MODEL DRIVEN SOFTWARE DEVELOPMENT MIT DEM TOOLPAKET SWAGGER

Herval Bernice Nganya Nana[1], Oussema Mzoughi[2], Aliridha Haouari[3], Christian Lange[4]
[1,2,3,4]*Fachbereich Informatik und Medien, Technische Hochschule Brandenburg, Deutschland*
[1]*nganyana@th-brandenburg.de,* [2]*oussema.mzoughi@th-brandenburg.de,* [3]*aliridha.haouari@th-brandenburg.de,*
[4]*langchri@th-brandenburg.de*

Keywords: Model Driven Software Development, Swagger, MDA

Abstract: Ein Softwareprojekt lebt und wird immer weiter entwickelt. Daher ist es wichtig, dass die Software vom Architekten gut geplant und ebenso auch vom Entwickler gut umgesetzt wird. Problematisch ist, dass die vom Architekten erzeugten Diagramme vom Softwareentwickler fehlinterpretiert werden können. Dadurch entsteht eine Archtiturerosion, die das Erweitern von Software erschwert oder sogar unmöglich macht. Um diesen Punkt entgegenzutreten, gibt es Model Driven Software Development. Die Software wird vom Architekten beschrieben und ein Build-Tool wird genutzt um aus dieser Beschreibung eine entsprechende Software zu generieren. Ein Beispiel dafür ist das Open-Source Toolpaket Swagger. Mit der Nutzung dieser Tools soll eine Serveranwendung entwickelt werden. Dabei wird überprüft, wie sich die neue Herangehensweise an Softwareentwicklung im Team ausprägt. Es konnte erfolgreich eine Serveranwendung geplant und entwickelt werden.

1 EINLEITUNG

1.1 Motivation

Viele Softwareprojekte stoßen bei der Erweiterung der eigenen Software auf große Probleme. Die Software wird vom Architekten geplant und von den Softwareentwicklern umgesetzt. Jedoch können die Anforderungen von dem Architekten missverstanden werden. Somit driften die geplante und die implementierte Software voneinander ab. Diese Archtikturerosion kann bei der Erweiterung der Software zu großen Problemen führen (Yazdanshenas and Khosravi, 2009). Diese dann auszubessern, kann sehr zeitaufwändig und teuer werden.

Zur Eindämmung solcher Probleme gibt es ein Verfahren, das die Software aufgrund von eindeutig beschriebenen Modellen generiert. Dieses Verfahren nennt sich Model Driven Software Development (MDSD). Die gesamte Software wird via Domain-Specific Language (DSL) beschrieben.

Die Planung der Software wird via Top-down vorangetrieben. Dadurch wird vom abstrakten Datenobjekt zur fertigen Funktionalität modelliert.

Es gibt bereits einige nützliche Werkzeuge, die solche DSLs zur Verfügung stellen. Mithilfe dieser Werkzeuge kann eine Software plattformunabhängig beschrieben und später auf einer bestimmten Plattform gebuildet werden.

Da die Software nun einen eindeutigen Rahmen besitzt, kann die Gefahr einer Architekturerosion verringert werden. Die Grundlage der Software ist ihre Beschreibung. Dadurch haben alle Parteien (z. B. Architekten und Entwickler) eine identische und stetig aktuelle Basis.

Ein Beispiel dafür ist das Open-Source Toolpaket Swagger (off,). Mit deren Hilfe können Serveranwendungen beschrieben und später auf eine beliebige Zielplattform gebuildet werden.

In dieser Ausarbeitung wird das Toolpaket Swagger getestet und eine Serveranwendung mithilfe dieser Tools beschrieben und implementiert.

1.2 Ziel

Diese Ausarbeitung beschäftigt sich mit der Entwicklung einer Software, die via MDSD entwickelt wird. Es soll überprüft werden, wie sich eine solche Mechanik in Projektarbeiten auswirken kann. Weiterhin soll überprüft werden, welche Probleme bei der Umsetzung des MDSD zum Vorschein kommen können.

1.3 Aufgabenstellung

Die zentrale Aufgabe ist es eine Serveranwendung zu entwickeln, die mit der DSL von Swagger beschrieben wurde. Um die Anwendung zu builden, muss ein Überblick über die Möglichkeiten mit Swagger erzeugt werden. Der nächste Schritt ist die Planung der Software. Dieses Projekt ist ein Kooperationsprojekt. Es soll eine Serveranwendung mit REST-Schnittstellen entwickelt werden, die wiederum vom anderen Team aufgerufen und für deren Projekt genutzt werden kann. Daher müssen auch die entsprechenden Schnittstellen gemeinsam geplant und umgesetzt werden. Die Daten sollen persistent gespeichert werden können. Dies fordert eine Planung und Integration einer Datenbank. Abschließend soll ein Fazit über die Arbeit mit MDSD gezogen werden.

1.4 Abgrenzung

Diese Ausarbeitung befasst sich leidglich mit dem Erstellen eines Softwareprojekts mit den Toolpaket Swagger. Die erstellten REST-Schnittstellen arbeiten mit keiner Verschlüsselung der Daten. Der Punkt Softwaresicherheit wird gänzlich ausgelassen. Weiterhin wird nicht geprüft, was beim erneuten builden der Software mit dem handgeschriebenen Code passiert.

1.5 Ergebnis

Abschließend soll ein Einblick über das Entwickeln einer Software mit MDSD gewonnen werden.

