DIPLOMARBEIT

Deklaratives GUI-Design für GWT mit XAML

Studiengang Telekommunikation und Medien

Institut für Medieninformatik

Fachhochschule St. Pölten

Betreuer: Dipl. Ing. Grischa Schmiedl

vorgelegt von

Bernhard Schauer

Dorfstraße 7

3142 Weißenkirchen

St. Pölten, im März 2009

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Diese Arbeit stimmt mit der von den Begutachtern beurteilten Arbeit überein.

Ort, Datum	*Unterschrift*

Danksagung

Mein Dank gilt meinem Betreuer Grischa Schmiedl, der Fachhochschule St. Pölten sowie allen Lehrbeauftragten.

Besonderen Dank möchte ich Conny und Lena sowie Martha und Franz aussprechen. Ohne ihre Unterstützung wäre mir dieses Studium nicht möglich gewesen.

St. Pölten, Österreich — Bernhard Schauer

3. März 2009

Kurzfassung

Viele Programmiersprachen weisen Nachteile auf, wenn sie für die Beschreibung von grafischen Benutzerschnittstellen eingesetzt werden. Sie erlauben im Quelltext eine andere Struktur bzw. Abfolge der grafischen Elemente, als jene der am Bildschirm tatsächlich angezeigten Seite. Auch ist es möglich, grafische Komponenten und Präsentationslogik zu vermischen. Eine deklarative Beschreibungssprache, zum Beispiel ein XML Dialekt, kann diese Nachteile ausmerzen.

In dieser Arbeit wird die beschriebene Problematik anhand der Technologien Google Web Toolkit (GWT) und eXtensible Application Markup Language (Xaml) behandelt. GWT ist ein Framework für die Erstellung von AJAX Applikationen, die in Java programmiert und dann zu JavaScript kompiliert werden. Xaml ist eine deklarative Sprache für die Beschreibung von Objektgraphen. Merkmale von Xaml sind eine spezielle XML Sytnax sowie eine Datenstruktur, die sich nach objektorientierten Strukturen ausrichtet.

Für die praktische Umsetzung wird ein Kompiler entworfen, der Xaml-Dateien in Java-Quelltext übersetzt. Für die Überprüfung einer Xaml Datei auf ihre Richtigkeit wird die GWT Klassenbibliothek herangezogen. Bei der Umsetzung zeigen sich einige Probleme, die zum Teil aus den unterschiedlichen Ansätzen von Xaml und Java in Bezug auf die Behandlung von Eigenschaften resultieren. Man kann aber feststellen, dass Xaml für die Deklaration von GWT-Benutzerschnittstellen verwendbar ist, wenn kleine Eingriffe in die GWT Klassenbibliothek vorgenommen werden.

Abstract

Many programming languages allow some unclean behaviour, when they are used to build graphical user interfaces. The source code of those user interfaces often shows a different structure than the page that is displayed on screen. Also it is possible to mingle graphical elements with presentation logic. Declarative languages, like XML-based languages, solve these problems.

This thesis is concerned with declaratively describing user interfaces for the Google Web Toolkit (GWT) using the eXtensible Application Markup Language (Xaml). GWT is a framework, which allows to develop AJAX applications in Java and to compile those sources to JavaScript. Xaml is a declarative language used to describe an object hierarchy. Features of Xaml are its special XML-syntax as well as its data-structure, which is designed for an object-oriented environment.

During the practical part of this theses a compiler is developed, which transforms Xaml files to Java source code. The necessary information for validating a Xaml file is obtained from GWTs class library. The implementation highlights some problems, which are partially caused by the different treatment of properties in Xaml and Java. After all it can be stated that Xaml can be used to describe user interfaces for GWT, if minor changes are made on GWTs class library.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Problemstellung und Motivation

Wann immer man ein Buch oder einen Artikel zu Software-Architektur liest, findet man die Forderung, verschiedene Aufgaben von einander zu trennen. Zu diesem Zweck werden oft Schichten gebildet. Ihr Vorteil liegt in der Austauschbarkeit einzelner solcher Schichten, was die Adaption der Software erleichtert (u.a. [Fow03], S 31f). Eine dieser Schichten ist die Präsentation, also die, heute meist grafische, Benutzerschnittstelle. Sie sollte von der Geschäftslogik getrennt sein, um eine Applikation zum Beispiel sowohl über eine Desktop-Anwendung als auch über eine Webseite bereitstellen zu können ohne dafür die gesamte Logik neu schreiben zu müssen. Doch auch innerhalb der Präsentationsschicht benötigt man Logik, um die Daten richtig darstellen zu können. Nun werden aber das grafische Design und die Präsentationslogik nicht notwendigerweise zur selben Zeit oder von den selben Personen erstellt. Eine Arbeitsteilung zwischen Programmieren und Grafikern bzw. Usability-Experten ist dabei vielleicht sogar wünschenswert. Daher möchte man auch den rein grafischen Teil der Präsentation von der nötigen Logik trennen.

Im Fall des Google Web Toolkit (GWT) ist diese Trennung eher weniger gut möglich. GWT ist ein Framework für AJAX Applikationenen, aus dem Haus Google (vgl. [GWT08q]). Es wird unter anderem von Google selbst für das Projekt Google Health, eine Applikation zum verwalten der eigenen Gesundheitsdaten (vgl. [Ram08]). Weiters

hat Red Hat angekündigt GWT als View Technologie in seine *JBoss Enterprise Application Platform* zu integrieren. Als Grund dafür wird angegeben, dass GWT, aus der Sicht Red Hats, Marktführer im Bereich der Rich Internet Applications Frameworks sei (vgl. [Hei08a]).

GWT ermöglicht dem Programmierer JavaScript zu vermeiden, und die gesamte AJAX Applikation in Java zu schreiben. Die grafische Oberfläche wird also durch Java Objekte, sogenannte Widgets, beschrieben. Eine Referenzierung dieser Objekte ist im Quelltext an jeder Stelle möglich, vorausgesetzt das Objekt wurde bereits instantiiert. Als Resultat davon wird der Programmierer zu keiner bestimmten Abfolge der Widgets gezwungen. Somit kann es schwierig werden aus einer Klasse, welche eine grafische Oberfläche beschreibt, herauszulesen wie diese Oberfläche tatsächlich aussieht.

Eine Möglichkeit eine klare Struktur zu forcieren, ist die deklarative Definition der grafischen Oberfläche. In einigen Frameworks wurde dieser Ansatz bereits umgesetzt. Zu nennen sind hier unter anderem XUL von Mozilla (vgl. [Boj07]), MXML von Adobe (vgl. [Coe04]) und Xaml von Microsoft (vgl. [MSD07m]). Alle genannten Sprache basieren auf XML. Sie erlauben es, die Definition einer grafischen Oberfläche von der damit verbundenen Logik zu trennen. Die Oberfläche wird dabei in einer XML Syntax beschrieben und mit einer weiteren Datei, welche den Code für die Eventbehandlung beinhaltet, verknüpft.

Der Autor musste in einem Webprojekt bei dem GWT zum Einsatz kam selbst die Erfahrung machen, wie unübersichtlich die Programmierung von grafischen Oberflächen mit einer Sprache wie Java ist. Es entstand der Wunsch nach einer Verbesserung. Eine deklarative Beschreibung mit einer XML artigen Syntax würde für eine übersichtliche Definition der Grafik sorgen, und auch die Präsentationslogik in eigene Klassen zwingen. Um nicht eine eigene XML Variante für diesen Zweck erfinden zu müssen, sollte eine bestehende Technologie für GWT adaptiert werden. In Frage kommen die drei bereits erwähnten XML Varianten. Für diese Arbeit ist die Wahl auf Xaml gefallen. Die *eXtensible Application Markup Language* wurde von Microsoft im Rahmen der *Windows Presentation Foundation* für die Beschreibung von grafischen Oberflächen entwickelt. Xaml ist aber mehr als das. Es ermöglicht eine Beschreibung von Objektgraphen, und kann daher nicht nur für den Spezialfall von grafischen Oberflächen ver-

wendet werden (vgl. [Rela]). Folgerichtig kommt Xaml auch in der *Windows Workflow Foundation* für die Definition von Workflows zum Einsatz (vgl. [MSD07k]).

1.2 Ziel und Aufbau der Arbeit

Im Folgenden soll ein praktikabler Weg aufgezeigt werden, wie GWT Oberflächen mit XAML beschrieben werden können. Generell muss die deklarative Beschreibung eine Benutzerschnittstelle eingelesen, ausgewertet und als Objektgraph dem jeweiligen Framework zur Verfügung gestellt werden. Abbildung 1.1 zeigt diese Struktur und soll in den weiteren Kapiteln dazu verwendet werden das behandelte Thema einzuordnen. Das angedeutete Xaml Information Set bezieht sich auf die Informationsstruktur, wie sie in der Xaml Spezifikation beschrieben wird (siehe 3.6).

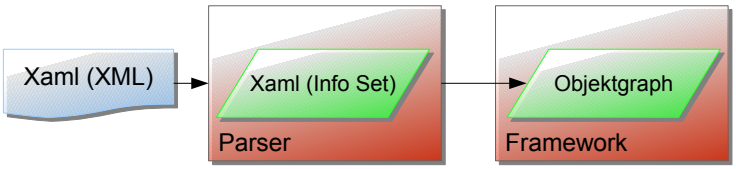

Abbildung 1.1: Überblick

Bevor der praktische Teil begonnen wird, sollen zuerst die verwendeten Technologien besprochen werden. Dazu erfolgt zu Beginn ein Vergleich der Merkmale von Xaml und anderen deklarativen XML Sprachen. Danach werden die Eigenschaften von Xaml erläutert. Um einen Eindruck über den Einsatz von Xaml für die Definition von graphischen Oberflächen zu gewinnen, geht ein weiteres Kapitel auf Xaml im Umfeld der WPF ein. Die Analyse der Technologien endet mit der Beschreibung von GWT. Im praktischen Teil der Arbeit wird zunächst eine Architektur für die Integration von GWT und Xaml entwickelt. Anschließend sollen die gewonnen Erkenntnisse in einer konkreten Implementierung umgesetzt werden. Es ist dabei jedoch nicht Ziel der Arbeit ein System für den Produktiv-Einsatz zu erstellen. Vielmehr sollen die Möglichkeiten und gegebenenfalls die Probleme bei der Realisierung dargestellt werden. Schließlich wird die Lösung hinsichtlich ihrer Machbarkeit und Konsistenz diskutiert.

Kapitel 2

Xaml, XUL, MXML - ein Vergleich

Dieses Kapitel stellt die drei Markupsprachen Xaml, XUL und MXML einander gegenüber. Es soll gezeigt werden, wo man Xaml einordnen kann, und ob es sich von anderen Markupsprachen unterscheidet. Thematisch ordnet sich dieses Kapitel am Beginn der in Abbildung 2.1 gezeigten Struktur ein.

Abbildung 2.1: Überblick

2.1 Xaml

Xaml wurde von Microsoft im Rahmen der WPF (siehe 4.1) entwickelt. Es dient aber nicht nur zum Beschreiben von grafischen Oberflächen, sondern kann allgemein zum Beschreiben von Objektgraphen eingesetzt werden. Ein Xaml Element entspricht dabei einer Klasse. Um die Eigenschaften dieser Klasse abzubilden kann die in XML übliche Attributsyntax verwendet werden. Es ist aber auch die Eigenschaftenelementsyntax möglich. Diese stellt eine Erweiterung gegenüber XML dar. Wie Listing 2.1 zeigt, wird dabei ein, aus dem Namen des Typen und jenem der Eigenschaft, zusammen gesetzter Elementname verwendet. Diese Schreibweise erlaubt die Deklaration komplexer Typen (siehe 3.3).

```
<Typname>
    <Typname.Attribute>
        <Attributwert />
    </Typname.Attribute>
</Typname>
```

Listing 2.1: Eigenschaftenelementsyntax

Mehrere Elemente auf der gleichen Ebene sind nur innerhalb von Aufzählungstypen möglich. Dieses Verhalten begründet sich auch in der Analogie zu Klassen und deren Eigenschaften (siehe 3.3). Ein Beispiel für eine in Xaml beschriebene Oberfläche ist in Listing 2.2 zu sehen (siehe 4.8). Mit dem ersten deklarierten Namespace werden WPF Elemente bekannt gemacht. Bei dem Namespace mit dem Präfix x handelt es sich um Standardvokabular von Xaml. Darin werden unter anderem Datentypen wie Double oder String definiert, aber auch Direktiven wie x:Class (siehe 3.5). Mit x:Class wird die Code-Behind Datei referenziert. Das ist jene Klasse, in der die Logik für die Oberfläche beschrieben wird. Darunter fällt auch die Eventbehandlung.

```
<Window x:Class="XAMLNamespacesProj.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:clr="clr-namespace:System;assembly=mscorlib"
    xmlns:prj="clr-namespace:XAMLNamespacesProj"
    Title="XAMLNamespacesProj" Height="300" Width="300">
```

```
    <StackPanel>
      <Button>
        <Button.Width>
          <clr:Double>
            200
          </clr:Double>
        </Button.Width>
        <prj:ZeichenFueller Anzahl="10" />
      </Button>
      <TextBox Width="200">
        <clr:String xml:space="preserve">    Hallo    </clr:String>
      </TextBox>
    </StackPanel>
  </Window>
```

Listing 2.2: Xaml Beispiel

([Fri07], S 52)

Neben jenen Elementen, die durch die WPF vorgegeben sind, können auch eigene verwendet werden. Dazu muss die Klasse referenziert werden, welche diesem Element entspricht. In Listing 2.2 werden zum Beispiel alle Klassen aus der .Net Klassenbibliothek eingebunden, die sich im Namensraum System befinden.

Die Methoden, welche Ereignisse behandeln, befinden sich in der Code-Behind Klasse. Sie werden mit Events verknüpft, indem der Methodenname als Attributwert des Events angegeben wird. Im Beispiel aus Listing 2.3 wird dem *Click* Event eines Buttons die Methode *ClickMich* zugewiesen (siehe 4.8).

```
<Button Click="ClickMich" Width="100">Klick mich </ Button >
```

Listing 2.3: Xaml Event

Weiters unterstützt Xaml Markuperweiterungen und Bindings. Dazu wird eine Notation mit geschweiften Klammern verwendet. Mit Markuperweiterungen können bereits instantiierte Objekte referenziert werden (siehe 4.7).

2.2 XUL

Mozilla verwendet in allen seinen Applikationen, wie zum Beispiel Firefox und Thunderbird, XUL. XUL ist eine XML basierte Sprache, die sich auf HTML, CSS, DOM und JavaScript stützt (vgl. [Boj07]). XUL Applikationen verwenden als Laufzeitumgebung XULRunner[1].

Innerhalb einer XUL Datei sind nicht nur die XUL-eigenen Tags erlaubt, welche Widgets darstellen, sondern es können auch andere XML basierte Sprachen wie XHTML und SVG verwendet werden (vgl. [Dea07c]). Von der Laufzeitumgebung wird die XUL Datei auf die gleiche Art wie HTML behandelt. Sie generiert einen DOM Baum, der dann dargestellt wird (vgl. [Dea07e]).

XUL Applikationen trennen meistens die Bereiche Layout, Stile und Skripte in eigene Dateien. XUL selbst dient hier für die Deklaration der Oberfläche, JavaScript für die Logik und CSS für die Gestaltung (vgl. [Dea07c]; [Boj07]).

Besonderheiten von XUL stellen XBL und Overlays dar. Mit Overlays kann man Teile einer bestehenden Oberfläche überschreiben. Es ist also möglich, eine Grundstruktur zu beschreiben und diese dann in einem Overlay anzupassen (vgl. [Boj07]). Die XML Binding Language (XBL) kann für zusammengesetzte Widgets verwendet werden. Dabei fügt man Elemente aus der Binding Datei in ein Element einer XUL Datei ein, wodurch diese Erweiterung versteckt wird. Ein Beispiel dafür zeigt Listing 2.4. Darin werden zwei Buttons aus der XBL Datei in das Box Element der XUL Datei eingebunden. Deklariert wird das Binding mittels CSS (vgl. [Dea07d]).

```
<!-- XUL (example.xul) -->
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
[...]
<window
     xmlns="http://www.mozilla.org/keymaster/gatekeeper/
     there.is.only.xul">
  <box class="okcancelbuttons"/>
</window>
```

[1] https://developer.mozilla.org/en/XULRunner

```
#CSS (example.css):
box.okcancelbuttons {
    -moz-binding: url(
        'chrome://example/skin/example.xml#okcancel');
}

<!-- XBL (example.xml): -->
<?xml version="1.0"?>
<bindings [...]>
  <binding id="okcancel">
    <content>
      <xul:button label="OK"/>
      <xul:button label="Cancel"/>
    </content>
  </binding>
</bindings>
```

Listing 2.4: XBL Beispiel

([Dea07d])

In Listing 2.4 ist auch der Standardnamensraum für XUL Applikationen zu sehen. Ebenso wird ein Stylesheet eingebunden, welches die Standardformatierungen vornimmt. Das Tag *window* stellt das Wurzelelement dar und ist mit HTMLs *Document* vergleichbar. Jede eigenständige XUL Datei beginnt mit diesem Element (vgl. [Dea07b]).

