild()
6         endPos = cursor.Node().EndPosition()
7         text := cursor.Node().Utf8Text([]byte(content))
8         split := strings.Split(text, "\n")
9         length := len(split)
10        for i := range split {
11            s := split[length-i-1]
12            if strings.ContainsFunc(s, func(r rune) bool {
13                switch r {
14                    case ' ', '\n':
15                        return false
16                    default:
17                        return true
18                }
19            }) {
20                endPos.Column = uint(len(split[length-i-1]))
21            } else {
22                endPos.Row--
23            }
24        }
25        break
26    }
27    if !cursor.GotoNextSibling() {
28        break
29    }
30 }
```

ReferenceService

Protokoll in Abschnitt ??

Um zwischen den Referenzen und den Deklarationen in der *IDE* in der Datei springen zu können, implementiert der ReferenzService die Methoden 'textDocument/references' und 'textDocument/declaration'. Die Parameter für beide Methoden enthalten das Textdokument und die Position im Dokument. Um die Referenz zu finden, wird der *AST* vom Blatt an der Position zur Wurzel durchlaufen. Dabei werden für folgende Node-Typen die erste Node gespeichert.

<i>AST</i> -Node Namen	Verwendung
reftype	Referenzen, Multireferenzen oder Parameter
identifier	Namen von Variablen oder Funktionen
package_block, struct_block, enum_block, entity_block, interface_block	PackageElemente

Mithilfe dieser Nodes kann der Name für den LookUp der PackageElemente und der Name des NamedElements innerhalb des PackageElements bestimmt werden.

Referenzen

Nachdem das PackageElement und der Name bestimmt wurden, wird unterschieden ob es sich um einen Typ oder um eine Variable handelt.

Für die **Variablen** wird das aktuelle Element und alle SubElemente (Elemente durch Abstraktion) durchsucht. In den Elementen wird nach der Variable über den NamensElemente-LookUp und in dem EntityIdentifier gesucht.

Für die Suche nach **Typ**-Referenzen müssen alle PackageElemente durchsucht werden. In den PackageElementen werden Referenzen, Multireferenzen, Parameter und die Abstraktion durchsucht.

Deklarationen

Auch bei den Deklarationen wird zwischen Typen und Variablen unterschieden.

Für die **Variablen** wird das Element und die SubElemente durchsucht. Für die **Typen** wird ???

HoverService

Protokoll in Abschnitt ??

Der HoverService lässt die Entwickler*innen Zusammenfassungen für jedes Element aufrufen. Um zu bestimmen, welche Zusammenfassung zu welchen Element übermittelt werden muss, wird zu Beginn jeder Hover-Anfrage die kleinsten Nodes verschiedener Typen bestimmt.

Zusammenfassung	<i>AST</i> -Node-Typen
Variable	arg_block, ref_block, multi_block
EntityIdentifier	identifier_statement
Konstante	enum_constant
Kommentar	comment_block
PackageElement	package_block, struct_block, enum_block, entity_block, interface_block
Import	import_statement
Model Deklaration	model_declaration
DMF Deklaration	dmf_declaration

Das Generieren der Zusammenfassungen folgt für jede Zusammenfassung dem gleichen Ablauf. Zuerst werden anhand der **AST**-Nodes die Elemente aus dem semantischen Modell bestimmt. Aus diesen Elementen wird eine spezifische Datenstruktur erstellt. Mithilfe dieser Datenstruktur kann das jeweilige Template ausgeführt werden.

Nutzung von Golang-Templates

Die Standard Bibliothek von Golang bietet eine Template-Engine das Generieren von Texten genutzt wird. Die **API** besteht dabei aus 3 Komponenten: Templates, Funktionen und Datenstrukturen.

Listing 4.13: Ablauf der Template-API