2 MODEL-DRIVEN-DEVELOPMENT

Modellgetriebene (Model-Driven) Entwicklung bedeutet in erster Linie, dass Modelle im Zentrum der Entwicklung stehen und aus ihnen Teile der zu erstellenden Software, automatisiert durch Codegeneratoren, erzeugt werden. Domänenspezifische Sprachen ermöglichen eine prägnante Modellierung der Fachlichkeit, helfen, deren Komplexität zu beherrschen, und tragen durch gute Integration mit Generatorwerkzeugen dazu bei, Software produktiver und in besserer Qualität zu erstellen (model driven software development,).

Dem gegenüber dienen Modelle als graphische Repräsentation, die beispielsweise in UML (Unified Modeling Language) notiert werden. Nur partiell erfolgt eine Generierung von Programmcode aus solchen Modellen, der dann noch um wesentliche Tei-

le manuell zu ergänzen ist. In seltenen Fällen werden Änderungen auf der Quellcodeebene auch auf der Modellebene nachgeführt (MDA THE ARCHITECTURE OF CHOICE FOR A CHANGING WORLD,). Mit der Model Driven Architecture (MDA) wurde ein Standardisierungsvorschlag der Object Management Group (OMG) erstellt, der die Repräsentation von Software, von der Programmcodeebene auf die Modellebene hebt. Um die Komplexität auf Modellebene zu reduzieren, werden Modelle und Plattformen verschiedener Abstraktionsebenen unterschieden, die aufeinander aufbauen.

Dabei kann die Realisierung eines plattformunabhängigen Modells durch Wahl einer Plattform, also der konkreten technischen Umsetzung, teilweise oder vollständig automatisiert erfolgen (MDA THE ARCHITECTURE OF CHOICE FOR A CHANGING WORLD,). Modelle werden durch die Auswahl von Plattformen in weniger abstrakte Modelle transformiert bis letztendlich ausführbarer Programmcode entsteht. MDA soll die Portierbarkeit und Wiederverwendung von Modellen und dadurch der Software verbessern.

2.1 Modelle (CIM/PIM/PSM)

In MDA wird hauptsächlich zwischen drei Typen von Modellen unterschieden: den computerunabhängigen (CIM, Computation Independent Model), plattformunabhängigen (PIM, Plattform Independent Model) und plattformspezifischen (PSM, Plattform Specific Model) Modellen (MDA THE ARCHITECTURE OF CHOICE FOR A CHANGING WORLD,).

CIM beschreibt ein Softwaresystem auf fachlicher Ebene. Es liegt in einer Sprache vor, die für die fachlichen Anwender des Systems verständlich ist, und dient zum Diskurs zwischen Softwarearchitekten und Anwendern über Leistungsumfang und Anforderungen. Das CIM legt fest, was ein Softwaresystem leistet. Es definiert nicht, wie es dies leistet oder wie das System strukturiert ist (Anneke Kleppe,).

PIM modelliert die Funktionalität einer Komponente unabhängig von der gegebenen Plattform. Damit enthält ein PIM also den Teil eines Systems, der sich beschreiben lässt, ohne die endgültige Zielplattform zu kennen (Anneke Kleppe,).

PSM kennt eine spezielle Plattform und realisiert ein PIM, wofür die von der Plattform bereitgestellten Schnittstellen genutzt werden (Anneke Kleppe,).

2.2 Transformationen

CIM, PIM und PSM sind die Bausteine des MDA-Ansatzes. Jedes dieser Modelle enthält Informationen, die zum Generieren des Codes der Anwendung benötigt werden. Der Code wird durch eine automatische Generierung aus dem PSM erhalten. Das PSM wird aus einer Transformation vom PIM erhalten. Das PIM wird vom CIM abgeleitet (siehe Abb. 1).

Die Modelltransformation ist ein wichtiger Schritt im MDA-Prozess. Jede Transformation besteht aus der Anwendung von Transformationsregeln, die sogenannten Mappings, die für die automatische Ausführung formal definiert sind.

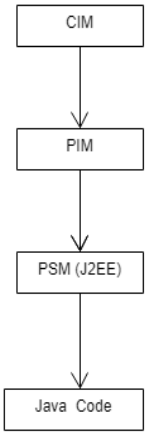

Abbildung 1: MDA-Transformationsworkflow

3 SWAGGER

3.1 Swagger

Swagger ist ein Open-Source Toolpaket, das um die OpenAPI-Spezifikation aufgebaut ist. Dieses Tool hilft dem Nutzer dabei, REST-APIs zu konzipieren, zu dokumentieren und zu konsumieren.

Swagger erlaubt die Beschreibung von RESTful Services über eine formale Spezifikation im JSON- oder YAML- Format. Diese Datei ist dadurch sowohl für Menschen als auch für Maschinen lesbar. Die enthaltenen Tools unterstützen bei der Definition dieser Spezifikation und erlauben die Generierung von Server- und Clientanwendungen in zahlreichen Zielsprachen (Java, JavaScript, Ruby, Scala,).

3.2 OpenAPI

OpenAPI-Spezifikation ist ein API-Beschreibungsformat für REST-APIs. Eine OpenAPI-

Datei ermöglicht dem Benutzer die Beschreibung der gesamten API, einschließlich: Verfügbare Endpoints (/ Users) und Operationen auf jedem Endpunkt (GET / Users, POST / Users) Betriebsparameter Eingabe und Ausgabe für jede Operation Authentifizierungsmethoden Kontaktinformationen Lizenz, Nutzungsbedingungen und andere Informationen. API-Spezifikationen können in YAML oder JSON geschrieben werden. Das Format ist sowohl für Menschen als auch für Maschinen leicht zu erlernen und lesbar. (off,)

3.3 Swagger Tools

Nachfolgend werden die Haupttools von Swagger beschrieben und erklärt.

