Generierung einer frontendseitigen GWT Anwendung unter Verwendung von dem MVP Pattern und Dependency Injection

Claudia Schäfer | Marcus Fabarius | Stephanie Lehmann
CMS

7. März 2014

INHALTSVERZEICHNIS

1. EINLEITUNG

2. GRUNDLAGEN

Text...

2.1 Model Driven Architecture

Model Driven Architecture (dt. Modellgetriebene Architektur), kurz MDA genannt, stellt einen bestimmten Ansatz zur Softwareentwicklung dar. Dieses Konzept ist 2001 von der Object Management Group (OMG) veröffentlicht worden und gilt heute als Standard. Hierbei werden Richtlinien zur Spezifikation in Form von Modellen vorgegeben. Aus diesen Modellen, die formal eindeutig sind, wird dann mithilfe von Generatoren automatisch der benötigte Code erzeugt. Ziel der MDA-Architektur ist es, den gesamten Prozess der Softwareerstellung in möglichst plattformunabhängigen Modellen darzustellen, so dass die Software zu einem hohen Anteil automatisch durch Transformationen von Modellen erzeugt werden kann. Die dabei entstehenden Transformatoren können eine hohen Wiederverwendbarkeit und Wartbarkeit sicher stellen. Bei den Modellen handelt es sich im Speziellen, um das Platform Independent Model und das Platform Specific Model, welche bei diesem Projekt auf das Metamodell der UML 2.4 Anwendung fanden. So kann in dieser Arbeit das Profil als PIM verstanden werden, welches dann durch das Klassenmodell (PSM) spezifiziert wurde. Was dies genau bedeutet und wie die verschiedenen Modelle zu verstehen sind, wird in dem folgenden Abschnitt erläutert

2.1.1 Platform Independent Model

Das Platform Independent Model (PIM, dt. Plattformunabhängiges Modell) stellt ein Softwaresystem dar, das unabhängig von der technologischen Plattform ist. Zudem wird die konkrete technische Umsetzung des Systems nicht berücksichtigt. In dem PIM sind alle Anforderungen erfasst. Alles, was es zu spezifizieren gibt im System, ist definiert, jedoch komplett frei von der später folgenden Implementierung. Somit ist nicht nur eine einzige Implementierung des Systems möglich, sondern durchaus mehrere unterschiedliche.

2.1.2 Platform Specific Model

Kombiniert man nun die Funktionalitäten, die im Platform Independent Model definiert sind, mit den Designanforderungen der gewünschten Plattform, so erhält man das Platform Specific Model (PSM, dt. Plattformspezifisches Model).

Dies geschieht über Modelltransformationen. Das nun entstandene PSM kann durch weitere Transformationen immer spezifischere Modelle erstellen, bis letztendlich der Quellcode für eine Plattform generiert wird. Im Gegensatz zum PIM, welches nur die fachlichen Anforderungen definiert, werden beim PSM auch die technischen Aspekte eingebunden.

Allerdings ist zu beachten, dass es sich bei PIM und PSM um relative Konzepte handelt. Ein PSM kann zwar beispielsweise spezifisch für eine Java EE sein, aber noch unabhängig von der Frage für welchen Applikationsserver es optimiert ist. Es stellt in diesem Fall also ein PIM bezüglich des konkreten Systems für die Applikationsserverplattform dar.

2.2 GWT

Das Google Web Toolkit, kurz GWT, ist ein open-source Projekt von Google. Es dient der Entwicklung von Webanwendungen mittels Java. Dabei übersetzt der GWT Compiler den gesamten Java Source-Code in JavaScript Code und DOM-Elemente. Während der Übersetzung des Java Codes zu JavaScript Code werden darüber hinaus Optimierungen vorgenommen wie das Löschen von dead-Code. Dies führt potenziell dazu, dass komplexe Anwendungen im Browser schneller ausgeführt werden können. Darüber hinaus bietet GWT noch weitere Möglichkeiten, die dem Entwickler einer Webanwendung zu Gute kommen. Dazu zählen u. A. das Integrieren von JavaScript Code oder von JavaScript Bibliotheken innerhalb des Java Codes durch das JavaScript Native Interface, kurz JSNI und das sogenannte Code-Splitting, welches einem Entwickler ermöglicht sogenannte Split Points innerhalb des Codes zu setzen, welche dazu führen, dass bei der Ausführung der Anwendung bestimmte Inhalte ab dem Split Point später nachgeladen werden und dadurch die Startladezeit verringern. Google bietet mit zu den genannten Eigenschaften weitere positive Software Engineering Aspekte. Durch GIN (GWT INjection)

2.2.1 MVP

2.2.2 UI-Binder

2.3 Dependency Injection mittels GIN

Dependency Injection alg.