Events werden in XUL mit JavaScript behandelt. JavaScript kann entweder innerhalb von Script Tags oder in eigenen Dateien geschrieben werden. Als Eventmodell verwendet XUL die Spezifikation für DOM Events des W3C[2]. EventListener können für ein Element als Attribut deklariert werden. Man verwendet dabei als Attributname das Präfix *on*, gefolgt vom Namen des Events. Alternativ kann der Listener auch per JavaScript angefügt werden. Listing 2.5 zeigt beide Varianten (vlg. [Dea07a]).

```
<!-- EventListener als Attribut-->
<button label="OK" oncommand="alert('Button was pressed!');"/>
```

[2]http://www.w3.org/TR/DOM-Level-2-Events/

```
<!-- EventListener per JavaScript-->
<button id="okbutton" label="OK"/>
<script>
    function buttonPressed(event){
        alert('Button was pressed!');
    }
    var button = document.getElementById("okbutton");
    button.addEventListener('command', buttonPressed, true);
</script>
```

Listing 2.5: XUL Events

([Dea07a])

2.3 MXML

MXML ist die Markupsprache von Flex. Flex wird von Adobe entwickelt und tritt als Framework für Rich Internet Applications an. Als Sprachen für Flex dienen die Skriptsprache ActionScript und eben MXML für das Markup. Flex Applikationen werden in swf Dateien kompiliert und laufen in Browsern, welche über ein Flash Player Plugin verfügen. Außerdem können sie mit Adobes Air auch als Desktop Applikation eingesetzt werden (vgl. [Ado]).

Der Namespace, unter dem das gesamte Vokabular von Flex bereitgestellt wird, lautet *http://www.adobe.com/2006/mxml*. Per Konvention wird für ihn das Präfix *mx* verwendet. Verwendet man die IDE Flex Builder, so wird aus MXML ActionScript generiert. Das bedeutet, dass die gesamte graphische Oberfläche auch in ActionScript geschrieben werden kann (vgl. [Bro08], S 39). MXML Tags korrespondieren also mit ActionScript Klassen (vgl. [Bro08], S 61). Listing 2.6 zeigt die Definition eines Labels in MXML und in ActionScript.

```
<mx:Label text="Welcome to Flex!" id="myLabel" />

var myLabel:Label = new Label();
myLabel.text = "Welcome to Flex!";
```

Listing 2.6: MXML und äquivalentes ActionScript

([Bro08], S 61)

Um die Logik für die Oberfläche zu definieren, wird ActionScript verwendet. Dieses kann innerhalb eines Script Tags angegeben werden. Für die Behandlung von Events kann ActionScript aber auch direkt als Attributwert angegeben werden. Verwendet man dabei geschwungene Klammern, so wird ein Data Binding deklariert. Listing 2.7 zeigt beide Varianten in einem Beispiel.

```
<!-- Action Script als Attributwert-->
<mx:Button label="Test" id="myButton" x="90" y="96"
    click="myLabel.text = 'The button is clicked' "/>

<!-- Data Binding -->
<mx:TextInput id="myName" />
<mx:Label text="{myName.text}" id="myLabel" />
```

Listing 2.7: MXML Events

([Bro08], S 63, 170)

Die Deklaration der Oberfläche erfolgt insgesamt sehr geradlinig. Layoutkomponenten und Steuerelemente erhalten je ein Tag. Durch die Verschachtelung dieser Tags ergibt sich die Zugehörigkeit zu den übergeordneten Elementen. Listing 2.8 zeigt den prinzipiellen Aufbau einer Oberfläche, die links ein Menü (Tree), sowie im rechten Bereich eine Tabelle (DataGrid) und darunter ein Textfeld (TextArea) aufweist.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:HBox>
        <mx:Tree/>
        <mx:VBox>
            <mx:DataGrid/>
            <mx:TextArea/>
        </mx:VBox>
    </mx:HBox>
</mx:Application>
```

Listing 2.8: MXML Layout

([Coe04])

2.4 Vergleich

Im Vergleich der drei Technologien zeigt sich, dass sich MXML und XUL sehr ähnlich sind. Xaml hingegen hebt sich doch etwas von den Anderen ab. XUL und MXML unterscheiden sich hauptsächlich durch die Frameworks, in denen sie eingesetzt werden. Events werden mit der jeweiligen Skriptsprache behandelt. Dabei können Befehle dieser Skriptsprache direkt als Attributwerte für einen EventListener angegeben werden.

Xaml besitzt mit der Eigenschaftenelementsyntax eine über XML hinausgehende Syntax. Es gibt ein spezielles Verhalten bei Aufzählungen und als Attributwerte für EventListener werden Methodennamen verwendet. Ausdrücke sind hier nicht vorgesehen. Xaml hebt sich vor allem durch seine starke Orientierung an einer Objektstruktur von den beiden anderen Markupsprachen ab.

MXML ist so wie Xaml an Klassen gebunden, da es zu ActionScript umgewandelt wird. Hier ist dieser Ansatz aber nicht so weit getrieben worden wie bei Xaml.

XUL schließlich setzt auf DOM, um die Elemente des Markups darzustellen. Es lässt auch andere Markupsprachen, wie zum Beispiel SVG, zu. Vom Ansatz her ist es nicht so sehr auf das Ersetzen von Code durch Markup ausgerichtet wie Xaml und MXML. Man merkt, dass die Wurzeln von XUL bei einem Browser liegen.

Kapitel 3

Xaml

Xaml ist aus dem Haus Microsoft und steht für eXtensible Application Markup Language. Aus dem Namen lässt sich bereits erkennen, dass sich Xaml auf XML (eXtensible Markup Language) stützt. Xaml umfasst dabei jedoch mehr als nur reine XML Dateien. Wie Abschnitt 3.2 erläutert, ist Xaml stark an objektorientierte Frameworks angelehnt. Dazu wird eine eigene Datenstruktur (siehe Abschnitt 3.6) verwendet. Die Datenstruktur, die XML Syntax (siehe Abschnitt 3.3), sowie eine Vorschrift für den Parser, um die erwähnte Datenstruktur aus einer Xaml-XML Datei zu erstellen (siehe Abschnitt 3.7), bilden den Inhalt dieses Kapitels.

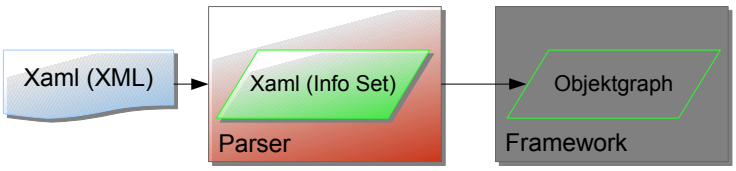

Abbildung 3.1: Überblick

3.1 Xaml im Überblick

Entwickelt worden ist Xaml im Rahmen der Windows Presentation Foundation (WPF) als deklarative Sprache zur Beschreibung von grafischen Benutzerschnittstellen. Ziel dieses Schrittes ist die Trennung von Darstellung und der dafür erforderlichen Logik. Eine Idee, die nicht neu ist. Objektorientierte Design Pattern wie Model View Controller schlagen ebenfalls eine derartige Trennung vor (vgl. [Fow03], S 367f).

Als Vorteil wird eine bessere Trennung zwischen den Tätigkeiten von Programmierern und Designern ins Treffen geführt. Designer sollen, unterstützt von entsprechenden grafischen Entwicklungswerkzeugen, einen für sie einfachen und vertrauten Zugang zur Erstellung von Benutzeroberflächen erhalten. Gleichzeitig können sich Programmierer um ihre Applikation kümmern, ohne dabei Zeit auf die exakte grafische Gestaltung verwenden zu müssen. Bei dieser Sichtweise stellt Xaml eine Schnittstelle dar, die durch ihre XML Syntax maschinenlesbar ist, und somit den Entwicklungswerkzeugen entgegenkommt. Weiters kann aber auch schlicht der Umfang des zu schreibenden Quelltextes durch den Einsatz von deklarativen Sprachen wie Xaml verringert werden. Der Grund dafür liegt darin, dass beim Interpretieren des Markups Funktionalität hinzufügt wird. Zum Beispiel werden einzelne Komponenten miteinander verknüpft, oder besonders prägnante Ausdrücke expandiert. (vgl. [Rela], S 14; [SH07])

Xaml wurde zwar im Rahmen der WPF entwickelt, ist aber nicht notwendigerweise an die WPF gebunden. Microsoft setzt Xaml im Rahmen der Windows Workflow Foundation zur deklarativen Beschreibung von Workflows ein (vgl. [MSD07k]). Darüber hinaus enthält Mircrosofts neue deklarative Programmiersprache M, Funktionen von Xaml. M weist eine C-artige Syntax auf, was sich nun gar nicht mit der oben beschriebenen XML Syntax von Xaml deckt (vgl. [Hei08c]; [Hei08b]). Was aber kein Problem darstellt, da Microsoft in der Spezifikation von Xaml die Darstellungsform nicht festlegt. XML wird lediglich als ein übliches Format beschrieben (vgl. [MS:08], S 7). Diese Arbeit wird sich auf die XML Darstellung beschränken. Zum einen tut das auch die WPF und die WF, zum anderen ist XML mittlerweile doch recht gängig und man ist den Umgang damit gewohnt.

3.2 Objektorientierung

Xaml ist, wie bereits erwähnt, an XML angelehnt und ein Xaml Dokument muss stets wohlgeformtes XML beinhalten. Es gibt jedoch auch einiges, worin sich Xaml von XML unterscheidet. Der Grund dafür liegt in der Objektorientierung von Xaml. Die Sprache ist darauf ausgelegt, einen Objektgraphen abzubilden. Xaml wird also üblicherweise nicht alleine verwendet, sondern steht im Kontext einer Klassenbibliothek. Die Elemente eines Xaml Dokuments werden bei seiner Analyse auf Objekte dieser Klassenbibliothek abgebildet bzw. wird beim Parsen für jedes Element ein zugehöriges Objekt instanziert. Elemente entsprechen also Objekten, die zugehörigen Attribute den Eigenschaften dieser Objekte. Um das zu ermöglichen, kennt Xaml im Gegensatz zu XML Typen. Als Attributwerte sind also nicht nur gewöhnliche Zeichenketten, sondern auch Objektdeklarationen möglich. Das erfordert eine spezielle Syntax, die in Abschnitt 3.3 beschrieben wird. (vgl. [SH07], S 19,72)

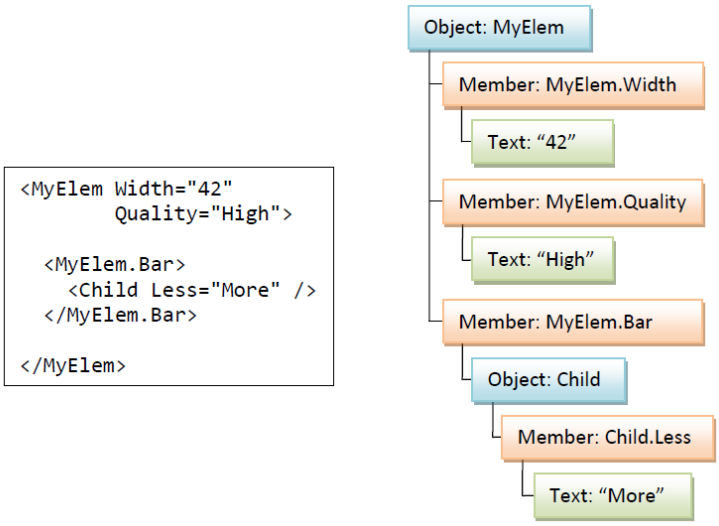

Abbildung 3.2: Struktur eines Xaml Objektgraphen ([MS:08], S 9)

Microsoft geht davon aus, dass zusammenhängende Elemente und Klassen denselben Namen tragen. Um diese Analogie zu unterstützen ist Xaml case sensitive. (vgl. [Relb]) Welches Element welcher Klasse zugeordnet ist, wird über XML Namespaces festgelegt (vgl. [XML06]). Somit wird auch das Vokabular einer speziellen Xaml Anwendung über die verwendeten Namespaces bestimmt (vgl. [MS:08], S 7). Von Microsoft wird zusätzlich zur individuellen Implementierung ein Standardvokabular mitgebracht. Dar-

auf wird im Detail in Abschnitt 3.5 eingegangen.

Da sich Xaml Elemente auf eine Klassenbibliothek stützen, können sie auch Attribute erben. Möchte man ein eine Schemadefinition mit DTD, XML Schema oder RELAX NG erstellen, muss man also die Verbungshierarchie auflösen (vgl. [MSD07m]). Ganz abgesehen davon sind Xaml Parser ausdrücklich dazu angehalten auf eine DTD-Direktive mit einem Fehler zu reagieren. (vgl. [MS:08], S 62) In diesem Zusammenhang sei noch eine weitere Abweichung von den XML Gepflogenheiten erwähnt. In Xaml Dateien wird die XML-Deklaration `<?xml version="1.0"?>` weggelassen (vgl. [Fri07], S 45).

3.3 Besonderheiten der Xaml Syntax

In diesem Abschnitt soll auf jene Aspekte der Xaml Syntax eingegangen werden, die sich von XML unterscheiden. Als Referenz wird die im Rahmen der WPF verwendete Syntax herangezogen, da dem Autor keine gesonderte Spezifikation einer Xaml-XML [1] Syntax vorliegt. Würde es eine solche geben, würde sie sich aber auch kaum von jener in der WPF eingesetzten unterscheiden, da ja beides aus dem Haus Microsoft stammt. Der Autor sieht daher die WPF in Hinblick auf die Xaml-XML Syntax als eine Referenzimplementierung an.

3.3.1 Attributsyntax

Die Attributsyntax ist die in XML übliche Art, den Wert eines Attributs anzugeben. Diese Schreibweise ist besonders kurz und daher ideal für alle Fälle, wo der Wert des Attributs in einer einfachen Zeichenkette dargestellt werden kann.

```
<Typname Attributename="Wert" />
```

Listing 3.1: Attributsyntax

Da Xaml typisiert ist, muss die als Attributwert angegebene Zeichenkette vom Parser in ein Objekt umgewandelt werden. Es gibt dafür zwei Möglichkeiten. Die erste ist

[1] An dieser Stelle sei nochmals darauf hingewiesen, dass die "Xaml Object Mapping Specification" nach [MS:08] lediglich die Datenstruktur festlegt. Es wird darin keine Aussage über die physische Repräsentation dieser Daten getroffen.

die Konvertierung der Zeichenkette in den Typ der, zu diesem Attribut gehörenden, Eigenschaft. Dabei muss die Syntax eingehalten werden, die das zugrunde liegende Framework für die Konvertierung vorgibt. Mit der *Typkonvertierung* können lange und mächtige Konstrukte erzeugt werden. Einerseits erreicht man so eine kompakte Schreibweise, andererseits leidet aber die Lesbarkeit. Stropek empfiehlt daher in ([SH07], S 81) bei komplexen Typen die Eigenschaftenelementsyntax zu verwenden. Die zweite Möglichkeit den Wert einer Eigenschaft in der Attributsyntax festzulegen, sind *Markuperweiterungen*. Hier wird eine Referenz auf ein Objekt, als Ausdruck, angegeben. Es wird also nicht notwendigerweise ein neues Objekt erzeugt, es kann auch ein bestehendes referenziert werden. Als syntaktisches Merkmal, sind Markuperweiterungen immer von geschweiften Klammern umschlossen. Neben eigenen Markuperweiterungen definiert Xaml bereits selbst einige über das Standardvokabular im Namespace *x*.

x:Array wird zur Definition von Arrays verwendet. Es stellt eine Ausnahme dar, da es üblicherweise als Element und nicht als Attributwert innerhalb der Attributsyntax geschrieben wird.

x:Null Einem Attribut wird der Wert *null* zugewiesen. `<object property="{x:Null}"/>`

x:Static Zuweisen von statischen Eigenschaften.

