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

Claudia Schäfer | **M**arcus Fabarius | **S**tephanie Lehmann
CMS

Team:

Claudia Schäfer	798603
Marcus Fabarius	798540
Stephanie Lehmann	798110

Beuth Hochschule für Technik Berlin
Software Engineering
Dozent: Ilse Schmiedecke
Abgabe: 10. März 2014

INHALTSVERZEICHNIS

1. EINLEITUNG

Dieses Projekt entstand im Rahmen des Software Engineering - Advanced Topic Kurses im Master Studiengang Medieninformatik, der Beuth Hochschule Berlin, im Wintersemester 2013/14.

Die Ausarbeitung beschäftigt sich mit der Generierung eines Google Web Toolkit (GWT) Projektes. Dies erfolgt durch eine Model-To-Text Transformation mittels eines Model Transformation Language (MTL) Generators, welchem mit der Object Constraint Language (OCL), weitere Funktionalitäten zur Verfügung stehen.

Dabei wird ein besonderes Augenmerk auf die Architektur gelegt, allerdeings ohne des Einsatzes von Layouting. Deshalb findet eine Generierung anhand einer vorgegebenen Grundstruktur bzw. -architektur statt. Mit dem in der vorliegenden Projektarbeit entwickelten Generator, ist es einem Entwickler möglich eine komplette GWT Struktur generieren zu lassen. Auf Grundlage eines, auf einem UML Profil basierenden, Klassendiagramms kann eine komplette GWT Frontend Webanwendung abgebildet und mit kleinen Änderungen schnell lauffähig gemacht werden. Das UML Profil wird so gestaltet, dass eine GWT Anwendung gemäß der vorgegebenen Architektur einfach umgesetzt werden kann. Dabei finden das Model-View-Presenter (MVP) Pattern sowie verschiedene Frameworks, welche weiterhin Architekturkonzepte, beispielsweise für Dependency Injection, liefern, als grundlegende Konstrukte Verwendung. Anstoß zu diesem Projekt war eine gegebene „Best Practice“ Lösung mehrerer Entwickler, die jedoch ziemlich aufwändig und fehleranfällig in der Umsetzung ist.

In dem weiterem Verlauf dieser Ausarbeitung werden die zugrundeliegenden Konzepte und Frameworks (vgl. Abschnitt 2) vorgestellt. Weiterhin wird die Idee und konzeptionellen Aspekte (vgl. Abschnitt 3), auch hinsichtlich der Ziel-Architektur, erläutert. Diese werden im Weiterem mit sich ergebenden Änderungen umgesetzt (vgl. Abschnitte 4, 5, 6). Anhand des Ergebnisses (vgl. Abschnitt 7) wird gezeigt, dass eine GWT Anwendung anhand zweier Anwendungsfälle generiert und wie diese lauffähig gemacht werden kann. Zusammenfassend folgt eine Bewertung des Generator Projektes mit einem Ausblick auf Erweiterungsmöglichkeiten (vgl. Abschnitt 8).

2. GRUNDLAGEN

Das Ziel dieser Arbeit besteht darin aus einem UML Modell ein GWT Projekt zu generieren. Zur Verständnis werden in diesem Kapitel einige wichtige Grundlagen erläutert.

2.1 Model Driven Architecture

Model Driven Architecture (dt. Modellgetriebene Architektur), kurz MDA, stellt einen 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, sodass die Software zu einem hohen Anteil automatisch durch Transformationen von Modellen erzeugt werden kann. Die dabei entstehenden Transformatoren können eine hohe Wiederverwendbarkeit und Wartbarkeit sicherstellen.[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 dem Metamodell der UML 2.4 Anwendung fanden. 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 im System zu spezifizieren gibt, ist definiert, jedoch komplett frei von der später folgenden Implementierung. Somit ist nicht nur eine einzige Implementierung des Systems möglich, sondern durchaus mehrere unterschiedliche. Werden nun die Funktionalitäten kombiniert, die im Platform Independent Model definiert sind, mit den Designanforderungen der gewünschten Plattform, so entsteht 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 JavaScript Objekt `XMLHttpRequest` 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 zur client-server Kommunikation mit GWT, auf die nicht genauer eingegangen wird, 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 zwei Vorteile, die die Nutzung durch GWT bietet. 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 ähnlich dem MVC (Model-View-Controller). Google beschreibt den Nutzen des MVP Patterns in der Ein-

[1] GWT wurde umbenannt zu Gwit Web Toolkit. In dieser Arbeit wird jedoch weiterhin von GWT gesprochen, da das Gwit Web Toolkit weiterhin auch unter GWT anzutreffen ist.

bindung von Testfällen in einer GWT Anwendung. Darüber hinaus kann dieses Pattern auch genutzt werden, um eine GWT Anwendung für verschiedene Plattformen verfügbar zu machen. Diese Plattformen können z.B. der Browser auf mobilen Endgeräten oder auf dem Desktop sein.

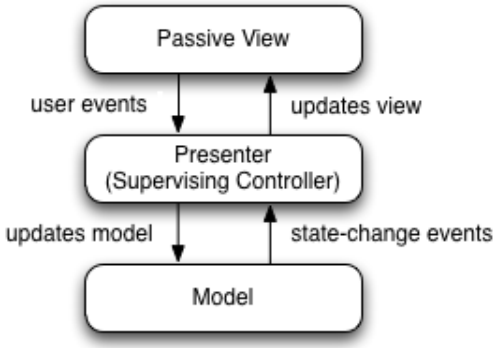

Fig. 2.1: Visualisierung des MVP Patterns [Chandel, 2009]

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

In der zu generierenden Anwendung soll dieses Pattern ohne ein Model implementiert werden, da der Fokus ausschließlich auf eine GWT Frontend Anwendung gerichtet ist. Die MVP Struktur ist dabei so umgesetzt, sodass der Presenter als Interface in dem View Interface und konkret über die Activity definiert wird. Die Activity regelt zusätzlich das Event Handling und die Datenbeschaffung. Die View Implementierung beinhaltet eine Instanz des Presenters, damit die Aktionen der View Komponenten (bei GWT Widgets genannt) an den Presenter übergeben und dadurch an die Activity weitergeleitet werden.

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 Webseiten. Dabei wird die Webseite nicht länger ausschließlich in Java Code gestaltet, sondern deren View Komponenten und Layout über eine spezielle, HTML ähnliche, Expression Language deklarativ formuliert. Dies geschieht innerhalb einer `.ui.xml`-Datei. In dieser Sprache können neben View 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 beispielsweise statische View Komponenten nur innerhalb der `.ui.xml`-Datei enthalten sind und somit ein Überladen der View Implementierung vermindert werden kann. Weiterhin können die View Komponenten innerhalb dieser Datei an die View Komponenten in der View Implementierung gebunden werden[Google, 2010]. Dazu folgende Codeauszüge 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`-Datei (vgl. Listing 2.2). Mit `@UiField` ist es möglich die dafür zuständige Instanz, der im UiBinder-Template definierten View Komponente, zu erhalten. Diese kann dann über den Java Code instanziiert oder Style Eigenschaften gesetzt werden. Die Annotation `@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 des UiBinder Frameworks, 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 erstmalig von Martin Fowler 2004 verwendet wurde [Martin Fowler, 2004]. Dabei handelt es sich um ein Verfahren, bei dem zur Laufzeit eines Programmes zusätzliche Informationen, beispielsweise beim Aufruf einer Funktion, zur Verfügung gestellt werden. Dies wird von einem extra Dependency Injection Framework vorgenommen, in dieser Arbeit handelt es sich dabei um das GIN Framework.

2.3.1 GIN

GIN 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. Die zu generierende Zielarchitektur nutzt GIN zur Dependency Injection, es wurde direkt in den GWT Generator eingebaut, wodurch Dependency Injection teilweise schon zur Compile Zeit erfolgen kann, dies sorgt dafür, dass es kaum Laufzeit Overhead gibt.