- begriff aus der objektorientierten Programmierung
- zuweisung von einer exterenen Instace, meist zur laufzeit
- eingeführt 2004 von Martin Fowler
  [$http://martinfowler.com/articles/injection.html$]
-

GIN [$http://code.google.com/p/google-gin/$]

- auto Dependency Injection für GWT
- durch einbau in den GWT Generator, zur compile zeit, kaum bis gar kein laufzeit overhead

3. IDEE

In erster Linie soll eine GWT Frontend Anwendung generiert werden, basierend auf einer vorgegebenen Architektur (vgl. Abschnitt 3.1). Dabei ist eines der Hauptziele die leichte Erstellung der GWT Anwendung, ohne Abhängigkeiten zu der zu generierenden Architektur. Darüber hinaus sollen Vorteile durch vorgegebene Architekturpatterns wie der leichte Austausch von Ansichten durch MVP weiterhin nutzbar sein. Zusätzlich ist es wünschenswert, das Verhalten von View Komponenten wie Buttons z.B. für eine Navigation innerhalb der Ansichten zu ermöglichen. Weitere dieser Verhaltensspezifikationen können das öffnen eines Popup's sowie die Übertragung von Daten sein. Auch hierbei steht die einfache Erstellung einer GWT Anwendung und die Konformität der Architektur im Vordergrund.

Es sollen alle von GWT vorgegebenen View Komponenten wie Buttons, Label und Menubars für den Entwickler verwendbar sein, welche zusätzlich untereinander zugeordnet werden können. Des Weiteren sollen Elemente, die auf jeder Ansicht zu sehen sind, wie z.B. ein Header implementiert werden. Dabei werden Layout- und Stylevorgaben nicht berücksichtigt, da dafür UI-Editoren existieren, mit denen eine "schöne, gestylte" Ansicht ermöglicht wird.

Es sollen weitere eigene View Komponenten erstellt werden können, damit, unabhängig der vorgegebenen Architektur, weitere architektonische Maßnahmen erfolgen können. Dazu zählt zum Beispiel die Erstellung einer Datentabelle, welche mehrmaligen Einsatz innerhalb der Anwendung findet. Darüber hinaus soll eine vorgegebene Paketierung sowie selbsterstellte Paketetierung für Views generiert werden.

Zur Erstellung einer GWT Frontend Anwendung soll ein UML-Modell erstellt werden (vgl. Abschnitt 3.3), basierend auf dem zum Generator-Projekt gehörendem UML-Profil (vgl. Abschnitt 3.2), welches durch den Generator (vgl. Abschnitt 3.4) generiert werden soll.

3.1 Ziel-Architektur

Die für die Generierung vorgesehene Architektur basiert auf Architekturkonzepten verschiedener Entwickler und entstand bei der Entwicklung von vorhergehenden GWT Projekten. Diese Architektur stellte sich dabei als Best Practice Lösung heraus, welche jedoch aufwändig und fehleranfällig bei der Umsetzung ist. Dies ist einer der Gründe einen Generator für GWT Frontend Anwendungen zu entwickeln. Im Folgendem wird die umzusetzende Architektur kategorisiert

und anhand der zu erstellenden Klassen und Dateien vorgestellt.

- einmalig vorhandene Dateien und Klassen
  - **index.html**
    HTML Seite über die, durch GWT, die in Java erzeugten View Klassen eingebunden werden.
  - **styles.css**
    CSS Datei für die Festlegung der Style-Eigenschaften.
  - **"Name".gwt.xml**
    Konfigurationsdatei in der u.A. verwendete Bibliotheken sowie Browsereinstellungen und die Zugriffsklasse eingetragen wird.
  - **AppEntryPoint.java**
    Zugriffsklasse, d.h. die Einstiegsklasse für die GWT Anwendung. In dieser Klasse wird u.A. die ContentView sowie die PermanentViews und die HistoryMapper definiert.
  - **AbstractView.java**
    Ist die Oberklasse aller View Klassen Implementierungen innerhalb der esrtellten GWT Anwendung. Diese definiert Eigenschaften für alle Views und ermöglicht als Oberklasse den Austausch der Ansichten.——————
  - **AbstractActivityDefaultImpl.java**
    Diese Klasse wird von allen View Activity Klassen erweitert und dient mittels *start*-Methode dazu die View Klassen aufzurufen und dem Zugriff auf die Views über den Browser mittels *Place*.———
  - **"Name"ViewActivityMapperImpl.java**
    Definiert die *PlaceControllerProvider* damit darüber der View Place aufgerufen werden kann und somit im Browser die View Implementierung erscheint. Diese Klasse existiert prinzipiell einmal. Kommt jedoch eine PermanentView hinzu so kommt je PermanentView eine weitere ViewActivityMapper-Implementierung dazu, welche sich jeweils ihren PermanentViewActivity als *PlaceControllerProvider* definiert. Diese *PlaceControllerProvider* werden dann nicht in der ursprünglich existierenden ViewActivityMapper-Implementierung definiert.————
  - **AppPlaceHistoryMapper.java**
    Ermöglicht über die View Places den Zugriff auf die gezeigte View Implementierung durch den Back-Button im Browser oder innerhalb der View Implementierung nachdem dies in der View Activity implementiert wurde.
  - **AppGinjector.java**
    Über diese Java Klasse wird es u.A. ermöglicht die ViewActivityMapper-Implementierungen sowie den EventBus zu erhalten, welcher die Navigations-Historie enthält bzw. speichert. Darüber können diese weiterhin in dem *AppEntryPoint* definiert werden.