```
<Button Background="{x:Static Brushes.Blue}"/>
```

x:Type Angabe eines Typen. `<Style TargetType="{x:Type Button}"/>`

(vgl. [Fri07], S 54; [MSD07i]; [Sch07], S 3)

3.3.2 Eigenschaftenelementsyntax

Die Eigenschaftenelementsyntax oder Property-Element-Syntax ermöglicht auch komplexe Datentypen als Attributwerte. Dabei wird ein Kindelement in dem Element angegeben, zu dem das Attribut gehört. Als Name dieses Attributelements dient der Typname (das ist der Name des Elements für welches das Attribut festgelegt wird), gefolgt von eine Punkt und dem Namen des Attributs. Im Gegensatz zu Xaml würde XML hier keinen Zusammenhang herstellen und einfach ein Kindelement einhängen.

Wird bei dieser Schreibweise eine Zeichenkette angegeben, so bleiben Zeilenumbrüche darin erhalten. (vgl. [MSD07i])

```
<Typname>
    <Typname.Attribute>
        <Attributwert />
    </Typname.Attribute>
</Typname>
```

Listing 3.2: Eigenschaftenelementsyntax

3.3.3 Inhaltssyntax

Jedem der bereits einmal mit XML gearbeitet hat, wird das Konstrukt `<Element>Textnode</Element>` bekannt vorkommen. In Xaml ist diese Art der Angabe einer Zeichenkette nicht ohne weiteres möglich. Der Grund dafür liegt wieder in der Objektorientierung von Xaml. Da ein Element ein Objekt darstellt, müssen alle darin deklarierten Werte auf Eigenschaften dieses Objekts umgelegt werden. Nun stellt sich die Frage, welche Eigenschaft zum Inhalt des Elements gehört. Darum wird eine Eigenschaft als Inhaltseigenschaft definiert. Vom Parser wird alles zwischen dem öffnenden und dem schließenden Tag eines Elements als Wert dieser Eigenschaft interpretiert. (vgl. [MSD07i]; [SH07], S 74f)

Weiters ist es in XML üblich auch mehrere Elemente als Kindelemente zu deklarieren. Auch das lässt sich nicht einfach so auf die Eigenschaft eines Objekts umlegen. Deshalb wird, wenn mehrere Kindelemente vorhanden sind, implizit ein Auflistungstyp erstellt. Alle Kindelemente werden diesem Auflistungstyp zugeordnet. Dadurch kann wieder die Zuordnung als Wert einer Objekteigenschaft erreicht werden. Diese Vorgangsweise wird als Inhaltssyntax für Auflistungstypen bezeichnet. Mit diesen Maßnahmen soll die gewöhnte XML Schreibweise erhalten bleiben. Außerdem wird die Struktur des Dokuments klarer und die notwendigen Zeilen an Quelltext minimiert. (vgl. [MSD07i])

```
<Element>
    <Auflistungstyp> <!-- diese Zeile kann entfallen -->
        <Kind>Wert1</Kind>
        <Kind>Wert2</Kind>
    </Auflistungstyp> <!-- diese Zeile kann entfallen -->
</Element>
```

Listing 3.3: Inhaltssyntax

Im Folgenden werden noch einmal Attributsyntax, Eigenschaftenelementsyntax und Inhaltssyntax dargestellt. Alle drei Beispiele liefern das gleiche Ergebnis.

```
<!-- Attributsyntax -->
<Element Content="Wert" />

<!-- Inhaltssyntax, wobei die Eigenschaft Content
    als Inhaltseigenschaft definiert wurde -->
<Element>Wert</Element>

<!-- Eigenschaftenelementsyntax kombiniert mit Inhaltssyntax -->
<Element>
    <Element.Content>Wert</Element.Content>
</Element>
```

Listing 3.4: Gegenüberstellung der Syntaxarten

3.3.4 Angefügte Eigenschaften

Angefügte Eigenschaften oder Attached Properties sind Eigenschaften eines Elternelements, die aber beim Kindelement deklariert werden. Als Syntax dient

`BesitzerTyp.Eigenschaftsname` .

Im Rahmen von Benutzerschnittstellen können angefügte Eigenschaften dazu verwendet werden, die Position eines Elements im Elternelement festzulegen (vgl. [Mac06], Abschnitt 3.4) [2]. Zur Veranschaulichung wird ein Beispiel aus der WPF (siehe Abschnitt 4) herangezogen. Zwei Buttons werden in ein DockPanel eingehängt. Dieser

[2]Dem Autor stand dieses Buch nur in Html Form auf www.safaribooksonline.com zur Verfügung. Daher können als Referenz nur die Abschnitte und keine Seitenzahlen verwendet werden.

Vorgang wird einmal in Xaml (Listing 3.5 und einmal in C# (Listing 3.6 dargestellt. Ein DockPanel erlaubt es, untergeordnete GUI Elemente an den vier Seiten des Panels oder in dessen Mitte zu platzieren, also "anzudocken". In diesem Beispiel wird ein Button links und einer rechts positioniert. Wie man im C# Code sieht, wird für die Angabe der Position die Methode *SetDock* der Klasse *DockPanel* aufgerufen. Als Parameter wird das Element, dessen Position festgelegt werden soll, sowie die Position im DockPanel verlangt. Somit zeigt sich die Notwendigkeit, das Attribut *Dock* bei den Kindelementen von *DockPanel* zuzuweisen, da ja ein Attribut in einem Element nur einmal angegeben werden kann. Außerdem fördert diese Schreibweise die Lesbarkeit. Ebenso wie Attribute können auch Events an ein übergeordnetes Element angefügt werden. (vgl. [MSD07m]; [MSD07c])

```
<DockPanel>
    <Button DockPanel.Dock="Left"
        Width="100"
        Height="20"
    >
        I am on the left
    </Button>
    <Button DockPanel.Dock="Right"
        Width="100"
        Height="20"
    >
        I am on the right
    </Button>
</DockPanel>
```

Listing 3.5: Angefügte Eigenschaften: Beispiel DockPanel Xaml

```
DockPanel myDockPanel = new DockPanel();

Button leftButton = new Button();
leftButton.Content = "I am on the left";
leftButton.Width = 100,0;
leftButton.Height = 20,0;

Button rightButton = new Button();
rightButton.Content = "I am on the right";
```

```
rightButton.Width = 100,0;
rightButton.Height = 20,0;

DockPanel.SetDock(leftButton, Dock.Left);
DockPanel.SetDock(rightButton, Dock.Right);

myDockPanel.Children.Add(leftButton);
myDockPanel.Children.Add(rightButton);
```

Listing 3.6: Angefügte Eigenschaften: Beispiel DockPanel C#

3.4 Code-Behind und Ereignisse

Der für die Darstellung der Benutzerschnittstelle erforderliche Code kann in so genannten Code-Behind Dateien ausgelagert werden. Code-Behind Dateien sind Klassen, die sich auf die Benutzerschnittstelle beziehen, und müssen mit dieser verknüpft werden. Wie diese Verknüpfung erfolgt ist vom verwendeten Framework abhängig. In der WPF werden die Code-Behind Dateien als partielle Klassen implementiert und über das Attribut x:Class referenziert (siehe Abschnitt 4.3). Ein wichtiger Bestandteil der Benutzerschnittstellenlogik ist die Behandlung von Ereignissen. Die Zuweisung von Eventhandler zum Ereignis erfolgt meist über die Attributsyntax. (vgl. [MSD07m])

