0.1 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. EMF setzt auf XML als Format für die Modelldateien und erleichtert dadurch die Bearbeitung durch grafische Editoren in Eclipse. Grafische Editoren funktionieren für das DMF nicht, da 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.

0.1.1 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 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.

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

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

Importblock

Es gibt zwei Gründe für den Import von vorhandenen Modellen:

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

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

Listing 2: 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.

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

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

Modellelemente erweitern Das DMF beinhaltet die Möglichkeit Elemente zu erweitern. Dies ermöglicht Code für das ursprüngliche Modell wiederzuverwenden und trotzdem Argumente, Referenzen und Funktionen hinzuzufügen.
Eine Erweiterung im DMF muss generell explizit gekennzeichnet sein. Dafür muss zuerst das Package, indem der Entwickler*in etwas erweitern 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.

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

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

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

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

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

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

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

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

Jedes PackageElement kann mit einem Kommentar, dem Keyword 'expand' und einem Override-Block versehen werden:

```
// Das ist ein Packages mit dem Pfad "de.beispiel"
expand package de.beispiel {

}
override {
}
```

Die Regeln für ein Package folgen der Namenskonvention. Der ‘package_string’ beinhaltet Regeln für die Separation der Packages mit Punkten.

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

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

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

```
struct Beispiel extends SuperBeispiel implements IBeispiel, IABeispiel {

}
```

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

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

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

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

Argument

```
arg int i;
```

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

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

Referenz

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

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

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

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

Multi-Referenz

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

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

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

Funktionen

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

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

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

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

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

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

Besonders bei der ‘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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

0.1.2 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.