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

3. KONZEPTION

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 View Implementierungen durch MVP, weiterhin nutzbar sein. Zusätzlich ist es wünschenswert das Verhalten von View Komponenten wie Buttons, beispielsweise für eine Navigation zwischen den Webseiten, 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 Menus für den Entwickler verwendbar sein, welche zusätzlich untereinander zugeordnet werden können. Des Weiteren sollen Elemente, die permanent auf jeder Webseite 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 auf M1 Ebene erstellt werden (vgl. Abschnitt 3.3), basierend auf dem zum Generator Projekt gehörendem UML Profil auf M2 Ebene (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 Architekturkonzepte 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. 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 Views 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. Darin sind elementare Strukturen festgelegt. Dazu zählt bespielsweise der grundlegende Webseiteninhalt, welcher zu dem `RootPanel` hinzugefügt und durch die `ViewActivityMapper` erreichbar ist, sowie die zum History Management gehörenden Komponenten, durch die die Startwebseite (durch Angabe des jeweiligen `Places`) registriert wird.
  - `AbstractView.java`
    Die Oberklasse aller View Implementierungen innerhalb einer GWT Anwendung. Darin können grundlegende Eigenschaften wie die Größe aller View Implementierungen definiert werden.
  - `AbstractActivityDefaultImpl.java`
    Diese Klasse ist ein GenericType mit dem Typparameter `Place` und wird von allen `Activity`-Klassen mit dem jeweiligem `Place` als Typparameter erweitert. Diese Klasse stellt eine Verbindung zwischen einem `Place` und der dazugehörigen `Activity` her.
  - `'Name'ViewActivityMapperImpl.java`
    In dieser Klasse werden die `Activity`-Klassen, außer die anderer `ViewActivityMapper`, instanziiert. Die Klasse existiert für jede vorhandene permanente View sowie prinzipiell einmal für alle normale Views. Durch die überschriebene Methode des GWT `ActivityMapper` `getActivity` wird mittels des `Place` die `Activity` zurückgeliefert. Dies wird benötigt, damit der Browserzugriff auf die jeweilige Webseite durch GWT umgesetzt werden kann.
  - `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` dem History Management und wird u.A. zur Registrierung der Start View benötigt.

  - `PlaceControllerProvider.java`
    Die Schnittstelle zu den View Places, welche wie erwähnt in den `ViewActivityMappern` aufgerufen werden und somit den Browserzugriff auf die View Implementierungen ermöglichen.
  - `ProductionGinModule.java`
    In dieser Klasse werden die für GIN typischen `bind`-Befehle verwendet. 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 und beinhaltet eine Instanz der `View` sowie des `PlaceController`. Durch den `PlaceController` wird z.B. die Navigation zwischen den Views ermöglicht mittels einer darauf angewendeten `goTo`-Methode, welcher ein `Place` übergeben wird.
  - `'Name'Place.java`
    Diese Implementierungen sehen prinzipiell immer gleich aus. Unterschieden wird hierbei die dazugehörige `View`. Über den `Place` wird die Navigation zwischen den Webseiten ermöglicht, wobei der Name des `Place` in der URI Zeile des Browsers steht.
  - `'Name'View.java`
    Hierbei handelt es sich um ein Interface, welches das Presenter Interface definiert 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 im Browser als Webseite sichtbare View Implementierung beinhaltet eine Instanz, des durch der jeweiligen `View` und `Activity` definierten Presenter. Dadurch wird die Kontrolle gemäß MVP abgegeben. Die Klasse kann entweder das gesamte GUI erstellen oder mittels UiBinder einen Teil der View Komponenten abgeben, wie im Fall von vordefinierten Labels.
  - `'Name'ViewImpl.ui.xml`
    Innherhalb dieser Datei können View Komponenten sowie dessen Style Eigenschaften definiert werden.

- Views und Elemente, die auf jeder Webseite zu sehen sind,
  - werden gemäß dem MVP Pattern und wie eine View erstellt.
  - sind 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	- 'Name'Activity.java
- AppPlaceHistoryMapper.java	- 'Name'Place.java
- ProductionGinModule.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 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 bei der Erstellung einer Vielzahl von Klassen schnell ein Eintrag vergessen werden kann.

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: `<source path='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 View Interfaces und ihre dazugehörigen Klassen 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 wie erwähnt sehr aufwändig und fehleranfällig. Beispielsweise kann über „Copy-Paste“ viel erstellt und implementiert werden. Jedoch ist dabei das Risiko erhöht, dass es vergessen wird Einträge abzuändern oder hinzuzufügen. Darüber hinaus kann es passieren, das Einträge enthalten sind, wie der 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 (des MVP) 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 wird. 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 einem 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 für die Verbindung mehrerer View Implementierungen gegeben werden muss. Aus diesem Grund soll das *View Interface* 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 ActivityDiagramm als M1 Modell zu nutzen. Dieses Mittel ermöglicht eine Übersicht über die Navigations- 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 Komponenten 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 `goToView` 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 Komponenten und *View Komponenten* sollen Properties sein, da eine *View Implementierung* diese View Komponenten enthalten kann.

Zur Kopplung von großen View Implementierungen 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 Implementierungen* enthalten sein kann.

Im Bereich von Views, z.B. ein Header, die auf allen Webseiten 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 Webseite (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 Implementierungen* von dem *View Interface* umgesetzt werden können. Dadurch wird zusätzlich der Austausch von permanenten Views gemäß MVP ermöglicht.

3.3 M1 Modell

Basierend auf dem UML Profil soll das Modell View Implementierungen enthalten, welche wiederum View Komponenten als Properties und eigene View Objekte beinhalten können. Zu den eigenen View Objekten können auch View Komponenten zugeordnet sein. 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 `goToView` 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 ein Menu enthalten, das durch MenuItems die Navigation zu verschiedenen Webseiten ermöglicht. Zusätzlich sollen View Interfaces, welche verschiedene Implementierungen haben, erstellt werden. Dadurch kann die Generierung des MVP Patterns getestet 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, dass aus dem M1 Modell die gesamte Ziel-Architektur generiert werden kann. Dies soll potenziell auch ermöglicht werden indem 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 abhängig) und andererseits die Views generiert. Gemäß dem Fall, dass View Interfaces in dem M1 Modell vorhanden sind, sollen die Klassen `'Interfacename'View.java`, `'Interfacename'Activity.java` und `'Interfacename'Place.java` den Interfacenamen tragen. Die Implementierungsklassen bzw. Dateien, der View Intefaces, `'Klassenname'ViewImpl.java` und `'Klassenname'ViewImpl.ui.xml`, haben die Implementierungsklassennamen. Darüber hinaus sollen die View abhängigen 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 Implementierung 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 dem Hauptinhalt bzw. der HauptView in dem `AppEntryPoint` 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` positioniert werden. Alle anderen vorhandenen permanenten Views sollen dazwischen eingetragen und müssen später durch den Entwickler im generiertem Code positioniert werden, da dies layoutspezifisch ist.

Der Entwickler soll weiterhin zusätzliche Änderungen vornehmen müssen. Diese betreffen die Startwebseite der GWT Webanwendung, welche in dem generiertem Code anstatt einer konkreten Angabe des Klassennamens, innerhalb des Befehls, durch „Start“ gekennzeichnet wird. Dieses Vorgehen kann das Verständnis für die Architektur stärken und es werden überflüssige und überfüllende Attribute innerhalb des UML 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 innerhalb der generierbaren Teile erfolgen. 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 zwei 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 Testfälle 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. Hierzu wird das UML Profil mit eigens definierten Meta Klassen erweitert. Der Grund hierfür ist, dass im späteren UML Modell Zuständigkeiten besser zugewiesen und erkannt werden können. Diese Erweiterungen werden als Stereotyp im Profil bezeichnet, die von vordefinierten Metaklassen abgeleitet werden.

Im folgenden Abschnitt wird das Profil detailierter beschrieben.