  - **PlaceControllerProvider.java**
    Ist die Schnittstelle zu den View Places. Damit können diese in der ViewActivityMapper-Implementierung aufgerufen werden und dadurch der Zugriff auf die View Implementierungen zur Ansicht im Browser ermöglicht werden. Dies wird über die Einbindung von GIN ermöglicht.
  - **ProductionGinModule.java**
    In dieser Klasse werden die für GIN typischen bind-Befehle definiert. Diese dienen u.A. dazu die View Interfaces an die View Implementierungen zu binden, sowie die Start-View festzulegen.

- View Klassen und Dateien, welche für jede View implementiert werden, basierend auf dem MVP-Pattern
  - **"Name"Activity.java**
    Dient der Implementierung des Presenters sowie der Definierung der View. Darüber hinaus wird innerhalb dieser Klasse der PlaceController definiert, worüber u.A. eine Navigation zwischen den Webseiten im Browser erfolgen kann mittels einer *goTo*-Methode.
  - **"Name"Place.java**
    Diese Implementierungen sehen im groben inhaltlich immer gleich aus. Die Unterscheidung ist hierbei die dazugehörige View. Über diese Klasse wird die Navigation innerhalb der Seiten bzw. View Implementierungen ermöglicht.
  - **"Name"View.java**
    Hierbei handelt es sich um ein Interface, welches des Presenter Interface enthält und ist die Oberklasse für die jeweiligen View Implementierungen. Diese Schnittstelle ermöglicht den einfachen Austausch der verschiedenen View Implementierungen für die Anwendung, welches zusätzlich über einen *bind*-Befehl innerhalb des *ProductionGinModule* festgelegt werden muss.
  - **"Name"ViewImpl.java**
    Dabei handelt es sich um die konkreten View Implementierungen, welche im Browser sichtbar sind. Diese implementieren die jeweilige View und enthalten den durch die View und Activity implementierten Presenter, wodurch die Kontrolle der View Implementierung seitens des MVP-Prinzips abgegeben wird. Darüber hinaus kann diese Klasse die notwendigen View Komponenten definieren, wenn dies erforderlich, z.B. zum Befüllen von Datentabellen, oder dies gewünscht, z.B. zum Zugriff auf Buttons oder zur inhaltsabhängigen Erstellung von View Komponenten, ist. Die View Implementierung ist an eine bestimmte fest definierte *"Name"ViewImpl.ui.xml*-Datei gebunden, durch die Angabe des gleichen Namens ""Name"ViewImpl". Zusätzlich können durch die Annotation *@UIField* die View Komponenten der *"Name"ViewImpl.ui.xml*-Datei innerhalb der View Implementierung aufgerufen und definiert werden.

  - **“Name“ViewImpl.ui.xml**
    Innherhalb dieser Datei können Style-Eigenschaften zu den View Komponenten sowie die View Komponenten definiert werden.———

- View Klassen und Elemente die auf jeder Ansicht zu sehen sind:
  - werden auch nach dem MVP Pattern, wie oben beschrieben, erstellt.
  - innerhalb des AppEntryPoint enthalten und definiert.
  - bilden jeweils eine eigene *“Name“ViewActivityMapper* Implementierung.

Basierend dieser Architektur muss zur Erzeugung einer View, diese eingetragen werden innerhalb der der folgenden Klassen:

- “Name“ActivityMapperImpl.java
- AppPlaceHistoryMapper.java
- ProductionGinModule.java

und folgende Klassen und Dateien erzeugt werden:

- “Name“Activity.java
- “Name“Place.java
- “Name“View.java
- “Name“ViewImpl.java
- “Name“ViewImpl.ui.xml

unter Betrachtung, dass ausschließlich eine View Implementierung existiert. Existieren zu einer View mehrere View Implementierungen so müssen mehrere Eintragungen innerhalb des *ProductionGinModule.java* getätigt werden und mehrere *“Name“ViewImpl.java* und *“Name“ViewImpl.ui.xml* erstellt werden. Dadurch wird eine gute Abstraktion und lose Kopplung geschaffen. Dies ist jedoch sehr aufwändig und fehleranfällig, da leicht ein Eintrag vergessen werden kann und viele Klassen erzeugt werden müssen.