3.3.1 Swagger Editor

Dieses Tool ist ein browserbasierter Editor, in den der Benutzer OpenAPI-Spezifikationen schreibt und es automatisch gegen die Swagger-Spezifikation vergleichen kann, siehe Abbildung 2. Alle Fehler werden markiert, und Alternativen werden vorgeschlagen.

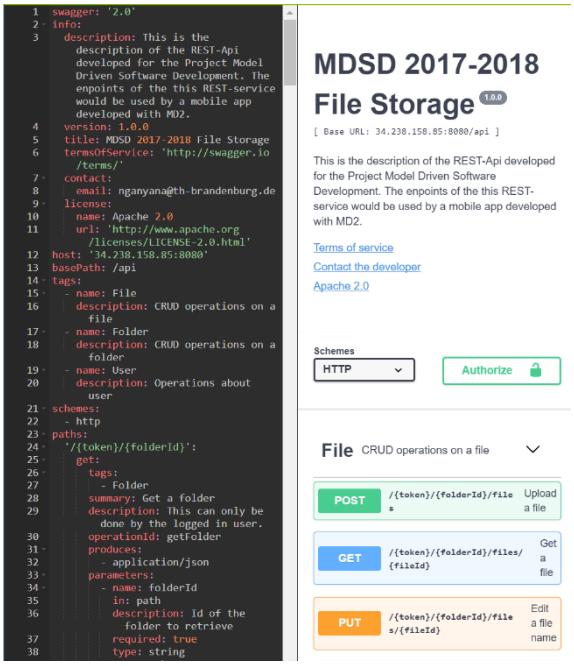

Abbildung 2: Swagger Editor, links OpenAPI-Spezifikation und rechts die Swagger Spezifikation

3.3.2 Swagger UI

Eine andere Verwendung der erzeugten API Spezifikationsdatei, ist die Bereitstellung einer benutzerdefinierten Benutzeroberfläche für Entwickler zum Erforschen der API. Die Swagger UI ermöglicht es den Nutzern der API, die verfügbaren Ressourcen mit HTTP-Verben, Eingabeparametern, Dokumentationen und sogar mit oAuth-Spezifikationen anzuzeigen. Die API wird direkt von der UI aufgerufen, um zu sehen, wie die Anfrage und die Antwort aussehen, siehe Abbildung 3.

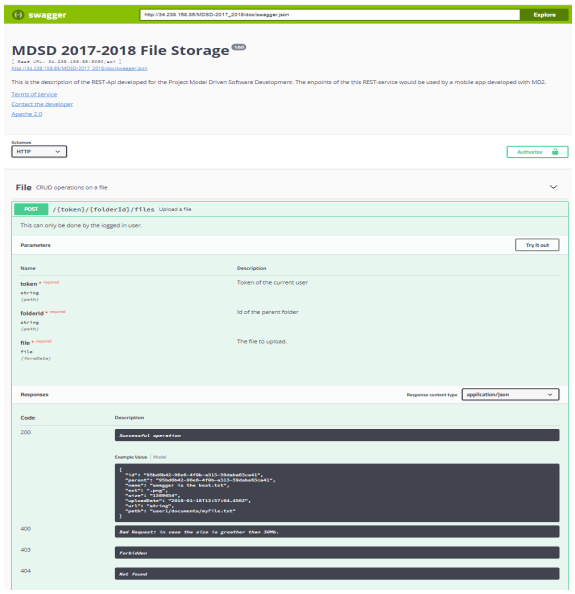

Abbildung 3: Beispiel einer Swagger UI

3.3.3 Swagger Codegen

Swagger Codegen generiert Server-Stubs und Client-Bibliotheken aus einer OpenAPI-Spezifikation. Design-First-Benutzer verwenden Swagger Codegen, um einen Server-Stub für die API zu generieren. Das einzige, was händisch implementiert werden muss, ist die Serverlogik. Swagger bietet die Möglichkeit, Server- und Clientcode aus einer vorhandenen Swagger-Spezifikationsdatei zu generieren. In der Kommandozeile muss der Entwickler den Pfad zur Spezifikationsdatei angeben, beispielsweise:

```
java -jar swagger-codegen-cli-2.2.1.jar generate
-i ../doc/swagger.json -l spring
-o ../MDSD-2017_2018-Swagger-rest-api/
```

4 HIBERNATE

Bei Hibernate handelt es sich um ein Framework zur Abbildung von Objekten auf relationalen Datenbanken für die Programmiersprache Java. Es wird auch als Object Relational Mapping Tool bezeichnet. Hibernate nutzt das Konzept der Reflexion, um

so zur Laufzeit sogenannte Meta-Informationen über die Struktur von Objekten und die zugrundeliegenden Klassen zu ermitteln, die dann auf die Tabellen einer Datenbank abgebildet werden (Hibernate official website,).

Mit Object Rational Mapping (OR-Mapping, ORM) wird eine Technik bezeichnet, bei der die Objekte einer auf Grundlage der objektorientierten Programmierung erstellten Software auf die Strukturen einer relationalen Datenbank abgebildet werden.

5 KONZEPT

5.1 Beschreibung der Software

Die erstellte Software soll ein Dateimanagmentsystem für Studierende sein. Die Studenten sollen sich einloggen und verschiedene Daten hoch- bzw. herunterladen, eine eigene komplexe Ordnerstruktur erzeugen und Dateien und Ordner wieder löschen können. Die Anwendung soll über REST-Schnittstellen zugänglich gemacht werden.

5.2 Planung der Software

Zu Beginn soll ein Überblick verschafft werden, was für die Erfüllung dieses Projekts gebraucht wird. Da die Applikation und Schnittstellen via Swagger beschrieben werden sollen, muss die Architektur klar strukturiert werden. Dafür werden ein Metamodell, ein Klassendiagramm, ein Komponentendiagramm und für die Datenbank ein ERM erzeugt.