```
<Button Click="ClickHandler" >OK</Button>
```

Listing 3.7: Eventhandler zuweisen

Auch die Ereignisbehandlung hängt vom verwendeten Framework ab. In den Abschnitten über WPF und GWT wird auf deren Art der Ereignisbehandlung genauer eingegangen.

3.5 Standardvokabular

Microsoft stellt unter dem Namespace http://schemas.microsoft.com/winfx/2006/xaml ein Standard Vokabular, auch Intrinsic Schema genannt, für Xaml bereit. Somit muss dieses Vokabular nicht von jeder Xaml Implementierung erneut definiert werden. Per Konvention wird dieser Namespace mit dem Präfix *x* versehen. Es werden darin Typen und Direktiven angegeben.

3.5.1 Typen

Bei den Typen handelt es sich um XamlType Information Items. Mit ihnen werden vier Bereiche abgedeckt. Zum ersten sind das Datentypen, die in den meisten Programmiersprachen verwendet werden. Darunter fallen zum Beispiel String, Double, Boolean oder Object. Zu Object sei angemerkt, dass es aus Xaml-Sicht nicht der Basistyp für alle anderen Typen ist. Soll ein Attribut den Typ x:Object aufnehmen können, muss dieses Attribut explizit so deklariert werden (siehe 3.6.2) (vgl. [MS:08], S 41).

Weiters werden Typen definiert, welche in Xaml Schema Information Sets verwendet werden. Darunter fallen zum Beispiel XamlType oder XamlEvent. Als dritte Gruppe gibt es Typen, die für das Einbetten von Code bzw. XML in eine Xaml verwendet werden. Das sind Code und XData. Schließlich kann mit eigenen Typen das Verhalten anderer Typen bestimmt werden. Zu nennen sind hier unter anderem ArrayExtension und NullExtension.

Nähere Informationen zu den Typen des x Namespace finden sich in [MS:08] ab Seite 39.

3.5.2 Direktiven

Bei Direktiven handelt es sich um XamlMember Information Items, die verwendet werden um ein Verhalten zu beschreiben. Bei vielen dieser Direktiven ist das genaue Verhalten nicht definiert. Es obliegt der jeweiligen Anwendung zu bestimmen, was mit den Werten in diesen Direktiven geschehen soll. In der WPF wird mit *Class* der Name

der Code-Behind Klasse angegeben. Bei der Xaml Implementierung für GWT hingegen dient *Class* zum Festlegen des Namens jener Klasse, die aus der Xaml Datei generiert wird.

Eine Liste mit allen Direktiven findet sich in [MS:08] ab Seite 46. An dieser Stelle werden nur jene XamlMember aufgelistet, die im weiteren Verlauf der Arbeit benötigt werden.

x:Item Eigenschaft, die verwendet wird, wenn ein Element mehrere Kinder enthält.

x:Name Eindeutiger Name für das Element, bzw. das daraus erstellte Objekt.

x:Class Angabe einer Klasse, die zu dieser Xaml Datei in Beziehung steht. Wird üblicherweise im Wurzelelement angegeben und kann für Code-Behind Dateien verwendet werden.

x:ClassModifier Kann Modifizierer, wie die Sichtbarkeit, zum Beispiel für generierte Klassen oder die Code-Behind Datei angeben. Das genaue Verhalten ist nicht spezifiziert und hängt von der jeweiligen Implementierung des Parsers ab.

3.6 Datenmodell von Xaml

In der "Xaml Object Mapping Specification" [MS:08] ist das Datenmodell einer Xaml Instanz beschrieben. Grundsätzlich gibt es zwei Arten von Datenmodellen. Ein *Xaml Information Set* repräsentiert die Daten einer Xaml Instanz. Parst man also zum Beispiel ein Xaml Dokument, welches in XML Syntax verfasst ist, so erhält man ein Xaml Information Set. (vgl. [MS:08], S 24]) Ein *Xaml Schema Information Set* beschreibt wie ein solches Xaml Information Set auszusehen hat. Es erfüllt also eine ähnliche Funktion wie *XML Schema* (vgl. [XML04b]). Ein konkretes Xaml Information Set wird als Instanz eines Xaml Schema Information Sets bezeichnet, wenn es sich an die darin vorgegebenen Regeln hält, und auch sonst wohlgeformt und valide ist. (vgl. [MS:08], S 13)

3.6.1 Xaml Information Set

Ein Xaml Information Set, also die Daten einer Xaml Instanz, setzt sich prinzipiell aus drei verschiedenen Elementen zusammen. Das sind Objekte, Eigenschaften und Text (vgl. [MS:08], S 9). In der Xaml eigenen Terminologie heißen diese Elemente *Object Node Information Item*, *Member Node Information Item* und *Text Node Information Item* (vgl. [MS:08], S 24). Abbildung 3.2 illustriert diesen Aufbau, und zeigt auch, dass ein Xaml Information Set eine Baumstruktur aufweist. Diesen Umstand kennt man bereits von XML her. Ebenfalls analog zu XML gibt es in Xaml ein Document Information Item. Im Folgenden wird nun auf die einzelnen Elemente des Xaml Information Set eingegangen.

Document Information Item

Das Document Information Item ist einmalig in einer Xaml Instanz. Es hat lediglich eine Eigenschaft, *document object*, welche das Wurzelobjekt der Xaml Instanz referenziert. Dieses Wurzelobjekt ist immer vom Typ Object Node Information Item.

Object Node Information Item

Object Node Information Items stellen die Objekte im Objektgraph einer Xaml Instanz dar. Alle Objekte referenzieren nicht nur ihre Kinder, sondern auch das Elternobjekt. Die Ausnahme dieser Regel stellt das Wurzelobjekt dar. Da es keine Eltern besitzt, ist seine Eigenschaft *parent member* gleich Null. Weiters ist das Wurzelobjekt jener Ort, an dem allgemein gültige Direktiven wie etwa x:Class definiert werden (vgl. [MS:08], S 32). Eine der Eigenschaften des Object Node Information Item ist *member nodes*. Darin werden alle Eigenschaften jenes Objektes aufgelistet, das von diesem Object Node dargestellt wird. Member Nodes erlaubt keine Dubletten. Diese Forderung ist auch einleuchtend, da andernfalls die Baumstruktur der Xaml Instanz verletzt würde. Um ein mit Xaml erstelltes Objekt ansprechen zu können, benötigt es einen, im zugehörigen Namespace Scope, eindeutigen Namen (vgl. [MS:08], S 34). Dieser kann entweder über die Eigenschaft x:Name oder über eine objekteigene Namenseigenschaft festgelegt werden. Beides zugleich ist jedoch nicht möglich. Objekte die in Xaml verwendet werden sollen, also durch einen Object Node representiert werden, müssen einen Standardkonstruktor aufweisen. Ansonsten können sie beim Auswerten von Xaml nicht instantiiert werden. (vgl. [MS:08], S 25f)

Member Node Information Item

Mit Hilfe von *Member Node Information Items* werden die Eigenschaften für ein Objekt angegeben. Wie Abbildung 3.2 zeigt, hängt an einem Member Node Information Item entweder ein Objekt oder Text. In der Eigenschaft *values* kann prinzipiell nur ein XamlMember Information Item angegeben werden. Soll eine Liste von Objekten erstellt werden, muss die Eigenschaft *member* entweder auf x:Items oder x:DirectiveChildren lauten. Außerdem sind auch für x:ConstructorArgs mehrere Werte möglich. Eine weitere Voraussetzung für Listen ist der richtige Typ des Elternobjekts. Genauer gesagt muss beim Typen entweder *is dictionary* oder *is list* auf True gesetzt sein. Typen werden im Xaml Schema Information Set beschrieben.

Im Fall eines Dictionary sind nur Objekte, die einen eindeutigen Schlüssel angeben, in *values* erlaubt. Angegeben wird dieser Schlüssel über die x:Key Direktive oder eine

entsprechende Eigenschaft des Objektes. Natürlich muss auch der Typ des Schlüssels zum Dictionary passen. Die Liste dieser Typen wird in der Eigenschaft *allowed key types* des Dictionary Objektes festgelegt. Zu den Member Node Information Items x:Subclass, x:ClassModifier und x:FieldModifier sei erwähnt, dass sie nur gültig sind, wenn auch die x:Class Direktive angegeben wird. (vgl. [MS:08], S 31f)

Text Node Information Item

Mit dem *Text Node Information Items* werden neben herkömmlichem Text auch XML Literale angegeben. Ebenso zählt Whitespace als Text Node. Wie Whitespace behandelt wird, findet sich im Abschnitt über den Xaml Parser. (vgl. [MS:08], S 36)

3.6.2 Xaml Schema Information Set

Ein *Xaml Schema Information Set* definiert das Schema für konkrete Xaml Instanzen. Es kann fünf Arten von Elementen enthalten. Diese sind: *Schema Information Item*, *XamlType Information Item*, *XamlMember Information Item*, *Text Syntax Information Item*, *Constructor Information Item*. Die konkrete Form der Darstellung eines Xaml Schema Information Set ist von Microsoft nicht spezifiziert. Im Folgenden werden die fünf genannten Elemente im Detail beschrieben. (vgl. [MS:08], S 13)

Schema Information Item

In jedem Xaml Schema gibt es genau ein *Schema Information Item*, welches das Wurzelelement darstellt. Für ein Schema können neben seinem Namespace auch die darin verfügbaren Typen und Direktiven aufgelistet werden. Bei beiden Listen dürfen keine Dubletten auftreten. Als Kriterium für die Eindeutigkeit dient der Name. Formal unterscheiden sich Direktiven von Typen durch die Eigenschaft *is directive*, welche auf *True* gesetzt ist. Neben den Typen in *types* gibt es noch jene in *assignable types*. Diese Typen bieten nicht die selben Möglichkeiten wie "normale" Typen. Was das genau bedeutet, wird in der Spezifikation aber verschwiegen. Schließlich werden noch kompatible Schemata aufgelistet. Diese dienen zur Versionierung, wobei ein neueres Schema

als kompatibel zu einem Älteren angegeben wird. Es ist damit aber auch möglich, in mehreren Schemata im gleichen Namensraum zu behandeln. (vgl. [MS:08], S 14f)

XamlType Information Item

Mit *XamlType Information Items* werden die Datentypen einer Xaml Instanz definiert. Es wurde bereits mehrfach auf den objektorientierten Ansatz von Xaml und die Verbindung zu Klassenbibliotheken hingewiesen. Es ist aber zu beachten, dass Xaml Vererbungshierarchien nicht implizit unterstützt. Kann einer Eigenschaft ein bestimmter Basistyp zugewiesen werden, ist damit nicht automatisch auch eine Zuweisung von Ableitungen dieses Typs möglich. Alle Typen müssen explizit in der Eigenschaft *types assignable to* aufgelistet sein. Wie bereits erwähnt, gibt es das Konzept der kompatiblen Schemata. Typen aus kompatiblen Schemata werden ihrerseits als kompatibel angesehen, wenn sie den gleichen Namen tragen. Um eine Eigenschaft einem Objekt zuweisen zu können, müssen die beiden Typen zusammenpassen. Dafür wird einerseits die Liste der *types assignable to* herangezogen. Andererseits wird auf die Kompatibilität der Typen geachtet. Zur Bestimmung von Zuweisbarkeit und Kompatibilität dienen die Methoden *isCompatibleWith(t1,t2)* und *isAssignableTo(tFrom,tTo)*.

Unter den Eigenschaften des *XamlType Information Items* gibt es einige die sich widersprechen können. So ist es nicht möglich gleichzeitig *content property*, *is list* oder *is dictionary* zuzuweisen bzw. True zu setzen. Weiters dürfen die Eigenschaften *allowed types* und *allowed key types* nur für Listen und Dictionaries verwendet werden. Wird in *types assignable to* eine Markuperweiterung angegeben, so muss es einen Rückgabewert geben. Andernfalls muss der Rückgabewert allerdings frei bleiben, oder in anderen Worten, *return value type* muss Null sein. Ebenso dürfen Konstruktorargumente nur in Zusammenhang mit einer Markuperweiterung angegeben werden. Außerdem darf es für einen Typ keine zwei *Constructor Information Items* mit der selben Anzahl von Argumenten geben.

XamlMember Information Item

XamlMember Information Items stellen entweder eine Eigenschaft eines Objekts oder eine Direktive dar. Eine Objekteigenschaft darf natürlich nur einmal pro Objekt vorkommen. Die Eindeutigkeit wird wieder über den Namen festgestellt.

Ist das *XamlMember Information Item* eine Direktive, ist also *is directive* gleich *True*, so gilt diese Einschränkung nicht. Zu welchem Typ ein *XamlMember Information Item* gehört wird über *owner type* festgelegt. Umgekehrt muss der Member auch beim Typen in *members* eingetragen sein. Da Direktiven nicht zu einem einzelnen Typen gehören, muss bei ihnen die *owner type* Eigenschaft *Null* sein.

Handelt es sich bei dem *XamlMember Information Item* um eine angefügte Eigenschaft, so muss mit *target type* bestimmt werden, welche Typen diese Eigenschaft verwenden dürfen. Auch hier gibt es wieder Attribute, die sich gegenseitig ausschließen. Ein *XamlMember Information Item* kann nur entweder eine angefügte Eigenschaft, ein Event oder eine Direktive sein. Weiters darf *is readonly* nur für statische Eigenschaften, Listen und Dictionaries auf *True* lauten. Für Listen und Dictionaries ist das Verweigern von Schreibzugriff möglich, da damit nur das komplette Ersetzen der Liste bzw. des Dictionaries verboten wird und nicht das Ändern ihrer Elemente. (vgl. [MS:08], S 18f)

Text Syntax Information Item

Ein *Text Syntax Information Item* beschreibt wie die Werte für Eigenschaften dargestellt werden müssen. Dafür gibt es zwei Möglichkeiten. Entweder wird ein Pattern oder einzelne Literale angegeben. Dafür stehen das *Value Syntax Information Item* und das *Pattern Syntax Information Item* zur Verfügung. Patterns sind dann geeignet, wenn ein Aufzählung der erlaubten Literale nicht sinnvoll ist. Zum Beispiel macht es keinen Sinn alle positiven, natürlichen Zahlen auflisten zu wollen. Die Pattern werden als reguläre Ausrücke beschrieben und entsprechen jenen aus den XML Schema Definitions (vgl. [XML04a]) . Ein *Text Syntax Information Item* kann mehrere Patterns und Values aufweisen. Ein Text wird dann als valid angesehen, wenn eine passende Beschreibung gefunden wird. (vgl. [MS:08], S 20f)

Constructor Information Item

Mit Hilfe des *Constructor Information Item* wird eine Parameterliste für den Konstruktor eines Typen angegeben. Wie bereits erwähnt dürfen keine zwei *Constructor Information Items* mit der selben Anzahl von Argumenten für den selben Typ definiert werden. Constructor Information Items werden nur für Markuperweiterungen eingesetzt. (vgl. [MS:08], S 23f)

3.7 Parser

In der "Xaml Objekt Mapping Specification" von Microsoft, werden in Abschnitt 6 ([MS:08], S 58f) die Anforderungen an einen XML-Xaml Parser beschrieben. Dieser Parser beschreibt mehrere Methoden (siehe Auflistung), in der Spezifikation Regel genannt, die jeweils Teilaspekte der Funktionalität abdecken. Für jede dieser Regeln sind Parameter und ein Rückgabewert definiert. Der Prozess beginnt mit der Regel *XML:document Processing*, welche als Parameter das Dokument-Element des XML Informtion Sets verlangt. Von dieser Regel aus werden dann alle weiteren bei Bedarf aufgerufen. Als Rückgabewert erhält man ein Document Information Item, das alle weiteren Elemente enthält.

- XML:document Processing
- Object Node Creation from an XML:element
- Member Node Creation from an XML:attribute
- Value Creation from Attribute Text
- Member Node Creation from an XML:element
- Member Node Creation from Content
- Object Node Creation from a Markup Extension in an Attribute
- Member Lookup
- Xml Namespace Mapping Conversion

Die Richtigkeit der XML Elemente wird durch Abfragen des Xaml Schemas bestimmt. Dabei wird im Schema nach einem Typ oder einem Member gesucht der dem Namen des XML Elements entspricht. Ist die Sucher erfolgreich kann das entsprechende Xaml Element erstellt werden. Andernfalls tritt ein Fehler auf. Es steht jeder Implementation dieses Parsers frei nach einem Fehler abzubrechen oder weiterzuarbeiten.

Kapitel 4

XAML in der Windows Presentation Foundation

Um die relativ allgemeine Beschreibung von Xaml etwas konkreter werden zu lassen, wird in diesem Kapitel beleuchtet, wie Xaml in die WPF eingebunden ist. Da Xaml im Rahmen der WPF entwickelt wurde, kann hiervon die Intention hinter den Möglichkeiten, welche Xaml bietet, abgelesen werden. Themen sind die Struktur einer WPF Applikation, der Build-Prozess sowie das Zusammenspiel von Xaml-Sprachelementen und WPF-Framework. Durch diese Betrachtungen soll ein Eindruck vom den konkreten Einsatz von Xaml und den damit verbunden Möglichkeiten vermittelt werden. Im Rahmen des .Net Framework können schließlich die verschiedensten Sprachen verwendet werden. Die vorliegende Arbeit beschränkt sich auf C#, da dieses durchaus gebräuchlich ist und aufgrund seiner C-artigen Syntax sowie einer gewissen Ähnlichkeit zu Java für die meisten Leser einfach nachzuvollziehen sein sollte.

Abbildung 4.1: Überblick

4.1 WPF im Überblick

Mit dem .Net Framework 3.0 wurden einige neue Frameworks in die .Net Programmierung eingeführt. Eines davon ist die Windows Presentation Foundation. Sie tritt als Nachfolgerin der Windows Forms an und ist somit für die Darstellung von graphischen Benutzerschnittstellen zuständig. Neben herkömmlichen Desktop-Fenstern, können mit der WPF auch 3D-Grafiken, Animationen, Dokumente und Browser-basierte Anwendungen erstellt werden. Die WPF übernimmt auch das Rendering und greift dafür auf DirektX zurück. Da die Ausgabe auf Vektoren basiert, kann sie unabhängig von der Auflösung des jeweiligen Displays erfolgen.

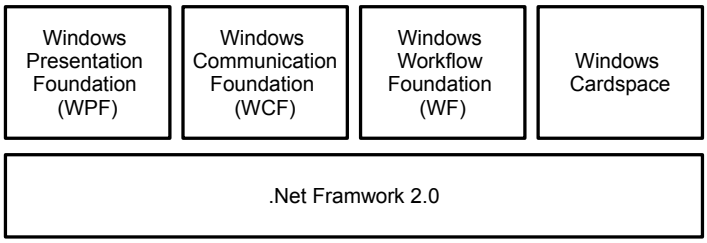

Abbildung 4.2: Säulen des .Net Frameworks 3.0 ([Fri07], S 16)

Die Klassenbibliothek befindet sich im Namespace *System.Windows*. Dort sind auch die alten Windows Forms enthalten, in *System.Windows.Forms*. Daraus könnte man ableiten, dass es gemeinsame Basisklassen für WPF und Windows Forms gibt, was aber nicht der Fall ist. WPF ist eine vollständig neue Implementierung. (vgl. [Sch07])

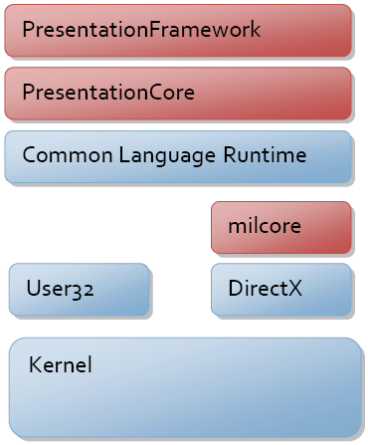

Abbildung 4.3: Aufbau WPF ([MSD07l])

4.2 WPF Projektaufbau und Build-Prozess

Den Einstiegspunkt einer WPF Applikation markiert ein Objekt der Klasse Application aus dem Namespace *System.Windows*. Application verwendet das Singleton Pattern und ist für Aufgaben zuständig, welche die gesamte Applikation betreffen. Darunter fallen zum Beispiel die Definition der ersten darzustellenden Seite über die Eigenschaft *StartupUri* oder das Verwalten von globalen Objekten. Um die Nachrichtenschleife für die Verarbeitung von Benutzereingaben zu starten, ruft man die Methode *run* des Application Objektes auf. Entweder erfolgt dieser Aufruf in der statischen main Methode, oder aber man deklariert auch das Application Objekt in einer Xaml Datei. (vgl. [SH07], S 42f)

```
<Application x:Class="MinimalXamlProj.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml">
</Application>
```

Listing 4.1: Application Definition File

([Fri07], S 34)

Die Deklaration der Startseite könnte dann zum Beispiel wie in Listing 4.2 aussehen. Wie man an beiden Beispielen sehen kann, werden im Wurzelelement immer zwei Namespaces deklariert. *http://schemas.microsoft.com/winfx/2006/xaml* ist für die bereits bekannte Angabe des Xaml Standardvokabulars zuständig. Im Namespace *http://schemas.microsoft.com/winfx/2006/presentation* befinden sich die meisten der WPF-Typen. Werden andere Typen benötigt, so können beliebige Klassen über ihren Namespace eingebunden werden. Dazu wird der CLR-Namespace der Klasse in Verbindung mit dem Schlüsselwort *clr-namespace* angegeben. Ein Beispiel dazu wäre `xmlns:clr="clr-namespace:System"`. Befindet sich die einzubindende Klasse in einer anderen Assembly, so wird diese zusätzlich angegeben.
`xmlns:clr="clr-namespace:System;assembly=mscorlib"` (vgl. [Fri07], S 48f)

```
<Window x:Class="MinimalXamlProj.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hallo von der WPF" Height="100" Width="230"
    Background="AliceBlue">
  <StackPanel>
      <Button Click="OnClick">Klick mich</Button>
  </StackPanel>
</Window>
```

Listing 4.2: Xaml Minimal

([Fri07], S 34)

Um aus Xaml Dateien eine ausführbare Anwendung zu erstellen, wird das Kommandozeilen-Werkzeug *MSBuild* verwendet. Als Parameter verlangt MSBuild eine **.csproj* Projektdatei. In ihr werden die für den Build relevanten Dateien angegeben. Das Application Definition File wird im Tag *ApplicationDefinition* definiert. In *Page* Tags folgen die Xaml Dateien. Code Dateien werden mit *Compile* angegeben. Handelt es sich um Code-Behind Dateien, so wird die Beziehung zur Xaml Datei mit dem *DependentUpon* Tag abgebildet. Mit den *targets* Einträgen in den letzten beiden Zeilen wird schließlich die Übersetzungslogik für C# und Xaml eingebunden. Verwendet man zum Entwickeln das Visual Studio, so erspart man sich den Konsolenaufruf von MSBuild und darüber hinaus auch noch das Schreiben der Projektdatei. Meist wird man auf diesen Komfort nicht verzichten wollen. Trotzdem kann kann man den Buildvorgang auch zu Fuß gehen. Eine Hilfestellung dafür bietet unter anderem die MSBuild Referenz [MSD07f]. (vgl. [MSD08]) (vgl. [Fri07], S 36f)

```
<Project DefaultTargets="Build"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

  <PropertyGroup>
    <AssemblyName>MinimalXamlProj</AssemblyName>
    <OutputType>winexe</OutputType>
    <OutputPath>.\bin\Debug\</OutputPath>
  </PropertyGroup>

  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="WindowsBase" />
    <Reference Include="PresentationCore" />
    <Reference Include="PresentationFramework" />
  </ItemGroup>

  <ItemGroup>
    <ApplicationDefinition Include="App.xaml" />
    <Page Include="Window1.xaml" />
  </ItemGroup>

  <ItemGroup>
    <Compile Include="App.xaml.cs" />
    <Compile Include="Window1.xaml.cs">
      <DependentUpon>Window1.xaml</DependentUpon>
      <SubType>Code</SubType>
    </Compile>
  </ItemGroup>

  <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
  <Import Project="$(MSBuildBinPath)\Microsoft.WinFX.targets" />

</Project>
```

Listing 4.3: WPF Projektdatei

([Fri07], S 37f)

Als Ergebnis des Build-Vorganges werden etliche Dateien erstellt. Die interessantesten davon sind jene, die auf *.baml* und *.g.cs* enden. Baml Dateien sind die in ein Binärformat umgewandelten Xaml Dateien. Da Xaml Oberflächen in .Net als Ressource betrachtet und zur Laufzeit eingelesen werden, bietet das Binärformat einen Geschwindigkeitsvorteil. Neben den Baml Dateien werden für alle Xaml Dateien auch noch besagte *.g.cs* erstellt. Für die Xaml Datei mit der Applikationsdefinition wird darin die Main-Methode angegeben, welche den Aufruf von *run* enthält. Für die übrigen Xaml Dateien wird die Methode *InitializeComponent* erstellt. Darin wird die Xaml-Ressource geladen.

Weiters erfolgt die Verknüpfung der Komponenten mit eventuell vorhandenen Code-Behind Dateien, sowie die Verknüpfung mit den Event-Handlern. In den folgenden zwei Listings sind Auszüge aus den, für die Xaml Dateien aus den Listings 4.1 und 4.2 generierten, C# Dateien zu sehen. (vgl. [SH07], S 29f) (vgl. [Fri07], S 36f) 4.1

```
namespace MinimalXamlProj {
    public partial class App : System.Windows.Application {

        public void InitializeComponent() {
            this.StartupUri = new System.Uri("Window1.xaml",
                System.UriKind.Relative);
        }

