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

Claudia Schäfer | Marcus Fabarius | Stephanie Lehmann
CMS

8. März 2014

INHALTSVERZEICHNIS

1. Einleitung 2

2. Grundlagen 3
2.1 Model Driven Architecture 3
2.1.1 Platform Independent Model und Plattform Specific Model 3
2.2 GWT 4
2.2.1 MVP 4
2.2.2 UiBinder 5
2.3 Dependency Injection mittels GIN 7
2.3.1 GIN 7

3. Idee 8
3.1 Ziel-Architektur 8
3.2 UML-Profil 12
3.3 M1-Modell 13
3.4 Generator 14
3.5 Anwendungsfall 15

4. UML Profil auf M2 Ebene 16

5. Aufbau und Struktur M1 Modell 20
5.1 Projetaufbau 20
5.2 Klassenaufbau 21
5.3 Seitenaufbau 23

6. Generator 25
6.1 Queries 25
6.2 Struktur 26
6.3 Funktion 27
6.4 Probleme 31

7. Ergebnis 32
7.1 Beispiel Anwendung 33

8. Fazit und Ausblick 36

9. Arbeitsaufteilung 38

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.[Andresen, 2004, S. 79 f.] 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. s Was dies genau bedeutet und wie die verschiedenen Modelle zu verstehen sind, wird in dem folgenden Abschnitt erläutert

2.1.1 Platform Independent Model und Plattform Specific 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. 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.[Efftinge et al., 2007, S.377 ff.][Google, 2013]

Allerdings ist zu beachten, dass es sich bei PIM und PSM um relative Konzepte handelt. In diesem Projekt sind sowohl M1 als auch M2 Modell im Bezug auf GWT als PIM zu betrachten. Die eigentliche Spezifizierung für GWT erfolgt erst bei der Model-To-Text Transformation im Generator.

2.2 GWT

Das Google Web Toolkit[1], kurz GWT, ist ein open-source Projekt von Google. Es dient der Entwicklung von komplexen Webanwendungen mittels Java. Dabei übersetzt der GWT Compiler den gesamten Java Source-Code in JavaScript Code. Zudem werden während der Übersetzung Code Optimierungen vorgenommen, welche u.A. das Löschen von nicht benötigtem Code, z.B beim Einsatz von mehreren Browsern als Plattformen, betreffen.

Weiterhin bietet GWT die Möglichkeit zur Interaktion mit JavaScript durch das JavaScript Native Interface, kurz JSNI. Dadurch können JavaScript Bibliotheken angebunden bzw. verwendet oder die Nutzung von JSON erleichtert werden [Hanson and Tacy, 2007, S. 4-9][Seemann, 2008, S. 237-238].

Darüber hinaus ist dadurch eine Kommunikation mit dem Backend möglich. Zusätzlich kann diese Kommunikation durch Remote Procedure Calls, kurz RCPs, u.A. mittels RequestBuilder oder GWT-RCP erfolgen. Beide setzen auf dem XMLHttpRequest JavaScript Objekt auf, welches die Kommunikation zwischen dem Browser und dem Server mittels Ajax erlaubt. Der RequestBuilder ist ein Wrapper für das genannte JavaScript Objekt und GWT-RCP ermöglicht den Austausch von konkreten Java Objekten [Hanson and Tacy, 2007, S. 16][Seemann, 2008, S. 222].

Dies zeigt einen kurzen Einblick in die verschieden Möglichkeiten mit GWT, welche folgend nicht näher erläutert werden, weil eine Generierung einer GWT Frontend Anwendung ohne Server Kommunikation erfolgen soll.

Der Einsatz mit GWT ist flexibel und durch den GWT Compiler wird eine bessere Laufzeitausführung der Webanwendung erlangt[Google, 2010]. Dies sind 2 Vorteile die Nutzung von GWT. Zusätzlich sind die durch Google gebotenen Architekturkonzepte durch u.A. Model-View-Presenter, kurz MVP, ein Ansatz und Grund einen Generator für GWT Frontend Anwnedungen zu schreiben. Dafür werden im weiterem Verlauf MVP, UiBinder sowie GIN erläutert, welche die Grundlage der umzusetzenden Architektur bilden.

2.2.1 MVP

MVP (Model-View-Presenter) ist ein Design Pattern. Es ist ähnlich dem MVC (Model-View-Controller). Google beschreibt den Nutzen des MVP Patterns in der Einbindung von Testfällen in einer GWT Anwendung. Darüber hinaus kann

[1] GWT wurde umbenannt zu Gwit Web Toolkit, seitdem es ein open-source Projekt ist. In dieser Arbeit wird jedoch weiterhin von GWT gesprochen, da das Gwit Web Toolkit weiterhin auch unter GWT anzutreffen ist.

dieses Pattern auch genutzt werden um eine GWT Anwendung für verschiedene Plattformen verfügbar zu machen z.B. für Browser auf mobilen Endgeräten oder auf dem Desktop.

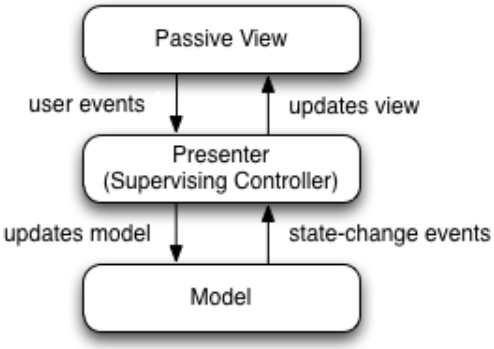

Fig. 2.1: Visualisierung des MVP Patterns [Chandel, 2009]

Bei MVP übernimmt der Presenter die Logik und die View ist einfach gehalten [Ramsdale, 2010]. Dies sorgt für eine klare Trennung zwischen Model und View (vgl. Abbildung 2.1). Wohingegen bei MVC die View das Model kennt [gwt-mvc, 2010]. Der Presenter steuert die View und übermittelt die Daten des Models zur View. Dadurch wird der einfache Austausch von Views ermöglicht ohne das weitere Änderungen vorgenommen werden müssen [Chandel, 2009][Ramsdale, 2010].

In der zu generierenden Anwendung soll das MVP Pattern ohne ein konkretes Model implementiert werden, da ausschließlich eine GWT Frontend Anwendung generiert werden soll. Die MVP Struktur ist dabei so umgesetzt, sodass der Presenter als Interface in dem View Interface enthalten ist und über eine Activity definiert wird, welche zusätzlich für das Event Handling und die Datenbeschaffung verantwortlich ist. Die konkrete Implementierung der View beinhaltet das Presenter Objekt, damit dadurch die Aktionen der View Komponenten (bei GWT Widgets) an den Presenter übergeben werden können.

2.2.2 UiBinder

Das UiBinder Framework für GWT Anwendungen ist ähnlich zu betrachten wie HTML und CSS. Dieses Framework ermöglicht das Layouting von GWT Websites. Dabei wird die Website nicht länger ausschließlich in Java Code gestaltet, sondern deren Komponenten und Layout über eine spezielle, HTML ähnliche, Expression Language deklarativ formuliert. Dies geschieht innerhalb einer ui.xml-Datei. In dieser Sprache können neben Komponenten und Layout auch Styles innerhalb eines ui:Style Tags (vgl. Listing 2.2) definiert werden. Dies bietet die Möglichkeit die View Implementierung zu entkoppeln. Darüber hinaus

existieren noch weitere Möglichkeiten zur Entkopplung der View Implementierung, sodass z.B. statische View Komponenten nur innerhalb der ui.xml Datei enthalten sind und somit ein Überladen der View Implementierung vermindert werden kann. Weiterhin können die Komponenten innerhalb dieser Datei gebunden werden an die Komponenten in der View Implementierung [Google, 2010]. Dazu folgender Codeauszug von einem vorangegangenem GWT Projekt zur Erläuterung dieses Zusammenhangs.

```
1  private static LoginViewImplUiBinder uiBinder = GWT
2      .create(LoginViewImplUiBinder.class);
3
4  interface LoginViewImplUiBinder extends
5      UiBinder<Widget, LoginViewImpl> {}
6
7  @UiField
8  TextBox name;
9
10 //Constructor
11 @Inject
12 public LoginViewImpl() {
13   content.add(uiBinder.createAndBindUi(this));
14 }
15 @UiHandler({ "button" })
16 void onButtonPressed(ClickEvent e) {
17   // do something
18 }
```

