0.1 Die Nutzung des LSP

0.1.1 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 1: Aufruf des CLI des LSP-Servers

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

Intellij

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

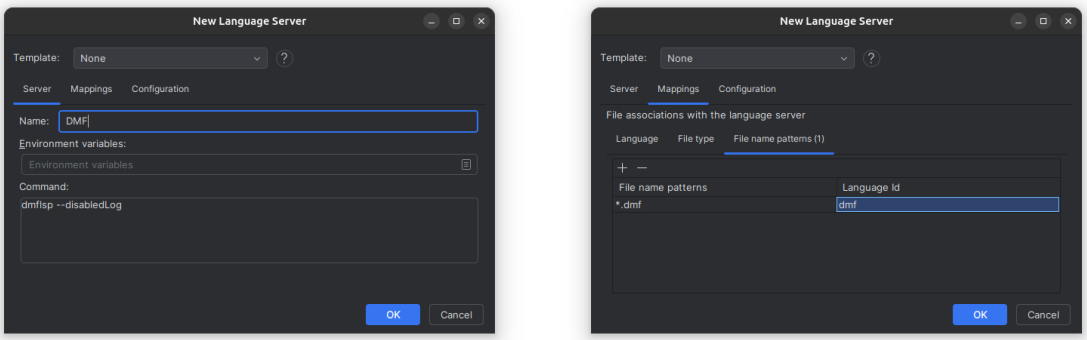


Abbildung 2: lsp4ij Konfiguration

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.

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.

0.1.2 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.

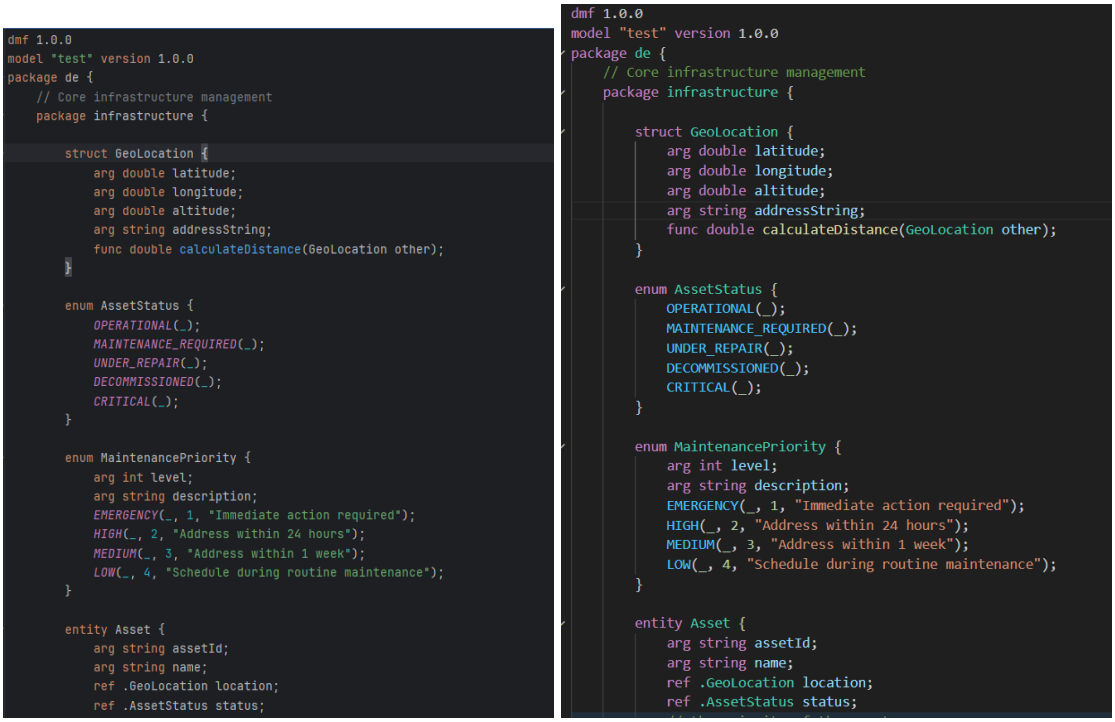


Abbildung 3: Eingefärbte Modelldateien in Intellij und Visual Studio Code

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.