        [System.STAThreadAttribute()]
        public static void Main() {
            MinimalXamlProj.App app = new MinimalXamlProj.App();
            app.InitializeComponent();
            app.Run();
        }
    }
}
```

Listing 4.4: Generated C# Datei für ein Application Definition File - Auszug

```
namespace MinimalXamlProj {
    public partial class Window1 :
          System.Windows.Window,
          System.Windows.Markup.IComponentConnector
    {
```

```
        void OnClick(object sender, RoutedEventArgs e) {
            MessageBox.Show("Funzt");
        }

          [System.Diagnostics.DebuggerNonUserCodeAttribute()]
          public void InitializeComponent() {
              if (_contentLoaded) {
                  return;
              }
              System.Uri resourceLocater = new System.Uri(
                  "/MinimalXamlProj;component/window1.xaml",
                  System.UriKind.Relative);
              System.Windows.Application.LoadComponent(this,
                  resourceLocater);
          }

          void System.Windows.Markup.IComponentConnector.Connect(
                  int connectionId, object target)
          {
              switch (connectionId)
              {
              case 1:
              ((System.Windows.Controls.Button)(target)).Click
                  += new System.Windows.RoutedEventHandler(
                      this.OnClick);
              return;
              }
              this._contentLoaded = true;
          }
      }
  }
```

Listing 4.5: Generated C# Datei für die Xaml-Datei aus Listing 4.2 sowie deren Code-Behind Datei aus Listing 4.6 - Auszug

4.3 Code-Behind

Um die Logik und den damit verbunden Code nicht in die Xaml Dateien einbetten zu müssen, kann man eine Code-Behind Datei mit einer Xaml Datei verknüpfen. In der Xaml Datei wird mit dem Attribut *x:Class* des Wurzelelementes auf den Code-Behind verwiesen. Dabei lautet der Wert von x:Class auf `<Namespace>.<Klassenname>`. Wie man in Listing 4.5 sieht, ist die für eine Xaml Datei generierte C# Klasse partiell. Ebenso ist auch die Code-Behind Klasse partiell. Das Konzept der partiellen Typen erlaubt es, die Beschreibung einer Klasse auf mehrere Dateien aufzuteilen (vgl. [MSD07g]). Möchte man also den Code aus der Xaml Datei in Listing 4.2 in eine Code-Behind Datei auslagern, so muss diese die selbe Klassensignatur wie die generierte C# Datei, siehe Listing 4.5, aufweisen. Weiters ist zu sehen, dass die Klasse vom Typ des Wurzelelementes der Xaml Datei erbt. Es wäre auch möglich diese Ableitung wegzulassen, da sie auch in der generierten Datei angegeben ist und so jedenfalls zum Zug kommt.

Zur Verarbeitung von Events muss eine Methode mit dem Namen erstellt werden, welcher in der Xaml Datei für dieses Ereignis zugewiesen wurde. Im vorliegenden Beispiel ist das die *Click* Eigenschaft sowie die Methode *OnClick*. Listing 4.6 zeigt diese Code-Behind Datei. Als Namenskonvention für Code-Behind Dateien wird der Name der Xaml Datei inklusive der Endung xaml verwendet. Im Fall von Listing 4.6 lautet der Dateiname also auf *window1.xaml.cs*. (vgl. [SH07], S 29f; [MSD07b])

```
namespace MinimalXamlProj
{
  public partial class Window1:System.Windows.Window
  {
    public Window1()
    {
      InitializeComponent();
    }

    private void OnClick(object sender, RoutedEventArgs e)
    {
      MessageBox.Show("Der Button wurde geklickt.");
    }
  }
```

```
}
```

Listing 4.6: Code-Behind zu Listing 4.2 - Auszug

4.4 Inhaltssyntax

Wie bereits oben beschrieben wird der Inhalt eines Elementes, also der Text zwischen öffnendem und schließendem Tag, der Inhaltseigenschaft des Elementes zugewiesen. Dazu muss der Typ dieses Elementes auch diese Inhaltseigenschaft festlegen. Zu diesem Zweck wird im Rahmen der WPF eine Eigenschaft der Klasse, welche den Typ spezifiziert, als diese Inhaltseigenschaft mit der Annotation *ContentProperty* gekennzeichnet. Listing 4.7 zeigt die Klasse *Film*, deren Eigenschaft *Title* als Inhaltseigenschaft angegeben wird. (vgl. [SH07], S 74f) (vgl. [MSD07a])

```
[ContentProperty("Title")]
public class Film
{
    [...]
    public string Title
    {
        get { return _title; }
        set { _title = value; }
    }
    private string _title;
}
```

Listing 4.7: Inhaltseigenschaft C#

(vgl. [MSD07e])

Werden mehrere Elemente als Inhalt angegeben, so kann die implizite Auflistung verwendet werden. Damit ein Xaml-Attribut eine Liste von Elementen enthalten darf, muss eine der folgenden Bedingungen erfüllt sein:

- Implementiert System.Collections.IList oder System.Collections.Generic.IList

- Implementiert System.Collections.IDictionary oder System.Collections.Generic.Dictionary
- Ist abgeleitet von System.Array (siehe dazu auch x:Array) [...]
- Implementiert die von WPF definierte Schnittstelle System.Windows.Markup.IAddChild

([SH07], S 79f)

4.5 Angefügte Eigenschaften

In der WPF werden angefügte Eigenschaften als Dependency Properties implementiert. (vgl. [Fri07], S 496f) Die WPF Architektur ist laut Microsoft stark auf Eigenschaften ausgelegt, wodurch Methoden in vielen Bereichen vermieden werden. Grund für diese Strategie ist die stärkere Betonung von Deklaration, welche durch eben diesen verstärkten Einsatz von Eigenschaften erreicht wird. Da aber nicht so viel Funktionalität durch herkömmliche Eigenschaften erfüllt werden kann, wurden Dependency Properties und das DependencyObject eingeführt. Dependency Properties werden als DependencyObject am WPF-Framework registriert. Vom Framework ausgehend können somit Informationen wie Ereignisse, Data-Binding und eben auch angefügte Eigenschaften an die registrierten Dependency Properties und somit an die mit ihnen verbundenen Objekte weitergegeben werden. (vgl. [MSD07l]) Möchte eine Klasse eine angefügte Eigenschaft anbieten, so muss sie für diese Eigenschaft zwei statische Methoden implementieren.

GetPropertyName erhält als Parameter das Objekt, auf welches das Attached Property angewandt wird. Der Rückgabewert muss dem Typ des Attached Property entsprechen.

SetPropertyName erhält als Parameter das Objekt, auf welches das Attached Property angewandt wird, sowie den Wert des Property.

([SH07], S 95f)

Listing 4.8 zeigt diese beiden Methoden in einem Beispiel.

```
namespace Samples
{
    [ContentProperty("Holidays")]
    public class HolidayCalendar
    {
        [...]
        private static List<Holiday> leisureDays =
            new List<Holiday>();

        public static List<Holiday> SetLeisureDay(
            Holiday holiday,
            bool isLeasureDay)
        {
            if (isLeasureDay)
                leisureDays.Add(holiday);
        }

        public static bool GetLeisureDay( Holiday holiday )
        {
            return leisureDays.Contains(holiday);
        }
    }
}
```

Listing 4.8: Implementierung Attached Property C#

([SH07], S 96)

4.6 Typkonvertierung

Wie bereites erwähnt wurde, ist Xaml streng typisiert. Es wird daher eine Typkonvertierung benötigt, um aus der Zeichenkette der XML Datei den vom Attribut geforderten Typ zu erzeugen. Bei primitiven Datentypen muss dazu nichts weiter angegeben werden. Der Xaml Parser der WPF kann diese Konvertierung eigenständig durchführen. Handelt es sich allerdings um einen komplexen Datentyp, so muss dem Parser mitgeteilt werden, was zu tun ist. In der WPF übernimmt die Klasse *TypeConverter* diese Aufgabe. TypeConverter stellt die zwei Methoden *ConvertFrom* und *CanConvertFrom* für die Konvertierung zur Verfügung. Dabei übernimmt ConvertFrom die eigentliche Umwandlung. Sie gibt entweder das gewünschte Objekt oder Null zurück. CanConvertFrom liefert einen boolschen Wert abhängig davon, ob vom übergebenen Typ aus in den Zieltyp konvertiert werden kann. Für den Typ String sollte also in jedem Fall *true* zurückgegeben werden. Möchte man eigene Klassen mit Xaml und WPF einsetzen, so muss man einen TypeConverter implementieren. Dazu leitet man eine Klasse von TypeConverter ab und überschreibt die beiden genannten Methoden. Eigenschaften wird ein TypeConverter durch Annotation zugewiesen. Die Listings 4.9 illustrieren diesen Sachverhalt. (vgl. [SH07], S 81f) (vgl. [MSD07j])

```
namespace Samples
{
    public class HolidayDateConverter : TypeConverter
    {
        public override bool CanConvertFrom(
            // stellt einen Formatierungskontext bereit
            ITypeDescriptorContext context,
            Type sourceType )
        {
            if (sourceType == typeof(string))
                return true;
            else
                return base.CanConvertFrom(context, sourceType);
        }

        public override object ConvertFrom(
            // stellt einen Formatierungskontext bereit
```

```
            ITypeDescriptorContext context,
            // u.a. Schriftsystem, verwendeter Kalender,
            // Formatierung für Datumsangaben
            CultureInfo culture,
            object value )
        {
            string dateString = (string) value;
            if (dateString == "Easter")
                return new Easter();
            else
            {
                string[] dateParts = dateString.Split('.');
                return new FixedHolidayDate(
                    Convert.ToInt32(dateParts[0]),
                    Convert.ToInt32(dateParts[1]));
            }
        }
    }

    public class Holiday
    {
        [...]
        [TypeConverter( typeof(HolidayDateConverter) )]
        public HolidayDate Date
        {
            get { return date; }
            set { date = value; }
        }
    }
}
```

Listing 4.9: Custom TypeConverter

([SH07], S 85f)

4.7 Markuperweiterungen

Markuperweiterungen liefern bei der Auswertung eine Referenz auf ein Objekt, nicht wie bei der üblichen Typkonvertierung das Objekt selbst. In der WPF werden Markuperweiterungen vor allem für Data-Binding sowie das Zuweisen von Ressourcen und Templates verwendet. Die zugehörigen Markuperweiterungen heißen: Binding, DynamicResource, StaticResource und TemplateBinding. (vgl. [Fri07], S 54) Markuperweiterungen werden als Klassen implementiert, die von System.Windows.MarkupExtension erben. Die gewünschte Objektreferenz wird von der Methode *ProvideValue* zurückgegeben. Als Parameter verlangt ProvideValue ein Objekt vom Typ System.IServiceProvider. Ein ServiceProvider stellt ein Objekt dar, welches Dienste für andere Objekte, in diesem Fall die Markuperweiterung, bereitstellt. (vgl. [MSD07h]) Der Name der Markuperweiterung ist der Klassenname. Es gilt dabei folgende Konvention: Endet der Klassennamen auf Extension, so wird diese Endung in Xaml weggelassen.

4.8 Events

Ereignisse können in der WPF auf verschiedene Arten behandelt werden.

- Direkt in Xaml über x:Code-Abschnitte.
- In Xaml wird ein Ereignis einer Komponente mit einer Routine in der Code-Behind-Datei verknüpft.
- Die Ereignisbehandlung erfolgt nur im Code.
- Innerhalb von Stilen werden so genannte Event-Setter und Event-Trigger verwendet (. . .).

([Fri07], S 70)

Eine C# Klasse deklariert die Ereignisse, welche von ihr ausgelöst werden können als öffentliche Eigenschaften. Genauer gesagt handelt es sich dabei um Delegates. Andere Klassen, welche auf diese Ereignisse reagieren möchten, implementieren Methoden für die Ereignisbehandlung. Um den Event mit dieser Methode in Beziehung zu setzen, erfolgt eine Verknüpfung der Methode mit dem Event Delegate durch den += Operator (vgl. [MSD07d]). Dieses Verhalten ermöglicht eine prägnante Schreibweise in Xaml Dateien. Mittels Attributsyntax wird dem Event ein Methodenname zugewiesen (vgl. [Fri07], S 71). In Listing 4.10 wird die Methode *ClickMich*, welche in der Code Behind Datei deklariert wird, mit dem Click Event der Klasse Button verknüpft.

```
// Xaml
[...]
<Button Click="ClickMich" Width="100">Klick mich</Button>
 <TextBox Name="TbInfo" Width="140">Hallo</TextBox>
[...]