Listing 2.1: Beispielcode UiBinder in View Implementierung

```
1  <!DOCTYPE ui:UiBinder SYSTEM
2    "http://dl.google.com/gwt/DTD/xhtml.ent">
3  <ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
4    xmlns:g="urn:import:com.google.gwt.user.client.ui"
5    xmlns:my="urn:import:myprojectpackage">
6    <ui:style>
7      .enterbutton {
8        font-size: 16px;
9        font-weight: bold;
10       padding: 10px;
11       color: #336699;
12     }
13   </ui:style>
14   <g:FlowPanel>
15     <g:Label text="Anmeldename"></g:Label>
16     <g:TextBox ui:field="name"></g:TextBox>
17     <g:Button text="Einloggen" ui:field="button"
18       styleName="{style.enterbutton}"></g:Button>
19   </g:FlowPanel>
20 </ui:UiBinder>
```

Listing 2.2: Beispielcode UiBinder in ui.xml

Die Annotationen @UiField und @UiHandler in der View Implementierung (vgl. Listing 2.1) ermöglichen den Zugriff auf die View Komponenten mit dem jeweiligem Attribut ui:field in der ui.xml (vgl. Listing 2.2). @UiField ist dabei dafür zuständig die Instanz, der im UiBinder-Template definierten Komponente, zu erhalten. Diese kann dann z.B. über den Java Code instanziiert oder Style Eigenschaften gesetzt werden. @UiHandler dagegen ermöglicht die Anmeldung einer Methode auf der Instanz. Darüber kann dann im Falle des Beispiels ein Klick Event auf dem Button ausgeführt werden.

Damit zeigen die Listings 2.1 und 2.2 nur kleine Beispiele für die Nutzung von dem UiBinder Framework, welche innerhalb des Generator Projektes umgesetzt werden sollen.

2.3 Dependency Injection mittels GIN

Bei Dependency Injection handelt es sich um einen Begriff aus der objektorientierten Programmierung, welcher zuerst von Martin Fowler 2004 verwendet worden ist [Martin Fowler, 2004]. Beim Dependency Injection handelt es sich um ein Verfahren, bei dem zur Laufzeit eines Programmes zusätzliche Informationen, beispielzweise beim Aufruf einer Funktion, zur Verfügung gestellt werden. Dies wird von einem extra Dependency Injection Framework vorgenommen, in dieser Arbeit wird handelt es sich dabei um das GIN Framework.

2.3.1 GIN

Das GIN-Framwork, ist ein Framwork für Dependency Injection, es wurde von Google für GWT entwickelt [Google, 2011, GIN]. GIN setzt auf Google Guice [Google, 2008, Guice] auf und erweitert die Java Dependency Injection für den speziellen Anwendungsfall von GWT. GIN wurde direkt in den GWT Generator eingebaut, wodurch es möglich ist die Dependency Injection teilweise schon zur Compile Zeit einzufügen, dies Sorgt dafür das es kaum Laufzeit Overhead gibt.

Guice wurde 2008 von Google für Dependency Injection mit Java entwickelt und war das erste Framwork, das Dependency Injection mit Hilfe von Annotationen ermöglicht hat. Bei Guice werden mittels bind-Befehlen eine Verbindung zwischen Interfasen und deren konkreten Klassen hergestellt, dadurch sind die konkreten Implementierungen leichter Austauschbar.

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 zwischen den Ansichten zu ermöglichen. Weitere dieser Verhaltensspezifikationen können das Öffnen eines Popups 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 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. Layout- und Stylevorgaben sollen nicht berücksichtigt werden, da dafür UI-Editoren existieren.

Es sollen weitere eigene View Komponenten, z.B. eine Datentabelle, die mehrmals verwendet wird, erstellt werden können, damit, unabhängig der vorgegebenen Architektur, weitere architektonische Maßnahmen erfolgen können. Darüber hinaus soll eine vorgegebene sowie eine durch den Nutzer erstellte Paketierung 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) mittels MTL 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 für dieses Generator Projekt mit OCL. 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 der AppEntryPoint eingetragen wird.
- **AppEntryPoint.java**
  Bildet die Einstiegsklasse für die GWT Anwendung und erstellt u.A. die MainView, die permanenten Views und die HistoryMapper.
- **AbstractView.java**
  Die Oberklasse aller View Implementierungen innerhalb einer GWT Anwendung. Dadurch wird der Wechsel der Views u.A. der genannten MainView über die index.html ermöglicht und Eigenschaften wie die Größe aller Views definiert.
- **AbstractActivityDefaultImpl.java**
  Diese Klasse wird von allen View Activities erweitert. Sie dient mittels einer *start*-Methode dem Aufruf der View Klassen und dem Browserzugriff der Views mittels des View *Places*.
- **"Name"ViewActivityMapperImpl.java**
  Instanziiert die *Provider* der Activities (außer die anderer ViewActivityMapper), um darüber den View Place, welcher die Browseradresse angibt, aufzurufen. Diese Klasse existiert prinzipiell einmal, außer es existieren permanente Views wie ein Header. Je permanenter View wird ein weiterer ViewActivityMapper implementiert, welcher jeweils einen *Provider* für die eigene Activity instanziiert.
- **AppPlaceHistoryMapper.java**
  Dient dem History Management, damit der Zugriff auf die View Implementierungen im Browser über den Place stattfinden und eine back-Funktionalität implementiert werden kann.
- **AppGinjector.java**
  Die Schnittstelle zum Zugriff u.A. auf die ViewActivityMapper sowie dem EventBus. Der EventBus dient wie der **AppPlaceHistoryMapper.java** dem History Management und wird u.A. zur Registrierung der Start View benötigt.
- **PlaceControllerProvider.java**
  Die Schnittstelle zu den View Places, welche in den ViewActivity-Mappern aufgerufen werden und dadurch den Browserzugriff auf die View Implementierungen ermöglichen.
- **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 und den UiBindern
  - **“Name“Activity.java**
    Diese Klasse implementiert den Presenter. Darüber hinaus werden die View sowie der PlaceController definiert. Durch den PlaceController wird z.B. die Navigation zwischen den Views ermöglicht mittels einer *goTo*-Methode.
  - **“Name“Place.java**
    Diese Implementierungen sehen prinzipiell immer gleich aus. Unterschieden wird hierbei die dazugehörige View. Über den Place wird die Navigation zwischen den Views ermöglicht, wobei der Name des Places in der URI Zeile des Browsers steht.
  - **“Name“View.java**
    Hierbei handelt es sich um ein Interface, welches das Presenter Interface beinhaltet und die Oberklasse für die jeweiligen View Implementierungen ist. Dadurch wird der einfache View Austausch durch MVP ermöglicht, welches zusätzlich über einen *bind*-Befehl innerhalb des *ProductionGinModule* festgelegt werden muss.
  - **“Name“ViewImpl.java**
    Die View Implementierung, welche im Browser sichtbar ist. Sie implementiert die jeweilige View und beinhaltet eine Instanz des durch View und Activity definierten Presenters, wodurch die Kontrolle gemäß MVP abgegeben wird. Die Klasse kann entweder das gesamte GUI erstellen oder mittels UiBinder einen Teil der View Komponenten abgeben z.B. im Fall von vordefinierten Labels.
  - **“Name“ViewImpl.ui.xml**
    Innherhalb dieser Datei können Style-Eigenschaften für View Komponenten sowie View-Komponenten definiert werden.
- View Klassen und Elemente die auf jeder Ansicht zu sehen sind:
  - werden gemäß dem MVP Pattern und wie eine View erstellt.
  - innerhalb des AppEntryPoint enthalten und definiert.
  - implementieren jeweils einen eigenen *ViewActivityMapper*.

Unter Betrachtung, dass ausschließlich eine View Implementierung existiert und basierend auf dieser Architektur muss zur Erzeugung einer View:

ein Eintrag dazu in den folgenden Klassen geschehen	folgende Klassen und Dateien erzeugt werden
- “Name“ActivityMapperImpl.java - AppPlaceHistoryMapper.java - ProductionGinModule.java	- “Name“Activity.java - “Name“Place.java - “Name“View.java - “Name“ViewImpl.java - “Name“ViewImpl.ui.xml

Existieren zu einer View mehrere View Implementierungen so müssen mehrere Eintragungen innerhalb des *ProductionGinModule* 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 innerhalb 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 und Dateien innerhalb ihrer Packages:

Package	Klassen und Dateien
"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 beschriebenen 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 erstellt 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 UML-Profil

Eines der Hauptziele ist wie erwähnt die erleichterte Erstellung von GWT Projekten unter Einbezug der durch die Architektur gegebenen Vorteile. Zu welchen auch der einfache Austausch von View Implementierungen unabhängig vom Model gehört. Des Weiteren sollen der Aufwand und die Fehleranfälligkeit bei der Erstellung eines GWT Projektes sowie zur Erstellung von Views (vgl. Abschnitt 3.1) minimiert 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 umzusetzenden MVP-Patterns innerhalb der View Klassen und Dateien *Activity.java*, *Place.java*, *View.java*, *ViewImpl.java* und *ViewImpl.ui.xml*. 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 Bereich als vorteilhaft herausstellende Architektur, dazu, dass dieses das unterste Element, die **View Implementierung**, ist. Sie 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 Implementierungen erfolgen, da hierbei eine Möglichkeit zu der Verbindung mehrerer View Implementierungen gegeben werden muss. 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 mehrere Attribute z.B. eine Value haben 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 Activity-Diagramm 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. Somit entsteht eine Redundanz innerhalb der verschiedenen M1-Modelle und der Mehrwert der Navigationsgenerierung wird gemindert 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. TreeItem, Button oder MenuItem, d.h. View Komponenten von GWT, welche für Navigation oder Verhaltensspezifikationen vorgesehen sein sollen.