```
1 // Inkludierung der Templates
2 //go:embed template/*
3 var tmplFiles embed.FS
4
5 // Vorbereiten der Templates
6 templates = template.Must(template.New("").
7     Funcs(funcMap).
8     ParseFS(tmplFiles, "template/*.go"))
9
10 // Generieren mit Templates
11 buffer := bytes.NewBuffer(make([]byte, 0))
12 err := templates.ExecuteTemplate(buffer, "variable", data)
13 return buffer.String()
```

Die Templates werden in eigenen Dateien abgelegt. Mithilfe der Golang-Embed-API werden diese Dateien in die ausführbare Datei kopiert und können während der Ausführung ausgelesen werden.

Ein Template wird immer mit einer Datenstruktur aufgerufen. Innerhalb eines Templates werden Kontrollstrukturen und Zugriffe auf die Datenstruktur innerhalb von Statements mit '{' und '}' gekennzeichnet. Statements in Templates folgen einer simplifizierten Golang Syntax. Das Ergebnis nach ihrer Evaluierung wird in das Ergebnis eingefügt. Die größte Simplifizierung sind die Blöcke der Kontrollstrukturen. Die Beginnen immer mit einem Statement in der Syntax des jeweiligen Keywords und enden mit dem '{{end}}'-Statement.

Ein Template wird definiert mit dem Define-Statement ('{{define "name"}}'). Der Name im Statement wird später zum Aufrufen des Templates genutzt.

Die Übergebene Datenstruktur wird immer mit '.' adressiert. So kann eine Variable der Datenstruktur mithilfe von '{{Variablenname}}' eingefügt werden. Es ist auch möglich lokale Variablen zu definieren, z.B. '{{\$importSlice := getImports .}}'. Ein Name einer lokalen Variable beginnt immer mit dem Zeichen '\$'.

Funktionen werden ohne Klammern aufgerufen, jedoch können Klammern zur Abgrenzung der Parameter verschiedener Funktionen genutzt werden. Das Statement 'ne (len \$importSlice) 0' wird zu einem Wahrheitswert evaluieren. Der Aufruf der Funktion 'len' befindet sich in Klammern sodass das Ergebnis als ein Parameter der Funktion 'ne' verwendet wird.

Mithilfe von 'if' und 'range' Statements kann der Ablauf eines Templates gesteuert werden. Ein 'if'-Statement funktioniert als Kontrollstruktur die einen Teil des Templates abhängig von einer Bedingung ausführt. Es kann mit einem 'else'-Statement erweitert werden. Ein 'range'-Statement übernimmt die Funktion der Schleifen. Es wird immer durch eine Slice iteriert, z.B. '{{range \$index, \$use := .Uses}}'.

Es können andere Templates aufgerufen werden. Dabei kann nur ein Parameter(=eine Datenstruktur) übergeben werden, weshalb häufig Methoden mehrere Parameter in eine Datenstruktur kombinieren. Ein Aufruf Statement enthält neben dem Parameter auch den Namen des Templates: '{{template "parameter"\$element}}'.

Die Funktionen werden in Go geschrieben. Eine Map dient zum Aufruf der Funktionen während der Generationen. Zwischen Funktionen, Templates und Datenstrukturen wird während der Kompilierung und der Vorbereitung der Templates keine Typ-Überprüfung durchgeführt. Besitzen die Parameter während der Generation die falschen Typen, wird die Generation mit einem Fehler abgebrochen.

Funktionen besitzen keine Referenzen zum Kontext indem die Generation gestartet wurde. Deshalb bleiben nur noch die globalen Elemente und die Parameter. Globale Elemente können jedoch diesen Kontext auch nicht speichern, wenn die Generation parallel ausgeführt wird. Somit wird jeder erforderlicher Kontext in den Parametern übergeben.

Die Datenstrukturen werden normal im Go-Code definiert. Sie enthalten neben den Daten aus dem semantischen Modell auch den evtuellen Kontext für den Aufruf für eine Funktion oder bündeln Parameter für den Aufruf eines Templates.

Bei der Generation wird ein Writer, der Name des Templates und die Datenstruktur übergeben. Das Writer-Interface bietet eine Abstraktion, um Ziel und Implementierungs unabhängig Schreibvorgänge abbilden zu können. Um die Zusammenfassung später als String übergeben zu können, wird ein Buffer genutzt.

Die verschiedenen Zusammenfassungen werden im Abschnitt ?? beschrieben.

CancelService

Protokoll in Abschnitt ??

Der CancelService wird zum Abbruch von nicht mehr benötigten anfragen genutzt. Um eine Abstraktion über alle Services zu ermöglichen, wird der Abbruch direkt in der Abstraktion der Verbindung implementiert. Innerhalb der Verbindung wird gepsichert, ob eine Antwort für eine Anfrage versendet werden soll. Wurde die Antwort vom CancelService blockiert, so wird die Antwort ignoriert. Um zu identifizieren, welche Antworten blockiert wurden, wird die ID der Anfrage genutzt. Diese wird auch in den Parametern der Abbruch-Anfrage übergeben.

4.2 Der Generator

Der Generator wird während des Build-Vorgangs ausgeführt und übersetzt eine Modelldatei in das ausgewählte Generationsziel.

4.2.1 Der Aufbau

Der Generator besteht aus 3 Komponenten: das **Command Line Interface (CLI)**, das Generator Modell und die Generationsziele.

Generator CLI

Damit die Generation des **DMFs** vom Build Tool unabhängig ist, wird der Generator mithilfe eines **CLI** gestartet. Der Generator nimmt bei der Ausführung drei Parameter:

Parameter	Funktion
basePath	Der Pfad unter dem die generierten Dateien ausgegeben werden.
modelFile	Die Modelldatei deren Modell generiert werden soll.
mode	Das Generationsziel.

Nachdem diese Parameter eingelesen wurden, kann die Generation vorbereitet werden. Zuerst wird dafür die Modelldatei eingelesen und verarbeitet. Benötigt das Generationsziel das Datenbankmodell wird dieses aus dem semantischen Modell erzeugt. Sollten bei der Verarbeitung Fehler entstehen, so werden diese Ausgegeben und die Ausführung beendet. Die Generation wird für jedes Generationsziel nach dem gleichen Schema implementiert. Es wird zunächst durch alle Elemente entweder aus dem Lookup des semantischen Modells (??) oder aus dem Datenbankschema (??) iteriert. Für jedes Element wird die Methode 'generateFile' aufgerufen. Dies geschieht immer in einer eigenen Routine. Die Ausführung

wird erst beendet, nachdem alle Routinen beendet wurden. Dies wird mithilfe einer ‘Wait-Group’ verwaltet.

Listing 4.14: Die Methode generateFile