Zu den genannten Architekturvorstellungen gehört zusätzlich eine Paketierung, die auch durch vorherige GWT-Projekte entstand. Der Vorteil der folgenden Gliederung der Pakete besteht darin, dass innherlab der *“Name“.gwt.xml* Konfigurationsdatei das source-Tag, welches den Pfad für den zu übersetzenden Java Code angibt, wie folgt: $< sourcepath =' client'/ >$ definiert werden kann. Folgend die Gliederung der Klassen innerhalb ihrer Packages:

- “projektname“
  - “Name“.gwt.xml

- “projektname“.client
  - AppEntryPoint.java
- “projektname“.client.common
  - AbstractView.java
  - AbstractActivityDefaultImpl.java
  - “Name“ViewActivityMapperImpl.java
  - AppPlaceHistoryMapper.java
- “projektname“.client.gin
  - AppGinjector.java
  - PlaceControllerProvider.java
  - ProductionGinModule.java
- “projektname“.client.view
  - “Name“Activity.java
  - “Name“Place.java
  - “Name“View.java
  - “Name“ViewImpl.java
  - “Name“ViewImpl.ui.xml

Jedoch soll für einen Entwickler die Möglichkeit bleiben innerhalb des view Packages, die Views in Packages zu gliedern. Aus diesem Grund soll an dieser Stelle das view Package nicht Generator-seitig tiefer gegliedert werden.

Anhand der beschrieben Architektur wird ersichtlich, dass die Erstellung eines GWT Projektes hauptsächlich im Bereich der einmalig vorhanden Dateien und Klassen und die Erstellung einer View im groben immer gleich ist. Dies bietet zwar den Vorteil der Vereinheitlichung mehrerer GWT Projekte und gewährleistet eine gewisse Übersichtlichkeit und kurze Einarbeitungszeit in verschiedenen GWT Projekten, ist aber sehr aufwändig und fehleranfällig. Beispielsweise kann über “Copy-Paste“ viel esrtellt und implementiert werden, jedoch ist dabei das Risiko erhöht, dass Einträge vergessen werden abzuändern oder hinzuzufügen. Darüber hinaus kann es passieren, das Einträge enthalten sind wie z.B. einer Bibliothek, welche jedoch nicht mehr benötigt werden und somit nicht im Build-Path enthalten sind. Diese Beispiele führen potenziell alle dazu, dass die gesamte GWT Anwendung nicht mehr startet und die Suche nach dem Fehler erschwert wird, da oftmals viele dieser Fehler flüchtig geschehen können.

3.2 Profil

Eines der Hauptziele ist wie erwähnt die erleichterte Erstellung von GWT Projekten unter Nutzung der durch die Architektur gegebenen Vorteile. Zu welchen auch der einfache Austausch von View Implementierungen unabhängig vom Model gehört. Des Weiteren soll der Aufwand und die Fehleranfälligkeit bei der Erstellung eines GWT Projektes sowie zur Erstellung von Views (vgl. Abschnitt 3.1) minimiert werden werden. Diese Ziele erfordern einen hohen Abstraktionsgrad innerhalb des Profils sowie weiterhin des M1-Modells.

Deswegen soll eine der ersten Überlegungen dazu führen, dass das gesamte GWT Projekt auf ein gemeinsames Element reduziert werden soll. Dieses Element erscheint im Rahmen des, aus der Architektur heraus, umzusetzenden MVP-Patterns in dem die View Klassen und Dateien *"Name"Activity.java*, *"Name"Place.java*, *"Name"View.java*, *"Name"ViewImpl.java* und *"Name"ViewImpl.ui.xml* das gemeinsame Element für alle anderen Klassen und Inhalte bieten. Jedoch erschien dadurch der Aufwand bei der Erstellung von Views nicht minimiert, da durch diese Stereotypenbildung, diese im M1-Modell weiterhin Einsatz finden müssen. Aus diesem Grund ist eine weitere Suche nach dem gemeinsamen Element erforderlich. Dies führt durch die, sich auch in diesem Rahmen als vorteilhaft herausstellende, Architektur dazu, dass dies das unterste Element ist, die **View Implementierung**. Diese erscheint ausreichend für die Erstellung der gesamten einmalig vorhandenen Klassen und Dateien sowie der View Klassen und Dateien, da innerhalb der View Implementierung und der simultanen und vereinheitlichten Namensbennenung alle notwendigen Teile generierbar wären.

Eine weitere Anforderung ist der Austausch der View Implementierungen durch den Einsatz von MVP. Dieser kann jedoch nicht ausschließlich durch die View Implementierung erfolgen, da hierbei eine Möglichkeit gegeben werden muss zu der Verbindung meherer View Implementierungen zueinander. Aus diesem Grund soll das **View Interfaces** genutzt werden. Dies stellt die Verbindung von mehreren View Implementierungen in Form einer Oberklasse her. Zusätzlich bietet dies die Möglichkeit bestimmte Methoden zu definieren, welche für jede implementierende View nützlich ist.

View Implementierungen haben zusätzlich View Komponenten und darüber hinaus ist die Umsetzung einer Navigation bzw. das Verhalten bei Interaktion mit View Komponenten ein weiteres umzusetzendes Ziel. In vorhergehenden Generator Projekten ging die Umsetzung dessen mittels Enumerations hervor. Diese sind sinnvoll wenn eine View Komponente mehere Attribute z.B. eine Value haben soll und kein konkreter Nachbau von bestehenden Frameworks erfolgen soll. In dem Profil wird ein hoher Abstraktionsgrad erwartet, damit eine Vereinfachung gewährleistet werden kann. Deswegen bildet der Einsatz von Enumerations als Typ für View Komponenten einen Mehrwert für die Umsetzung des Profils. Zusätzlich wird ein Layouting als Ziel ausgeschlossen, wodurch die Nachteile der Verwendung von Enumerations verringert werden. Dadurch ergibt sich zusätzlich der Einsatz von **View Komponenten** als Stereotypen mit dem Attribut *type* vom Typ **Enumeration View Objekt Typ**.