Navigations View Komponente und **View Komponenten** sollen Properties sein, 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 mehrerer View Implementierung enthalten sein kann.

Im Bereich von Views, z.B. ein Header, die auf allen Views innerhalb des Browsers sichtbar sind, soll eine weitere Stereotypenbildung stattfinden. Diese Views sind **permanente Views** und sollen sich zusätzlich in **Header** und **Footer** gliedern, da diese, durch ihre Positionierung in einer View (Header oben, Footer unten), konkrete permanente Views sind. Diese Views verhalten sich simultan zu den normalen Views (vgl. Abschnitt 3.1), 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 und eigene

View Objekte zu diesen View Komponenten zugeordnet sein können. Darüber hinaus sollen über Attribute innerhalb von dem Stereotyp Navigations View Komponenten, im M1-Modell enthaltene View Implementierungen angegeben werden. Somit kann z.B. über das Attribut goTo die Navigation zu dieser View Implementierung durch die Generierung dessen erfolgen. Weiterhin 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 View Interfaces, welche verschiedene Implementierungen haben, erstellt werden, damit eine Auslegung der angezeigten Views in der GWT Anwendung für verschiedene Plattformen auf verschiedenen Devices getestet werden kann. Diese Plattformen können z.B. ein Browser auf einen Desktop und ein Browser auf einem mobilem Endgerät sein. Dieser Anwendungsfall würde zusätzlich das Generator Projekt auf höchste Ebene prüfen.

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 ausschließlich 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. In dem Fall, dass View Interfaces existieren, sollen die Klassen *"Interfacename"View.java*, *"Interfacename"Activity.java* und *"Interfacename"Place.java* den Interfacenamen tragen. Die Implementierungsklassen bzw. Dateien *"Klassenname"ViewImpl.java* und *"Klassenname"ViewImpl.ui.xml* der View, haben die Implementierungsklassennamen. Darüber hinaus sollen die viewabhängigen Inhalte, wenn mehr als eine View Implementierung vorhanden ist, mittels eines boolean *binding* dementsprechend unterschieden werden. Dadurch sollen alle Inhalte basierend auf View Implementierungen mit dem *binding* gleich false auskommentiert werden. 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 abhängigen Inhalte mittels des Implementierungsklassennamens 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 *MainView* in dem *AppEntryPoint.java* eingetragen werden müssen. 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, sollen diese durch den Generator dementsprechend in dem *AppEntryPoint.java* positioniert werden. 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. Diese betreffen die Startseite der GWT Webanwendung, welche in dem generiertem Code anstatt einer konkreten 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.

Darüber hinaus soll der Generator auch strukturelle und redundanzfreie Anforderungen erfüllen. Dazu sollen Util Generatoren für u.A. Package Declarations, Constants und abfragebedingter Query-Blöcke, welche mehrfach Verwendung finden, geschrieben werden. Zusätzlich sollen weitere Trennungen erfolgen innerhalb der generierbaren Teile. Zu diesen gehören die Generierung der View Interfaces und die Generierung der konkreten View Implementierungen. Wobei hierbei nochmals anzumerken ist, dass die Interface Generierung für die View Implementierungen auf die Generierung der konkreten View Implementierungen zugreift. Eine weitere Trennung soll innerhalb der einmalig vorhandenen Klassen und Dateien erfolgen, da manche Klassen und Dateien nicht View abhängige Inhalte haben.

3.5 Anwendungsfall

Prinzipiell soll es über den Generator möglich sein verschiedene Anwendungsfälle zu generieren. Beispielhaft soll dementsprechend eine einfache Homepage mit u.A. einem Portfolio und einer Newsseite sowie einem Header über dem alle Seiten erreichbar sind als Anwendungsfall dienen. Weiterhin soll mit 2 M1-Modellen gearbeitet werden, in dem der Anwendungsfall inhaltlich leicht abgeändert wird. Ein M1-Modell dient Testzwecken und das andere der 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 Test- und verschiedene 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.

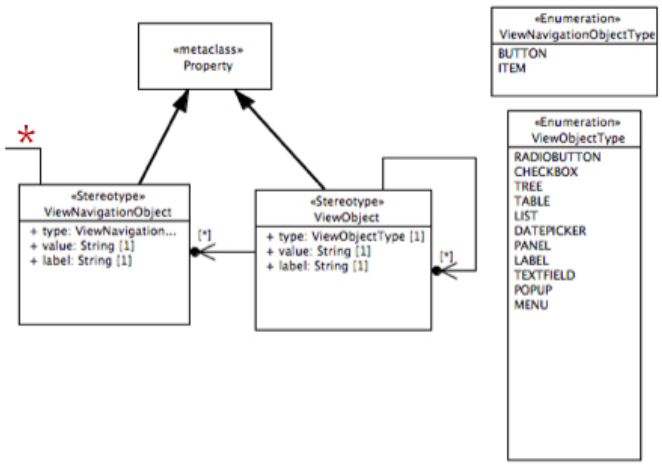

Fig. 4.1: Darstellung der Properties und Enumerations im Profil

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 **Per-**

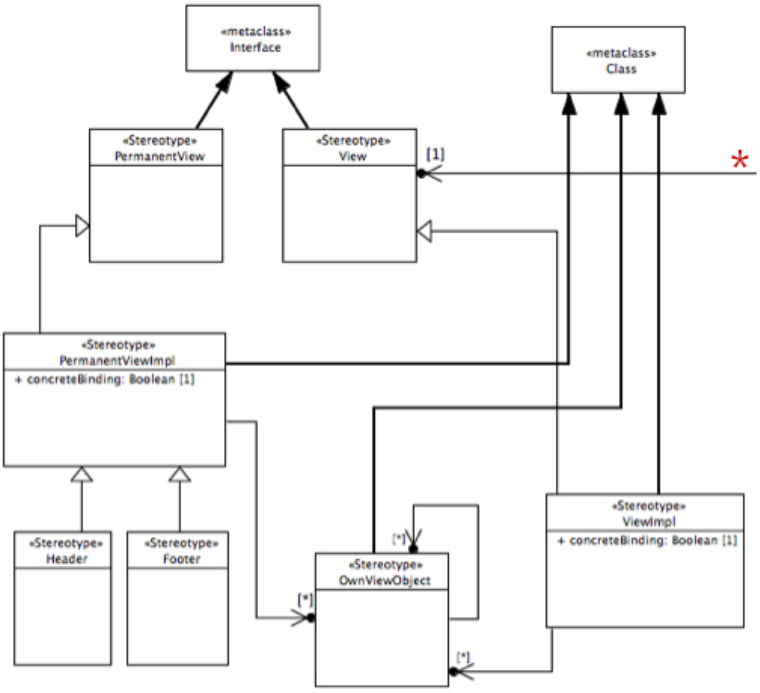

Fig. 4.2: Darstellung der Classes und Interfaces im Profil

manentViewImpl 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 stehen soll. Es dient somit als eigenständiges 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

Beim M1-Modell handelt es sich um die direkte Darstellung der zu generierenden Objekte. Dieses Modell wird direkt durch den Generator, siehe Abschnitt 6, in Quellcode umgesetzt und dient somit als Grundlage des zu generierenden Projektes. In dem in dieser Arbeit beschriebenen Generator, ist es nicht vorgesehen ein vollständig lauffähiges Projekt zu erzeugen, aus diesem Grund ist auch das M1-Modell nicht als vollständige Abbildung für das Projekt zu verstehen.