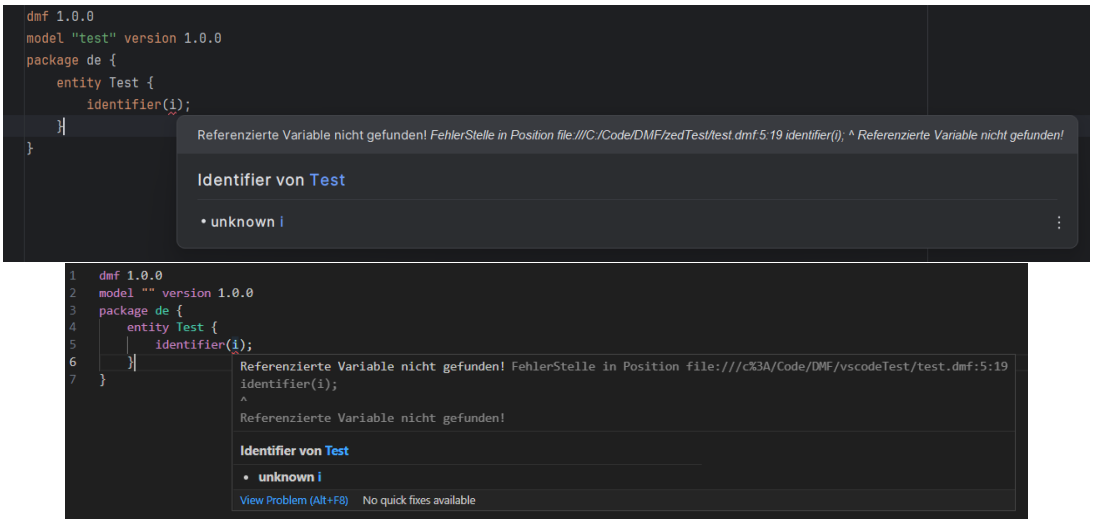


Abbildung 4: Beispiele aus Intellij und Visual Studio Code

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.

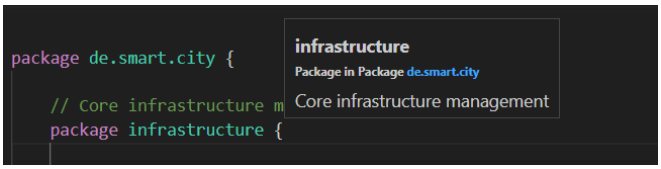


Abbildung 5: Beschreibung eines Package Elements

Bei PackageElementen enthält die Beschreibung den Kommentar des Elements sowie das Package, in dem es liegt.

Abbildung 6: Beschreibung einer Enum Konstante ohne Kommentar

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.

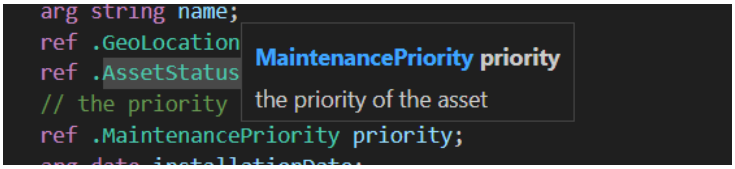


Abbildung 7: Beschreibung einer Referenz

Referenzen enthalten den Kommentar sowie den Typen und Namen der Variable. Argumente und MultiReferenzen verhalten sich gleich.

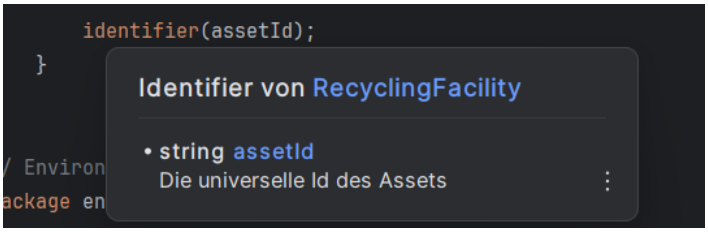


Abbildung 8: Beschreibung eines Entity Identifiers

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.

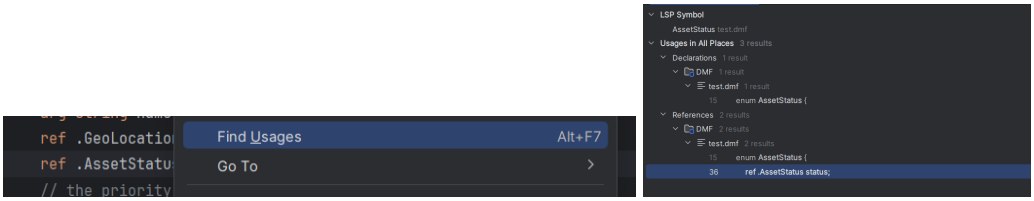


Abbildung 9: Aufruf der Referenzen

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.