```
1 func generateFile(file *os.File, f func(writer io.Writer) error) {
2     if file != nil {
3         writer := bufio.NewWriter(file)
4         err := f(writer)
5         if err != nil {
6             panic(errors.Join(err, errors.New(file.Name())))
7         }
8         err = writer.Flush()
9         if err != nil {
10            panic(err)
11        }
12    }
13    operations.Done()
14 }
```

Durch die Abstraktion des Generierens des Dateinhalts über eine Funktion kann die Methode für alle Elemente genutzt werden. Um dieses Pattern zu erleichtern wird die Methode ‘apply’ genutzt.

Listing 4.15: Die apply-Methode

```
1 func apply[E any](f func(writer io.Writer, e E) error, data E) func(writer io.Writer) error {
2     return func(writer io.Writer) error {
3         return f(writer, data)
4     }
5 }
```

Mit dieser Funktion können Funktionen, die einen weiteren Parameter benötigen, zu einer Funktion verpackt werden, die für die generateFile-Methode genutzt wird. Die ‘apply’-Methode wird mit den Methoden der verschiedenen Templates der Generationsziele (siehe ??) genutzt.

Um die Dateien zu Erzeugen werden mehrere Funktionen verkettet. Die Funktionen ‘createFile’ und ‘createFileIfNotExists’ erstellen die Dateien und nutzen die Funktionen ‘buildJavaPath’, ‘buildTsPath’ oder ‘CreateDelegatePath’ um den Dateipfad für einen gegebenen ModelPath zu erzeugen.

Zusammen sieht der Aufruf für die Generation einer Entity mit dem Generationsziel Java so aus:

Listing 4.16: Generation einer Entity

```
1 go generateFile(createFile(basePath, element.Path, buildTsPath), apply(template.GenerateEntity, element))
```

Das Generator-Modell

Um die Entwicklung der Generationsziele zu vereinfachen werden, stellt das Generator-Modell verschiedene Komponenten und Funktionen bereit.

ImportKontext

Importe müssen für jede Datei berechnet werden. Deshalb stellt das Generator-Modell den ImportKontext und Funktionen zum Erstellen des Kontextes bereit.

Listing 4.17: ImportKontext

```

1  type ImportKontext struct {
2      ImportLookup ImportLookup
3      Path         base.ModelPath
4      HasDelegate  bool
5  }
6  type ImportLookup map[string]Import
7  type Import struct {
8      OriginalName base.ModelPath
9  }
10 func CreateImportKontext(pElement packages.PackageElement,
11     handleMultiReferenzImport func(handleImport func(path base.ModelPath), typ packages.MultiReferenzType),
12     handleArgument func(*ImportLookup, packages.Argument)) ImportKontext {}

```

Der ImportKontext beinhaltet den Pfad der aktuellen Datei, ein Kennzeichen, ob es sich um ein Delegate handelt, und einen ImportLookup, indem die Importe unter dem kompletten ModelPath des importierten Elements gespeichert werden. Die Funktion CreateImportKontext durchläuft alle Elemente eines PackageElements. Die Logik für Referenzen und Abstraktionen kann allgemein für alle Generationsziele implementiert werden. Für Argumente und Multireferenzen müssen abhängig vom Generationsziel Importe hinzugefügt werden. Deshalb wird diese Logik mithilfe der Funktionen in den Parametern bereitgestellt.

Jedes Generationsziel enthält eine Funktion, welche CreateImportKontext mit den passenden Parametern aufruft.

Variablen

Die Definitionen von Variablen folgen unabhängig vom Typ dem gleichen Schema. Deshalb bietet das Generation-Modell auch die 'FieldData'-Struktur. Diese kann mithilfe der ToFields-Funktion erstellt werden.

Listing 4.18: Definition von FieldData und Konstruktor

```

1  type FieldData struct {
2      Typ      string
3      Name     string
4      Value    *string
5      Kommentar *base.Comment
6  }
7  func ToFields(argumente []packages.Argument, referenzen []packages.Referenz,
8      multiReferenzen []packages.MultiReferenz, kontext ImportKontext,
9      primitiveTypeMapping func(base.PrimitivType, ImportKontext, bool) string,
10     buildGenericType func(element *packages.MultiReferenz, kontext ImportKontext) (typ string, value string))
11     []FieldData {}

```

'ToFields' enthält wie auch 'CreateImportKontext' Funktionen als Parameter, um sprachspezifische Mechanismen nutzen zu können.

Delegate

Damit Delegates Teile der Templates von anderen PackageElementen nutzen können, wurde das DelegateElement auch als PackageElement implementiert.

Listing 4.19: Definition des DelegateElements