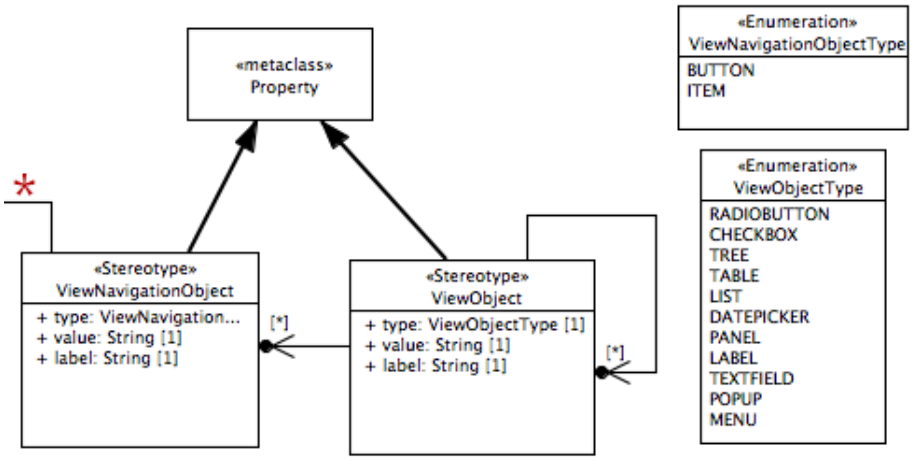

Fig. 4.1: Darstellung der Properties und Enumerations im Profil

In dem Profil wurden zwei Stereotypn als Properties definiert (Abbildung 4.1). Das *ViewObject* und das *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 *ViewNavigationObject*s können auch für andere Verhaltensspezifikation dienen, die aber im Rahmen dieses Projektes nicht umgesetzt wurden. Die Trennung dieser 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 sollte. Denn diese beschreiben,

mit Button und Item, schon die einfachsten Interaktionselemte. Den Typ der Property wurde, wie bereits im Konzeptionsteil erläuert (vgl. Abschnitt 3.2), mit Hilfe von Enumerations festgelegt. Zudem ermöglichen diese eine Reduzierung der unterschiedlichen Modellelemente und gewährleisten so eine bessere Übersichtlichkeit. Im Unterschied zur Konzeption wurde festgelegt, dass Multi-Navigationselemente (wie Table, List, Menu oder Tree) als *ViewObject* definiert sind, die dann wiederum *ViewNavigationObject*s vom Typ `ITEM` oder `BUTTON` besitzen können. Die Gruppierung als Items, die zuvor in der Konzeption als `MENUITEM` oder `TREEITEM` aufgeführt wurden (vgl. Abschnitt 3.2), ermöglichte es, die Generierung einfacher zu gestalten. Denn alle Items können nun gleich generiert werden, unabhängig davon, welchem *ViewObject* sie angehören. Zusätzlich können `TREEs`, `LISTs` sowie `TABLEs` auch einfache Widgets wie beispielsweise Textfelder enthalten. Somit müssen diese also keine Interaktionselemente beinhalten, wodurch diese Trennung durch `ITEM` und `BUTTON` erforderlich wurde.

Bei den Widgets lag der Fokus auf den jeweils wichtigsten Elemente (`BUTTON`, `ITEM`, `TEXTFIELD`, `TABLE` etc.), die im späteren Verlauf des Projektes auch alle generierbar sein sollten. Diese lassen sich in zukünftigen Versionen einfach erweitern, indem man zusätzliche EnumerationLiterals hinzufügt und die Erstellung dieser im Generator anpasst. Die weiteren Attribute wie `value` oder `label` wurden dann in dem entsprechenden Stereotyp 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 Stereotypn *View* und eine *PermanentView* von der Metaklasse Interface abgeleitet. Diese bieten hilfreiche Schnittstellen für deren Implementierungen. Die beiden Interfaces werden als Stereotyp *ViewImpl* und *PermanentViewImpl* der Meta Klasse class implementiert (Abbildung 4.2). Durch Erstellen des Boolean Attributs `concreteBinding`(in der Konzeption 3.4 noch binding genannt), ist es z.B. möglich, den entsprechenden `bind`-Befehl über den Generator zu setzen. Der Defaultwert des Attributs ist `true`, da im Regelfall kein Interface existiert oder es wenigstens eine Implementierung von einem speziellen Interface gibt. In diesem Fall gibt es immer nur diese eine View Implementierung, die angezeigt werden kann. Sollte es aber mehrere Implementierungen zu einem Interface geben, muss das `concreteBinding` dieser auf `false` gesetzt werden. Dadurch wird dieses im Anwendungsfall entsprechend berücksichtig 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 *View* 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.

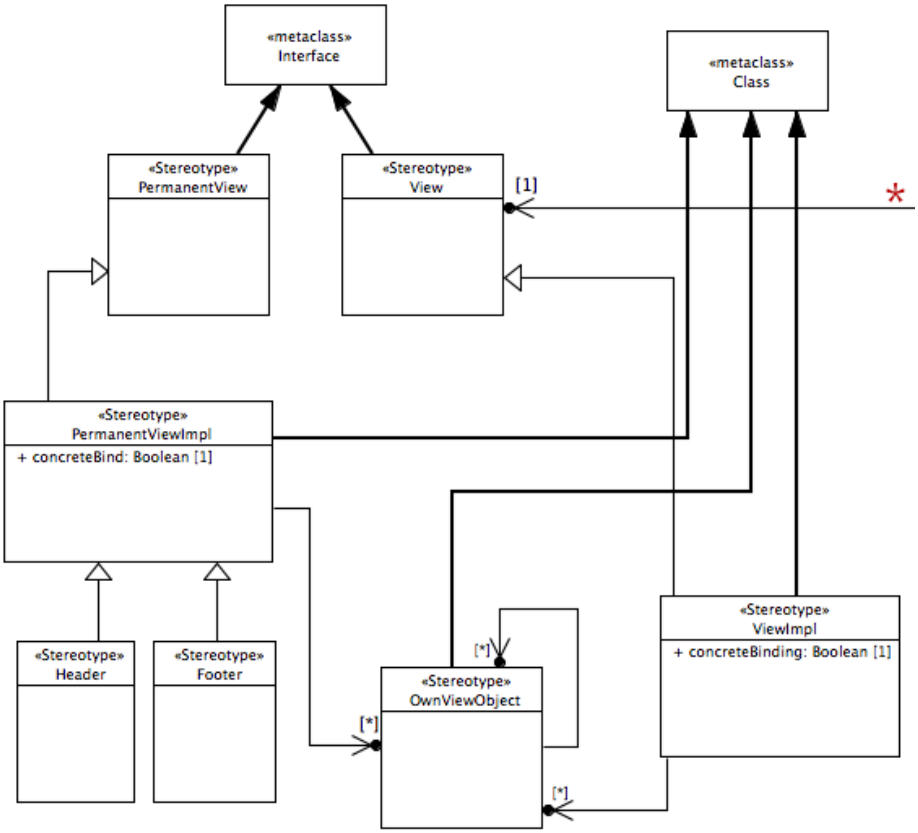

Fig. 4.2: Darstellung der Classes und Interfaces im Profil

Für spezielle Klassen wurde noch der Stereotyp *OwnViewObject* hinzugefügt, der es dem Entwickler später ermöglichen soll, diese zu View Implemntierungen hinzuzufügen. Die *OwnViewObjects* dienen als eigenständige Widgets, die die Möglichkeit bieten 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 Stereotypn, wie in der Ziel Architektur 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 Generators werden diese automatisch zur entsprechenden View generiert. Auch der direkte Stereotyp Model, welcher beispielsweise die Schnittstelle zu einer Datenbank repräsentiert hätte, entfällt in dem Profil. Denn im Projekt war diese Backend-Anbindung nicht vorgesehen.

5. AUFBAU UND STRUKTUR M1 MODELL

Beim M1 Modell handelt es sich um die direkte Darstellung der zu generierenden Views für die Webanwendung. Dieses Modell wird direkt durch den Generator, siehe Abschnitt 6, umgesetzt und um alle zusätzlichen Dateien erweitert, die für die Ziel-Architektur notwendig sind. Das Ziel, in dieser Ausarbeit, lag vor allem darin die Ziel-Architektur mit möglichst wenig Aufwand für den Entwickler zu generieren, somit ist das M1 Modell nicht als vollständige Abbildung des Projektes zu verstehen. Das Erzeugen des Modells sollte übersichtlich gestaltet werden, um einerseits den Programieraufwand zu minimieren und anderseits die Fehlerquote zu verringern.

In dieser Arbeit, soll der Generator aus dem M1 Modell, die im Abschnitt 3.1 beschriebene mögliche Grundstruktur für ein GWT Projekt erzeugt werden. Dieses kann durch kleine Änderungen lauffähig gemacht werden. Das M1 Modell ist so angedacht, dass alle Seiten, der Webanwendung, als eine eigenständige View dargestellt werden. Des Weiteren können einfache Elemente, wie beispielsweise Label und Textfelder den Views hinzugefügt werden. Zudem ist es möglich die Navigation zwischen den Seiten, beispielsweise mit Hilfe eines Menus, in das Modell mit einzubringen.