5.3 Metamodell

Um einen gesamten Überblick über den Ablauf des Programms zu gewinnen, wurde folgendes Metamodell erzeugt (siehe Abb. 4). Dieses Modell gehört zum CIM. Die Applikation besteht aus einer oder mehrerer Schnittstellen (Api). Diese Schnittstellen werden vom Nutzer via REST aufgerufen. Diese Schnittstellen rufen den entsprechenden Service auf. Dieser bekommt von der Schnittstelle die Parameter als ValueObjects übergeben. Der Service arbeitet mit dem ModelRepository, um Zugriff zu einzelnen Entitäten des Modells zu bekommen. Das ModelRepository liest dafür die Daten aus einer Datenbank und erstellt die einzelnen Entitäten. Der Service kann diese Entitäten nun manipulieren, um so die gewünschte Aufgabe zu erfüllen.

Auf diese Art sollen alle Aufgaben gelöst werden können.

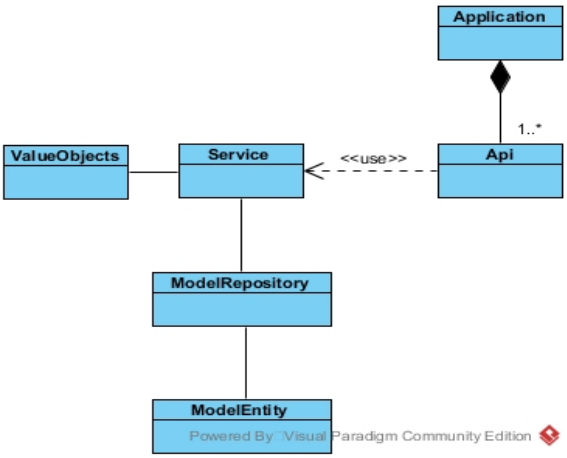

Abbildung 4: Metamodell vom Softwareprojekt

5.3.1 Klassendiagramm

Ein Überblick über die gesamten Datenstrukturen ist ein wesentlicher Bestandteil bei der korrekten Planung einer Software. Aus diesem Softwareprojekt erstellt sich folgendes Klassendiagramm (siehe Abb. 5). Dieses Diagramm gehört zum PIM.

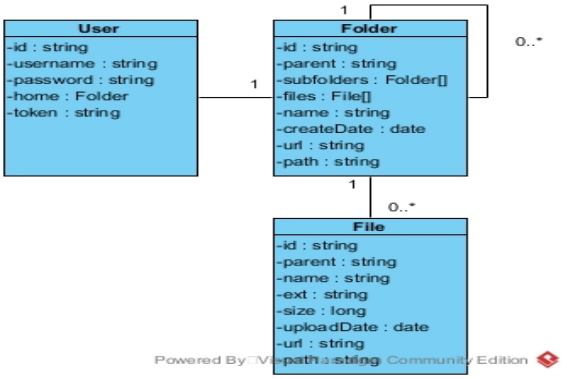

Abbildung 5: Klassendiagramm vom Softwareprojekt

Aus Überlegungen über eine optimale Datenstruktur, kristallisierten sich die Klassen User, Folder und File heraus. Der User soll alle Daten enthalten, die gebraucht werden, um ihn zu identifizieren und zu autorisieren. Daher müssen Id, Username und Password gesetzt sein. Unsere Software plant einen Login-Vorgang, daher muss für die Autorisierung auch ein entsprechendes Token generiert werden. Damit jeder Nutzer auch wirklich auf seine eigenen Ordner zugreifen kann, wurde das Home-Verzeichnis zum User hinzugefügt.

Jeder Folder besitzt eine Id um ihn eindeutig referenzieren zu können. Weil der Nutzer eigene Ordnerstrukturen anlegen kann, muss der Folder, seinen

Parent kennen. Weiterhin kann jeder Folder beliebig viele Unterordner und Files besitzen. Es werden auch allgemeine Daten, wie das Erstellungsdatum, der eigentliche Pfad und die korrekte URL gespeichert. Der Unterschied zwischen Pfad und URL besteht darin, dass der Pfad den Weg von der Wurzel zum Ordner beschreibt, während bei der URL der konkrete Speicherort hinterlegt wird. Damit die Files eindeutig bleiben, wird auch in diesem Objekt eine eindeutige Id hinterlegt. Der Parent der File ist der Ordner, in dem sich die Datei befindet. Es werden weitere Metadaten wie die Dateiendung, die Größe der Datei, das Uploaddatum, die URL und der Pfad gespeichert.

5.3.2 Schnittstellenbeschreibung

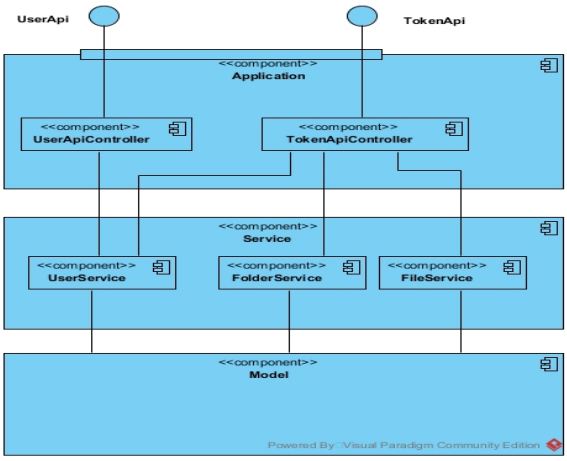

Abbildung 6: Komponentendiagramm vom Softwareprojekt

Die Planung der Schnittstellen ist eine der wichtigsten Aufgaben in diesem Projekt. Sie spiegelt die Funktionalität nach außen wider und muss daher so kompakt wie möglich sein. Während dieser Planung haben sich folgende Aufgaben für die zu erstellende Software ergeben:

1. Userlogin und -logout
2. Registrierung eines neuen Users
3. Auflisten aller Datenstrukturen in einem Ordner
4. Erstellung neuer Ordner
5. Umbenennung von Ordnern
6. Löschen von Ordnern
7. Hochladen von Dateien
8. Herunterladen von Dateien
9. Ändern von Dateien
10. Löschen von Dateien.