```

1  type DelegateElement struct {
2      base.PackageElement

```

```
3      ExtendsNamedElements *map[ string ] base . NamedElement
4      Caller               base . ModelPath
5  }
```

Das DelegateElement enthält die NamedElements des geerbten Elements und den Pfad des Elements deren Delegate dargestellt wird. Mithilfe der NamedElemente des geerbten Elements lässt sich während der Generation unterscheiden, ob Funktionen schon im Delegate des geerbten Elements implementiert wurden.

4.2.2 Die Generationsziele

Die Generationsziele sind unterteilt in Zielsprachen und in Datenmodell und Delegates. Für jedes Generationsziel müssen Templates und Funktionen bestimmt werden.

Java

Was soll ich hier schreiben?

4.2.3 Maven Plugin

Um die Ausführung des Generators in das Maven Build-Tool zu integrieren, muss ein Maven Plugin implementiert werden.

Ein Maven Plugin besteht aus Mojos. Jedes Mojo ist eine Funktion, die während des Builds ausgeführt werden kann. Für die Generierung enthält das DMF-Plugin zwei Mojos: GenerateModelMojo und GenerateDelegatesMojo. Beide Erben vom AbstractGeneratorMojo. Dieses enthält die Konfigurationsparameter und die Ablauflogik. Nur die Ausführung des Generators wird in den Unterklassen bestimmt.

Zu Beginn der Generation wird zunächst der Generator vorbereitet. Da der Generator eine ausführbare Datei ist, muss diese im Dateisystem vorhanden sein. Um globale Installationen zu Vermeiden, werden die verschiedenen Varianten, für die verschiedenen Kompilationsziele, in der JAR des Plugins gespeichert. Bei der Ausführung wird die für das Betriebssystem passende Variante aus der JAR entpackt und in das temporäre Verzeichnis 'target/dmf/install' kopiert.

Damit Maven auch alle generierten Klassen kompiliert, wird das Ziel-Verzeichnis zu den Source-Verzeichnissen von Maven hinzugefügt.

Nun kann die Generation gestartet werden. Beide Mojos nutzen die gleiche Methode, um den Output des Generators zu verarbeiten. Die Ausgabe wird über den Logger von Maven ausgegeben. Sollte ein Fehler vom Generator ausgegeben wird dieser über den Fehlerkanal des Loggers ausgegeben und in einer Exception zurückgegeben. Das Auftreten dieser Exception hält die Ausführung von Maven an und der Text der Exception wird in der Konsole als letztes ausgegeben, sodass Entwickler*innen ihn direkt sehen.


```
[INFO] Generator exited with code 0
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time: 1.937 s
[INFO] Finished at: 2025-03-14T17:10:23+01:00
[INFO] -----
[ERROR] Failed to execute goal de.alex-brand.dmf:dmf-generator-plugin:0.0.1-SNAPSHOT:generate-model (java-gen) on project Modell:
[ERROR] FehlerStelle in Position ../riesig.dmf:30:57
[ERROR] EMERGENCY(_, 1, "Immediate action required", false);
[ERROR]                                     AAAAA
[ERROR] mehr Werte in der Konstante angegeben als im Enum definiert sind
```

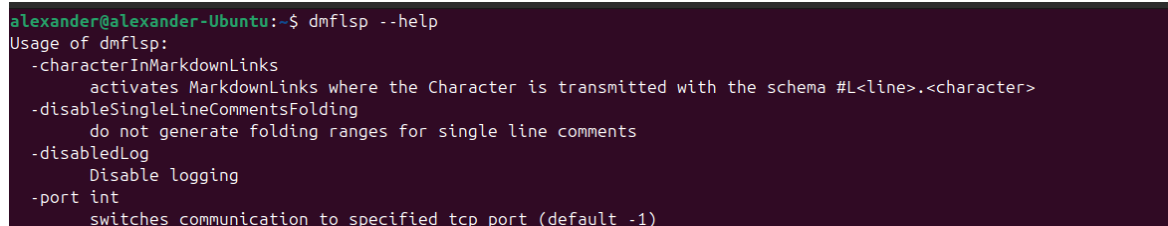
Abbildung 4.1: Ausgabe eines Fehlers

5 Benutzung des DMF

5.1 Die Nutzung des LSP

5.1.1 Installationsmöglichkeiten

Der LSP-Server benötigt keine zusätzlichen Strukturen und kann direkt als Datei ausgeführt werden. Deshalb ist die normale Vorgehensweise das Ablegen des LSP-Servers in einem Zentralen Server und das Hinzufügen des Pfades zu dem Umgebungsvariablen. Im Beispiel wurde die Datei zu 'dmflsp' umbenannt, damit der Name genauer das Programm beschreibt und nicht mit anderen LSP-Server kollidieren kann.