Durch die Verwendung eines Profils sind in dem M1 Modell keine Assoziationen zu sehen. Die Navigation wird ausschließlich über die Properties in den Navigations Elementen erstellt, in dieser Arbeit werden diese Elemente mit dem Stereotyp *ViewNavigationObject* gekennzeichnet. Der einfache Grundaufbau des M1 Modells, sorgt für eine leichtere Modellierung, erschwert allerdings das Erkennen der Navigation.

5.1 M1 Modellaufbau

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

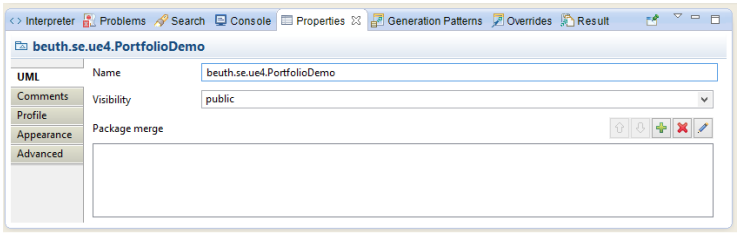

Fig. 5.1: Name des Modells und gleichzeitig Hauptpaket des GWT Projekts

5.2 Klassenaufbau

Die grundlegendsten Elemente in dem Modell sind *ViewImpl* Klassen. Ein Beispiel dafür ist in Abbildung 5.2 zu sehen. Für jede dieser Klassen, werden durch den Generators, alle notwendigen Dateien erzeugt die für den späteren Aufruf, der Seite, notwendig sind.

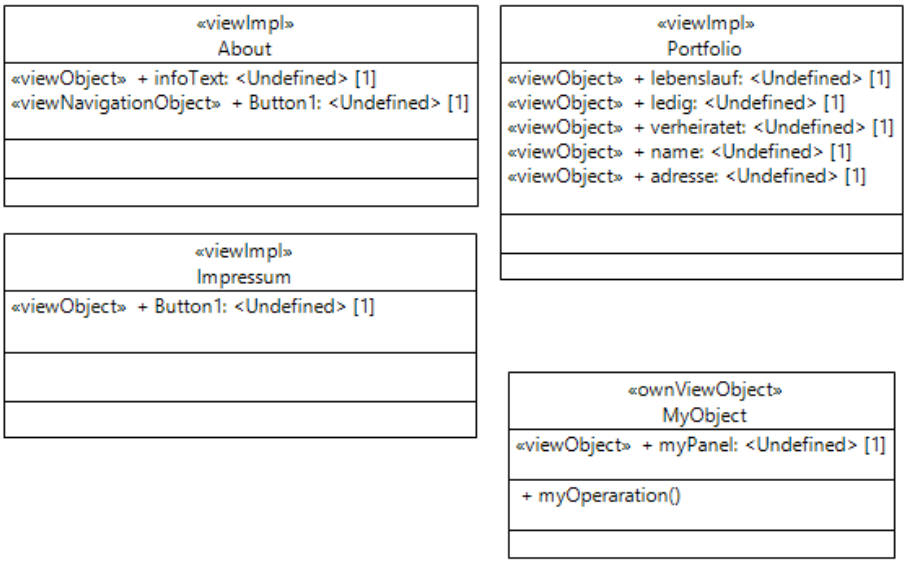

Fig. 5.2: ViewImpl Klassen zur Erzeugung von Seiten für den Webauftritt.

Des Weiteren ist in der Abbildung 5.2 ein Beispiel für ein *OwnViewObject* zu sehen. Diese sind für komplexere oder mehrfach verwendete UI Strukturen gedacht und werden vom Generator als eigenständige Klasse generiert.

Die Ziel-Architektur bietet, durch Verwendung des MVP Pattern, eine einfache Möglichkeit View Implementierungen auszutauschen. Um dieses Verfahren

in jedem Fall für den Entwickler nutzbar zu machen, wurde sich dazu entschlossen, neben der Generierung von View Implementierungen, eine zusätzliche Methode einzubauen. Diese Methode bietet die Möglichkeit beliebig viele alternativ Seiten für ein View Interface zu erzeugen (vgl. Abbildung 5.3). Die Möglichkeit der einfachen Austauschbarkeit von Views, wurde bei GWT unter anderem für Testzwecke angedacht.

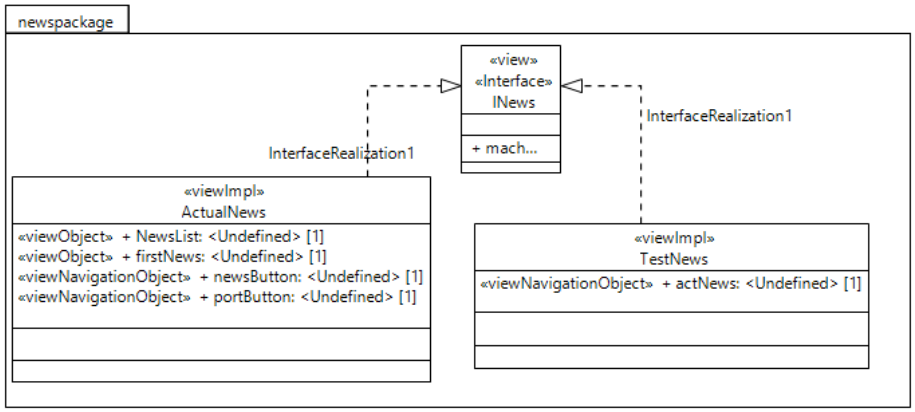

Fig. 5.3: Ein Beispiel für die Verwendung der `ViewInterface`-Klassen um mit Hilfe einer Vielzahl von `ViewImpl`-Klassen eine austauschbare Ansicht zu erzeugen.

Bei dem, in Abbildung 5.3, gezeigtem M1 Modell Ausschnitt wird die komplette Klassenstruktur nicht zweimal erzeugt, anders als im allgemeinem Fall (siehe Abbildung 5.2). 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 für jede realisierende `ViewImpl`-Klasse generiert.

Darüber hinaus ist in Abbildung 5.3 zu sehen, dass es möglich ist, Klassen zu Paketen zuzuordnen. Auf diese Weise wird es ermöglicht die zu generierenden Dateien zu gruppieren.

Das letzte in den Modellen genutzte Klassenkonstrukt, sind die sogenannten *PermanentViewImpl*-Klassen. Diese sind dazu gedacht, um zum Beispiel Menus zu erzeugen. Diese haben die Eigenschaft das sie auf allen Webseiten zur Verfügung stehen und bei der Generierung nicht mehrfach in die normalen Views eingebunden werden sollen.

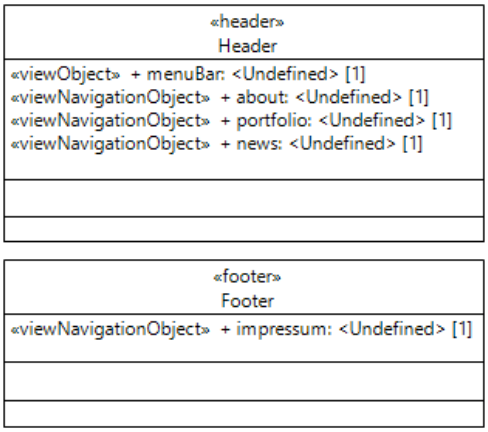

Fig. 5.4: PermanentViewImpl-Klassen am Beispiel eines Headers, mit Menu, und eines Footers.

5.3 Seitenaufbau

Je nach Art der View muss zuerst definiert werden, um was für einen Typ von View es sich handelt. In Abbildung 5.5 ist ein Beispiel dafür zu sehen, welches eine *ViewImpl* und ein *Header* zeigt, eine spezialisierte Form der *PermanentViewImpl*. Wichtig ist hier die `concreteBinding`-Eigenschaft. An dieser Stelle kann bei Verwendung eines separaten Interfaces festgelegt werden, welche konkrete Implementierung dargestellt werden soll. Um keine Laufzeitfehler zu erzeugen muss zu jeder View exakt eine *ViewImpl* angegeben werden. Diese Einstellung kann später im Quellecode geändert werden.