Alle diese Aufgaben sind mit REST konform zu lösen. Damit ein Userlogin und logout korrekt ausgeführt wird, muss eine extra Schnittstelle für die Nutzerverwaltung erstellt werden. Da der Nutzer die anderen Ordner- und Dateifunktionalitäten erst benutzen darf, nachdem er eingeloggt ist, wurden alle Operationen auf eine extra Schnittstelle verlegt. Da jeder Nutzer einen Token nach dem Einloggen erhält, wurde dieses Token als Basis genommen und somit alle Funktionalitäten unter einer Token-Schnittstelle gebündelt.

Aus diesen Informationen konnte das Komponentendiagramm erzeugt werden, welches zum PIM gehört (siehe Abb. 6).

Für jede der beiden Schnittstellen gibt es einen entsprechenden Controller, der die entsprechenden Services aufruft. Die Services setzen die gewünschte Aufgabe um. Anschließend wird dem Nutzer ein Response zurückgeschickt.

Damit der REST-Service auch korrekt aufgerufen werden kann, müssen noch die entsprechenden URLs aufgebaut werden. Dieses Beschreibung würde formal dem PSM zugeordnet werden.

Für die Registrierung neuer User muss die URL /users aufgerufen werden. Ein POST Befehl an dieser Schnittstelle erzeugt mit korrektem Body eine Registrierung. Das eigentliche Login findet an der Schnittstelle /users/login statt. Auch hier werden Nutzername und Password im Body mitgeschickt. Serverseitig wird ein neuer Token angelegt. Damit sich der Nutzer korrekt ausloggen kann, muss er die URL /users/logout/token via DELETE-Befehl aufrufen. Durch das mitgeschickte Token kann die Software den korrekten User ausloggen.

Die Folder und File-Operationen verlaufen alle ähnlich. Es werden die CRUD-Operationen via /token/folderidaufgerufen. Das Token authentifiziert den Nutzer und die Folderid den korrekten Speicherort. Für die Folder werden die CRUD Operationen mit entsprechenden Body genutzt.

Beim File wird die URL zu /token/folderid/files erweitert. Mit einem POST-Befehl kann eine Datei zu dem bestimmten Ordner hochgeladen werden. Die restlichen CRUD-Operationen müssen auf die einzelnen Files spezialisiert werden: /token/folderid/files/fileid. Damit können Dateien heruntergeladen, geändert und gelöscht werden.

5.3.3 Persistenz

Damit der Server mit der gesamten Anwendung nicht überfordert wird, soll die Speicherung der Dateien ausgelagert werden. Dafür eignet sich der S3-Service von Amazon Web Services. Es soll eine S3 Instanz

aufgebaut und eine serverseitige Verbindung hergestellt werden. Weiterhin soll diese Lösung auch die Datenbank nicht überfordern, indem dort Dateien beliebiger Größe gespeichert werden.

Für den Aufbau der Datenbank wurde überlegt, wie die Objektstruktur des Klassendiagramms effektiv in eine Datenbank gespeichert werden kann. Im Zuge der Überlegungen wurde folgendes ERM-Diagramm erzeugt (siehe Abb. 7).

Die Tabelle User besitzt einen Fremdschlüssel von Folder. Folder kann auf sich selbst referenzieren und jede File muss auf einen Folder referenzieren.

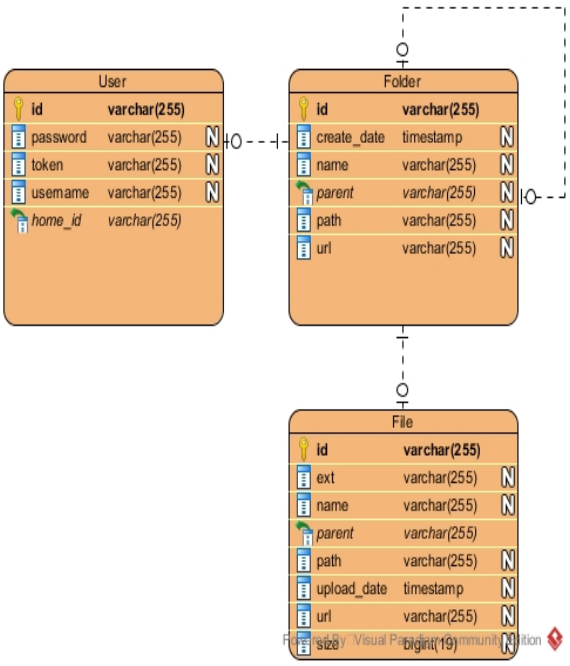

Abbildung 7: ERM vom Softwareprojekt

5.3.4 Erreichbarkeit

Da dieses Projekt in Kooperation mit einem anderen Projekt steht, wurde beschlossen, dass diese Anwendung auf eine Amazon EC2 Instanz installiert werden soll, damit das andere Team stetig mit der Software arbeiten kann.