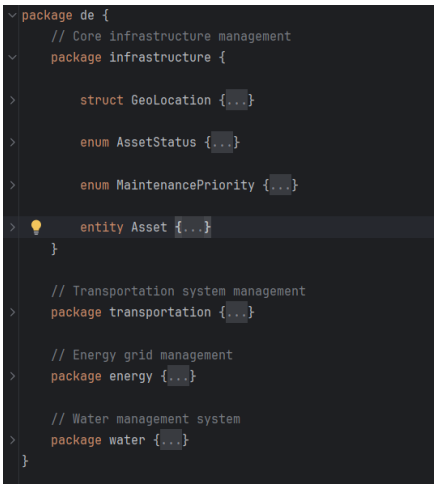


Abbildung 10: Nutzung der Faltbereiche

Auswahlbereiche

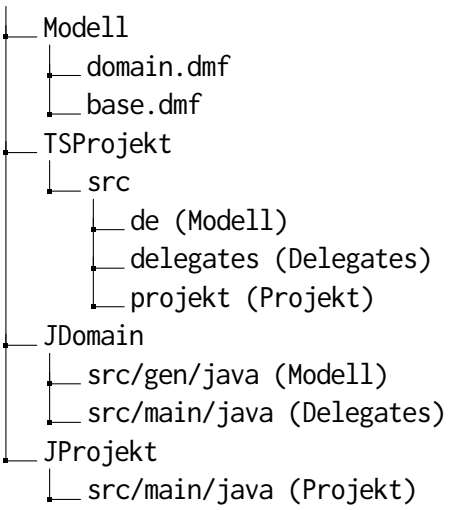
Damit die Entwickler*innen auch verschiedene Elemente gut Auswählen können, werden die Auswahlbereiche von LSP-Server berechnet.

Abbildung 11: Nutzung der Auswahlbereiche

0.2 Nutzung des DMF

Zum Darstellen der Benutzung des DMF wird das Anlegen eines Projektes mit einem Java und einen Typescript Programm beschrieben. Es wird dafür das Modell aus dem Abschnitt ??.

Abbildung 12: Dateiaufbau für ein Beispielprojekt



Im Modell Ordner wird das Modell abgelegt. Die Bereitstellung kann an die Organisation angepasst werden, da die restlichen Projekte per relativen Pfad auf die Datei zugreifen. Entscheidend ist dabei die Organisation der Git Repositories. Werden die verschiedenen Komponenten (Modell, Typescript Projekt, Java Projekt) in verschiedenen Repositories abgelegt, so handelt es sich um eine Poly-Repository Struktur. **?** Die Verwaltung der verschiedenen Repositories kann manuell durch Entwickler*innen oder durch Build Scripte gesteuert werden. Konträr zur Poly-Repository Struktur ist die Mono-Repository Struktur. Bei dieser werden alle Projekte in einem Repository verwaltet. **?** Hier entfällt die Synchronisation des Modells.

In unserem Beispiel wird das vorher beschriebene Modell (siehe **??**) zusammen mit einem Basismodell genutzt.

Listing 1: Das Basismodell

```
1  dmf 1.0.0
2  model "base" version 1.0.0
3
4  package de.base {
5      interface IBeispiel {
6          func string printBeispiel();
7      }
8  }
```

Das Typescript Projekt enthält sowohl die generierten Dateien als auch den Source Code des eigentlichen Projekts. Dies kann besonders in kleinen Projekten den Aufwand reduzieren.

Das Java Projekt wird in mehrere Artefakte unterteilt. Dies dient der Strukturierung von größeren Projekten und wurde exemplarisch genutzt, obwohl das Beispiel Projekt nur ein weiteres Artefakt besitzt. Innerhalb des JDomain Artefakts werden Modell und Delegates in verschiedene Ordner generiert, damit nur die Delegates von der Versionsverwaltung beachtet werden können und IDEs die Dateien richtig verwalten können.