```
alexander@alexander-Ubuntu:~$ dmflsp --help
Usage of dmflsp:
  -characterInMarkdownLinks
    activates MarkdownLinks where the Character is transmitted with the schema #L<line>.<character>
  -disableSingleLineCommentsFolding
    do not generate folding ranges for single line comments
  -disabledLog
    Disable logging
  -port int
    switches communication to specified tcp port (default -1)
```

Abbildung 5.1: Aufruf des CLI des LSP-Servers

Es gibt Editoren die eine native Anbindung eines LSP-Servers ermöglichen. TODO Zed Beispiel

IntelliJ

IntelliJ unterstützt nur wenige LSP-Funktionen ohne zusätzliche Plugins. Mit von 'lsp4ij' können LSP-Server direkt in der Oberfläche konfiguriert werden.

Um diese Konfiguration automatisch anzulegen und den Dateien ein passendes Icon zu geben, kann das IntelliJ-Plugin für das DMF verwendet werden. Es enthält die verschiedenen Versionen des Servers und kann sie automatisch an den konfigurierten Pfad ablegen. Der Pfad zum LSP-Server kann entweder in den Einstellungen des Plugins oder über die Umgebungsvariable 'DMF_LSP' konfiguriert werden.

5 Benutzung des DMF

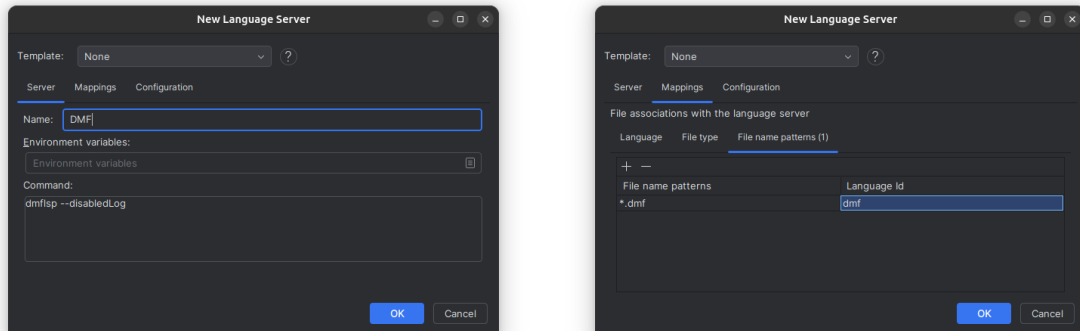


Abbildung 5.2: lsp4ij Konfiguration

Visual Studio Code

Um den LSP-Server in Visual Studio Code nutzen zu können, wird die Erweiterung für das DMF benötigt. Dieses enthält die Logik zum Verbinden zum Server und die verschiedenen Versionen. Die benötigte Version kann direkt ausgeführt werden und benötigt keine zusätzliche Konfiguration.

Im Gegensatz zu den bisherigen Konfigurationen nutzt die Visual Studio Code Erweiterung eine TCP-Verbindung.

5.1.2 Funktionen im Editor

Während des Bearbeitens der Modelle können die Funktionen des LSP-Servers genutzt werden. In diesem Abschnitt werden die Funktionen präsentiert.

Einfärbung des Textes

Mithilfe der semantischen Token kann der Editor den Text der Modelldatei einfärben. Da der Server nicht die Farbe, sondern nur die Funktion, vorgibt, wird der Text anhand der Einstellungen der Entwickler*innen eingefärbt.

Diagnosen

Wenn der Fehler in der Datei existieren werden diese automatisch im Editor markiert. Es wird auch eine Beschreibung und die detaillierte Beschreibung, welche auch im Generator ausgegeben wird, übertragen.