In dieser Arbeit, soll der Generator aus dem M1-Modell, lediglich eine möglichen Grundstruktur für ein GWT-Projekt erzeugen. Das M1-Modell ist so angedacht das alle Seiten eine eigenständige Klasse darstellen. Des Weiteren können einfache Elemente, wie unter anderem Label und Textfelder den Seiten hinzugefügt werden. Zu dem ist es möglich die Navigation zwischen den Seiten, beispielsweise mit Hilfe eines Menüs, in das Modell mit ein zu bringen.

Durch die Verwendung eines Profils, sind in dem M1-Modell keine Assoziationen zu sehen. Dies sorg dafür das keine Verbindung zwischen den einzelnen Seiten zu sehen ist. Die Navigation wird ausschließlich über die Properties der Navigation Elemente, hier mit `viewNavigationObject` benannt. Dies erschwert das Erkennen der Navigation, sorgt allerding dafür das Unabhängigkeit der Seiten untereinander gut zu erkennen ist.

5.1 Projetaufbau

Wenn ein neues Projekt angelegt wird muss in den Properties des M1-Modelles neben der Angabe des Profils, ein Name definiert werden. Dieser Name stellt später den Hauptpaketnamen des GWT-Projektes da und legt somit den Grundstein für die Paketstruktur (siehe Abbildung 5.1).

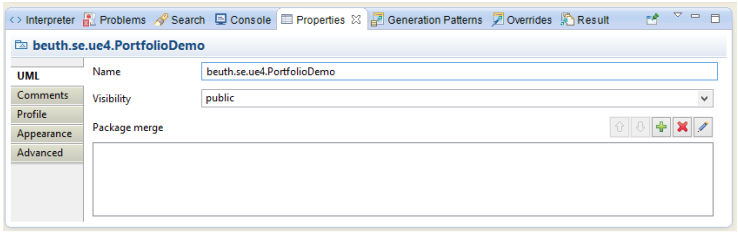

Fig. 5.1: Name des Modelles und gleichzeitig Hauptpakte des GWT-Projekts

5.2 Klassenaufbau

Die Einfachsten Elemente in dem Modell sind `ViewImpl`-Klassen. ein Beispiel dafür ist in Abbildung 5.2 zu sehen. Für jede dieser Klassen-Objekte, werden durch den Generator, alle notwendigen Dateien erzeugt die für den späteren Aufruf, der Seite, notwendig sind.

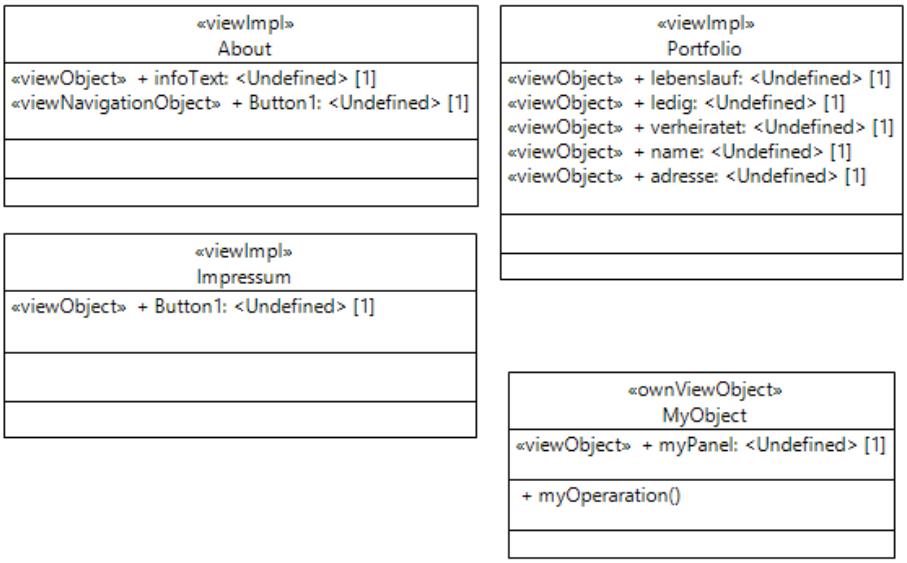

Fig. 5.2: `ViewImpl`-Klassen zur Erzeugung von Seiten für den Webauftritt.

GWT bietet, unter Verwendung von Gin, eine einfache Möglichkeit Seiten auszutauschen. Um dieses Verfahren mit zu generieren wurde sich dazu entschlossen, neben der Standardgenerierung von Seiten, eine zusätzliche Methode einzubauen. Diese Methode bietet die Möglichkeit beliebig viele alternativ Seiten für eine Ansicht im Webauftritt zu erzeugen (siehe Abbildung 5.3).Dieses Möglichkeit ist unter anderem für Testzwecke gedacht.

Bei dem, in Abbildung 5.3, gezeigtem M1-Model Ausschnitt. wird anderes als im allgemeinen Fall (siehe Abbildung 5.2) nicht die komplette Struktur zweimal erzeugt. Die “Name“Activity.java, “Name“Place.java und “Name“View.java Klassen, werden hierbei nur einmal für das Interface und die “Name“ViewImpl.java und “Name“ViewImpl.ui.xml Dateien werden für jede `ViewImpl`-Klassen generiert.

Darüber hinaus ist ein Abbildung 5.3 zu sehen, das es möglich ist Elemente in Paketen zu verpacken, auf diese Weise wird es ermöglicht die zu generierenden Dateien zu gruppieren.

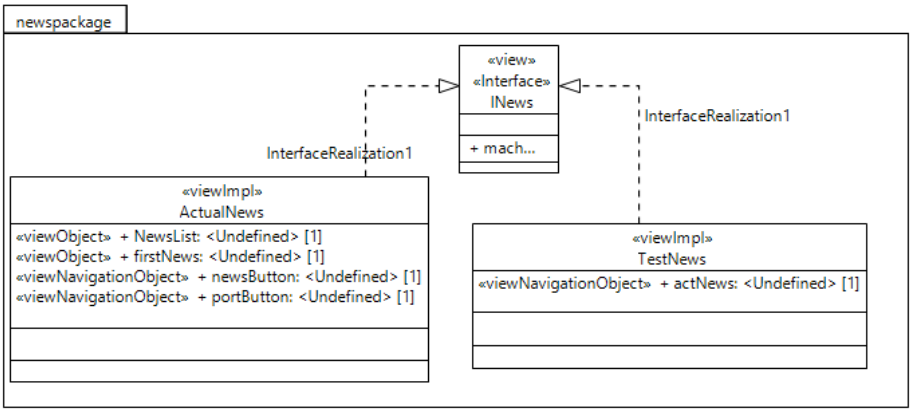

Fig. 5.3: Ein Beispiel für die Verwendung der `ViewInterface`-Klassen um mit Hilfe einer Vielzahl `ViewImpl`-Klassen eine Austauschbare Ansicht zu erzeugen.

Das Letzte Klassenkonstrukt, sind die sogenannten `PermanentView`-Elemente, diese sind dazu gedacht um zum Beispiel Menüs zu erzeugen. Dies haben die Eigenschaft das sie auf allen Seiten zur Verfügung stehen und nicht mehrfach generiert werden sollen.

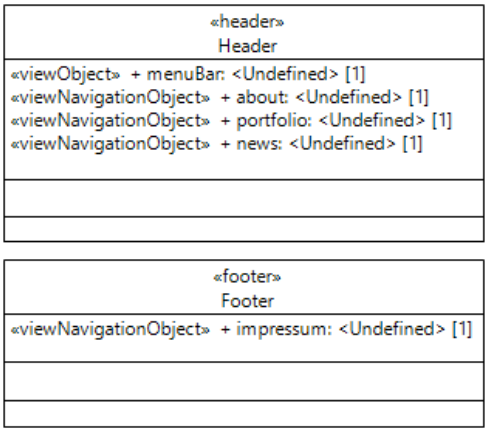

Fig. 5.4: `PermanentView`-Elemente am Beispiel eines Header, mit Menu, und eines Footers.

5.3 Seitenaufbau

Je nach Art der Seite muss zuerst deferiert werden, um was für einen Typ von Seite es sich handelt. In Abbildung 5.5 ist ein Beispiel dafür zu sehen, hier ist eine `ViewImpl` und ein `Header`, eine spezialisierte Version der `PermanentView`, zu sehen. Wichtig ist hier die „concreteBinding“ Eigenschaft, hier kann bei Verwendung eines separaten Interfaces festgelegt werden, ob diese Klasse eingebunden werden soll oder nicht. Diese Einstellung kann später im Quellecode weiter angepasst werden.