Für Navigationselemente wie Buttons und Umsetzung der Navigation ist

eine Überlegung ein weiteres Profil für ActivityDiagramme und somit ein ActivityDiagramm als M1 Modell zu nutzen. Dieses Mittel ermöglicht eine Übersicht über die Navigationsstruktur bzw. Verhaltensstruktur einer Anwendung und ist somit gut geeignet. Dennoch ist das Ziel der Vereinfachung nicht gegeben, da für jedes Navigationselement ein extra Eintrag in einem ActivityDiagramm erfolgen muss und somit eine Redundanz innerhalb der verschiedenen M1-Modelle entsteht und der Mehrwert der Navigationsgenerierung seitens des Entwicklers, welcher das Generator Projekt nutzen möchte, mindert aufgrund des Mehraufwandes. Die Verwendung von Enumerations für View Objekte ermöglichte die Idee, dass für navigierbare bzw. verhaltensbasierte Elemente eine ähnliche Handhabung dienlich sein kann. Diese ermöglichen die Erfüllung der Zielsetzung und bieten über die Angabe eines Attributes *goTo* die Navigation sowie über weiterer Attribute z.B. *openPopup* eine Generierung verhaltensbezogener Inhalte über das M1-Modell. Dies führt zu der Stereotypenbildung der **Navigations View Komponenten** mit **Enumeration Navigations View Objekt Typen**. Diese Enumeration enthält dann z.B. Tree, Button oder MenuBar, d.h. View Komponenten von GWT, welche für Navigationen oder Verhaltensspezifikationen vorgesehen sind.

Navigations View Komponente und **View Komponenten** sind Properties, da eine View Implementierung diese View Komponenten enthalten kann.

Zur Kopplung von großen Views und zur Erweiterung der vorgegebenen Mittel anhand der Architektur soll eine weitere Klasse als Stereotyp dienen, die **eigenen View Komponenten**. Darin sollen bestehende View Komponenten enthalten sein und somit eine eigene View Komponente, z.B. eine große Datentabelle, bilden, die innerhalb einer View Implementierung enthalten sein kann.

Im Bereich von Views z.B. ein Header die auf allen Views innerhalb des Browsers sichtbar sind, soll eine weiterer Stereotypenbildung stattfinden. Diese Views sind **Permanente Views** und gliedern sich zusätzlich in **Header** und **Footer**, da diese konkrete permanente Views sind durch ihre Positionierung in einer View (Header oben, Footer unten). Diese Views verhalten sich simultan zu den normalen Views, weshalb die permanenten Views als View Implementierung von dem View Interface umgesetzt werden können. Dadurch wird zusätzlich der Austausch von View Implementierungen ermöglicht.

3.3 M1-Modell

Basierend auf dem UML-Profil soll das Modell View Implementierungen enthalten, welche wiederum View Komponenten als Properties beinhalten können sowie eigene View Objekte zu diesen zugeordnet sein können. Darüber hinaus sollen über Attribute innerhalb von Navigations View Komponenten, im M1-Modell enthaltene, View Implementierungen angegeben werden, sodass z.B. über das Attribut goTo die Navigation zu dieser View Implementierung durch Generierung erfolgen kann. Darüber hinaus sollen permanente View Implementierungen in Form z.B. eines Headers erstellt werden, welche dann auf jeder View innerhalb des Browsers sichtbar sind. Eine Implementierung eines Headers soll

eine MenuBar enthalten, die durch MenuItems die Navigation zu verschiedenen Seiten ermöglicht. Zusätzlich sollen, z.B. für die Implementierung einer GWT Anwendung, für verschiedene Devices z.B. einen Browser auf einen Desktop oder einem Browser auf einem Smartphone, View Intefaces rstellt werden, welche mehrere Implementierungen haben, sodass diese für verschiedene Devices ausgelegt angezeigt werden.

3.4 Generator

Im Bereich des UML-Profils sowie des M1-Modells sind außer der Views und ihren Implementierungen und ihren View Komponenten die weiteren Klassen und Dateien ungeachtet geblieben. Aus diesem Grund muss der Generator so geschrieben werden, sodass aus dem M1-Modell ein gesamtes GWT Projekt generiert werden kann. Dies soll potenziell auch ermöglicht werden in dem nur eine View Implementierung erstellt wird, welche ohne Attribute und Methoden ausgestattet ist. Dazu soll eine Unterleitung erfolgen, welche einerseits die einmalig vorhandenen Klassen mit Inhalten (auch View abhängiger Inhalte) generiert und andererseits die Views generiert. Existieren View Interfaces so sollen die Klassen *"Interfacename"View.java*, *"Interfacename"Activity.java* und *"Interfacename"Place.java* den Interfacenamen tragen und die Implementierungsklassen bzw. Dateien *"Klassenname"ViewImpl.java* und *"Klassenname"ViewImpl.ui.xml* der View, die Implemntierungsklassennamen. Darüber hinaus sollen die View abhängigen Inhalte, wenn mehr als eine View Implementierung vorhanden ist, mittels eines boolean *binding* dementsprechend unterschieden werden, sodass alle Inhalte basierend auf View Implementierungen auskommentiert werden, bei binding gleich false. Dadurch kann ein einfacher Austausch bei einem Wechsel der View über das MVP-Pattern ermöglicht werden, ohne Code hinzufügen zu müssen. In dem Fall, dass kein View Interface zu einer Implementierung gehört, so werden alle Klassen, Dateien und View abhängige Inhalte mittels des Implementierungsklassennamen generiert.