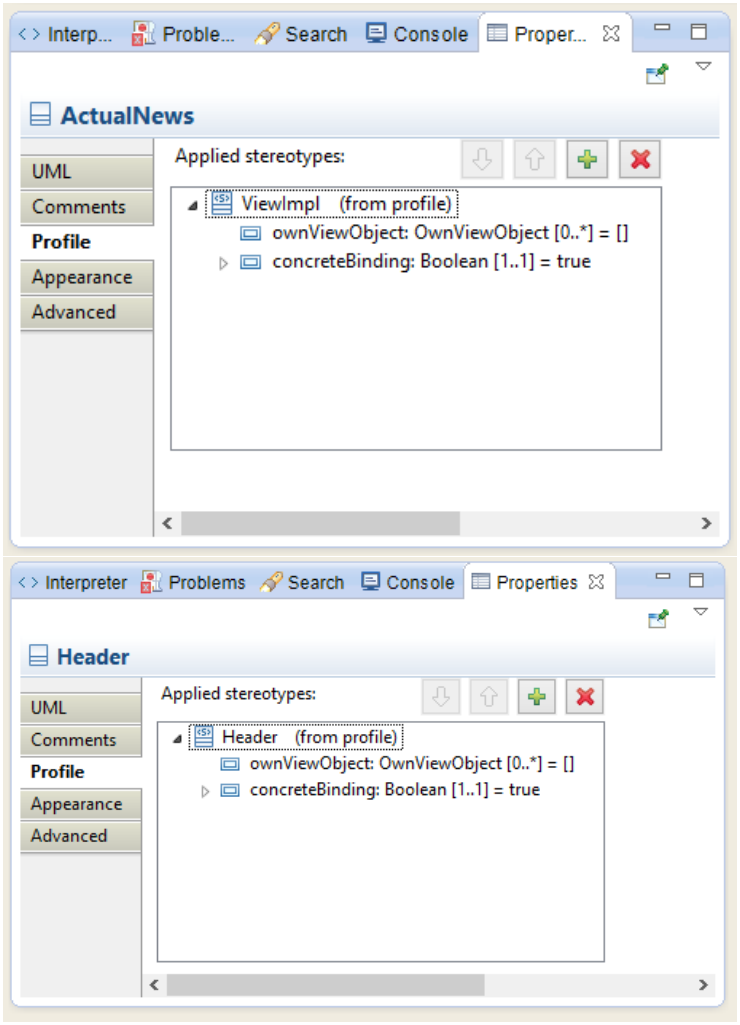

Fig. 5.5: Klassen mit definierten Stereotypen, oben eine *ViewImpl* und unten ein *Header*

Um diese Views 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.

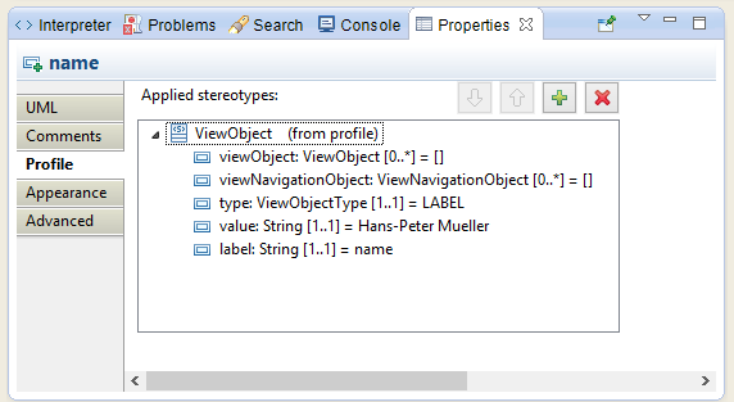

Fig. 5.6: Eigenschaften eines *ViewObject*-Elementes

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.

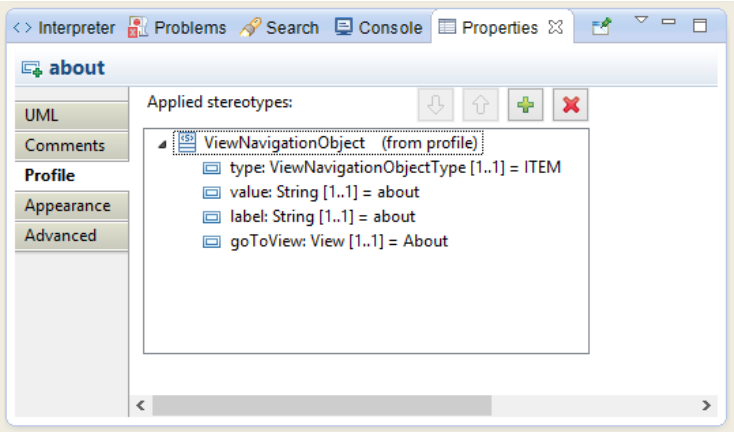

Fig. 5.7: Eigenschaften eines *ViewNavigationObject*-Elementes

6. GENERATOR

Der in diesem Projekt benutzte "Model Transformation Language" (MTL) Generator, dient unter anderem dazu aus UML Modellen Quellcode zu generieren. 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. 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 Struktur

Der Generator ist so konzipiert, dass er mit Hilfe des kleinsten gemeinsamen Elements, der *ViewImpl*, alle GWT relevanten Klassen erzeugen kann. Daher ist es auch möglich mit nur einer einzigen Klasse vom Stereotyp *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
`generate-BasicStatic-Structure.mtl`	Generierung aller einmaligen Klassen, die View unabhängig sind	Erstellung der `.gwt.xml`, der `index.html` oder der `style.css`
`generate-Structure.mtl`	Generierung der einmalig vorhandenen Klassen, welche inhaltlich Modell bzw. View abhängig sind.	`AppGinjector.java`, `AppEntryPoint.java` oder`ProductionGin-Module.java`
`generateMVP.mtl`	Generierung aller MVP spezifischen Objekte mit zwei verschiedenen Templates (`generate-ConcreteView.mtl` und `generateView-Interface`).	wie `Activity`, `Place`, `View`, `ViewImpl` und `ui.xml`
`generateOwnView-Object.mtl`	Template, welches sich um die gesonderte Generierung des *Own-ViewObject* kümmert.	Generierung einer *Own-ViewObject* Java-Klasse mit im M1 Modell festgelegten Widget-Attributen und Operationen

Die oben erwähnten Generatoren bedienen sich mehrerer Hilfsgeneratoren, darunter eine `util.mtl`, die beispielsweise alle Queries zusammenfasst (vgl. Abschnitt 6.3) oder auch Template-Aufrufe für die Protected Regions bereitstellt. Zudem existiert eine `packageDeclaration.mtl` die sich darum kümmert, dass für Klassen oder Interfaces die richtigen Namen entsprechend ihrer Paketierung zurückgegeben werden.

6.2 Funktion

Um die Struktur des Generators zu verdeutlichen, wird folgend auf die wichtigsten Funktionen ausführlicher eingegangen.

Ein wichtiges Element ist die Klasse des `AppEntryPoints`. Denn diese beschreibt die Zugriffsklasse für GWT und es werden von dieser gleichzeitig die ersten UI-Panel gesetzt. Hierbei ist im Generator zwischen einer *Permanent-ViewImpl* und einer Content View zu unterscheiden. Es können durchaus mehrere *PermanentViewImpls* vorhanden sein. Der Contentbereich ist dagegen immer nur einmal vorhanden und dient so als Container für alle *ViewImpls*. So müssen alle *PermanentViewImpls* mittels `for`-Schleife durchlaufen werden. Zudem sind die Spezialfälle der *PermanentViewImpl*, *Footer* und *Header*, auch unterschiedlich zu behandeln, da diese eine feste Positionierung besitzen. Alle anderen Layoutspezifischen Entscheidungen müssen vom späteren Entwickler gesetzt werden. Sämtliche *PermanentViewImpls* und der Content werden dann als `SimplePanel` zu dem GWT `RootPanel` hinzugefügt (vgl. Listing 6.1).

```
[for (class : Class | pack.ownedElement)]
  [if ( class.getAppliedStereotypes()->asOrderedSet()
      ->first().name.endsWith('Header'))]

    RootPanel.get().add([getClassName(class).toLowerFirst()/]);
  [/if]
[/for]
```