0.2.1 Anlage des Typescript Projektes

Das Beispiel Typescript Projekt nutzt die Bibliothek 'Express' um einen kleinen Webserver zu implementieren. Ziel ist es die Verwendung des DMF im Kontext einer simplen Restschnittstelle zu zeigen.
Das Projekt wurde mithilfe der einer Anleitung von **?** angelegt.
Zuerst wird die 'package.json'-Datei mithilfe der NPM-CLI angelegt. In ihr wird das Projekt beschrieben. Dazu gehören unter anderem der Name, Version, Abhängigkeiten und Ausführungskonfigurationen.
Zu den Abhängigkeiten werden Dotenv, Typescript und Express hinzugefügt. Die Abhängigkeiten zu den Typen von Typescript und Express werden als 'DevDependecies' hinzugefügt. So werden sie nur während der Entwicklung genutzt.
Der nächste Schritt der Einrichtung ist die Konfiguration von Typescript. Es werden die Ordner für Source Code und Output sowie die beabsichtigte Version des JavaScript-Standards angegeben.
Nun kann die Beispielimplementierung aus der Anleitung angelegt werden. Diese kann mit folgendem Befehl ausgeführt werden.

```
1  npx ts-node src/index.ts
```

Kann der Server aufgerufen werden, war die bisherige Anlage erfolgreich. Als nächstes kann das DMF hinzugefügt werden. Dafür werden die Ausführungskonfigurationen erweitert.

Listing 2: Ausführungskonfigurationen in package.json

```
1  "scripts": {
2      "run": "npx ts-node src/index.ts",
3      "build": "npm run generateModell && npm run generateDelegates && npx tsc",
4      "start": "node dist/index.js",
5      "generateModell": "generator --basePath ./src --mode ts --modelFile ../Modell/domain.dmf",
6      "generateDelegates": "generator --basePath ./src --mode tsDelegates --modelFile ../Modell/domain.dmf",
7    },
```

Es werden zwei neue Konfigurationen hinzugefügt: generateModell und generateDelegates. Sie rufen den Generator mit den Parametern für den Source Code Ordner, dem Pfad der Modelldatei und dem jeweiligen Generationsziel auf. Da die Konfigurationen in der Build Konfiguration eingebaut wurden, werden sie bei jedem Build automatisch ausgeführt.

Die generierten Modelldateien werden in den Ordnern nach ihrem Package Pfad abgelegt. Aus dem verwendeten Modell werden folgende Dateien generiert.

Listing 3: Aufgabe.ts

```
import { Beispiel } from "./Beispiel";
import * as delegate from "../../delegates/de/beispiel/AufgabeDelegate";

export class Aufgabe {
    beispiel: Beispiel;
    frage: string;
    antwort: string;
    id: number;

    constructor(frage: string, antwort: string, id: number, beispiel: Beispiel) {
        this.frage = frage;
        this.antwort = antwort;
        this.id = id;
        this.beispiel = beispiel;
    }

}
```

Bei der Aufgabe handelt es sich um eine Entity. Es werden alle Argumente und Referenzen (und evt. Funktionen) generiert. Typescript initialisiert Variablen nicht bei ihrer Definition. Sie müssen von Entwickler*innen gesetzt werden. Deshalb generiert das DMF einen Konstruktor mit allen Variablen.

Listing 4: Beispiel.ts

```
import { IBeispiel } from "../base/IBeispiel";
import { BeispielTyp } from "./BeispielTyp";
import * as delegate from "../../delegates/de/beispiel/BeispielDelegate";

export class Beispiel implements IBeispiel {
    i: number;
    inhalt: string;
    typ: BeispielTyp;

    constructor(i: number, inhalt: string, typ: BeispielTyp) {
        this.i = i;
        this.inhalt = inhalt;
        this.typ = typ;
    }


    printBeispielMarkdown(): string {
        return delegate.printBeispielMarkdown(this);
    }

    printBeispiel(): string {
        return delegate.printBeispiel(this);
    }
}
```

In der Aufgabe wird das Struct Beispiel referenziert. Structs werden im Typescript Generator equivalent zu Entities generiert. Die Beispiel-Klasse demonstriert den Aufruf der Delegate Methoden.

Listing 5: BeispielTyp.ts

```
export enum BeispielTyp {
    CODE = 0,
    TEXT = 1
}

export interface BeispielTypDetails {

}


export const BeispielTypInfo : Record<BeispielTyp, BeispielTypDetails> = {
    [BeispielTyp.CODE]: {  },
    [BeispielTyp.TEXT]: {  },

}
```

Am Enum BeispielTyp kann die Emulation von weiteren Argumenten, ohne native Unterstützung, präsentiert werden, obwohl das Enum keine weiteren Argumente besitzt. Das eigentliche Enum besitzt nur Einträge mit den Datenbankindizes. Das Details-Interface würde eventuelle Argumente deklarieren. Im Info-Record werden die Enum Konstanten Details-Instanzen zugeordnet, welche die Werte aus dem Modell enthalten.
Die Type (Interface/Record) müssen für jede Sprache ersetzt werden.

