0.1 Aufbau der DMF-DSL

Im Kern des DMFs sind die Modelldateien. Sie enthalten alle Informationen und die Schnittstelle zwischen DMF und Entwickler*innen. Deswegen ist das Format essentiell. EMF setzt auf XML als Format für die Modelldateien und erleichtert die Bearbeitung durch Grafische Editoren in Eclipse. Grafische Editoren funktionieren für das DMF nicht, da DMF mithilfe des LSP-Protokolls nur Text-Editoren unterstützt. Deshalb muss sich auch das Format der Modelldateien anpassen.

0.1.1 Die DSL und die EBNF

Die DMF-DSL besteht aus vielen Elementen die in einer EBNF dargestellt werden können. In diesem Abschnitt werden die verschiedenen Elemente und ausgewählte Regeln der EBNF Grammatik erläutert.

Die EBNF Grammatik nutzt "[]"für Optionale Elemente, *"für optionale Wiederholungen (mindestens 0 Elemente) und "+ërforderliche Wiederholungen (mindestens 1 Element).
Sie beginnt mit der Regel ßource_file:

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

Header

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

Listing 1: Header einer DMF-Modelldatei

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

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

Listing 1: Die EBNF Regeln

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

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

Imports

Es gibt zwei Gründe für den Import von vorhandenen Modellen:

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

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

Listing 2: Import des Package de.base

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

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

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

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

Packages

Ein DMF Modell besteht aus Packages. Diese Unterteilen die restlichen Elementen und representieren Ordner im späteren Generat. Innerhalb eines Package können weitere Packages, Structs, Entities, Enums und Interfaces enthalten sein. Diese verschiedenen Elemente werden im weiterem als PackageElemente bezeichnet.

```
1 // Das ist ein Packages mit ddem Pfad "de.beispiel"
2 package de.beispiel {
3
4 }
```

Jedes PackageElement kann mit einem Kommentar, dem Keyword ëxpandünd einem Override-Block versehen werden. Die Auslagerung dieser AST-Elemente in das "package_content-Element verinfacht das Verarbeiten des ASTs, da ein allgemeines Vorgehen für alle PackageElemente genutzt werden kann.
Der Kommentar dient zur Dokumentation des jeweiligen Elements.
Das ëxpandKeyword muss genutzt werden, wenn ein importiertes Element erweitert werden soll.
Der Override-Block lässt den Entwickler Zielsprachen Spezifische Anpassungen vornehmen. Dadurch können Spezielle Details die in der Syntax für alle Sprachen nicht abgebildet werden können, trotzdem generiert werden. Es können auch Bibliotheken in der Zielsprache eingebunden werden.

Alle Elemente folgen dem Schema das die *_content"Regel immer den Inhalt, der in unbestimmter Reihenfolge enthalten sein kann, enthält und die *_block"Regel die Elemente enthält, die den Inhalt einschließen.

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

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

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

Structs

Ein Struct fasst Variablen zusammen ohne dabei eine Identität zu garantieren. Structs können Argumente, (Multi-)Referenzen und Funktionen beinhalten. Zusätzlich unterstützen Structs die Implementierung von mehreren Interfaces und sie können von einem anderen Struct erben.

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

Wie bei den PackageElementen gibt auch beim ßtruct_content"wieder einen optionalen Kommentar und Override-Block. Dieses Schema wird auch bei allen anderen PackageElementen fortgeführt.

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

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

Entity

Eine Entity ist ein spezializiertes Struct. Es besitzt eine Identität in der Datenbank. Die restlichen Elemente werden aus dem Struct übernommen. Jedoch kann eine Entity von Structs und Entities erben.

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

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

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

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

Interface

Interfaces beinhalten Funktionen und können von Structs und Entities implementiert werden. Interfaces können auch andere Interfaces implementieren. Dabei werden die Methoden ins Interface übernommen.

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

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

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

Enum

Enum sind Aufzählungen. Jedem Eintrag können mithilfe von Argumenten weitere Werte zugeordnet werden.

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

⟨*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*⟩