Listing 6.1: Hinzufügen eines Panels zum `RootPanel`, am Beipsiel eines *Header*s

Ein weiteres wichtiges Element ist der von GIN stammende `bind`-Befehl, der sicherstellt, das GWT automatisch weiß welches Interface zu welcher Implementierung instanziiert wird. Damit der `bind`-Befehl in der `ProductionGinModule`-Klasse, richtig gesetzt wird, muss mittels Generator aus dem M1 Modell das `concreteBinding`-Attribut einer *ViewImpl*- Klasse ausgelesen und entsprechend seines Wertes behandelt werden. Wenn die `concreteBinding`-Variable den Wert `true` hat, wird das View Interface an die entsprechende View Implementierung gebunden. Sollte der Wert auf `false` gesetzt sein, so wird dieser `bind-Befehl` trotzdem in den Code geschrieben, allerdings in kommentierter Form. So ist es später möglich beim Wechsel einer View, den entsprechenden `bind`-Befehl auszutauschen, ohne neuen Code schreiben zu müssen. Die Generierung dieses Befehls ist im Listing 6.2 zu sehen. 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 „YourStartHereßteht. Hier ist der Entwickler gezwungen, seine Startwebseite einzutragen. Dies geschieht auch bei den `bind-Befehlen`, bei denen es wichtig ist, dass zuerst die Startseite 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))]
    ...
    [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]
```

Listing 6.2: Auszug aus der Generierung des `bind`-Befehls

Als weitere Funktion wird im Folgenden die Generatorseitige Umsetzung des MVP Patterns beschreiben. Dieser wurde nach zwei Templates getrennt. Zum einen werden die View Interfaces mit entsprechenden `Place` und `Activity` Klassen und zum anderen die View Implementierungen mit dazugehörigen `.ui.xml` Dateien generiert. Jedes View Interface besitzt ein Interface `Presenter` und die entsprechende `setPresenter`-Methode sowie andere Operationen, soweit im M1 Modell festgehalten. Zusätzlich enthält das Interface für jedes Navigations Objekt eine `on'Name'Clicked`-Methode, zu sehen im Listing 6.3. Auch hier wird zusätzlich wie bei dem `bind`-Befehl die gleiche Methode als Kommentar geschrieben, sofern deren `concreteBinding`-Attribute auf `false` gesetzt ist (siehe Listing 6.3). 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 voneinander unterschieden und entsprechend behandelt. Es wurde sich bewusst für die etwas unschöne Art der Programmierung mittels `label`-Abfrage des Buttons in der `if`-Bedingung entschieden, da nicht davon ausgegangen wird, dass mehr als fünf Buttons sinnvoller Weise auf einer Webseite angebracht werden sollten. Wenn dies doch einmal vorkommt beispielsweise durch eine Datentabelle, würde diese eh vom Entwickler implementiert und nicht im M1 Modell festgelegt werden. 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 `on'Name'Clicked`-Methode die `@Override`-Annotation hinzugefügt werden. Diese dient dann als Wrapper Methode für die eigentliche `goTo`-Methode.

```
...
[if (class.getValue(class.getAppliedStereotypes()
  ->asOrderedSet()->first(), 'concreteBinding')
  .oclAsType(Boolean))]

    @Override
    void on[property.name.toUpperFirst()/]Clicked(){
      placeController.gotTo(new [getClassName
        (property.getTaggedValue(...)/]Place());
    }
[else]

    // @Override
    // void on[property.name.toUpperFirst()/]Clicked(){
    //  placeController.gotTo(new [getClassName
    //    (property.getTaggedValue(...)/]Place());
    // }
[/if]
...
```

Listing 6.3: Auszug der Generieung der `on'Name'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 konkretenViewImpl geschieht mittles @UIField das „Binding"von *ViewObject* oder *ViewNavigationObject* an die ui.xml. Zur Vereinfachung der zu generierenden XML Elemente wurde beispielsweise für eine Liste oder eine Tabelle das GWT UI Element grid verwendet. Darüber hinaus bietet dieses mehr Möglichkeiten verschiedenste Elemente, wie Buttons oder Textfelder einzubinden. Des Weiteren ist noch zu beachten, dass wenn ein Tree oder Menu mit Navigationselementen vorhanden ist, für jedes innere Element ein TreeItem oder MenuItem erzeugt werden muss. 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 der viewXML aufgerufen wird. Als Beipiel für die Generieung eines UI Elementes ist im Listing 6.4 ein Menu zu sehen.

```
1  [template public newMenu (property : Property) ]
2  [if (isValueExists(property, 'type', 'MENU'))]
3
4  <g:MenuBar ui:field="[property.name/]">
5
6  [if(property.getValue(property.getAppliedStereotypes()->
7    asOrderedSet()->first(), 'viewNavigationObject')
8      ->notEmpty())]
9    [for (prop : Property | property.getTaggedValue
10     ('ViewObject','viewNavigationObject', false).oclAsSet().
11     oclAsType(Property))]
12
13   <g:MenuItem ui:field="[prop.name/]" text="[prop.getValue
14     (prop.getAppliedStereotypes()->asOrderedSet()->
15     first(), 'value')/]"/>
16   [/for]
17 [/if]
18
19 </g:MenuBar>
20 [/if]
21 [/template]
```

Listing 6.4: Template für die XML - Generierung eines Menus

Abschließend ist zum OwnViewObject zu sagen, dass dieses durch Vererbung der GWT isWidget Klasse fast wie ein eigenständiges Widget behandelt werden kann. Das OwnViewObject 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 M1 Modell vorhandene UI Elemente als eigene Attribute und Operationen generiert.

6.3 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 Ausdrucks eingesetzt werden. Die verwendeten Queries dieses Projektes sind in der `util.mtl` zusammengefasst. So wird beispielsweise in `isNotViewExisting` (vgl. Listing 6.5) über die Klasse abgefragt, ob diese von einem Interface des Stereotypes View implementiert wurde. Ist dies der Fall, gibt der Query `false` zurück anderenfalls `true`. Dieser längere Ausdruck kann auf diese Weise später kompakt als Query an einer geeigneten Stelle verwendet werden.

```
[query public isNotViewExisting(class : Class) : Boolean =
  if(class.interfaceRealization->notEmpty())
  then if(class.interfaceRealization
      .target.getAppliedStereotypes()->asOrderedSet()
      ->first().name.endsWith('View'))
    then false
    else true
    endif
  else true
  endif
\]
```

Listing 6.5: 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. Der Query musste für dieses Projekt soweit angepasst werden, als dass zu diesem ein weiterer Boolean Parameter `isClass` ergänzt wurde. Dies war notwendig, da der 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 einer Liste (vgl. Listing 6.6).

```
private void addToResult(String property, Boolean isClass,
     List<Object>result, Object values) {
   if(values instanceof DynamicEObjectImpl) {
     final DynamicEObjectImpl objectImpl =
       (DynamicEObjectImpl) values;
     final EClass c = objectImpl.eClass();
     EStructuralFeature sf = null;

     if(isClass) sf = c.getEStructuralFeature("base_Class");
     else sf = c.getEStructuralFeature("base_Property");