// Code Behind
private void ClickMich(object sender, RoutedEventArgs e)
{
    TbInfo.Text = "Der Button wurde geklickt";
}
```

Listing 4.10: WPF: Eventhandler in Xaml zuweisen

(vgl. [Fri07], S 71f)

Müssen Events in Elementen behandelt werden, die oberhalb oder unterhalb des Elements liegen, an dem der Event auftritt, so kommen *Routed Events* zum Einsatz. Diese können sowohl durch *Bubbling*, als auch durch *Tunneling* propagiert werden. Beim *Bubbling* wandert der Event vom Element, an dem er auftritt, höher bis er beim Wurzelelement angelangt ist. Im Fall des *Tunneling* ist es genau umgekehrt: Der Event wandert vom Wurzelelement abwärts. Beide Varianten können auch gemeinsam verwendet werden. Jener Event, der dabei *tunneled*, wird als *Preview*-Event bezeichnet (vgl. [Fri07], S 76f).

Kapitel 5

Google Web Toolkit

Das Google Web Toolkit, kurz GWT, ist ein Framework zum Erstellen von AJAX Applikationen. GWT stellt das Zielframework, der in dieser Arbeit angestrebten Lösung, dar. Dieses Kapitel soll einen Einblick darin geben, wie jene Benutzerschnittstellen aussehen, welche mit Xaml deklariert werden sollen. Darüber hinaus geht es um einen Überblick über die Möglichkeiten welche GWT bietet.

Abbildung 5.1: Überblick

GWT ermöglicht dem Entwickler JavaScript basierte Browser Anwendungen in Java zu schreiben. Der Fokus von GWT liegt am Browser. Als Backend kann eine beliebige Technologie verwendet werden (vgl. [GWT08g]). Im Mai 2006 wurde der erste Release Candidate der Version 1.0 von GWT veröffentlicht. Seit August 2008 ist GWT in der Version 1.5 erhältlich (vgl. [GWT08r]). Ab dieser Version werden Java 5 Sprachmerkmale, wie Generics, Annotations, For-Each Loops und Autoboxing unterstützt (vgl. [Joh08]). Für das erste Quartal 2009 ist bereits die Version 1.6 angekündigt, die unter anderem leichteres Deployment in war Dateien, neue Widgets sowie Geschwindigkeitsvorteile verspricht (vgl. [GWT08b]). Builds sind für die Betriebssysteme Windows, Mac OS X und Linux erhältlich. Für den Hosted Mode (vgl. 5.3) ist eine 32-bit Java vir-

tuelle Maschine erforderlich. Lizenziert ist GWT seit Version 1.3 unter der Apache 2.0 Lizenz (vgl. [GWT08r]). Das Framework bedient sich jedoch einiger Projekte, welche unter anderen Lizenzen veröffentlicht wurden. Darunter fallen unter anderem Apache Tomcat, Eclipse Standard Widget Toolkit (SWT) und Mozilla Rhino (vgl. [GWT07b]).

5.1 GWT im Überblick

GWT verspricht den Entwickler einer Webapplikation von JavaScript zu befreien, so dass er seine Java Umgebung nie verlassen muss (vgl. [GWT08q]). Als Vokabular stehen dem Entwickler dabei nicht nur selbst geschriebene und GWT eigene Klassen zur Verfügung, sondern auch Teile der Java SE. Google hat ausgewählte Klassen der Pakete *java.lang*, *java.util*, *java.io* und *java.sql* in JavaScript emuliert. Eine Liste der verfügbaren Klassen findet sich auf der GWT Webseite (vgl. [GWT08n]). Man kann also für die Logik einer Webapplikation auf die Java Standardbibliothek zurückgreifen.

Bietet GWT etwas Spezielles nicht an, so kann man über das *JavaScript Native Interface* (JSNI) Methoden auch direkt in JavaScript implementieren. Mit Hilfe von GWTs Java zu JavaScript Kompiler kann man den Java Quelltext der erstellten Oberflächen samt Events und Logik zu JavaScript kompilieren. Um die Applikation zu veröffentlichen, müssen diese JavaScript Dateien schlicht auf einen Webserver gestellt werden. Abbildung 5.2 zeigt die drei eben erwähnten Komponenten als Kern von GWT.

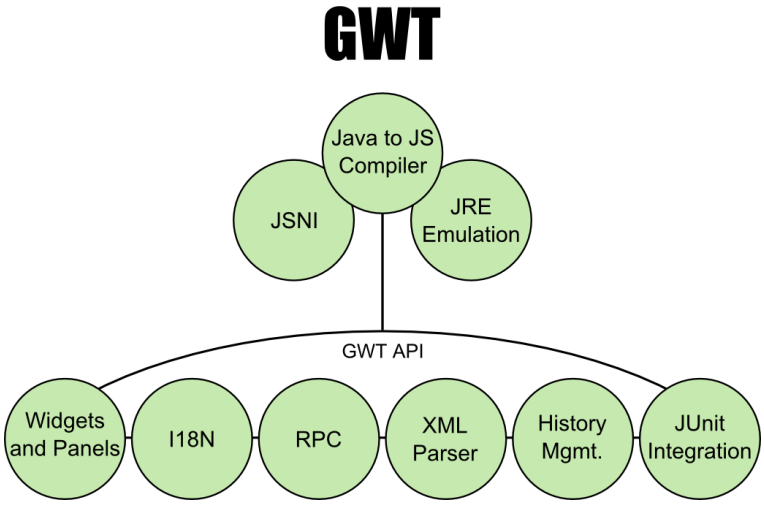

Abbildung 5.2: Überblick GWT ([HT07], S 6)

Weiters zeigt Abbildung 5.2 Module (vgl. 5.2), welche man in einer GWT Applikation nutzen kann. Um etwa die Oberfläche der Applikation zu beschreiben, stellt GWT Widgets bereit, welche in Panels angeordnet werden können (vgl. 5.3). Muss man trotz des Einsatzes von Widgets direkt auf die DOM Struktur zugreifen, so kann man die DOM API von GWT verwenden (vgl. [WJ08]). Da AJAX Applikationen nicht in statischen Seiten organisiert sind, führt der Back Button des Browsers meist nicht zu jener Seite, die sich der Benutzer erwartet. Über das GWT History Management kann die Applikation zu passenden Zeitpunkten einen Punkt in die Historie des Browsers einfügen. Vom Drücken des Back Buttons kann sich ein Objekt durch das Registrieren eines HistoryListeners informieren lassen. Es obliegt dann der Applikation den richtigen Zustand herzustellen.

GWT unterstützt auch Properties Dateien mit Internationalisierungsdaten. Diese können mit Hilfe der Interfaces *Constants* und *Messages* auf Klassen umgelegt werden, sodass pro definierter Eigenschaft aus den Properties Dateien eine Getter Methode vorhanden ist (vgl. [HT07], S 14f). Möchte die GWT Applikation im Browser mit ihrem Server kommunizieren, kann sie GWT-RPC verwenden. Es handelt sich dabei um Remote Procedure Calls die Java Objekte zwischen Browser und Server austauschen (vgl. 5.4). Aber auch andere Formen von Serveraufrufen sind möglich, weshalb GWT die Arbeit mit XML (vgl. [GWT08v]) und JSON Daten (vgl. [GWT08o]) unterstützt. Und schließlich sind auch Unit Tests mit dabei. Somit können jene Methoden, welche die Logik der Browser Applikation enthalten mit JUnit getestet werden (vgl. [Glo07]).

5.2 Projektaufbau und Build-Prozess

5.2.1 Verzeichnisstruktur

GWT Applikationen verwenden die gleiche Verzeichnisstruktur, wie Java Pakete. Durch diese Struktur wird ein Projekt in Module, sowie Code für den Browser und solchen für den Server unterteilt. Tabelle 5.1 listet die Verzeichnisse beispielhaft für den Paketnamen *com.example.cal.Calendar* auf. Es gibt also einen definierten Bereich für den clientseitigen Code. Das dient zum einen der Übersichtlichkeit, zum anderen werden standardmäßig auch nur jene Java Dateien kompiliert, welche sich im Client Verzeichnis befinden. Alle generierten JavaScript Dateien werden im Verzeichnis *www* abgelegt. Dort landen beim Kompilieren auch Ressourcen wie Bilder, die für die Entwicklung im Verzeichnis *public* abgelegt werden. Für den Hostet Mode kommt der Servlet-Container *Apache Tomcat*[1] zum Einsatz. Die dafür notwendigen Dateien befinden sich im Verzeichnis *tomcat*.

Directory	**Purpose**
src/com/example/cal/	The project root package contains module XML files
src/com/example/cal/client/	Client-side source files and subpackages
src/com/example/cal/server/	Server-side code and subpackages
src/com/example/cal/public/	Static resources that can be served publicly. Files in the public directory are copied into the same directory as the GWT compiler output.
www/com.example.cal.Calendar/	Directory where the GWT compiler writes output and files on the public path are copied.
test/	Directory for unit test class source files
tomcat/	Directory created for hosted mode's internal tomcat server. This directory is created automatically when hosted mode is run sucessfully for the first time.

Tabelle 5.1: Ordnerstruktur eines GWT-Projekts ([GWT08h])

[1] tomcat.apache.org/

5.2.2 Wichtige Dateien

GWT Applikationen werden in Modulen (vgl. 5.2.4) organisiert. Für jedes Modul gibt es eine Konfigurationsdatei im XML Format, deren Name auf *.gwt.xml* endet. Sie befindet sich im dem Modul übergeordneten Verzeichnis. Im genannten Beispiel lautet der Pfad also *src/com/example/cal/Calendar.gwt.xml*. Ein Modul kann einen Einstiegspunkt definieren. Dazu gibt man in der Modulkonfiguration eine Klasse an, welche das Interface *EntryPoint* implementiert. Typischerweise wird diese Datei, bei obigem Beispiel bleibend *src/com/example/cal/client/Calendar.java* heißen. Im Verzeichnis *public* wird neben den bereits erwähnten Ressourcen auch eine HTML Datei hinterlegt. Sie bildet den Startpunkt der Applikation. In einem Script-Tag wird ein JavaScript mit der Endung *.nocache.js* eingebunden. Es ist dafür verantwortlich alle notwendigen GWT JavaScripte zuladen. Damit startet es die eigentliche JavaScript Applikation. Auf jene Dateien, welche beim Kompilieren erzeugt werden, und die wie bereits erwähnt, im Ordner *public* landen, wird in Abschnitt 5.2.6 eingegangen. (vgl. [GWT08h]; [HT07], S 30f,72f)

5.2.3 Tools

Das Erstellen der beschriebenen Verzeichnisse und Dateien muss der Entwickler nicht von Hand ausführen. Google stellt einige recht nützliche Kommandozeilen Tools bereit, welche den initialen Aufwand verringern.

projectCreator Dieses Tool ist vor allem dann sinnvoll, wenn Eclipse als IDE eingesetzt wird. Wird die Option *-eclipse* angegeben, so erstellt das Tool eine Eclipse Projekt-Datei, eine Classpath-Datei sowie *src* und *test* Ordner. Mit der Option *-ant* wird ein Ant Build-Skript erzeugt.

applicationCreator Mit diesem Skript wird die bereits erwähnte Verzeichnisstruktur erstellt. Darüberhinaus wird diese auch gleich mit Beispieldateien befüllt. Somit entsteht bereits eine lauffähige Applikation. Darüber hinaus werden die beiden Kommandozeilen Skripte *ModulName-shell* und *ModulName-compile* er-

stellt. Diese dienen zum Starten des Hosted Mode bzw. zum Kompilieren der Applikation.

i18nCreator Hiermit werden die für Internationalisierung nötigen Dateien erstellt.

junitCreator Ein JUnit Test sowie Skripte für dessen Start in Hosted Mode und Web Mode werden mit diesem Tool erstellt. Die konkreten Testfälle können dann nach dem somit vorgegeben Schema erstellt werden.

(vgl. [GWT08u]; [HT07], S 44f)

5.2.4 Module

GWT Applikationen bieten die Möglichkeit der Modularisierung. Somit wird die Wiederverwendbarkeit des Quelltextes erleichtert (vgl. [HT07], S 318). Module sind XML Dateien, welche die Konfiguration für ein GWT Projekt beinhalten (vgl. [GWT08p]). Um ein GWT Projekt kompilieren zu können, muss es über eine Modul-Datei mit der Endung **.gwt.xml* verfügen. Abgeschlossene Teile einer großen Applikation können mit eigenen Modulen ausgestattet werden, wodurch sie kleine eigenständige GWT Projekte bilden. Es ist dabei von Vorteil, wenn man die empfohlene Projektstruktur einhält, und somit das Projekt in Java-Paketen organisiert. Die Modul-Datei sollte sich, wie in Abschnitt 5.2.2 beschrieben, im Wurzelpaket befinden. Alle Quelltexte liegen also in Pakten unterhalb der Modul-Datei. Hält man sich an diese Vorgaben, kann die Modul-Datei recht kurz ausfallen, wie Listing 5.1 zeigt.

Für das Kompilieren der Applikation muss neben der Modulkonfiguration auch der Classpath richtig gesetzt sein. Es müssen neben den Class Dateien auch die Quelltexte über den Classpath auffindbar sein, da sie der GWT-Kompiler benötigt um die JavaScripte zu erzeugen (vgl. [HT07], S 325). Um eine Applikation, welche aus mehreren Modulen besteht, zusammenzufügen, erstellt man ein übergeordnetes Modul. In dieses bindet man alle für die Applikation benötigten Module ein. Es ist auch möglich andere Module direkt als JavaScript einzubinden. Dadurch enstehen allerdings Nachteile. Zum einen dauert das Laden länger, da zwei Applikationen geladen werden müssen, zum anderen werden GWT Bibliotheken mehrfach eingebunden (vgl. [GWT08f]).