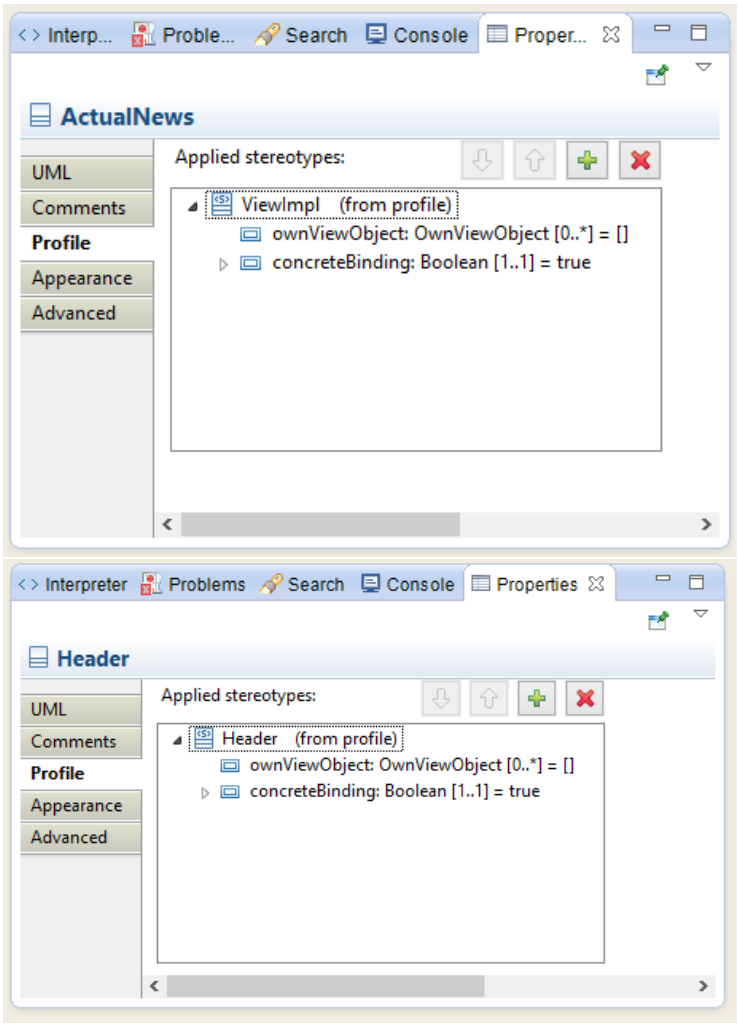

Fig. 5.5: Definition der Stereotypen einer Klassen, oben eine `ViewImpl` Definition und unten ein `Header`

Um deine Seite mit Inhalten zu befüllen, können `ViewObject`s und `ViewNavigationObject`s als Attribute hinzugefügt werden.

Bei `ViewObject`-Elementen können die in Abbildung 5.6 dargestellten Eigenschaften verändert werden.

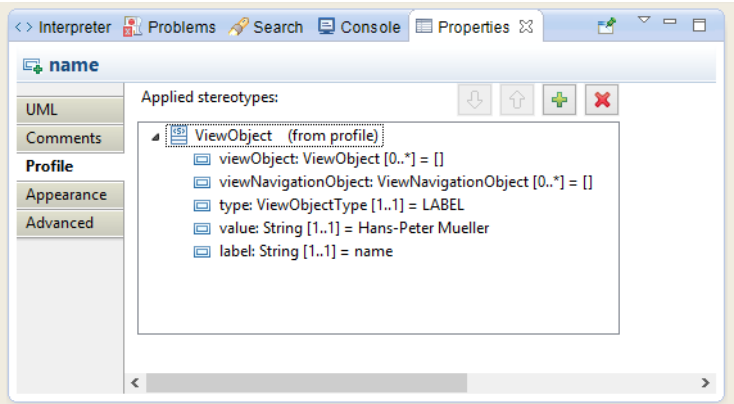

Fig. 5.6: Eigenschaften eines `ViewObject`-Elementes

Bei `ViewNavigationObject`-Elementen können die in Abbildung 5.7 dargestellten Eigenschaften verändert werden. Hier ist vor allem die letzte Eigenschaft „goToView“ zu beachten, welche angibt wohin dieses navigieren soll.

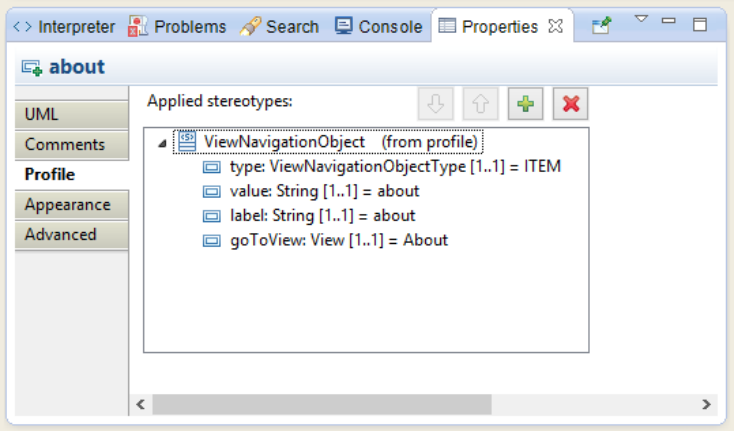

Fig. 5.7: Eigenschaften eines `ViewNavigationObject`-Elementes

6. GENERATOR

Der "Model Transformation Language" (MTL) Generator, in welchem ein zuvor definiertes UML Model mit einem verknüpften UML-Profil verarbeitet werden kann, dient zur Erstellung von Quellcode. 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 mehreren UML Modellen mit UML Profil angelegt werden. Darüber hinaus kann es bzw. soll es auch möglich sein, aus verschiedenen UML Modellen, die sich an das UML Profil halten und entsprechend implementiert wurden, Quellcode zu generieren.

Die nachfolgenden Quellcodeauszüge 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 daraus ein konkreteres Ergebnis zu erhalten.

6.1 Queries

Um das Arbeiten innerhalb des MTL Generators zu erleichtern, werden für häufig verwendete bzw. auch spezielle Ausdrücke sog. Queries benutzt. Diese können dann in dem Generator an Stelle des komplexeren Ausdruck eingesetzt werden. Die verwendeten Queries dieses Projektes sind in der **util.mtl** zusammengefasst. So wird beispielsweise in **isNotViewExisting** über die Klasse abgefragt, ob diese von einem Interface des Stereotypes View implementiert wurde. Ist dies der Fall, gibt das Query ein false zurück anderenfalls ein true. Dieser längere Ausdruck kann auf diese Weise später kompakt als Query an einer geeigneten Stelle verwendet werden.

```
1 [query public isNotViewExisting(class : Class) : Boolean =
2    if(class.interfaceRealization->notEmpty())
3   then if(class.interfaceRealization.target.
        getAppliedStereotypes()->asOrderedSet()->first().name.
        endsWith('View'))
4    then false
5    else true
6       endif
7   else true
```

```
8   endif\]
```

Listing 6.1: Query für isNotViewExisting

Mithilfe der zur Verfügung gestellten Query **getTaggedValue** und deren Java-Methode konnten Tagged Values von Stereotypen über Assoziationen als Liste wiedergeben werden. Die Query musste für dieses Projekt soweit angepasst werden, als dass zu dieser ein weiterer Boolean Parameter **isClass** ergänzt wurde. Dies war notwendig, da die Query sowohl für Klassen als auch für Properties benötigt wurde. Die Hilfsmethode **addToResult** in der **PropertyHelper.java** speichert anhand dieser Boolean Variable die richtigen Elemente als Klasse oder Property in eine Liste.

```
1  private void addToResult(String property, Boolean isClass,
2       List<Object> result, Object values) {
3     if(values instanceof DynamicEObjectImpl) {
4       final DynamicEObjectImpl objectImpl = (DynamicEObjectImpl
            ) values;
5       final EClass c = objectImpl.eClass();
6       EStructuralFeature sf = null;
7       if( isClass ){
8         sf = c.getEStructuralFeature("base_Class");
9       } else {
10        sf = c.getEStructuralFeature("base_Property");
11      }
12      if( sf !=null ){
13        result.add(objectImpl.eDynamicGet(sf, true));
14      }
15    }
16  }
```

Listing 6.2: Hilfsmethode addToResult der PropertyHelper.java

6.2 Struktur

Der Generator ist so konzipiert, dass er mit Hilfe des kleinsten gemeinsamen Nenners, der ViewImpl, alle GWT relevanten Klassen erzeugen kann. Daher ist es auch möglich mit nur einer einzigen Klasse vom Stereotype ViewImpl eine gesamte GWT-Anwendung zu generieren. Der Hauptgenerator bzw. das Haupttemplate im MTL **generate.mtl** ist die Einstiegsstelle zum Verarbeiten, wenn eine Klasse aus dem UML Modell gelesen wird. In diesem Template werden folgenden Templates aufgerufen, die die Struktur für GWT und damit auch die des Generators sicherstellen:

MTL	Zuständigkeit	Beispiele
generateBasicStaticStructure.mtl	Generierung aller einmaligen Klassen, die unabhängig von M1 Modell sind	Erstellung der XML für GWT, der index.html oder der style.css
generateStructure.mtl	Generierung der über das M1 Modell abhängigen Klassen, speziell der ViewImpl-Klassen, anhand der GWT Struktur.	AppGinjector.java, AppEntryPoint.java oder ProductionGinModule.java
generateMVP	stellt die Generierung aller MVP spezifischen Objekte sicher.	View Interface mit entsprechenden Activty und Place Klassen, Implementierungen der konkreten ViewImpls mit dazugehörigen ui.xml Dateien sowie die Implementierung des Presenters über die ‚View'Activity.java
generateOwnViewObject.mtl	Template, welches sich um die gesonderte Generierung des OwnViewObject kümmert.	Generierung einer OwnViewObject Java-Klasse mit im M1 festgelegten Widget-Attributen und Operationen

6.3 Funktion

Um die Funktion des Generations zu verdeutlichen, wird im nächsten Abschnitt auf wichtigsten Strukturelemente ausführlicher eingegangen.

Eines dieser wichtigen Elemente stellt die Klasse des **AppEntryPoint** dar. Durch diese wird die Zugriffsklasse für GWT beschrieben und es werden von dieser gleichzeitig die ersten UI-Panel gesetzt. Hierbei ist im Generator zwischen einer PermanentView und einer ContentView zu unterscheiden. Es können durchaus mehrere PermanentViews vorhanden sein. Der Contentbereich ist dagegen immer nur einmal vorhanden und dient so als Container für alle /textifViewImpls. So müssen alle PermanentViews mittels for-Schleife durchlaufen werden. Zudem sind die Spezialfälle der PermanentView, Footer oder Header, auch unterschiedlich zu behandeln, da diese eine feste Positionierung besitzen. Alle anderen Layout-spezifischen Entscheidungen müssen vom späteren Entwickler gesetzt werden. Sämtliche PermanentViews und der Content werden dann als SimplePanel zu dem GWT **RootPanel** hinzugefügt.

```
[for (class : Class | pack.ownedElement)]
  [if ( class.getAppliedStereotypes()->asOrderedSet()->first()
      .name.endsWith('Header') )]
    RootPanel.get().add( [getClassName(class).toLowerFirst()/]
        );
  [/if]
[/for]
```

Listing 6.3: Hinzufügen eines Panels zum RootPanel, am Beipsiel eines Header

Ein weiteres wichtiges Element ist der aus GIN stammende bind-Befehl, der sicherstellt, dass im Anwendungsfall die richtige Seite (View) angezeigt wird. Damit der bind-Befehl in der ProductionGinModule-Klasse, richtig gesetzt wird, muss mittels Generator aus dem M1 Modell das /textifconcreteBinding Attribut einer ViewImpl Klasse ausgelesen und entsprechend seines Wertes behandelt werden. Wenn die **concreteBinding** Variable den Wert true annimmt, wird das View Interface an die entsprechende View Implementierung gebunden. Sollte der Wert auf false gesetzt sein, so wird dieser bind-Befehl trotzdem ausgeführt, allerdings in kommentierter Form. So ist es später möglich bei Wechsel einer View, den entsprechenden bind-Befehl auszutauschen, ohne neuen Code schreiben zu müssen. Zudem werden im Generator bewusst Fehler erzeugt, die dem Entwickler später darauf hinweisen sollen, an welchen Klassen noch von Hand Änderungen vorzunehmen sind. So passiert dies beispielsweise immer dann, wenn in einer Klasse ein ‚YourStartHere' steht. Hier ist der Entwickler gezwungen, den Namen seiner Start-View einzutragen. Dies geschieht auch bei den bind-Befehlen, bei denen es wichtig ist, dass zuerst die Start-View das binding erhält und anschließend den anderen View Interfaces ihr jeweiliges binding an deren Implementierung zugewiesen wird.

```
bind(YourStartHereActivity.class);
[for (class : Class | pack.ownedElement)]
  [if (not isNotViewExisting(class))]
    [for (p : Property | class.getAppliedStereotypes()->
        asOrderedSet()->first().attribute)]
    [if (p.name.endsWith('concreteBinding'))]
      [if(not class.getValue(class.getAppliedStereotypes()->
          asOrderedSet()->first(),p.name).oclAsType(Boolean))]
    // bind([getClassName(class)/]View.class).to([class.name/]
        ViewImpl.class).in(Singleton.class);
      [else]
    bind([getClassName(class)/]View.class).to([class.name/]
        ViewImpl.class).in(Singleton.class);
      [/if]
    [/if]
    [/for]
  [else]
    bind([getClassName(class)/]View.class).to([class.name/]
        ViewImpl.class).in(Singleton.class);
  [/if]
[/for]
```

Listing 6.4: Auszug aus der Generierung des bind-Befehls

Als weiteres Strukturelement wird im Folgenden die Generator-seitige Umsetzung des MVP Patterns beschreiben. Dieser wurde nach zwei Templates getrennt. Zum einen nach den View Interfaces mit entsprechenden Place und Activity Klassen und zum anderen nach den View Implementierungen mit dazugehörigen ui.xml Dateien. Jedes View Interface besitzt einen Interface Presenter und die entsprechende set-Methode setPresenter () sowie andere Methoden, soweit im Modell festgehalten. Zusätzlich enthält das Interface für jedes Navigations Objekt eine onClicked Methode. Auch hier wird zusätzlich wie bei dem bind-Befehl die gleiche Methode als Kommentar geschrieben, sofern deren concreteBinding-Attribute auf false gesetzt ist. Ein Sonderfall bietet der Button, deren onButtonClicked() Methode nur einmal generiert wird, unabhängig davon wie viele Buttons existieren. Denn die Buttons werden während der Implementierung mit Hilfe von if-Bedingungen innerhalb dieser Methode von einander unterschieden und entsprechend behandelt. In der Activity Klasse wird der Presenter implementiert und über die goto-Methode des PlaceControllers die Navigation zwischen den Webseiten geschaffen. Durch die Implementierung der View Methoden, muss vor jeder onClicked() Methode die @Override Annotation hinzugefügt werden. Diese dient dann als Wrapper Methode für die eigentliche goto-Methode. Innerhalb der zuvor erwähnten onButtonClicked() Methode wird geprüft, ob der übergebende**buttonName** identisch mit dem String der Ziel-Seite ist, denn nur so ist sichergestellt, dass es sich um ein Navigations-Button handelt und dieser dann auch mittels goto-Methode auf die richtige Seite verweisen kann.

```
1  [for (class: Class | interface.getTargetDirectedRelationships
       ().source.oclAsType(Class))]
2      [for (property: Property | class.attribute)]
3        [if (property.getAppliedStereotypes()->asOrderedSet()->
             first().name.endsWith('NavigationObject'))]
4          [if (not getValue(property.getAppliedStereotypes()->
               asOrderedSet()->first(), 'type').oclAsType(uml::
               EnumerationLiteral).name.endsWith('BUTTON'))]
5            [if (class.getValue(class.getAppliedStereotypes()->
                 asOrderedSet()->first(), 'concreteBinding').
                 oclAsType(Boolean))]
6      @Override
7      void on[property.name.toUpperFirst()/]Clicked(){
8        placeController.gotTo(new [getClassName(property.
             getTaggedValue(property.getApplicableStereotypes()->
             asOrderedSet()->first().name, 'goToView', true)->
             asOrderedSet()->first().oclAsType(uml::Class))/]Place
             ());
9      }
10           [else]
11     // @Override
12     // void on[property.name.toUpperFirst()/]Clicked(){
13     //  placeController.gotTo(new [getClassName(property.
```

```
        getTaggedValue(property.getApplicableStereotypes()->
        asOrderedSet()->first().name, 'goToView', true)->
        asOrderedSet()->first().oclAsType(uml::Class))/]Place()
        );
14    // }
15          [/if]
16        [/if]
17      [/if]
18    [/for]
19  [/for]
```