Die Delegate Funktionen werden von den Modellklassen importiert und in den Funktionen der Modellklassen aufgerufen.

Listing 6: BeispielDelegate.ts

```
import {Beispiel} from "../../../de/beispiel/Beispiel";
import {BeispielTyp} from "../../../de/beispiel/BeispielTyp";


export function printBeispiel(caller: Beispiel): string {
    return `${caller.typ == BeispielTyp.TEXT ?
    "Text-" : "Code-"}Beispiel ${caller.i}`;
}

export function printBeispielMarkdown(caller: Beispiel): string {
    return `## Beispiel\n${caller.typ == BeispielTyp.TEXT ? "" :
    "```javascript\n"}${caller.inhalt}${caller.typ == BeispielTyp.TEXT ?
    "" : "\n```"}`;
}
```

In der Delegatedatei werden die Funktionen mit dem zusätzlichen Caller generiert. Die Inhalte der Funktionen müssen die Entwickler*innen selbstständig implementieren.

Im Server kann nun ein neuer Endpunkt hinzugefügt werden, welcher eine Aufgabe mit einem Beispiel zu Markdown rendert und ausgibt. Die Aufgaben werden in einem Record gespeichert. In einem realen Projekt würden an dieser Stelle die Prozesse des Programms aufgerufen werden.

Listing 7: Neuer Endpunkt in index.ts

```
const aufgaben: Record<number, Aufgabe> = {
    1: new Aufgabe(
        "Was ist der Unterschied zwischen 'let' und 'const' in JavaScript?",
        "'let' erlaubt Variablen nach der Deklaration zu  ndern , ...",
        1,
        new Beispiel(1,`let i = 1;\nconst ii = 2`, BeispielTyp.CODE)
    ),
    2: new Aufgabe(
        "Erkl re das Konzept der Vererbung in der objektorientierten Programmierung.",
        "Vererbung ist ein Mechanismus, bei dem eine Klasse die ...",
        2,
        new Beispiel(1, `Ein Pferd ist genauso ein Tier wie ein Hund.`, BeispielTyp.TEXT)
    ),
}
app.get("/:id", (req: Request, res: Response) => {
    const id: number = Number(req.params["id"]);

    const aufgabe = aufgaben[id];

    if (!aufgabe) {
        res.status(404).send({
            error: 'Aufgabe not found'
        });
        return
    }
    res.setHeader('Content-Type', 'text/markdown');

    res.send(`# ${aufgabe.frage}\n${aufgabe.beispiel.printBeispielMarkdown()}`);
});
```

0.2.2 Anlage des Java Projektes

Für das Java Projekt wird Maven als Build Tool verwendet. Deshalb beginnt die Anlage der beiden Maven Artefakte mit der Anlage der 'pom.xml'-Dateien. In ihnen werden die Namen, Versionen, die Plugins und Abhängigkeiten definiert.
Im JDomain Artefakt wird nun die Ausführung des DMF-Plugins hinzugefügt.

Listing 8: Konfiguration des DMF-Plugins

```
<build>
    <plugins>
        <plugin>
            <groupId>de.alex-brand.dmf</groupId>
            <artifactId>dmf-generator-plugin</artifactId>
            <version>0.0.1-SNAPSHOT</version>
            <executions>
                <execution>
                    <id>java-gen</id>
                    <goals>
                        <goal>generate-model</goal>
                    </goals>
                </execution>
                <execution>
                    <id>java-delegates-gen</id>
                    <goals>
                        <goal>generate-delegates</goal>
                    </goals>
                    <configuration>
                        <tempSources>./src/gen/java</tempSources>
                    </configuration>
                </execution>
                <execution>
                    <id>schema-gen</id>
                    <goals>
                        <goal>generate-model</goal>
                    </goals>
                    <configuration>
                        <zielsprache>sql</zielsprache>
                        <tempSources>./src/test/resources</tempSources>
                    </configuration>
                </execution>

            </executions>
            <configuration>
                <modelPath>../Modell/domain.dmf</modelPath>
            </configuration>
        </plugin>
    </plugins>
</build>
```