```
<module>
    <inherits name="com.google.gwt.user.User" />
    <entry-point class="com.example.cal.client.Calendar" />
 </module>
```

Listing 5.1: minimale Modul Datei

Wie Listing 5.1 zeigt, werden andere Module mit dem Tag *inherit* eingebunden. Genauer gesagt wird von ihnen geerbt. Als Name für das Modul dient der sogenannte *logical name*, welcher dem Javapaket plus dem Namen des Moduls ohne Dateierweiterung entspricht. In den meisten Fällen wird das Modul *com.google.gwt.user.User* eingebunden werden, weil dieses Modul seinerseits die meisten Module der GWT Bibliothek einbindet.

Weiters kann ein Modul Einstiegspunkte definieren. Es können keiner bis mehrere dieser Einstiegspunkte pro Modul vorhanden sein. In einer Applikation sollte es aber zumindest einen geben, da sonst die Applikation nicht starten würde. Alle Klassen, welche im Modul als EntryPoint definiert werden, müssen das Interface EntryPoint implementieren und einen Standard-Konstruktor aufweisen. Beim Laden des Moduls werden diese Klassen instantiiert und ihre *EntryPoint.onModuleLoad()* Methoden ausgeführt. (vgl. [HT07], S 321f; [GWT08p])

Als Standard-Paket für die zu kompilierenden Quelltexte wird das *client* Paket verwendet, welches sich direkt unterhalb des Moduls befindet. Weicht man von dieser Struktur ab, so kann mit dem Tag `<source path="path"/>` ein anderes Paket angegeben. Der Wert des Parameters *path* ist ein Paketname. Werden zusätzliche Module eingebunden, so werden alle source paths zusammen gefügt, sodass das einbindende Modul Zugriff auf die Quelltexte der eingebundenen Module hat. Nur jene Quelltexte, welche sich im source path befinden, werden zu JavaScript kompiliert. Ebenso kann der Standardwert des public path mit dem Tag `<pulic path="path"/>` geändert werden. Voreingestellt ist hier das Paket *public* unterhalb des Moduls. Auch die public paths der eingebundenen Module werden mit dem Pfad des Einbindenden kombiniert. (vgl. [GWT08p]; [HT07], S 325)

Möchte man nicht alle Dateien aus den source bzw. public Pakte kompilieren bzw. kopieren, kann man Filter angeben. Diese werden analog zum Ant Tag *FileSet* angegeben

(vgl. [HT07], S 326). Es ist aber zu beachten, dass nicht die vollen Möglichkeiten des Ant Äquivalents genutzt werden können. Eine Liste mit den unterstützten Optionen findet sich unter [GWT08j].

Muss man zusätzliche JavaScripte oder CSS Dateien referenzieren, so kann man diese ebenfalls im Modul angeben. Dazu werden die Tags `<script src="js-url"/>` sowie `<stylesheet src="css-url"/>` verwendet. JavaScripte die auf diese Weise referenziert werden, erscheinen in der Host-HTML Seite, als wären sie direkt dort über das HTML script-Tag eingebunden. Weiters werden diese Scripte geladen, bevor *onModuleLoad()* aufgerufen wird. (vgl. [GWT08s])

Kommuniziert eine Applikation mit einem Java Servlet, so ist für das Debuggen dieser Anwendung von Vorteil, wenn der Server verfügbar ist. Im Hosted Mode (vgl. 5.2.5) wird die Applikation in einem Apache Tomcat gestartet. Mit Hilfe des Tags *servlet* kann der Serverteil der Applikation bei jedem Start des Hosted Modes mit in den Tomcat deployed werden. Somit ist sichergestellt, dass bei jedem Start des Hosted Modes die Kommunikation mit dem Server funktioniert. (vgl. [HT07], S 326f)

5.2.5 Hosted Mode

Beim Entwickeln einer Applikation, ist es wichtig auf einfachem Weg das Resultat der Arbeit ansehen und debuggen zu können. GWTs Hostet Mode soll diese Anforderungen befriedigen. Im Hosted Mode wird die Applikation nicht als JavaScript in einem Browser geladen, sondern läuft als Java Bytecode in einem Managed Environment. Dafür wird ein Embedded Browser zum Anzeigen der Browser-Applikation, sowie die Tomcat Servlet Engine zum Ausführen des serverseitigen Codes verwendet. Weil man somit die Java-Umgebung nicht verlässt, ist es möglich sowohl den Client- als auch den Severcode aus der IDE heraus zu debuggen.

Gestartet wird der Hosted Mode über das Kommandozeilenskript *modulName-shell*, welches beim Anlegen des Projektes vom *applicationCreator* (5.2.3) erstellt wurde. Das Skript führt dabei die Klasse *com.google.gwt.dev.GWTShell* aus, welche sich im Paket *gwt-dev-OS.jar* befindet. *OS* steht dabei für das verwendete Betriebssystem. Unter Linux heißt dieses Paket also *gwt-dev-linux.jar*. Im Skript kann man die Parameter

für die JVM, wie zum Beispiel den verfügbaren Speicher, anpassen. Weiters stehen einige Parameter für *GWTShell* zur Verfügung, welche das Verhalten des Hosted Mode verändern. So ist es möglich den Port des Tomcat Servers zu ändern. Auch kann man verhindern, das Tomcat überhaupt startet. Das ermöglicht den Einsatz eines anderen Web oder Application Servers. Eine vollständige Liste der möglichen Parameter bietet [GWT08k].

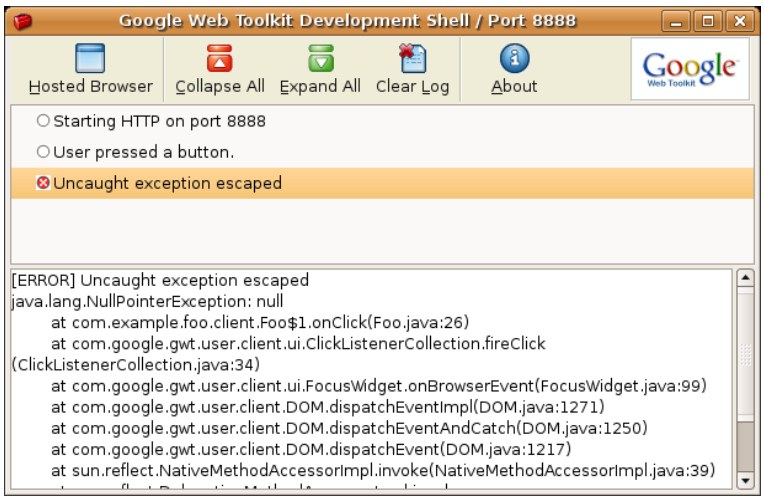

Abbildung 5.3: GWT DevelopmentShell ([GWT08l])

Beim Ausführen des Hosted Mode wird zuerst die Developement Shell (Abbildung 5.3) geöffnet. In diesem Fenster werden Logging Informationen und Exceptions angezeigt. Dann wird der eingebettete Tomcat gestartet, um die Host HTML Seite und den severseitigen Code auszuführen. Schließlich wird der Hosted Browser (Abbildung 5.4) ausgeführt, der die GWT Applikation darstellt. Der Hosted Browser bietet neben den üblichen Schaltflächen eines Browsers wie Back, Forward und Refresh auch Compile/-Browse. Damit kann die gerade laufende Applikation per Knopfdruck kompiliert und in einem richtigen Browser geöffnet werden, was auch als Web Mode bezeichnet wird. Ändert man den Quelltext der Applikation genügt ein Klick auf die Schaltfläche *Refresh* um die neue Version in den Hosted Browser zu laden. Diese Vorgehensweise ist um einiges schneller als den Hosted Mode bei jeder Änderung neu zu starten. Möchte man eigene Debug Informationen in der Development Shell ausgeben, so kann man das mit dem Aufruf *GWT.log()* tun. (vgl. [HT07], S 86f; [GWT08l])

Abbildung 5.4: GWT Hosted Browser ([GWT08l])

5.2.6 Deployment

Um die GWT Applikation in den Produktiv-Betrieb überzuführen, verlässt man den Hosted Mode und geht über zu JavaScript. Als Allererstes muss die Applikation also kompiliert werden. Dazu gibt es drei Möglichkeiten:

- Nutzen des Button Compile/Browse im Hosted Browser.
- Aufruf des Kommandozeilen-Skript *ProjektName-compile*, welches beim Erstellen des Projektes mit *applicationCreator* generiert wurde.
- Ausführen der Java Klasse *com.google.gwt.dev.GWTCompiler*.

(vgl. [GWT08g])

Die Klasse *com.google.gwt.dev.GWTCompiler* bietet einige Parameter, mit denen der Erstellungsprozess beeinflusst werden kann. Unter anderem kann die Genauigkeit des Loggings und das Ausgabe-Verzeichnis angegeben werden. Das voreingestellte Verzeichnis für die Ausgabe ist *www* direkt unterhalb der Modul-Datei. (vgl. [GWT08c])

Als Einstiegspunkt für den Java-JavaScript Kompiler dient die Klasse *com.google.gwt.dev.GWTCompiler*. Kompiliert werden nur Quelltexte, welche sich im source path befinden. Begonnen wird der Kompilierungsprozess bei jenen Klassen, welche als entry point deklariert wurden. Es werden anschließend nur jene Klassen und Methoden

kompiliert, welche auch tatsächlich verwendet werden. Das ermöglicht es, große Bibliotheken zu verwenden, und trotzdem die Größe der JavaScript Dateien gering zu halten, da meist nicht jedes einzelne Feature einer Bibliothek eingesetzt wird. Das vom Kompiler erzeugte JavaScript ist sehr prägnant. Es enthält statt der ursprünglichen Klassen- und Variablennamen nur noch einzelne Buchstaben. Dieser Modus wird als *obfuscate* bezeichnet, und ermöglicht kleine Dateigrößen. Außerdem erschwert er es Dritten den Quelltext der Applikation zu kopieren. Während der Entwicklung, kann es nötig sein, den JavaScript Code zu evaluieren. Dafür gibt es die Modi *pretty* und *detailed*, welche lesbareres JavaScript erzeugen. (vgl. [HT07], S 6f)

Als Ergebnis einer kompilierten GWT Applikation, landet eine Vielzahl von Dateien im Ausgabeverzeichnis. Warum das soviele sind, bzw. welche Aufgabe diese Dateien haben, soll im Folgenden erläutert werden. Zuvor sind jedoch einige erklärende Worte vonnöten.

Deferred Binding

GWT hat es sich als Aufgabe gesetzt zu allen gängigen Browsern kompatibel zu sein. Da diese Browser nicht gänzlich standardkonform sind und ihre Eigenheiten aufweisen, erstellt GWT für jeden unterstützten Browser eine eigene JavaScript Version der Applikation. Verwendet man Internationalisierung, wird sogar pro Sprache und Browser ein eigenes JavaScript generiert. Es wurde aber nicht die komplette GWT Klassenbibliothek für alle unterstützten Browser neu geschrieben. Vielmehr ist die Funktionalität von GWT durch Interfaces definiert. Von jedem Interface gibt es dann eine oder mehrere Implementierungen, je nachdem ob sich die Browser bei dieser Funktionalität unterscheiden oder nicht. Für die Entscheidung welche Klasse nun für die jeweilige Version des JavaScripts verwendet wird, kommt das sogenannte *Deferred Binding* zum Einsatz. Deferred Binding kann auf zwei Arten verwendet werden. Erstens kann mit *Replacement* die Implementierung eines Interface durch eine andere ersetzt werden. Dieses Verfahren kommt zur Auswahl von browserspezifischen Implementierungen zum Einsatz. Zweitens kann mit *Generatoren* eine Implementierung für ein Interface erstellt werden. Eine Anwendung davon ist zum Beispiel die Generation von Klassen mit Gettern und Settern für Properties Dateien bei der Internationalisierung. Um Deferred

Binding zu etablieren, muss es in der Modulkonfiguration (5.2.4) deklariert werden. In den meisten Fällen ist ein Anwendungsprogrammierer jedoch nicht damit konfrontiert. Daher wird nicht genauer auf dieses Thema eingegangen. Deferred Binding wird jedenfalls zur Kompilierzeit ausgewertet. Im Produktivbetrieb auf einem Webserver sind fertige, sozusagen, statische JavaScripte vorhanden. (vgl. [GWT08e]; [Nee08])

Ausgabeverzeichnis

Die folgende Liste erklärt jene Dateien, welche sich im *www* Verzeichnis eines GWT Projekts befinden. Die Reihenfolge orientiert sich daran, wie eine GWT Applikation geladen wird.

Host HTML Seite Die Host Seite ist jene HTML Datei, die das Skript **.nocache.js* einbindet. Diese HTML Seite bildet den Startpunkt für die Applikation, da alle GWT JavaScripte durch sie geladen werden.

***.nocache.js** Dieses JavaScript wählt aus den am Server verfügbaren Skripten (**.cache.html*) jenes aus, das dem Browser und der Sprache des Benutzers entspricht und lädt es herunter. Der Dateiname beginnt mit dem logischen Modulnamen. Im Beispiel vom Beginn dieses Abschnitts wäre der Name des Skripts also *com.example.cal.Calendar.nocache.js*. Der Name nocache stammt daher, dass diese Datei nicht vom Browser gecached werden sollte. Werden Änderungen an der Applikation durchgeführt, liegen neue JavaScript Dateien auf dem Server. Diese können von einer veralteten nocache Datei nicht gefunden werden.

***.cache.html** Diese Dateien enthalten die eigentliche Applikation. Es gibt mehrere Dateien mit der Endung cache.html, da, wie bereits erwähnt, pro Browser und Sprache ein eigenes JavaScript erstellt wird. Die JavaScripte sind in HTML eingebettet, da sie so komprimiert werden können.

history.html Diese Datei wird für die Verwaltung der Browser-History benötigt. Sollte diese Funktionalität nicht verwendet werden, so kann man diese Datei auch löschen.

hosted.html Im Web Mode wird diese Datei nicht benötigt, da sie nur für den Hosted Mode relevant ist.

(vgl. [GWT07a]; [HT07], S 543)

Um die Applikation schließlich veröffentlichen zu können, müssen alle erforderlichen HTML, JavaScript, CSS und Bild Dateien auf einen Server gestellt werden. Da GWT nicht notwendigerweise mit Java als Servertechnologie betrieben werden muss, muss auch der Server kein Java Applikation Server sein (vgl. [GWT08g]). Möchte man allerdings GWT-RPC verwenden, muss zumindest der Serverteil der Applikation in einem Applikation Server laufen. Man kann in diesem Fall auch die HTML und JavaScript Dateien des Frontends mit in die War Datei packen. Das ist aber nicht in jedem Fall ratsam. Oft arbeitet ein Webserver bei der Verarbeitung statischer Daten effizienter. Auch ist zu beachten, das in eine War Datei nur jene GWT Bibliotheken inkludiert werden, welche auch tatsächlich am Server erforderlich sind, sprich *gwt-servlet.jar* wenn GWT-RPC verwendet wird. Keinesfalls sollten die clientseitigen Bibliotheken *gwt-user.jar* und *gwt-dev-*.jar* mit in das War verpackt werden, da sie Tomcat-Bibliotheken enthalten und daher Probleme verursachen können. (vgl. [HT07], S 551)

5.3 Grafische Oberflächen schreiben

5.3.1 Widgets und Panels

Für das Erstellen von grafischen Benutzerschnittstellen verwendet man in GWT Widgets und Panels. Panels werden benutzt um Widgets anzuordnen. Sie sind also Layout-Komponenten und ermöglichen zum Beispiel das horizontale (Horizontal Panel) oder vertikale (Vertical Panel) Positionieren von Widgets. Aber auch komplexere Strukturen sind möglich, wie etwa mit Hilfe von DockPanel, Grid oder TabPanel. Widgets stellen Elemente dar, die für Inhalte sowie für die Interaktion mit dem Benutzer verwendet werden. Zum einen sind das Elemente, die man von HTML her kennt wie TextBox, Button oder Checkbox, zum anderen hat Google aber auch etwas aufwändigere Elemente wie zum Beispiel das RichTextArea inkludiert, welches einen Texteditor für den Browser bietet. (vgl. [HT07], S 12) Eine Übersicht über alle Widgets und Panels, welche GWT von Haus aus mitbringt, bietet der GWT-Showcase [2] . Hier kann man sich einen ersten Eindruck über die Möglichkeiten von GWT verschaffen. Darüber hinaus sind zu jedem Beispiel der Quellcode und die CSS Datei angegeben.

Wie man in den Abbildungen 5.5 und 5.6 sehen kann, erben alle grafischen Elemente von der Klasse *UIObject*. UIObject befindet sich wie alle anderen Widget und Panel Klassen auch im Paket *com.google.gwt.user.client.ui*. Von dieser Klasse werden Basisfunktionalitäten wie Lesen und Setzen von Position, Größe und Sichtbarkeit des Elements sowie Methoden zum Anfügen und Entfernen von CSS Stilnamen geerbt. Wie in Java üblich, wird auf die einzelnen Eigenschaften über Getter und Setter Methoden zugegriffen. Dabei verlangen die meisten Methoden Parameter vom Typ String. Am Beispiel der Methode *setHeight* ist dies ersichtlich (vgl. Listing 5.2). Durch den Einsatz von String können von HTML her gewohnte Angaben wie *"100px"* verwendet werden. Weiters ist am Beispiel von setHeight ersichtlich, dass beim Setzen der Eigenschaft Höhe auf den DOM Baum zugegriffen wird. Jedes Widget ist nicht nur als Java Objekt vorhanden, sondern muss auch im DOM Baum des Browsers aufscheinen. Die Methode *getElement*, die ebenfalls von UIObjekt angeboten wird, liefert die nötige Referenz auf das dem jeweiligen Java Objekt entsprechende DOM Objekt (vgl. [HT07], S 115f). GWT

[2] http://gwt.google.com/samples/Showcase/Showcase.html

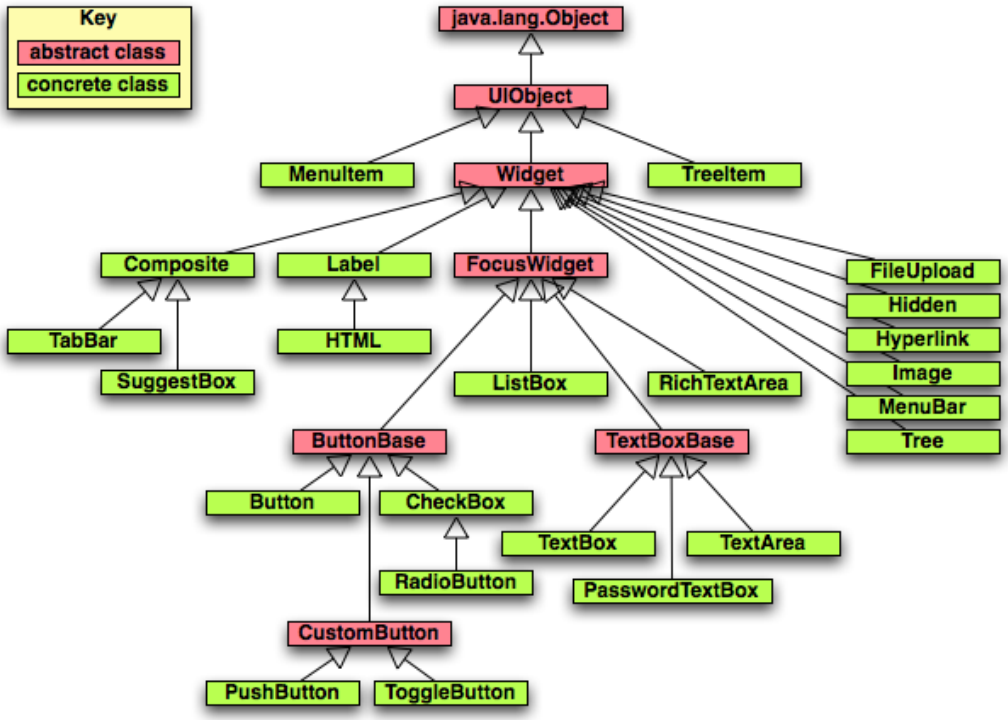

Abbildung 5.5: GWT Widgets - Auszug ([Vol07])

verfügt über Klassen zur Manipulation von DOM. Auch hier muss also in den meisten Fällen kein JavaScript geschrieben werden. Enthalten sind diese Klassen im Paket *com.google.gwt.dom.client*. Alle Klassen im DOM Paket erben von JavaScriptObject, welches native JavaScript Objekte kapselt. (vgl. [WJ08])

```
public void setHeight(String height) {
    assert extractLengthValue(height.trim().toLowerCase())
        >= 0 : "CSS heights should not be negative";
    DOM.setStyleAttribute(getElement(), "height", height);
}
```

Listing 5.2: UIObject.setHeigth

In der Klassenhierarchie folgt nach UIObject die Klasse *Widget*. Bis auf die Ausnahme von TreeItem und den MenuItems erben alle grafischen Elemente, einschließlich der Panels, von Widget. Widget ist dabei für jene Funktionalität verantwortlich, die ein Widget zur im DOM Baum manipulierbaren Einheit macht. Diese Klasse bietet Methoden zum Hinzufügen und Entfernen von Kind-Widgets, zum Anfügen an das bzw. Entfernen vom Eltern Element, sowie zum Ein- bzw. Aushängen aus dem DOM Baum

inklusive dem Feststellen ob das Widget gerade eingehängt ist. Weiters gibt es Methoden, welche das Verhalten beim Einhängen und Laden festlegen. Schließlich wird mit *onBrowserEvent* die Behandlung von Ereignissen ermöglicht.

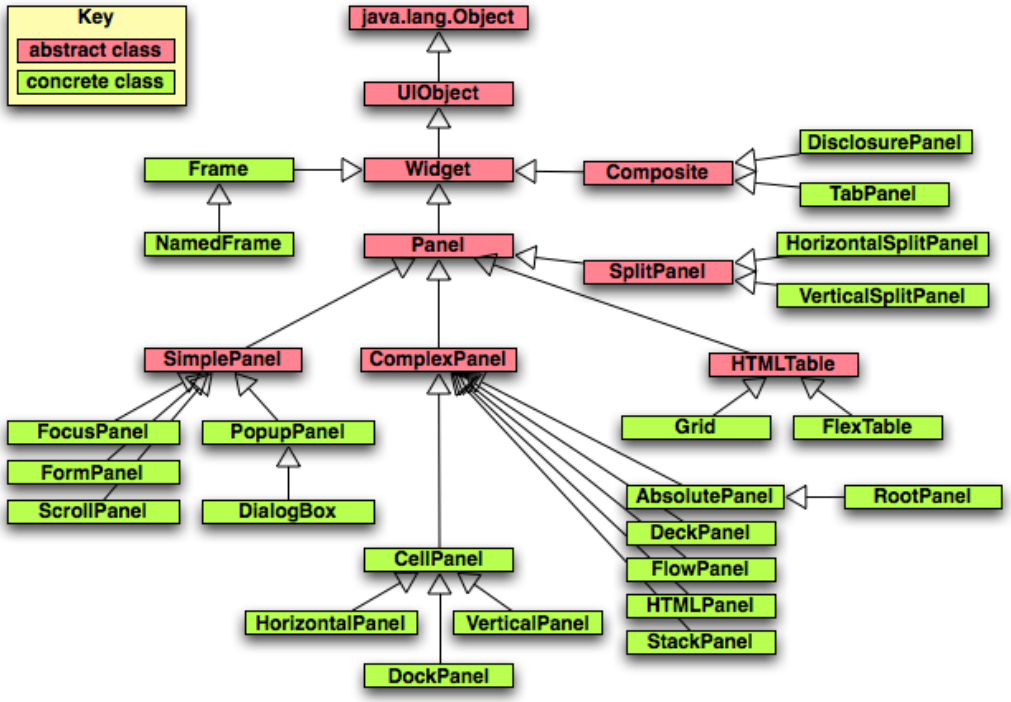

Abbildung 5.6: GWT Container - Auszug ([Vol07])

Blickt man nun weiter nach unten in der Vererbungshierarchie, wird ersichtlich, das gemeinsame Funktionalitäten von Widgets in Basisklassen zusammengeführt sind. *ButtonBase* und *TextBoxBase* sind zum Beispiel solche Basisklassen für alle Buttons und TextBoxen. Kommen wir nun schließlich zu einem konkreten Widget. Es wurde zuvor erwähnt, dass zu jedem Java Widget ein DOM Objekt existiert. Erstellt wird dieses DOM Objekt im Konstruktor des Widgets. Exemplarisch zeigt Listing 5.3 den Konstruktor der Klasse Button. Ebenfalls ist ersichtlich, dass dem Widget ein Stylename für das Standardaussehen zugewiesen wird. (vgl. [HT07], S 113f)