Listing 6.5: Auszug der Generieung der on"Clicked() Methode

Sollte zu einer View Implementierung kein View Interface vorhanden sein, wird View, Activity und Place anhand der Implementierung ähnlich zu dem Interface generiert. In der konkreten ViewImpl werden die ViewObjects und ViewNavigationObjects als **@UIField** entsprechend ihres Types gesetzt. Zur Vereinfachung wurde beispielsweise für eine Liste oder eine Tabelle das GWT-UI Element grid verwendet. Des Weiteren ist noch zu beachten, dass wenn ein Tree oder Menu vorhanden ist, für jedes weitere innere Element ein TreeItem oder MenuItem erzeugt werden musste. So ist sichergestellt, dass die Struktur der Widgets erhalten bleibt und entsprechend ihre GWT Deklarationen funktionieren. Darüber hinaus werden für alle UI-Elemente die ui.xml zur entsprechenden ViewImpl in einem separaten **viewXML**-Template generiert. Hierfür wurde für jedes UI Element ein eigenes Template geschrieben, das dann in dem **viewXML**-Template aufgerufen wird.

```
1  [template public newMenu (property : Property) ]
2  [if (isValueExists(property, 'type', 'MENU'))]
3  <g:MenuBar ui:field="[property.name/]">
4  [if(property.getValue(property.getAppliedStereotypes()->
       asOrderedSet()->first(), 'viewNavigationObject')->notEmpty
       ())]
5    [for (prop : Property | property.getTaggedValue('ViewObject'
         ,'viewNavigationObject', false).oclAsSet().oclAsType(
         Property))]
6    <g:MenuItem ui:field="[prop.name/]" text="[prop.getValue(
         prop.getAppliedStereotypes()->asOrderedSet()->first(), '
         value')/]"/>
7    [/for]
8  [/if]
9  </g:MenuBar>
10 [/if]
11 [/template]
```

Listing 6.6: Template für die XML - Generierung eines Menus

Abschließend ist zum OwnViewObject zu sagen, dass dieses durch Vererbung der GWT - Widget Klasse als eigenständiges Widget erzeugt wird. Das OwnView-Object besitzt keinen eigenen Konstruktor, da keine Kenntnis von deren Inhalt bekannt ist. Denn das Aussehen dieser Klasse wird komplett dem Entwickler überlassen und steht zum Zeitpunkt der Generierung noch nicht fest. Dies hat

zur Folge, dass die Generierung dieser Klasse weitestgehend als eine normale Klasse behandelt wird, die schon aus dem Modell vorhandene UI Elemente als eigene Attribute und Operationen generiert.

6.4 Probleme

Ein umfangreiches Softwareprojekt wie dieses führt unweigerlich eine Reihe unterschiedlichster Probleme herbei. Speziell im Generator sind einige davon aufgetreten. Eines der größten Probleme war in MTL der Umgang mit Variablen. So war es nicht möglich, trotz etlicher verschiedener Versuche, eine Boolean Variable anzulegen, die nicht final ist. Es wurde beispielsweise mit dem von MTL zur Verfügung gestellten /textitlet[] Block versucht, unterschiedliche Variablen anzulegen und innerhalb dieses Blockes zu verändern. Dies gestaltete sich als unmöglich, da im Let-BLock definierte Variablen immer final gesetzt werden. Anschließend wurde getestet, ob man mit Hilfe einer Java Klasse (/textitBooleanHelper.java), eine einfache Klasse mit einem Getter und Setter für ein Boolean Wert anlegen und abfragen konnte. Die Setter-Methode funktioniert soweit einwandfrei. Das Problem bestand aber darin, diese zuvor gesetzte Variable mittels Getter-Methode wieder auszulesen. Beim Ausführen der Queries kam der Verdacht auf, das sobald die Setter-Methode aufgerufen wurde, dieser Vorgang eine neue Instanz der Klasse erzeugte, welche nicht mehr den zuvor gesetzten Wert gespeichert hat. Als nächsten möglichen Lösungsschritt wurde versucht die Java Klasse als Singleton zu gestalten, aber auch dies führte zu unterschiedlichsten Fehlermeldungen oder gar gänzlicher Verweigerung der Ausführung. Dieses Problem konnte somit im zeitlichen Rahmen dieses Projektes vorerst nicht gelöst werden, sodass speziell in der /textitui.xml die doppelten UI-Elemente per Hand entfernt werden müssen.

Ein weiteres Problem ist die Redundanz im Generator. Bei der Entwicklung wurde der Fokus auf die Funktionalität des Generators gelegt. Aber durch das doch sehr umfangreiche Projekt, fehlte am Ende die Zeit, eine gründliche Optimierung vorzunehmen, so dass noch einige unnötige Redundanzen auftreten. Dieses erschwert zwar die Lesbarkeit des Generators erschwert, aber seine Funktionalität ist trotzdem gegeben.

7. ERGEBNIS

Das Ziel bestand darin eine Grundstrucktur für ein GWT-Projekt zu erzeugen. Um das Projekt lauffähig zu machen, sind noch ein paar Handgriffe zu erledigen.

- **Imports**
  Zum einen ist es notwendig die noch fehlenden Imports in den Java-Klassen einzufügen.
- **AppEntryPoint.java**
  Hier muss defieniert werden welche der Views die Start Seite werden soll.
- **AbstractView.java**
  In dieser Klasse ist es möglich eine Standard Größe für die Seiten zu bestimmen.
- **ProductionGinModule.java**
  Auch hier muss wie in der AppEntryPoint-Klasse noch einmal die Start Seite definiert werden.
- ***.ui.xml**
  In diesen Dateien müssen doppelte Elemente entfernt werden, die der Generator zu viel eingefügt hat.
- **index.html und style.css**
  Diese beiden Dateien müssen nach dem durchlauf des Generators in den „war“ Ordner kopiert werden. Sie dienen als Ausgangspunkt für die Webanwendung.
- **Zusätzliche Bibliotheken**
  Zusätzlich zu den „GWT SDK“ und „App Engine SDK“ müssen die hier aufgeführten .jar Bibliotheken in das Projekt hinzugefügt werden.
  - **gin-2.0.jar**
  - **guice-3.0-no_aop.jar**
  - **guice-assistedinject-3.0.jar**
  - **gwt-servlet.jar**
  - **javax.inject.jar**

Sind diese Änderungen vorgenommen, ist das Projekt lauffähig. Nun können weitere Änderungen vorgenommen werden, zum Beispiel können weitere Texte eingefügt werden, natürlich kann auch das Design angepasst werden, dieses wurde von Generator nur rudimentär, in Form einer einfaschen css-Datei angelegt.

7.1 Beispiel Anwendung

Für das in diesem Abschnitt gezeigte Beispiel wurde das Modell aus Abbildung 7.1 verwendet. Zu sehen sind 5 Seiten, welche einzeln in Paketen liegen, darüber hinaus wurden zwei PermanentViews modelliert, eine für den Header, mit eingebautem Menu und eine für den Footer, in dem eine Navigation zum Impressum vorgesehen ist.

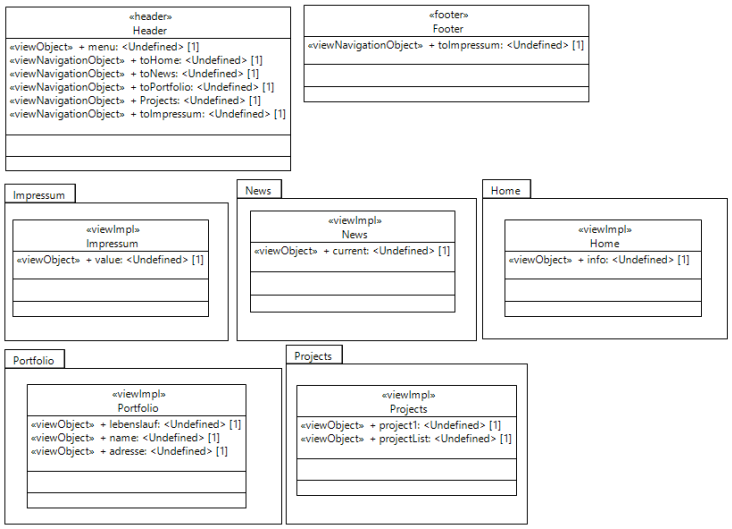

Fig. 7.1: Verwendetes Modell

Durch die extra eingebauten Pakete erweitert sich die Grundstrucktur des Projektes, in Abbildung 7.2 ist auf der linken Seite die Komplette Packet-Struktur des Projektes zu sehen. Hier ist zu erkennen das es für die fünf Seiten separate Pakete gibt, wobei für den Header und den Footer keine zusätzlichen Pakete angelegt worden sind. Auf der rechten Seite der Abbildung 7.2, wird der Inhalt einiger der Pakete näher gezeigt, hier ist zu erkennen das die beiden `ViewImpl` die nicht in Paketen liegen direkt in `view` Packet liegen, zusätzlich sind die erweiterten Klassen `Activity`, `Place`, `View` und die `.ui.xml` Datei in dem Packet zu sehen. Am Beispiel der „Home“ Seite ist gezeigt, dass die hierfür generierten Klassen und Dateien in einem separaten unter Packet von `view`, nämlich `home` liegen.