Die Darstellung wird von der IDE übernommen. In IntelliJ und Visual Studio Code wird die Hover-Beschreibung zusätzlich angezeigt.

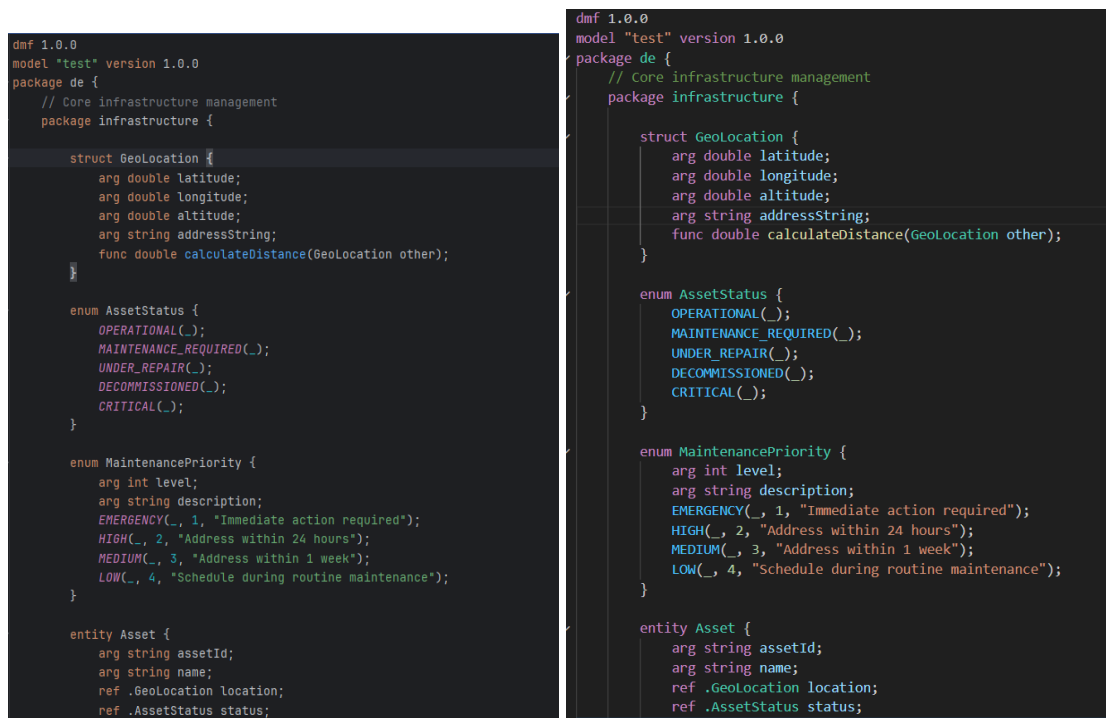


Abbildung 5.3: Eingefärbte Modelldateien in IntelliJ und Visual Studio Code

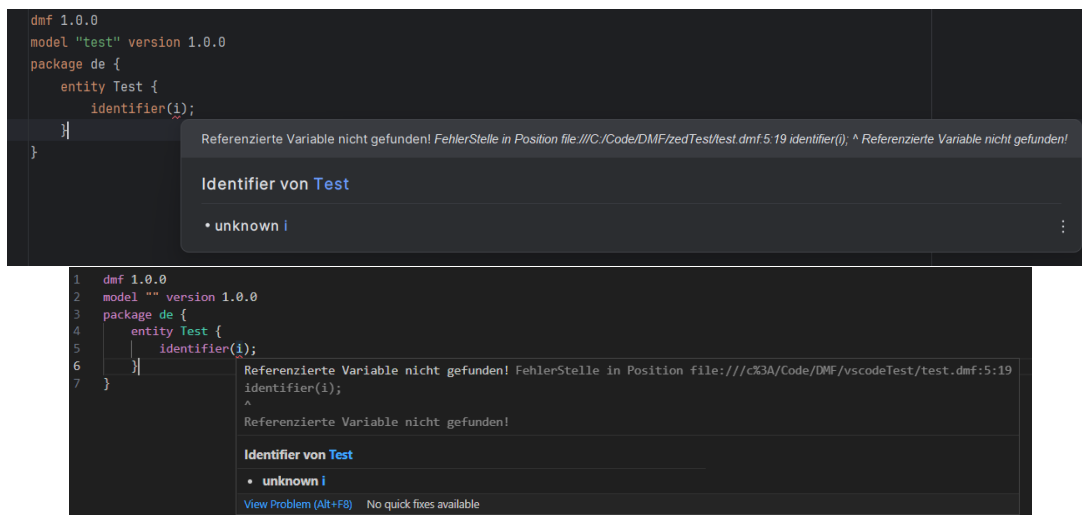


Abbildung 5.4: Beispiele aus IntelliJ und Visual Studio Code

Hover-Beschreibungen

Um Informationen über ein Element bereitzustellen, kann mit dem Mauszeiger über einem Element gehovered werden. Für alle PackageElemente, EntityIdentifier, Argumente, Referenzen, MultiReferenzen und Kommentare können Beschreibungen angezeigt werden. Mithilfe von Links in den Beschreibungen kann direkt zum erwähnten Element navigiert werden. Bei PackageElementen enthält die Beschreibung den Kommentar des Elements sowie das Package, in dem es liegt. Die Beschreibung von Enum Konstanten enthält das Enum, falls vorhanden den Kommentar und die Argumente des Enum mit den Werten der Konstante. Der erste Wert ist der Index, welcher in der Datenbank gespeichert wird. Referenzen enthalten den Kommentar sowie den Typen und Namen der Variable. Argumente und MultiReferenzen verhalten sich gleich. Bei einem Entity Identifier werden die referenzierten Variablen ihren Kommentaren angezeigt.

Referenzen

In den IDEs können die Referenzen aufgerufen werden. Der DMF-LSP-Server findet Referenzen, Deklarationen und Verwendungen in Parametern und EntityIdentifier.

Faltbereiche

Damit Entwickler*innen in großen Dateien die Übersicht behalten können unterstützt der LSP-Server die Übermittlung von Faltbereichen. Die Steuerung der Faltbereiche ist IDE spezifisch.

Auswahlbereiche

Damit die Entwickler*innen auch verschiedene Elemente gut Auswählen können, werden die Auswahlbereiche von LSP-Server berechnet.

5.2 Nutzung des DMF

Zum Darstellen der Benutzung des DMF wird das Anlegen eines Projektes mit einem Java und einen Typescript Programm beschrieben.

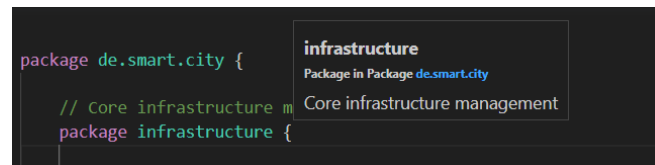


Abbildung 5.5: Beschreibung eines Package Elements



Abbildung 5.6: Beschreibung einer Enum Konstante ohne Kommentar

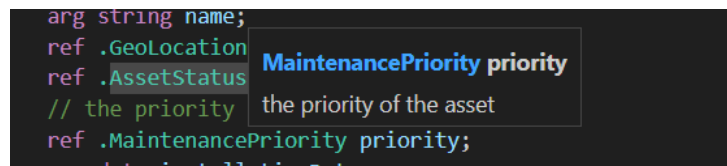


Abbildung 5.7: Beschreibung einer Referenz



Abbildung 5.8: Beschreibung eines Entity Identifiers

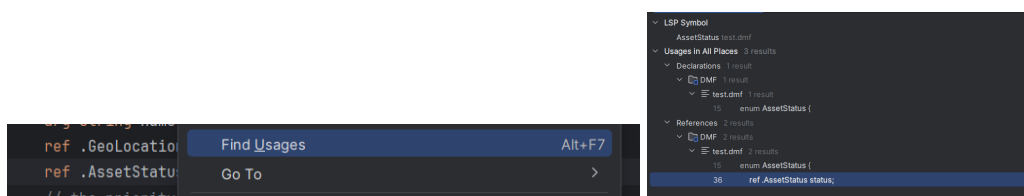


Abbildung 5.9: Aufruf der Referenzen