6 PROTOTYP

In diesem Kapitel geht es um die Beschreibung des Weges zur Erzeugung eines Prototyps. Dieses wurde mittels der Werkzeuge Swagger (off,) (Swagger UI, Swagger Editor und Swagger Codegen), Amazon Web Services und Hibernate erarbeitet. Für eine modellgetriebene Entwicklung wurde während der Durchführung dieses Projekts nicht Bottom-Up entwickelt, sondern Top-Down. Also wurde direkt von angefertigten Modellen zum generierten Produkt entwickelt.

6.1 Funktionalität

Da das Ziel des ganzen, ursprünglichen Projekts ein Anwendung zur Datenverwaltung ist, ermöglicht das in dieser Dokumentation beschriebene Rest-Service, der dafür verantwortlich ist Schnittstellen zu diesem Zweck bereitzustellen, folgende Funktionalitäten:

1. einen Benutzer zu erstellen
2. sich als Benutzer ein- und auszulogen
3. Dateien hochzuladen, herunterzuladen, zu löschen, umzubennenen und aufzulisten
4. Ordner zu erstellen, zu löschen, umzubennenen und zu visualisieren.

Insgesamt sind von dem Rest-Service 11 Schnittstellen zur Verfügung gestellt.

6.2 Schnittstellenerzeugung

Der Prozess zur Erzeugung der Schnittstellen wurde in folgenden Schritten durchgeführt:

1. Zuerst wurden mittels des Werkzeugs Swagger Editor die Modelle, Schnittstellen sowie ein paar Meta-Daten des Projekts unter anderem der Titel, die Beschreibung, die Version beschrieben. All das wird entweder unter einer JSON- oder YAML-Datei gespeichert. In diesem Projekt wurde die JSON-Datei ausgewählt.
2. Anschließend wurde aus dieser Beschreibung eine graphische Dokumentation der Schnittstellen als Webseite mittels Swagger UI erzeugt. Die enthält einerseits die Schnittstellenbeschreibung, andererseits die Modellbeschreibung.
3. Dann wurde ebenfalls aus der JSON-Beschreibung der Quellcode anhand des Swagger-Codengen-Werkzeugs generiert. Swagger Codegen ermöglicht, den Quellcode in vielen Programmiersprachen (Backend sowie Frontend) zu erzeugen. Für dieses Projekt wurde ein Spring-Projekt gewählt, da den angestrebten Prototypen ein Java-basierter Rest-Service sein soll.
4. Danach wurden in dem generierten Spring-Projekt Maven-Abhängigkeiten hinzugefügt. Diese sind unter anderem AWS S3, MySQL und Spring-Data.
5. Der nächste Schritt war dann die Verbindung zu der MySQL-Datenbank und die Modelle gemäß

der verschiedenen Attribute, die in der Datenbank zu speichern sind, zu annotieren. Für diesn Schritt wurde Hibernate benutzt.

6. Abschließend wurde jede von Swagger automatisch generierte Funktion ausgefüllt. Bei der Generierung der Schnittstellen wird von Swagger eine leere Funktion pro Schnittstelle erzeugt. Es bleibt nur noch diese Funktionen auszufüllen.

6.3 Erweiterung der Schnittstellen

Der Rest-Service könnte noch Funktionalitäten bereitstellen, die in diesem Prototypen noch nicht entwickelt worden sind.

6.3.1 Encryption-Möglichkeit

Das UserLogin-Modell kann um das Feld encryption erweitert werden. Dieses Feld ist so gedacht, dass die übertragenen Daten vor der Übertragung verschlüsselt werden. Bei True sollen die Daten chiffriert werden und bei False nicht. Über das Verschlüsselungsverfahren sollen sich die Entwickler entscheiden.

6.4 Rebuild des Projektes nach Schnittstellenbearbeitung

Das Projekt basiert auf einer Menge von Schritten, die nach und nach durchgeführt wurden. Gleich nach der Beschreibung der Modelle und der Schnittstellen bis zum generierten Projekt wird jede Code-Zeile automatisch generiert. Was ist denn, zu tun, falls Änderungen in der Beschreibung vorkommen? Der neu generierte Code soll integriert werden, ohne den bereits selbstgeschriebenen Code zu überschreiben. Zu diesem Zweck werden in dem Paper (Timo Greifenberg and Wortmann, 2015a) ein paar Methoden vorgestellt, die in solchen Fällen nützlich sind. Im Rahmen dieser Arbeit wurde aus Zeitgründen keine dieser Methoden implementiert, aber eine davon hat sich trotzdem durch die fünf von diesem Paper vorgestellten Kriterien (C1 bis C5) (Timo Greifenberg and Wortmann, 2015b) zu diesem Projekt passend gezeigt. Die heißt Generation Gap (Timo Greifenberg and Wortmann, 2015b). Bei diesem Verfahren werden beispielsweise für eine Klasse NotePad ein Interface NotePad und eine Standard-Implementierung NotePadBaseImpl generiert. Diese Klasse zur Implementierung unterscheidet sich von der eigenen Implementierung, die zum Beispiel in der Klasse NotePadImpl geschrieben ist, in sofern als die dazugehörigen Codes unterschiedlich sind und die Klasse NotePadImpl wird zusätzlich bei einer neuen Erzeugung des Codes nicht überschrieben (Abbildung 8). NotePadImpl ist die Implementierung, die von beiden Codes (der generierte und der Selbstgeschriebene) verwendet werden soll.

