0.1 Zusammenfassung

Ausgehend vom bisherigen Stand des DMFs konnten die zuvor gestellten konkreten Anforderungen erfüllt werden.
Mithilfe des Vergleichs verschiedener Programmiersprachen konnte eine Abstraktion zur Modellierung der Datenstrukturen gefunden werden. Basierend auf der Abstraktion konnte eine Syntax und Semantik entworfen werden, welche eine kompakte und ausdrucksvolle Modellierung ermöglicht.
Durch die Inklusion der Override Funktion konnte auch eine Flexibilität im Generat erreicht werden.
Die Anforderung einer Wiederverwendung und Erweiterung von Modellen für das PLE konnte mithilfe des Importierens und der 'expand'-Funktion erreicht werden.

Mithilfe des LSP-Servers können Entwickler*innen DMF-Modelle in den meisten IDEs einsetzen. Durch die zusätzlichen Plugins ist der Einsatz in Intellij und Visual Studio Code nochmals stark vereinfacht worden.
Ein Nebeneffekt des LSP ist die schnellere Entwicklung von Erweiterungen für nicht kompatible IDEs. Da dieses Protokoll so weit verbreitet ist, kann auf mehr Dokumentation für die Entwicklung zurückgegriffen werden.
Dank des schnellen und inkrementellen Parsen durch die Nutzung von Treesitter kann eine reaktionsschnelle und intuitive Nutzung der IDEs geboten werden. Dabei spielt auch die Wahl von Golang als Implementierungssprache eine große Rolle. Mithilfe verschiedener Strukturen der Sprachen ließ sich der LSP-Server sehr einfach parallelisieren. Durch die Kompilierung von Golang wurde der Server zusätzlich beschleunigt und die Verwaltung der Installation vereinfacht.

Diese Effekte der Wahl von Golang lassen sich auch beim Generator feststellen. Durch die parallele Generierung von Dateien erwies sich der Generator als sehr Zeitsparend in der Nutzung. Dadurch wird die Fokussierung der Entwickler*innen auf die Entwicklung erleichtert und es wird die Produktivität gesteigert. Der Generator ist in der Lage komplette Modelle in der Startzeit einer JVM zu generieren.
Mit der Unterstützung von Java und Typescript kann das DMF in verschiedenen Projekten eingesetzt werden. Dabei ermöglicht das Maven Plugin auch einen reibungslosen Einsatz in einem Framework mit einer anderen Implementierungssprache.
Dank der Delegate Strukturen können Teile des Modells immer neu generiert werden, ohne die Implementierung der Funktionen zu überschreiben. Entwickler*innen können so risikofrei den Generator nutzen.
Durch die Generierung eines Datenbankschemas ist die korrekte Einrichtung einer Datenbankinstanz auch in großen Modellen sehr leicht.

Das beschriebene Beispielprojekt zeigt, dass das DMF schon heute einsatzbereit ist. Gerade wenn ein Modell sowohl in Typescript als auch in Java genutzt werden soll, bietet es entscheidende Vorteile gegenüber vergleichbaren Frameworks. Es steigert dabei abhängig von der Modell- von Team-größe die Produktivität und spart so Kosten.

Doch es gibt auch Schwachstellen der bisherigen Implementierung. Deswegen bietet sich auch ein Ausblick in eine potenzielle Entwicklung des DMFs an.

0.1.1 Ausblick

Bisher werden keine automatischen Mappings für ORM Frameworks erzeugt, wodurch die komplette Persistenz manuell implementiert werden muss. Deshalb wird es sich als besonders schwierig herausstellen, ein großes Projekt mit einer Datenbankverbindung zu pflegen. Die zusätzlichen Kosten würden den Vorsprung, den das DMF durch die flexibilität des Frameworks besitzt, überschatten.

Die Flexibilität ist auch ein weiteres Ziel, welches stark ausgebaut werden kann. Durch die Implementierung neuer Generationsziele können mehr Projekte das DMF nutzen und bisherige Projekte erhalten mehr Freiheiten in der Technologiewahl. Es würden sich Golang, Kotlin, Rust, Python oder C anbieten.
Um den Entwickler*innen die vollständige Kontrolle über die Generation zu geben, wäre die Implementierung einer Schnittstelle zur Definition eigener Generationsziele denkbar. Denkbar wäre das Laden neuer Golang Templates mit einer Schnittstelle für die Funktionen in der Generation. Für die Implementierungssprache der Funktionen kämmen mehrere Sprachen infrage. Sowohl Javascript und Python sind weit verbreitet.
Jedoch sind die Schnittstellen zu Javascript für die Ausführung in einem Browser ausgelegt. Zur Nutzung müsste der Generator in () kompiliert werden.
Python Funktionen lassen sich über eine C-Schnittstelle ausführen.
Deutlich weiterentwickelt ist die Integration zwischen Lua und Golang. Somit wären Golang-Templates und eine Lua Schnittstelle zur Definition wahrscheinlich.

Durch die Implementierung neuer Override Funktionen würde sich das Framework an andere Architekturen besser anpassen können. Infrage kämen Optionen für Typescript und die Datenbank.
Zusätzlich wären auch Modellweite Overrides, die Einstellungen für eine bestimmte Elementgruppe treffen, denkbar.

Um die Zusammenarbeit in größeren Teams oder zwischen verschiedenen Teams im Kontext von PLE zu erleichtern, wäre eine automatische Generierung einer Dokumentation hilfreich. Dabei könnten neben der reinen Generation aus dem Modell auch die Implementierungen der Delegates genutzt werden, um einen schnellen Überblick über Aufbau, Funktion und Implementierung zu bieten.

Langfristig wäre auch die Unterstützung von Migrationen wichtig. Mithilfe der Informationen aus zwei verschiedenen Modellversionen könnten automatisch Migrationsscripte für das Datenbankschema generiert werden. Die Migration der Daten einer Datenbank wäre jedoch wahrscheinlich nicht automatisch möglich ohne weiteren Kontext während der Generierung einzubinden.