     if(sf!=null) result.add(objectImpl.eDynamicGet(sf,true));
   }
 }
```

Listing 6.6: Hilfsmethode `addToResult` der `PropertyHelper.java`

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. Diese Variable sollte auf `false` gesetzt und in einer `if`-Bedingung verändert werden. Dazu wurde beispielsweise versucht, mit dem von MTL zur Verfügung gestellten `let`-Block zu arbeiten. Allerdings mit mittelmäßigen Erfolg, da es unmöglich ist im `let`-BLock Variablen zu verändern. Denn definierte Variablen werden immer als final gesetzt. Anschließend wurde getestet, ob man mit Hilfe einer Java Klasse (`BooleanHelper.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 `.ui.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, sodass noch einige unnötige Redundanzen auftreten. Dieses erschwert zwar die Lesbarkeit des Generators, aber seine Funktionalität ist trotzdem gegeben.

7. ERGEBNIS

Das Ziel bestand darin eine Grundarchitektur für ein GWT Projekt zu erzeugen, welche es dem Entwickler vereinfachen soll seine Anwendung gemäß dieser Architektur zu erstellen. Um das Projekt vollständig lauffähig zu machen, sind noch ein paar Handgriffe vorzunehmen.

- `AppEntryPoint.java`
  Hier muss definiert werden, welche der Views die Startseite werden soll.
- `AbstractView.java`
  In dieser Klasse ist es möglich Standardeigenschaften für alle View Implementierungen zu bestimmen, der Befehl für die Größe ist hier vorgeneriert und muss vervollständigt werden.
- `ProductionGinModule.java`
  Auch hier muss wie in der `AppEntryPoint`-Klasse noch einmal die Starteite definiert werden.
- `index.html und style.css`
  Diese beiden Dateien müssen nach dem Durchlauf des Generators in den `war`-Ordner verschoben werden. Sie dienen als Ausgangspunkt für die Webanwendung.
- `Zusätzliche Bibliotheken`
  Zusätzlich zu dem `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`

Derzeit sind noch folgende Schritte notwendig.

- `import`
  Es ist notwendig die noch fehlenden Imports in den Java Klassen einzufügen, da diese aus Zeitgründen nicht vollständig eingearbeitet worden sind, um nicht überflüssige Imports zu generieren.

- `'Name'ViewImpl.ui.xml`
  In diesen Dateien müssen doppelte Elemente entfernt werden, die der Generator zu viel eingefügt hat, näheres dazu in Abschnitt 8 zweiter Absatz.

Sind diese Maßnahmen durchgeführt, ist das Projekt lauffähig. Zusätzlich können weitere Änderungen vorgenommen werden, zum Beispiel können weitere Elemente eingefügt oder das Design angepasst werden, dieses wurde von dem Generator nur rudimentär, in Form einer `css`-Datei angelegt.

7.1 Beispiel Anwendung

Für das in diesem Abschnitt gezeigte Beispiel wurde das M1 Modell aus Abbildung 7.1 verwendet. Zu sehen sind fünf Seiten, welche in Pakete gegliedert sind. Darüber hinaus wurden zwei *PermanentViewImpl* modelliert. Eine betrifft den Header, mit einem eingebautem Menu und eine den Footer, in dem eine Navigation zum Impressum vorgesehen ist.

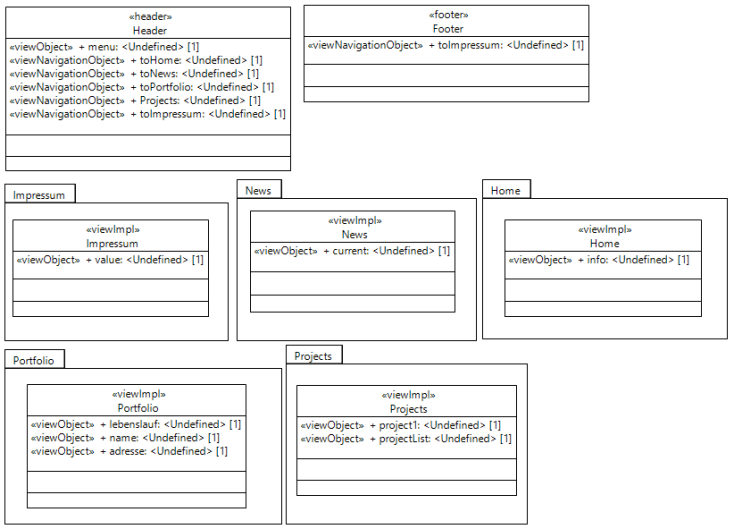

Fig. 7.1: Verwendetes Modell

Durch die extra eingebauten Pakete erweitert sich die Grundstruktur des Projektes. In Abbildung 7.2 ist auf der linken Seite die komplette Paketstruktur des Projektes zu sehen. Hier ist zu erkennen, dass für die fünf Seiten separate Pakete existieren, welche in Abbildung 7.1 zusehen sind. Jedoch für den Header und den Footer keine zusätzlichen Pakete angelegt worden sind. Auf der rechten Seite in Abbildung 7.2, wird der Inhalt einiger der Pakete näher gezeigt. Hier ist zu erkennen, dass die beiden `ViewImpls`, die nicht in Paketen liegen, direkt in dem `View`-Paket liegen. Zusätzlich sind die Klassen `Activity`, `Place`, `View` und die `.ui.xml`-Datei in dem Paket zu sehen. Am Beispiel der `Home`-Seite ist

gezeigt, dass die hierfür generierten Klassen und Dateien in einem separaten Paket von `view`, nämlich in `home` liegen.

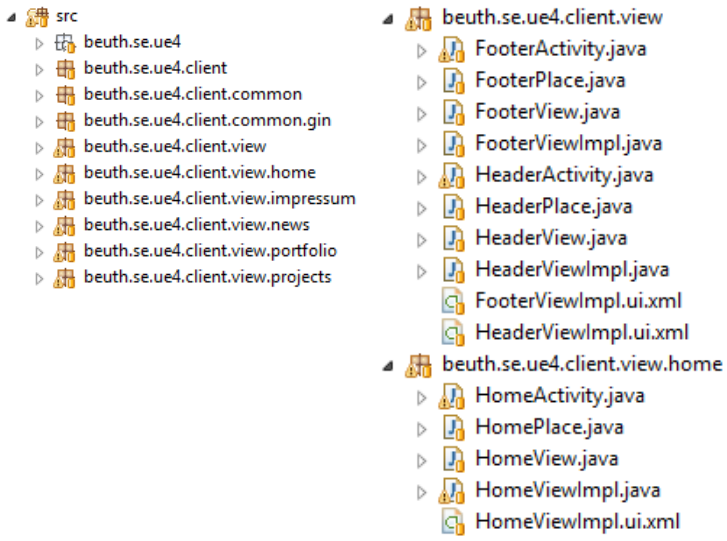

Fig. 7.2: Generierte Paketstruktur, links gesamte Paketstruktur im Überblick; rechts Beispiel für das Strukturieren im Modell, mit zusätzlichen Paketen

In diesem Beispiel soll die `Home`-Seite die Startseite für die Webanwendung darstellen. Aus diesem Grund wurde entsprechend in der `AppEntryPoint`- und der `ProductionGinModule`-Klasse die jeweilige Klasse für die StartView angegeben (vgl. Listing 7.1 und 7.2).

```
1 [..]
2   historyHandler.register(injector.getPlaceController(),
3       eventBus, new HomePlace());
4 [..]
```

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

```
1 [..]
2   bind(HomeView.class);
3 [..]
```

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

Zusätzlich wurde an einigen Stellen der Inhalt der Seiten erweitert. Dies ist einerseits dadurch möglich, dass innerhalb der `ViewImpl`-Klasse zusätzlicher Inhalt eingefügt werden kann (vgl. Listing 7.3) und andererseits auch innerhalb der `.ui.xml`-Datei (vgl. 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

```
[..]
 <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 in der `.ui.xml`-Datei

Dieses sind nur zwei Arten wie die Anwendung weiter bearbeitet werden kann, nachdem der Quellecode aus dem M1 Modell generiert worden ist. Zum Einen unterstützen der Generator und das Profil derzeit nicht alle von GWT bereitgestellten UI Elemente, sodass diese nachträglich eingefügt werden müssen. Da diverse UI Editoren existieren, welche das Erzeugen einer Benutzeroberfläche grafisch unterstützen, wurde sich in dieser Ausarbeitung bewusst gegen das Erzeugen einer Webanwendung, die vollständig designt wurde, entschieden.

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 Ziel-Architektur (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 beispielsweise durch Aus- und Einkommentierung bei dem Austausch von View Implementierungen. Zur Gestaltung der Webseite 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 zwei M1 Modelle deutlich wird. Es können eigene View Elemente in Form von *OwnViewObjects* erstellt und eingebunden werden, welche die Architekturkonzepte zusätzlich unterstützen. Des Weiteren wird die Navigation über *ViewNavigationObjects* innerhalb der View Implementierung generiert, jedoch sind weitere Verhalten wie das Öffnen eines Popups nicht umgesetzt. Zusätzlich sind nicht alle View sowie View Komponenten abhängige `imports` vollständig umgesetzt und müssen somit vervollständigt werden.