Gründe für die Wahl dieser Methode sind:

1. Die Struktur des endgültigen Quellcodes in diesem Projekt weicht nur von dem ursprünglichen Code um die Pakete io.swagger.service, io.swagger.repository, io.swagger.configuration. Ähnlich NotePadImpl in (Timo Greifenberg and Wortmann, 2015b) sind die Klassen wie FileService, FolderService und UserService die selbstgeschriebenen Klassen, die nicht überschrieben werden sollten. Dies erfüllt das Kriterium C1.

2. Die anderen Klassen in io.swagger.api, io.swagger.api, io.swagger.model und io.swagger sind hier generierte Pakete. Ihre Inhalte können beliebig überschrieben werden. Das Kriterium C2 ist erfüllt.

3. Die generierten Schnittstellen können im Laufe des Projekt erweitert werden, ohnen den selbstgeschriebenen Code zu beeinflüssen. Das Kriterium C3 ist erfüllt.

4. Das Kriterium C4 ist ebenfalls erfüllt, da der Selbstgeschriebene Code unabhängig von dem generierten Code ist. Dies ist auf die Tatsache, dass der generierte Code keinen Einfluss auf den Selbstgeschriebenen hat.

5. In diesem Projekt wurde der Code in Spring (Java) generiert. Nach einer neuen Erzeugung ist der generierte Code immernoch auf die gleiche Programmiersprache. Dies fordert deshalb keine zusätzliche Programmiersprache. So ist das Kriterium C5 erfüllt.

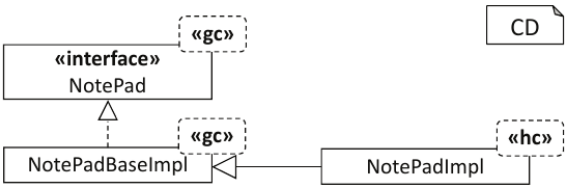

Abbildung 8: Generation Gap Muster für das NotePad-Beispiel
gc: generated code, hc: hand code

7 Ergebnisse und Probleme

7.1 Ergebnisse

Schließlich liegt nach der Arbeitsweise der gezielte Rest-Service vor. Dieser ist in der Lage die gewünschten Funktionalitäten anzubieten, die vorher abgesprochen waren. Die Daten sind durch die Datenbank und Speicherung der Daten auf einer AWS-S3-Instanz tatsächlich persistent. Der Rest-Service kann ebenfalls entsprechende Antworten zurückgeben:

1. 200 OK
2. 400 Bad Request
3. 403 Forbidden
4. 404 Not Found

7.2 Probleme

Während der Entwicklung ist das Team auf Schwierigkeiten gestoßen. Diese waren einerseits aufgrund der Verwendung eines Modellgetriebene-Software-Entwicklungswerkzeugs ausgelöst andererseits aufgrund der Datensicherheit bei der Nutzung vom AWS S3.

7.2.1 Unübersichtlicher, generierter Code

Der von Swagger generierte Code ist an vielen Stellen unübersichtlich. Der Code in den Controllern und Interfaces ist schwer zu verstehen (Abbildung 9).

```
    @ApiOperation(value = "Edit a folder", notes = "This can only be done by the logged in user.", response = Folder.class, tags
    @ApiResponses(value = {
        @ApiResponse(code = 200, message = "Successful operation", response = Folder.class),
        @ApiResponse(code = 401, message = "Unauthorized", response = Folder.class),
        @ApiResponse(code = 404, message = "Not found", response = Folder.class) })
    @RequestMapping(value = "/{token}/{folderId}",
        produces = { "application/json" },
        method = RequestMethod.PUT)
    ResponseEntity<Folder> editFolder(
@ApiParam(value = "Id of the folder to edit",required=true ) @PathVariable("folderId") String folderId


,
@ApiParam(value = "token of the current user",required=true ) @PathVariable("token") String token


,

@ApiParam(value = "Edit a folder in data storage" ,required=true ) @RequestBody FolderCreate folder

);


    @ApiOperation(value = "Get a file", notes = "This can only be done by the logged in user.", response = File.class, tags={ "F
    @ApiResponses(value = {
        @ApiResponse(code = 200, message = "Successful operation", response = File.class),
        @ApiResponse(code = 401, message = "Unauthorized", response = File.class),
        @ApiResponse(code = 404, message = "Not found", response = File.class) })
    @RequestMapping(value = "/{token}/{folderId}/files",
        produces = { "application/json" },
        method = RequestMethod.GET)
    ResponseEntity<File> getFile(
@ApiParam(value = "Token of the current user",required=true ) @PathVariable("token") String token


,
@ApiParam(value = "Id of the parent folder",required=true ) @PathVariable("folderId") String folderId


);
```

Abbildung 9: Quellcode der Schnittstellen getFile und editFile

7.2.2 Projektstruktur

Jeder Entwickler und jede Organisation können ihre eigene Paketstruktur besitzen. Daher ist die von Swagger angebotene Paketstruktur immer auf dem ersten Blick ungewöhnlich (Abbildung 10). In dieser Abbildung ist das Paket io.swagger.service nicht von Swagger sondern von dem Entwicklungsteam generiert.