```
package de {  
    // Core infrastructure management  
    package infrastructure {  
  
        struct GeoLocation {...}  
  
        enum AssetStatus {...}  
  
        enum MaintenancePriority {...}  
  
        entity Asset {...}  
    }  
  
    // Transportation system management  
    package transportation {...}  
  
    // Energy grid management  
    package energy {...}  
  
    // Water management system  
    package water {...}  
}
```

Abbildung 5.10: Nutzung der Faltbereiche

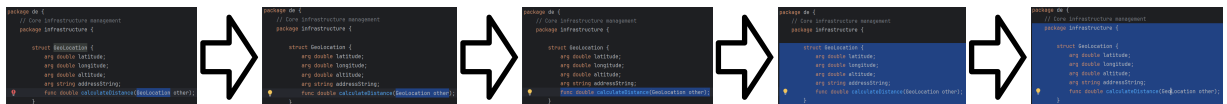


Abbildung 5.11: Nutzung der Faltbereiche

6 Anhang

6.1 EBNF Grammatik für DMF

$\langle \text{source_file} \rangle$	$::= \langle \text{dmf_declaration} \rangle \langle \text{new_line} \rangle \langle \text{model_declaration} \rangle \langle \text{new_line} \rangle$ $[\langle \text{import_block} \rangle] \langle \text{model_content} \rangle$
$\langle \text{dmf_declaration} \rangle$	$::= \text{'dmf'} \langle \text{version_number} \rangle$
$\langle \text{model_declaration} \rangle$	$::= \text{'model'} \langle \text{string_value} \rangle \text{'version'} \langle \text{version_number} \rangle$
$\langle \text{import_block} \rangle$	$::= \langle \text{import_statement} \rangle^+$
$\langle \text{import_statement} \rangle$	$::= \text{'import'} \langle \text{package_string} \rangle \text{'from'} \langle \text{string_value} \rangle \langle \text{new_line} \rangle$
$\langle \text{model_content} \rangle$	$::= \langle \text{package_content} \rangle^+$
$\langle \text{package_content} \rangle$	$::= [\langle \text{comment_block} \rangle] [\text{'expand'}] \langle \text{package_block} \rangle [\langle \text{override_block} \rangle]$ $ [\langle \text{comment_block} \rangle] [\text{'expand'}] \langle \text{struct_block} \rangle [\langle \text{override_block} \rangle]$ $ [\langle \text{comment_block} \rangle] [\text{'expand'}] \langle \text{enum_block} \rangle [\langle \text{override_block} \rangle]$ $ [\langle \text{comment_block} \rangle] [\text{'expand'}] \langle \text{entity_block} \rangle [\langle \text{override_block} \rangle]$ $ [\langle \text{comment_block} \rangle] [\text{'expand'}] \langle \text{interface_block} \rangle [\langle \text{override_block} \rangle]$
$\langle \text{comment_block} \rangle$	$::= \langle \text{comment} \rangle^+$
$\langle \text{comment} \rangle$	$::= \text{R'//.*\n'}$
$\langle \text{override_block} \rangle$	$::= \text{'override'} \text{'{' } (\langle \text{java_override} \rangle \langle \text{typescript_override} \rangle)^* \text{'}'}$
$\langle \text{java_override} \rangle$	$::= \text{'java'} \text{'{' } (\langle \text{java_annotaion} \rangle \langle \text{java_extends} \rangle \langle \text{java_implements} \rangle$ $ \langle \text{java_class} \rangle \langle \text{java_name} \rangle \langle \text{java_type} \rangle \langle \text{java_doc} \rangle)^* \text{'}'}$
$\langle \text{java_annotation} \rangle$	$::= \text{'annotations'} \langle \text{stringValue} \rangle$
$\langle \text{java_doc} \rangle$	$::= \text{'javaDoc'} \langle \text{stringValue} \rangle$