In diesem Beispiel soll die `home` Seite Den Start Punkt für die Webanwendung darstellen, aus diesem Grund wurde entsprechend in der `AppEntryPoint` und der `ProductionGinModule` die jeweilige Klasse in für den Startpunkt angegeben. (Listing 7.1 und 7.2)

```
1 [..]
2   historyHandler.register(injector.getPlaceController(),
```

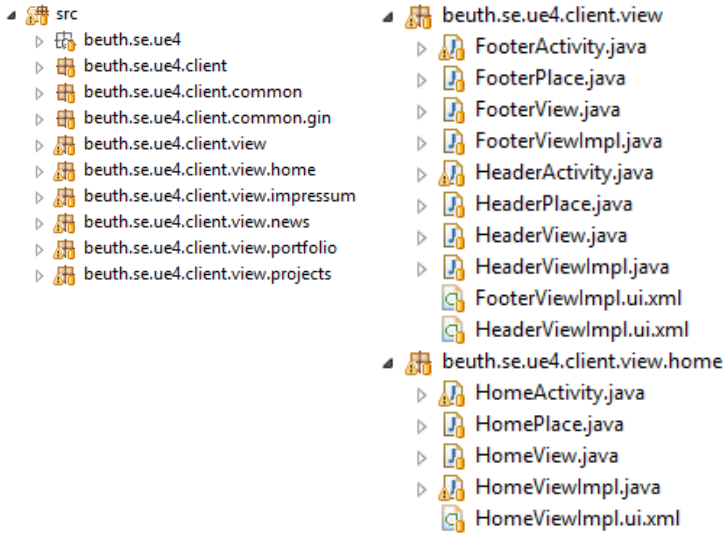

Fig. 7.2: Generierte Paketstrucktur, links gesamte Paketstrucktur im Überblick; rechts Beispiel für das Strukturieren im Modell, mit zusätzlichen Paketen

```
      eventBus,
      new HomePlace());
[..]
```

Listing 7.1: Änderung an der AppEntryPoint Klasse zur Bestimmung der Startseite

```
[..]
  bind(HomeView.class);
[..]
```

Listing 7.2: Änderung an der ProductionGinModule Klasse zur Bestimmung der Startseite

Zusätzlich wurde an einigen Stellen der Inhalt der Seiten erweitert, dies ist einer seits dadurch möglich das innerhalb der `ViewImpl` Klasse zusätzlicher Inhalt eingefügt werden kann, aber auch innerhalb der `.ui.xml` Datei, beide verfahren werden hier anhand eines Beispiels gezeigt. Änderungen am Java-Code siehe Listing 7.3 und innerhalb der UI-Datei siehe Listing 7.4.

```
[..]
  HTML html = new HTML("<div><h1>Brandhei&szlig;e News</h1>"
  +"Lorem ipsum [..] amet."
  +"<div/>");
  content.add(html);
[..]
```

Listing 7.3: Einfügen von Inhalten auf einer Seite durch Veränderungen am Java-Code,

hier in der `NewsViewImpl.java`

```
[..]
 <g:FlowPanel>
  <!-- InteractionElements -->
  <!-- Start of user code Home
       Start protectetRegion -->
      <g:Label text=" Lorem [..] facilisi. "/>
      [..]
  <!-- End of user code -->
  </g:FlowPanel>
[..]
```

Listing 7.4: Einfügen von Inhalten auf einer Seite durch Veränderungen am UI.XML-Datei, hier in der `HomeViewImpl.ui.xml`

Dies sind nur zwei Arten wie die Anwendung weiter bearbeitet werden kann, nach dem der Quellecode aus dem M1-Model generiert worden ist. Zum einen unterstützen der Generator und das Profil nicht alle, von GWT bereitgestellten, UI-Elemente so, dass diese nachträglich eingefügt werden müssen. Des Weiteren wurde sich in dieser Arbeit bewusst gegen das Erzeugen einer Ausdesignten Webanwendung entschieden. Aus dem Grund, dass es diverse Editoren gibt, welche das Erzeugen einer Benutzer Oberfläche graphisch unterstützen.

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 Anwendungsfälle 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 sind weitere 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 Generator Projektes, bietet jedoch noch keine Komplettlösung, sollten Views auch ohne Datenmodelle generierbar sein.

Innerhalb des Generators wurde bereits in Abschnitt ... ersichtlich, dass nicht alle Optimierungen im Bereich Struktur und Redundanz vorgenommen wurden. Dies muss weiterhin geschehen, unterliegt jedoch trotzdem dem Aspekt der Verwendbarkeit von OCL. Dies liegt daran, dass viele unterschiedliche if-

Bedingungen und viele if-Bedingungen mit ähnlichen if-else-Zweigen enthalten sind. Eine Überlegung wie dies mit OCL lösbar wäre ist weiterhin erforderlich, wobei ein Test mit anderen Generierungssprachen, die diesbezüglich mehr Möglichkeiten zur Strukturierung bieten, denkbar ist.

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 hervorgehoden. 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 Activitydiagramme zu generieren. Dies ermöglicht einen weiteren Überblick über die generierte Navigation.

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

Backen nicht. Datenmodelle nicht

9. ARBEITSAUFTEILUNG

In diesem Projekt wurde der schriftliche Teil klar getrennt.

Stephanie hat die Kapitel MDA, UML-Profil auf M2-Ebene und Generator geschriben. Marcus hat die Kapitel Dependency Injection, Aufbau und Struktur M1-Modell und das Ergebnis gechrieben. Claudia hat die Kapitel GWT, Idee ... un Fazit und Ausblick geschriben.

Dagegen wurde der praktische Teil und die Konzeption des Projektes nicht strikt aufgeteilt. In den meisten Fällen waren alle gemeinsam an dem Projekt beteiligt.

Wir würden uns jedoch gerne nach der Notenbekanntgabe die Möglichkeit offen halten, die Noten innerhalb des Teams evtl. umverteilen zu dürfen.

LITERATURVERZEICHNIS

[Andresen, 2004] Andresen, A. (2004). *Komponentenbasierte Softwarentwicklung: mit MDA,UML2 und XML*. Carl Hanser Verlag München Wien, München, Germany. Auflage 2.

[Chandel, 2009] Chandel, S. (2009). Testing. `http://www.gwtproject.org/articles/testing_methodologies_using_gwt.html`. Letzter Aufruf: 06.03.2014.

[Efftinge et al., 2007] Efftinge, S., Haase, A., Stahl, T., and Völter, M. (2007). *Modellgetriebene Softwarentwicklung: Techniken, Engineering, Management*. dpunkt.Verlag GmbH, Heidelberg, Germany. Auflage 2.

[Google, 2008] Google (2008). Guice. `https://code.google.com/p/google-guice/`. Letzter Aufruf: 08.03.2014.

[Google, 2010] Google (2010). Overview. `http://www.gwtproject.org/overview.html`. Letzter Aufruf: 07.03.2014.

[Google, 2010] Google (2010). Ui Binder. `http://www.gwtproject.org/doc/latest/DevGuideUiBinder.html`. Letzter Aufruf: 07.03.2014.

[Google, 2011] Google (2011). Gin. `https://code.google.com/p/google-gin/`. Letzter Aufruf: 08.03.2014.

[Google, 2013] Google (2013). MDA. `http://www.itwissen.info/definition/lexikon/modell-driven-architecture-MDA.html`. Letzter Aufruf: 07.03.2014.

[gwt-mvc, 2010] gwt-mvc (2010). gwt-mvc: MVC layer built on top of GWT. `http://code.google.com/p/gwt-mvc/wiki/MVCvsMVP`. Letzter Aufruf: 06.03.2014.

[Hanson and Tacy, 2007] Hanson, R. and Tacy, A. (2007). *GWT in Action: Easy Ajax with the Google Web Toolkit*. Manning Publications Co., Greenwich, CT, USA.

[Martin Fowler, 2004] Martin Fowler (2004). `http://martinfowler.com/articles/injection.html`. Letzter Aufruf: 08.03.2014.

[Ramsdale, 2010] Ramsdale, C. (2010). MVP Part1. `http://www.gwtproject.org/articles/mvp-architecture.html`. Letzter Aufruf: 06.03.2014.

[Seemann, 2008] Seemann, M. (2008). *Das Google Web Toolkit: GWT*. O'Reilly Verlag GmbH & Co. KG, Köln, Germany. Auflage 1.