Durch Abschnitt 6.4 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 (vgl. Abschnitt 6.4). Eine weitere Möglichkeit der Optimierung an dieser Stelle könnte darin bestehen, Datenmodelle generierbar zu machen und zu dem UML Profil hinzuzufügen. Diese können als Grundlage für View Komponenten dienen. Anhand eines Beispiels kann das Datenmodell einer Produktklasse mit den Attributen Name, Preis und Verkaufsort dazu genutzt werden, dass innerhalb einer dementsprechenden View Implementierung 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 View Komponenten auch ohne Datenmodelle generierbar sein.

Weiterhin wäre es denkbar ein Backend genierbar zu machen und die genannten Datenmodelle zusätzlich darauf zugreifen zu lassen. Damit würden diese das Model im MVP bilden und eine weitere architektonisch sinnvolle Abgrenzung zwischen Frontend und Backend stattfinden.

Innerhalb des Generators wurde bereits in Abschnitt 6.4 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, beispielsweise das Umsetzen der `index.html` oder das Hinzufügen von Bibliotheken z.B. GIN (vgl. Abschnitt 7), 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 View Implementierungen 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 hervorgehoben. Darüber hinaus sind viele Teile wie z.B. die einmalig vorhandenen Klassen sowie die View Klassen `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.

9. ARBEITSAUFTEILUNG

In diesem Projekt wurde der schriftliche Teil klar getrennt.

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

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.

10. ANHANG

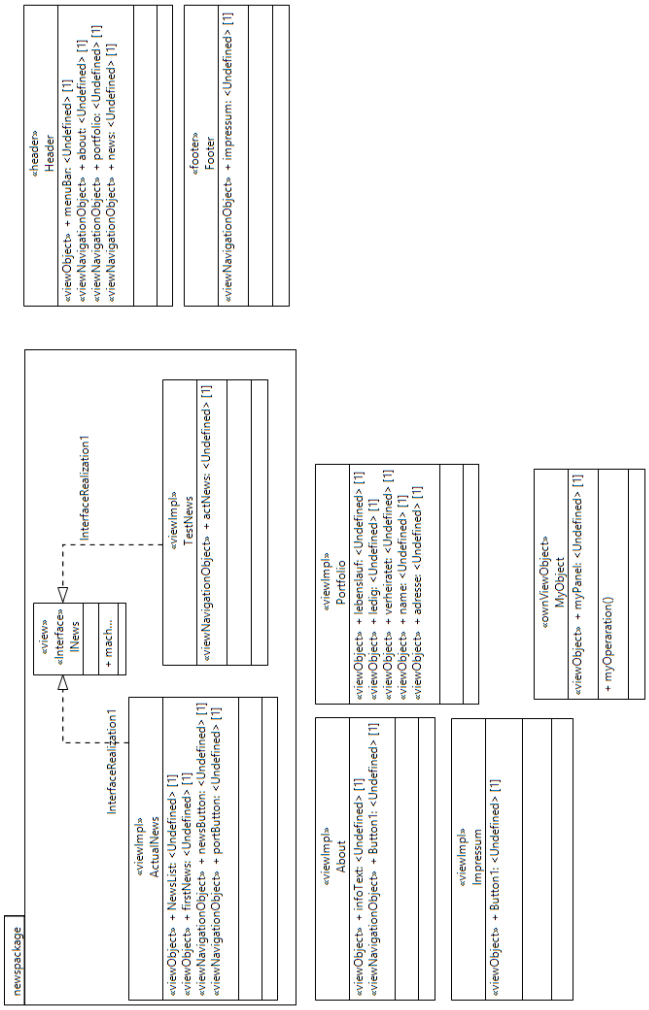

Fig. 10.1: Darstellung des Testmodells

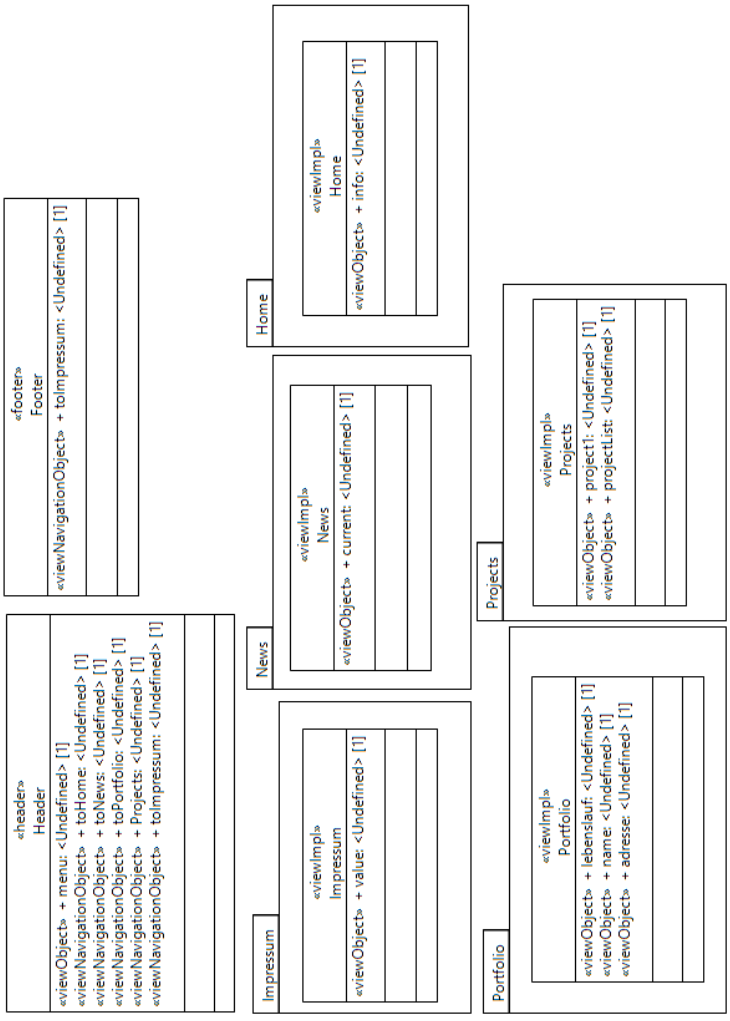

Fig. 10.2: Darstellung M1 Modell des vollständigen Anwendungsfalls

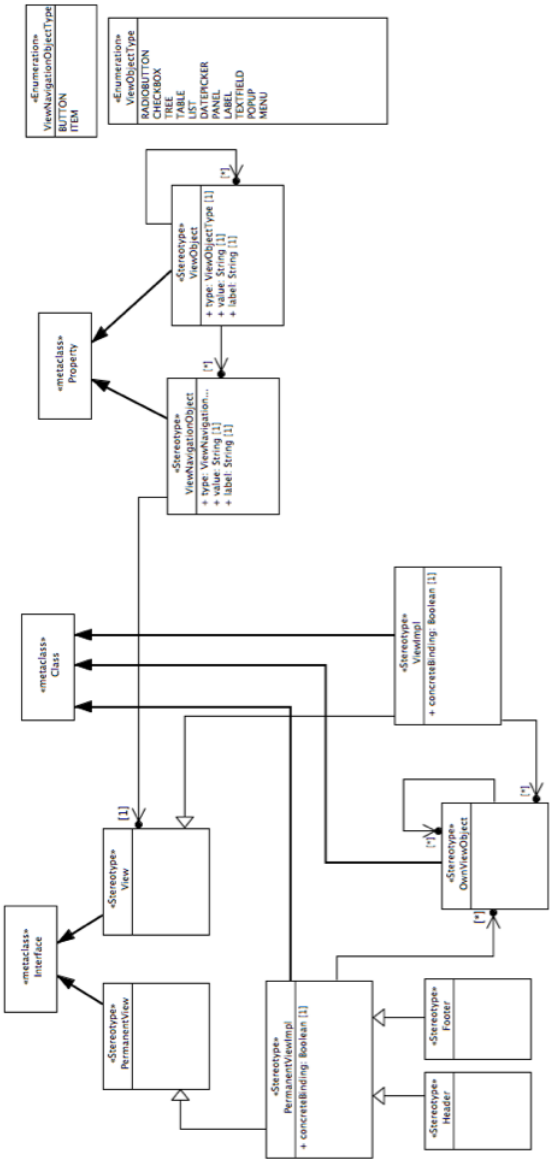

Fig. 10.3: Darstellung des gesamten UML Profil