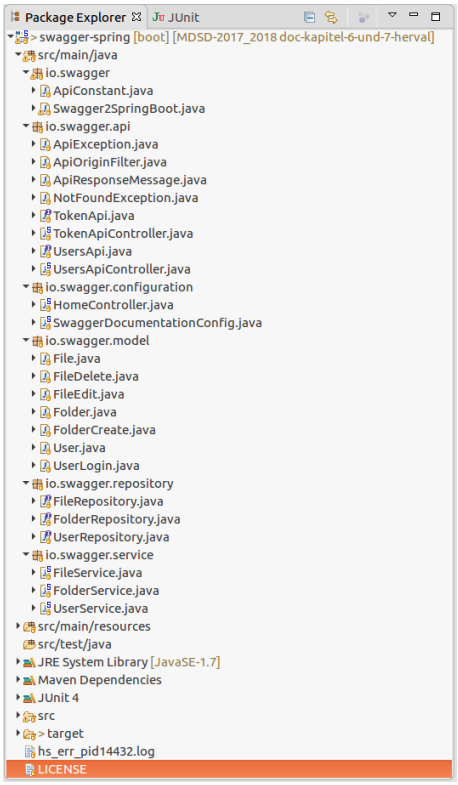

Abbildung 10: Von Swagger generierte und vom Entwicklungsteam veränderte Paketstruktur

7.2.3 Sicherheit

AWS bietet den S3-Dienst an, der ermöglicht Dateien auf eine Cloud zu speichern. Dies erfolgt per Quellcode mittels Access- und Secret-Keys. Damit das Team Transaktionen mit S3 während der Entwicklung durchfüren kann, soll es diese Schlüssel besitzen. Das Problem an der Stelle ist die Verteilung dieser Schlüssel. Diese kann zu bösen Zwecken verwendet werden und schließlich hohe Kosten für den offiziellen Besitzer verursachen. Außerdem wären sie öffentlich für Hacker durch GitHub gewesen, da GitHub zur Versionsverwaltung verwendet wurde. Aus diesen Gründen wurde ein Nexus-Repository entwickelt und geheim gehalten, damit diese Access-

und Secret-Keys nicht verteilt werden. Dieses Repository stellt ein Object S3Transaction zur Verfügung, das seinerseits Funktionen wie upload, rename und delete zur Verfügung stellt. Die S3Transaction-Abhängigkeit ist anhand der folgenden Dependency aufzurufen:

```
<dependency>
<groupId>
com.mdsd-2017-2018.s3-transactions
</groupId>
<artifactId>
s3-transactions
</artifactId>
<version>1.2.2</version>
</dependency>
```

8 Zusammenfassung und Ausblick

Die vorliegende Arbeit beschreibt das Design und die Realisierung einer Software mit dem MDSD Framework Swagger. Dieses Projekt beschäftigte sich mit den Transformationenphasen von CIM zu PIM, von PIM zu PSM und von PSM zum fertigen Code.

Zunächst wurde in Form einer rechnerunabhängigen Plattform (CIM) ein Metamodell erstellt, um die Anforderungen an unsere Funktionalitäten abzubilden. Anschließend wurde ein Klassendiagramm und ein Komponentendiagramm als PIM erstellt. Über das Klassendiagramm konnte ein Eindruck über die benötigten Strukturen und deren Assoziationen zueinander erweckt werden. Durch das Komponentendiagramm wurden die Schnittstellen und deren Zusammenspiel mit den Services beschrieben. Basierend auf dieser Phase wurde eine API-Spezifikationsdatei für REST-APIs via Swagger erstellt (PSM).

Abschließend wurde aus dieser Datei ein fertiger Spring-Code generiert.

Daraufhin wurden im generierten Code eine Implementierung der Serverlogik und eine Integration in die Datenbank vorgenommen.

Insgesamt können der Systementwurf und die Implementierung als gelungen bezeichnet werden. Die wesentlichen Funktionalitäten konnten im Rahmen der Arbeit realisiert und die wichtigsten Anwendungsfälle abgedeckt werden.

Aufgrund der modularen Entwicklung der Software stellt es keinerlei Probleme dar, diese beliebig weiterzuentwickeln. Funktionen wie beispielsweise, die Suche nach Dateien unter Vorgabe eines Suchbegriffs oder Dienste wie Speicherzuweisung für jeden Benutzer können der Anwendung durch Entwicklung einer neuen Spezifikation in die API-Spezifikation Datei hinzugefügt werden.

Eines der bei der Durchführung dieses Projekts getroffenen Probleme war die Überschreibung des vorher generierten Codes sowie des selbstgeschriebenen Codes. Der in dieser Dokumentation beschriebene Prototyp kann weiterhin mithilfe des Generation-Gap-Ansatzes, beschrieben im Kapitel 6, weiter entwickelt werden.

REFERENCES

Offizielle spezifikation von swagger, openapi specification 3.0.0, zuletzt aufgerufen am 19.01.2018. page https://swagger.io/specification/.

Anneke Kleppe, Jos Warmer, W. B. Mda explained: The model driven architecture : Practice and promise. *Addison Wesley*, page 192.

Hibernate official website, Z. a. a. . page http://www.omg.org/mda/.

MDA THE ARCHITECTURE OF CHOICE FOR A CHANGING WORLD, . page http://www.omg.org/mda/.

model driven software development, I. O. . M. pages http://www.itwissen.info/MDSD–model–driven–software–development–Modellgetriebene–Software–Entwicklung.html.

Timo Greifenberg, Katrin Hlldobler, C. K. M. L. P. M. S. N. K. M. A. N. P. D. P. D. R. A. R. B. R. M. S. and Wortmann, A. (2015a). *A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages*. Software Engineering, RWTH Aachen University, Institute for Building Services and Energy Design, TU Braunschweig.

Timo Greifenberg, Katrin Hlldobler, C. K. M. L. P. M. S. N. K. M. A. N. P. D. P. D. R. A. R. B. R. M. S. and Wortmann, A. (2015b). *A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages, Page 76, Chapter 3.1*. Software Engineering, RWTH Aachen University, Institute for Building Services and Energy Design, TU Braunschweig.

Yazdanshenas, A. R. and Khosravi, R. (2009). Using domain-specific languages to describe the development viewpoint of software architectures. *2009 14th International CSI Computer Conference*, pages 146–151.