Für alle Permanenten Views gilt ein ähnliches Vorgehen wie bei den normalen Views. Jedoch haben diese die Besonderheit, dass sie in jeder View zu sehen sind und somit zusätzlich zu der einen *AbstractView* in dem *AppEntryPoint.java* eingetragen werden. Dazu erfolgt eine durch das Profil vorgegebene Unterscheidung zwischen normalen permanenten Views und Header und Footer. Da Header (oben) und Footer (unten) eine feste Positionierung auf einer Webseite haben, werden diese durch den Generator dementsprechend in dem *AppEntryPoint.java* positioniert. Alle anderen vorhandenen permanenten Views sollen dazwischen eingetragen werden und müssen später durch den Entwickler positioniert werden, da dies layoutspezifisch ist.

Der Entwickler soll weiterhin zusätzliche Änderungen vornehmen müssen bzgl. der Startseite der GWT Webanwendung, welches in dem generiertem Code anstatt konkreter Angabe des Klassennamens innerhalb des Befehls über "Start" gekennzeichnet wird. Dieses Vorgehen kann das Verständnis für die Architektur stärken und es werden überflüssige und überfüllende Attribute innerhalb des

Profils und M1-Modells vermieden.

allg. zum Generator z.B. Strukturierung, Redundanz usw.

3.5 Anwendungsfall

Prinzipiell soll es über den Generator möglich sein verschiedene Anwendungsfälle zu generieren. Beispielhaft dient als Anwendungsfall eine einfache Homepage mit u.A. einem Portfolio und einer Newsseite sowie einem Header über dem alle Seiten erreichbar sind und einem Footer um das Impressum zu erreichen. Weiterhin soll mit 2 M1-Modellen gearbeitet werden, in dem der Anwendungsfall inhaltlich leicht abgeändert wird. Ein M1-Modell dient Testzwecken und das andere zur Vervollständigung der gesamten Homepage mit Änderungen in dem generiertem Code z.B. zur Gestaltung der Homepage und einer vollständigen und geordneten Projektstruktur. Dadurch können alle Anwendungsfälle abgedeckt werden, inklusive der Einbindung von View Komponenten, und eine Trennung zwischen notwendigen Änderungen im generiertem Code, durch ein einheitliches Projekt, erfolgen.

4. UML PROFIL AUF M2 EBENE

In diesem Projekt wurde vorab entschieden, den bestehenden Sprachumfang der UML durch UML-Profile zu erweitern. Ein UML-Profil ist genau genommen eine Erweiterung dessen Metamodells und somit des Standard Sprachumfangs der UML und es ist gleichzeitig ein UML-Modell.

Das UML Profil wird mit eigens definierten Meta Klassen erweitert. Grund hierfür ist, dass im späteren UML Modell Zuständigkeiten besser zugewiesen und erkannt werden können. Diese Erweiterungen werden als Stereotype im Profil bezeichnet, die von vordefinierten Metaklassen abgeleitet werden.

Im folgenden is die Endfassung des Profils zu sehen, welche anschließend genauer erläutert wird. [profil.png] In dem Profil wurden zwei Stereotypen für Properties definiert. Die **ViewObject** und die **ViewNavigationObject**, die die Widgets in GWT repräsentieren. Durch die Frontendbetrachtung gibt es nur Elemente, die entweder zum Anzeigen von Informationen (**ViewObject**) oder zum Navigieren auf andere Views (ViewNavigationObject) dienen. Die Trennung der doch ziemlich ähnlichen Elemente musste erfolgen, da es nur so möglich wurde, dass zwar **ViewObjects** andere **ViewObjects** oder **ViewNavigationObjects** enthalten können, bei **ViewNavigationObjects** dies wiederum aber nicht möglich sein darf. Im Sinne einer besseren Übersichtlichkeit wurde sich darauf geeinigt, den Typ der Property mit Hilfe von Enumerations festzulegen. Diese ermöglichen eine Reduzierung der unterschiedlichen Modellelemente. Es wurde hierbei bewusst auf die Darstellung der Widget als eigene Klassen verzichtet, um es später für den GWT Entwickler einfacher zu gestalten. Im Unterschied zur Konzeption wurde festgelegt, dass Multi-Navigationselemente (wie Table, List, Menu oder Tree) als **ViewObject** definiert sind, die dann wiederum ViewNavigationObjects von type **ITEM** besitzen. Die Gruppierung als Items, die zuvor in der Konzeption als Menuitems oder treeitems aufgeführt wurden, ermöglichte es, die Generierung einfacher zu gestalten. Denn alle Items konnten nun gleich generiert werden, unabhängig davon, welchem **ViewObject** sie angehören.

