0.1 Der Generator

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

0.1.1 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 ‘WaitGroup’ verwaltet.

Listing 1: 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 Dateiinhalts über eine Funktion kann die Methode für alle Elemente genutzt werden. Um dieses Pattern zu erleichtern wird die Methode ‘apply’ genutzt.

Listing 2: Die apply-Methode

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

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 Die Generationsziele) 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 3: Generation einer Entity

```
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 ImportKotext und Funktionen zum Erstellen des Kontextes bereit.

Listing 4: ImportKontext

```
type ImportKontext struct {
    ImportLookUp ImportLookUp
    Path         base.ModelPath
    HasDelegate  bool
}
type ImportLookUp map[string]Import
type Import struct {
    OriginalName base.ModelPath
}
func CreateImportKontext(pElement packages.PackageElement,
    handleMultiReferenzImport func(handleImport func(path base.ModelPath), typ packages.MultiReferenzType),
    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 5: Definition von FieldData und Konstruktor

```
type FieldData struct {
    Typ         string
    Name        string
    Value       *string
    Kommentar *base.Comment
}
func ToFields(argumente []packages.Argument, referenzen []packages.Referenz,
    multiReferenzen []packages.MultiReferenz, kontext ImportKontext,
    primitiveTypeMapping func(base.PrimitivType, ImportKontext, bool) string,
    buildGenericType func(element *packages.MultiReferenz, kontext ImportKontext) (typ string, value string))
    []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 6: Definition des DelegateElements

```
type DelegateElement struct {
    base.PackageElement
    ExtendsNamedElements *map[string]base.NamedElement
    Caller               base.ModelPath
}
```

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.

0.1.2 Die Generationsziele

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

0.1.3 Maven Plugin