$\langle \text{java_extends} \rangle$::= 'extends' $\langle \text{stringValue} \rangle$
$\langle \text{java_implements} \rangle$::= 'implements' $\langle \text{stringValue} \rangle$
$\langle \text{java_class} \rangle$::= 'class' $\langle \text{stringValue} \rangle$
$\langle \text{java_name} \rangle$::= 'name' $\langle \text{stringValue} \rangle$
$\langle \text{java_type} \rangle$::= 'type' $\langle \text{stringValue} \rangle$
$\langle \text{package_block} \rangle$::= 'package' '{' $\langle \text{package_content} \rangle^*$ '}'
$\langle \text{struct_block} \rangle$::= 'struct' $\langle \text{identifier} \rangle$ [$\langle \text{extends_block} \rangle$] [$\langle \text{implements_block} \rangle$] '{' $\langle \text{struct_content} \rangle^*$ '}'
$\langle \text{extends_block} \rangle$::= 'extends' $\langle \text{reftype} \rangle$
$\langle \text{implements_block} \rangle$::= 'implements' $\langle \text{reftype} \rangle$ (',' $\langle \text{reftype} \rangle$)+
$\langle \text{struct_content} \rangle$::= [$\langle \text{comment_block} \rangle$] $\langle \text{arg_block} \rangle$ [$\langle \text{override_block} \rangle$] [$\langle \text{comment_block} \rangle$] $\langle \text{ref_block} \rangle$ [$\langle \text{override_block} \rangle$] [$\langle \text{comment_block} \rangle$] $\langle \text{multi_block} \rangle$ [$\langle \text{override_block} \rangle$] [$\langle \text{comment_block} \rangle$] $\langle \text{func_block} \rangle$ [$\langle \text{override_block} \rangle$]
$\langle \text{arg_block} \rangle$::= 'arg' $\langle \text{primitive_type} \rangle$ $\langle \text{identifier} \rangle$ ','
$\langle \text{ref_block} \rangle$::= 'ref' $\langle \text{reftype} \rangle$ $\langle \text{identifier} \rangle$ ','
$\langle \text{multi_block} \rangle$::= 'ref' $\langle \text{multi_name} \rangle$ '<' $\langle \text{primitive_type} \rangle$ [',' $\langle \text{primitive_type} \rangle$] >' $\langle \text{identifier} \rangle$ ',' 'ref' $\langle \text{multi_name} \rangle$ '<' $\langle \text{reftype} \rangle$ [',' $\langle \text{primitive_type} \rangle$] '>' $\langle \text{identifier} \rangle$ ',' 'ref' $\langle \text{multi_name} \rangle$ '<' $\langle \text{primitive_type} \rangle$ [',' $\langle \text{reftype} \rangle$] '>' $\langle \text{identifier} \rangle$ ',' 'ref' $\langle \text{multi_name} \rangle$ '<' $\langle \text{reftype} \rangle$ [',' $\langle \text{reftype} \rangle$] '>' $\langle \text{identifier} \rangle$ ','
$\langle \text{func_block} \rangle$::= 'func' $\langle \text{reftype} \rangle$ $\langle \text{identifier} \rangle$ '(' [$\langle \text{param_definition} \rangle$ (',' $\langle \text{param_definition} \rangle$)*] ')' ',' 'func' $\langle \text{primitive_type} \rangle$ $\langle \text{identifier} \rangle$ '(' [$\langle \text{param_definition} \rangle$ (',' $\langle \text{param_definition} \rangle$)*] ')' ','

	'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>

6.2 Beispiel

<statement>	::= <ident> '=' <expr> 'for' <ident> '=' <expr> 'to' <expr> 'do' <statement> '{' <stat-list> '}' <empty>
<stat-list>	::= <statement> ';' <stat-list> <statement>