Bei den Widgets lag aus Zeitgründen der Fokus auf den jeweils wichtigsten Elementen (**BUTTON**, **ITEM**, **TEXTFIELD**, **TABLE** etc.), da im späteren Verlauf des Projektes auch alle Elemente generierbar sein sollten. Dies lässt sich aber in zukünftigen Versionen einfach erweitern, indem man zusätzliche Enumerations hinzufügt und die Erstellung dieser im Generator anpasst. Weitere Attribute wie value oder label wurden dann in dem entsprechenden Stereotype als Attribut hinterlegt. Hierbei ist anzumerken, dass das ViewNavigationObject in der ersten Fassung noch ein zusätzliches Attribut goToView

besaß. Im Verlauf der Implementierung des Generators, stellte sich das als fehlerhaft heraus, da dies eine 1:1 Beziehung beschrieb und eine View nur von einem einzigen ViewNavigationObject angesteuert werden konnte. Um dieses Problem zu beheben, wurde eine Assoziation im Profil hinzugefügt. So ist es jetzt beispielsweise möglich, dass unterschiedliche Impressum-Buttons auf die gleiche ViewImpl Impressum verweisen können.

Es wurden die Stereotypen **View** und eine statische **PermanentView** von der Metaklasse Interface abgeleitet. Diese bieten eine hilfreiche Schnittstelle für deren Implementationen. Die beiden Interfaces werden als Stereotype **ViewImpl** und **PermanentViewImpl** der Meta Klasse class implementiert. Durch Erstellen des Boolean - Attributs **concreteBinding** ist es möglich, den entsprechenden bind-Befehl über den Generator zu setzen. Da im Regelfall kein Interface besteht oder es nur eine Implementierung von einem speziellen Interface gibt, ist dieser Wert per Defaultwert auf true gesetzt. In diesem Fall gib es immer nur diese eine spezielle View, die angezeigt werden kann. Sollte es aber mehrere Implementierungen zu einem Interface geben, muss das **concreteBinding** dieser per Hand angepasst werden. Dadurch wird dieses im Anwendungsfall entsprechend gesetzt und gleichzeitig nur die richtige View ausgewählt. Anfangs gab es für die Interfaces mehrere Lösungsansätze. In der ersten Fassung beispielsweise erbten auch **Footer** und **Header** direkt vom **PermanentView** Interface. Da **Footer** und **Header** aber eigentlich **PermanentViewImpl** mit vordefinierten, festen Positionen sind, war es sinnvoller diese direkt von der **PermanentViewImpl** erben zu lassen. So wie es letztendlich in der Endfassung auch umgesetzt wurde.

Für spezielle Klassen wurde noch der Stereotype **OwnViewObject** hinzugefügt, der es dem Entwickler später ermöglichen soll, als Hülle für mehrere unterschiedliche **ViewImpl** oder auch **PermanentViewImpl** zur Verfügung zu stellen. Es dient somit als Widget, das aber die Möglichkeit bietet andere GWT Widgets zu enthalten.

Damit auch hier der Umfang des Arbeitsaufwandes für den GWT Entwickler so gering wie möglich gehalten wird, wurden im Profil die Stereotypen, wie von GWT vorgesehen, Activity und Place nicht angelegt. So müssen sich die Entwickler im M1 Modell zu diesen Klassen keine Gedanken mehr machen, denn mit Hilfe des Generator werden diese automatisch zur entsprechenden View generiert. Auch der direkte Stereotype Model, welcher beispielsweise die Schnittstelle zu einer Datenbank repräsentiert hätte, entfällt in dem Profil. Im Projekt war diese Backend-Anbindung nicht vorgesehen. Daten können später direkt in dem UML M1 Modell angegeben werden oder über XML Dateien geladen werden.

5. AUFBAU UND STRUKTUR M1 MODELL

6. GENERATOR

Der "Model Transformation Language" (MTL) Generator dient zur Erstellung von Quellcode, indem ein zuvor definiertes UML Model mit einem verknüpften UML-Profil verarbeitet werden kann. Hierbei können eigens definierte Regelungen und Vereinbarungen beachtet und miteinbezogen werden. Dies heißt, dass im UML-Model etwas definiert und im MTL Generator entsprechend behandelt wird. Es können somit verschiedene MTL Generatoren zu einem und mehr UML Modellen mit UML Profil angelegt werden. Es kann bzw. soll auch möglich sein, verschiedene UML Modelle, die sich an das UML Profil halten und implementiert haben, Quellcode zu generieren.

Die nachfolgenden Quellcodeauszuüge aus dem MTL Generator werden mit dem Eclipse Plugin Acceleo ausgeführt. Mit der "Object Constraint Language" (OCL) stehen einem MTL Generator diverse zusätzliche Funktionalitäten zur Verfügung. So können verschiedene String oder Mengenoperationen angewendet werden, um ein feineres oder konkreteres Ergebnis zu erhalten. Wiederkehrende und in sich geschlossene Abfragen ohne Seiteneffekte können als "Queries" definiert werden. Diese können anschließend an fast beliebiger Stelle im MTL Generator aufgerufen werden, die stets immer das gleiche Ergebnis liefern. Gleiche Aufrufe werden ebenso gecached, so dass eine schnellere Ausfuührung der Quellcodeerzeugung erzielt wird.

7. ERGEBNIS

8. FAZIT UND AUSBLICK

Anhand des Ergebnisses wird deutlich, dass mit dem Generator Projekt und einigen kleineren Änderungen im generiertem Code eine funktionierende GWT Frontend Anwendung erzeugt werden kann. Durch einen hohen Abstraktionsgrad innerhalb des UML-Profils wird die Entwicklung einer solchen Anwendung vereinfacht und die Fehleranfälligkeit auf ein geringes Maß minimiert ohne Einschränkungen bei der vorgegebenen Zielarchitektur (vgl. Abschnitt 3.1). Darüber hinaus sind die durch GWT und MVP gegebenen Vorteile z.B. der einfache Austausch von Views weiterhin und teilweise leichter nutzbar z.B. durch Aus- und Einkommentierung bei dem Austausch von Views. Zur Gestaltung der Website stehen einem Entwickler alle Möglichkeiten z.B. die Nutzung von Ui-Editoren weiterhin ohne Einschränkungen zur Verfügung und die Anwnendungsfälle sind frei wählbar, wie anhand der 2 M1-Modelle deutlich wird. Es können eigene View Elemente erstellt und eingebunden werden, welches die Architekturkonzepte zusätzlich unterstützt. Des weiteren wird die Navigation über ViewNavigationObjects innerhalb der Seiten generiert, jedoch ist weiteres Verhalten wie z.B. das Öffnen eines Popups nicht umgesetzt.

Durch Abschnitt ... ref wird ersichtlich, dass die vorzunehmende Änderung im Bereich der View Komponenten innerhalb der ui.xml Datei keine optimale Lösung ist. Dies wird hervorgerufen dadurch, dass ViewObjects weitere ViewObjects beinhalten können und somit in der ui.xml mehrfach enthalten sein können. Dies führt zu dem Gedanken, dass eine andere Generierungssprache außerhalb von OCL potenziell besser geeignet wäre, da keine weitere Möglichkeit gefunden werden konnte einen optimalen Lösungsansatz umzusetzen u.A. durch das verändern einer Variable innerhalb einer if-Bedingung. Eine weitere Möglichkeit der Optimierung an dieser Stelle könnte darin bestehen, dass Backend generierbar zu machen. Dadurch können Datenmodelle zu dem UML-Profil hinzugefügt werden, welche auch die Grundlage für View Komponenten bilden können. Anhand eines Beispiels kann das Datenmodell einer Produktklasse mit den Attributen Name, Preis und Verkaufsort dazu genutzt werden, dass innerhalb einer View automatisch eine Tabelle generiert wird, welche als Spalten die Attribute beinhaltet und alle Produkte anzeigt. Dieser Lösungsansatz wäre einerseits eine Weiterentwicklung des Generatorprojekten bietet jedoch noch keine Komplettlösung, sollten Views auch ohne Datenmodelle generiebar sein.

Weiterhin müssen strukturelle Änderungen z.B. das Umsetzen der index.html oder das Hinzufügen von Bibliotheken z.B. GIN (vgl. Abschnitt ref ...) innerhalb des GWT Projektes vorgenommen werden. Diese Änderungen können durch die Verbindung des Generator Projektes mit Maven vermieden werden.

Das zu generierende M1-Modell weist keine Assoziationen auf, wodurch viele Eigenschaften wie das Beinhalten von eigenen View Objekten in Views versteckt bleiben. Aus diesem Grund wäre eine Generierung eines UML Klassendiagramms als Weiterentwicklung ein wichtiger aspekt. Dadurch wird zusätzlich der im Fokus liegende architektonische Aspekt des Generator Projektes hervorhebt. Darüber hinaus sind viele Teile wie z.B. die einmalig vorhandenen Klassen sowie die View Klassen z.B. Activity, Place und ViewImpl.ui.xml in dem M1-Modell nicht ersichtlich bzw. nicht vorhanden, wodurch das generierte unübersichtlich werden kann. Weshalb ein UML Klassendiagramm weiterhin stark unterstützend wirkt.

Eine weitere Möglichkeit Übersicht zu schaffen besteht zusätzlich darin Activitätsdiagramme zu generieren. Dies ermöglicht einen weiteren Überblick über die generierte Navigation.

Zusammenfassend ist zu erwähnen, dass das Generatorprojekt eine gute Grundlage für die Weiterentwicklung darstellt unter der Vorrausetzung, dass die Redundanz und Strukturierung des Generators verbessert wird.