```
public Button() {
    super(Document.get().createButtonElement());
    adjustType(getElement());
    setStyleName("gwt-Button");
}
```

Listing 5.3: Konstruktor Button

Damit die Elemente einer Oberfläche angezeigt werden, muss der Objektgraph aus Panels und Widgets dem *RootPanel* hinzugefügt werden. RootPanels werden nicht erzeugt sondern über die statische Methode *RootPanel.get()* referenziert (vgl. [GWT08a]). Listing 5.4 zeigt eine Auswahlliste mit mehreren RadioButtons. Die grafischen Elemente werden in der Methode *onModuleLoad()* instantiiert. Alle RadioButtons sowie die beiden Überschriften werden in ein *VerticalPanel* eingehängt, welches für die vertikale Anordnung sorgt. Dieses VerticalPanel wird in das RootPanel eingehängt. Der Quelltext aus Listing 5.4 ist an das Radio Button Beispiel aus dem GWT-Showcase angelehnt[3]. Die aus dem Beispiel resultierende Oberfläche im Hosted Mode zeigt Abbildung 5.7.

```
public class RadioButtonExample implements EntryPoint {
    public void onModuleLoad() {
        // Create a vertical panel to align the radio buttons
        VerticalPanel vPanel = new VerticalPanel();
        vPanel.add(new HTML(
            "<b>Select your favorite color:</b>"));

        // Add some radio buttons to a group called 'color'
        String[] colors = {"blue", "red", "yellow", "green"};
        for (int i = 0; i < colors.length; i++) {
            String color = colors[i];
            RadioButton radioButton = new RadioButton(
                "color", color);
            if (i == 2) {
              radioButton.setEnabled(false);
            }
            vPanel.add(radioButton);
        }

        // Add a new header to select your favorite sport
        vPanel.add(new HTML("<br>" +
            "<b>Select your favorite sport:</b>"));

        // Add some radio buttons to a group called 'sport'
        String[] sports = {"Baseball", "Basketball", "Football",
            "Hockey", "Soccer", "Water Polo"};
```

[3] http://gwt.google.com/samples/Showcase/Showcase.html#CwRadioButton

```
            for (int i = 0; i < sports.length; i++) {
                String sport = sports[i];
                RadioButton radioButton = new RadioButton(
                    "sport", sport);
                if (i == 2) {
                  radioButton.setChecked(true);
                }
                vPanel.add(radioButton);
            }
            RootPanel.get().add(vPanel);
        }
    }
```

Listing 5.4: GUI Beispiel Radio Button

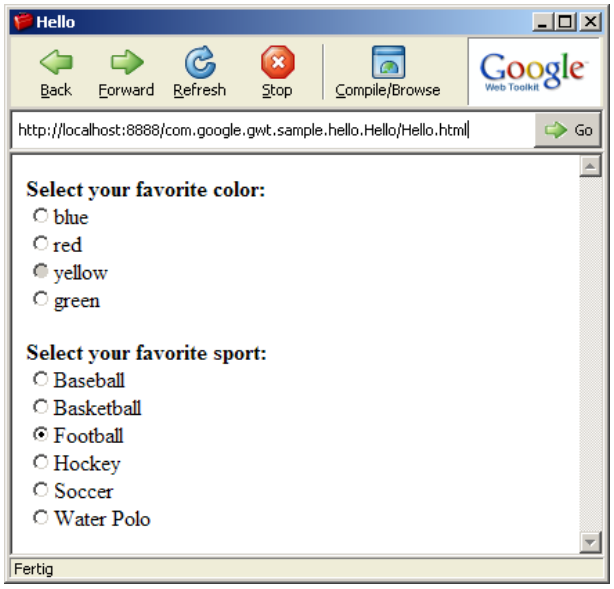

Abbildung 5.7: Beispiel RadioButton - Ergebnis aus Listing 5.4

5.3.2 Eigene Widgets

Wenn die Widgets, welche GWT mitbringt für einen Anwendungsfall nicht ausreichen, oder wenn man andere Funktionalitäten benötigt als die GWT-Widgets bieten, muss man selbst Widgets schreiben, oder Bibliotheken von Drittanbietern verwenden. GWT bietet drei Möglichkeiten um eigene Widgets zu erstellen.

- Direkte Manipulation des Dom Baums.
- Erweitern eines bestehenden Widgets.
- Kombination mehrere Widgets.

Um Widgets direkt aus Dom-Objekten zu erstellen, kann man entweder JavaScript über das JSNI aufrufen, oder GWTs Dom API nutzen. Dabei erstellt man also mit JavaScript HTML-Elemente. Auch wenn man die GWT DOM API verwendet ist das der Fall, da diese API ja nur einen Wrapper für JavaScript bildet. Um das neue Widget zu den übrigen GWT Widgets kompatibel zu halten, soll es direkt oder indirekt von Widget erben. Die in GWT enthaltenen Widgets wurden ebenfalls auf diese Weise erstellt. (vgl. [HT07], S 142f)

Ist bereits ein Widget vorhanden, welches fast den Anforderungen entspricht, kann man dieses auch erweitern. Durch das Implementieren von Interfaces oder das Hinzufügen oder Überschreiben von Methoden kann die fehlende Funktionalität ergänzt werden (vgl. [HT07], S 146f).

Einen ähnlichen Ansatz verfolgt die Klasse *Composite*. Composite dient dabei als Wrapper für ein bestehendes Widget, welches mit *initWidget()* gesetzt wird. Die neue Klasse verhält sich beim Einhängen in ein Panel wie dieses eingeschlossene Widget. Der Vorteil liegt darin, einen ganzen Baum an Widgets hinter einem Widget zu verbergen. Das sozusagen Oberste dieser Widgets wird meist ein Panel sein, dem dann weitere Widgets hinzugefügt werden, um ein komplexes Ganzes zu bilden. Die Methoden der einzelnen enthaltenen Widgets können von außerhalb des Composites nicht aufgerufen werden. Als Kommunikationsmittel innerhalb des Composites können Events verwendet werden (vgl. [HT07], S 247f; [GWT08d]; [GWT08a]).

5.3.3 Events

Standardkonforme Browser sollten zwei Modelle für die Ausbreitung von Events unterstützen. Bei *Event capture* wandert der Event von der Spitze des Dom Baumes nach unten, bei *Event bubbling* tritt er zuerst am betroffenen Element auf, und wandert

dann nach oben (vgl. [W3C00]). Da nicht alle Browser tatsächlich beide Möglichkeiten implementieren, übernimmt GWT die Verbreitung der Events. Dabei implementiert es selbst das Modell des Event capturing. GWT greift auf Dokument-Ebene die Events auf und lässt sie dann in der Widget-Hierarchie nach unten wandern bis sie behandelt werden. Die Eventbehandlung erfolgt über Listener. Ein Widget hört auf einen bestimmten Typ von Event, wenn es dafür einen Listener registriert hat. Um einen Listener zu registrieren, kann man die Notation mit inneren Klassen verwenden, wie Listing 5.5 zeigt (vgl. [HT07], S 195f; [GWT08i]).

```
public void anonClickListenerExample() {
    Button b = new Button("Click Me");
    b.addClickListener(new ClickListener() {
        public void onClick(Widget sender) {
          // handle the click event
        }
    });
}
```

Listing 5.5: Eventlistener mit innerer Klasse

([GWT08i])

Erstellt man eigene Widgetklassen, wie es zum Beispiel bei einem Composite der Fall ist, kann man den Listener auch innerhalb der Klasse registrieren. In diesem Fall ist es auch möglich, den selben Event für mehrere Widgets, innerhalb dieses Composites gemeinsam zu behandeln. Dazu implementiert man den EventListener und fügt diesen Listener mit *addEventListener(this)* an die gewünschten Widgets an. Durch die Referenz auf das eigene Objekt kann die behandelnde Methode in der Klasse selbst implementiert werden. Da beim Aufruf dieser Handler-Methode das Widget übergeben wird, welches den Event ausgelöst hat, kann wenn nötig selektiert werden (vgl. Listing 5.6). Auf diese Weise spart man Klassen ein, was sich Ressourcen schonen auswirkt. (vgl. [GWT08i])

```
public class ListenerExample extends Composite
        implements ClickListener {
    private FlowPanel fp = new FlowPanel();
    private Button b1 = new Button("Button 1");
    private Button b2 = new Button("Button 2");
```

```
    public ListenerExample() {
        initWidget(fp);
        fp.add(b1);
        fp.add(b2);
        b1.addClickListener(this);
        b2.addClickListener(this);
    }

    public void onClick(Widget sender) {
        if (sender == b1) {
            // handle b1 being clicked
        } else if (sender == b2) {
            // handle b2 being clicked
        }
    }
}
```

Listing 5.6: Eventlistener in einem Composite

([GWT08i])

5.4 Kommunikation mit dem Server

Da die meisten Webapplikationen nur in Verbindung mit dem Server sinnvoll arbeiten können, muss man mit diesem Daten austauschen. GWT bietet dazu zwei Möglichkeiten: *GWT-RPC* und die sogenannten *HTTP Requests*. Beide bauen auf dem XmlHttpRequest Objekt des Browsers auf, welches Ajax Applikationen die asynchrone Kommunikation mit dem Server erlaubt. Kommt am Server ebenfalls Java zum Einsatz, so können mit GWT-RPC Java-Objekte zwischen Server und Browser verschickt werden. Welche Klassen für einen solchen Remote Procedure Call erforderlich sind, zeigt Abbildung 5.8. Um die möglichen Calls, sowohl auf dem Client als auch auf dem Server bekannt zu machen, werden sie in einem Interface definiert, welches sich im Client-Paket befindet. In einem weiteren Interface werden die asynchronen Callback Methoden definiert. Das eigentliche Service muss nun am Server implementiert werden, um es vom Browser aus aufrufen zu können. Hinter den Kulissen wird dabei von der abgeleiteten Klasse *RemoteServiceServlet* das Empfangen bzw. Versenden sowie das Serialisieren bzw. Deserialisieren des Textes übernommen. Ein Methodenaufruf, welcher das Passwort eines Benutzers ändert, könnte dabei wie in Listing 5.7 aussehen. (vgl. [HT07], S 18f)

```
service.changePassword("jdoe", "abc123", "m@tr1x",
    new AsyncCallback()
    {
        public void onSuccess (Object result){
            Window.alert("password changed");
        }
        public void onFailure (Throwable ex){
            Window.alert("uh oh!");
        }
    });
```

Listing 5.7: GWT-RPC Aufruf

([HT07], S 19)

Da GWT aber eine reine View Technologie ist, und somit nichts über den Server aussagt, kann dort natürlich auch eine andere Technologie als Java zum Einsatz kommen. Für diesen Fall unterstützen HTTP Requests die Kommunikation. Dazu muss in der

Modulkonfiguration das HTTP Paket aufgenommen werden,

`<inherits name="com.google.gwt.http.HTTP"/>`.

Ein Objekt der Klasse *RequestBuilder* kapselt dabei die Funktionalität für Senden und asynchrones Empfangen. Listing 5.8 zeigt ein Beispiel für einen GET Request. Um Daten besser aufzubereiten und somit die Kommunikation zu erleichtern, bietet GWT Tools zum Parsen von JSON und XML. Näher soll an dieser Stelle nicht auf die Kommunikation mit dem Server eingegangen werden. Weiterführende Informationen dazu finden sich in der Online-Dokumentation [4] von GWT (vgl. [HT07], S 16f; [GWT08m]).

```
import com.google.gwt.http.client.*;
...

String url = "http://www.myserver.com/getData?type=3";
RequestBuilder builder = new RequestBuilder(
    RequestBuilder.GET, URL.encode(url));
try {
    Request request = builder.sendRequest(
            null, new RequestCallback() {
        public void onError(Request request, Throwable exception){
            // Couldn't connect to server
```

[4] http://code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5

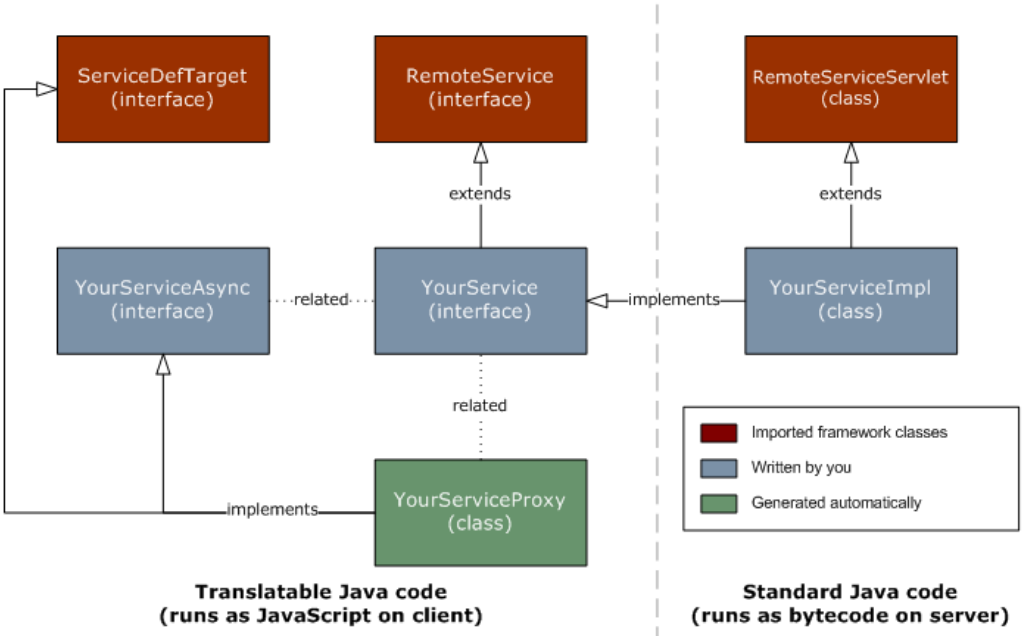

Abbildung 5.8: Struktur einer GWT-RPC Implementierung ([GWT08t], S 9)

```
            // (could be timeout, SOP violation, etc.)
        }

        public void onResponseReceived(Request request,
                Response response) {
            if (200 == response.getStatusCode()) {
                // Process the response in response.getText()
            } else {
                // Handle the error. Can get the status text from
                // response.getStatusText()
            }
        }
    });
} catch (RequestException e) {
    // Couldn't connect to server
}
```

Listing 5.8: GWT HTTP Request

([GWT08m])

Kapitel 6

Xaml für GWT

Nachdem nun die verwendeten Technologien einzeln besprochen wurden, soll im Folgenden dargestellt werden, wie Xaml und GWT zusammengeführt werden können. Dazu wird zuerst besprochen, welche Unstimmigkeiten zwischen Xaml bzw. Xaml, wie es in der WPF eingesetzt wird, und GWT bestehen. Daraus werden Bedingungen, Anforderungen und schließlich eine Architektur entwickelt. Wie aus der Übersichtsgrafik (Abb. 6.1) ersichtlich ist, geht es nun vor allem darum, die Verbindung zwischen der Xaml Datei und dem Framework zu schaffen. Es werden also alle Teile behandelt, die notwendig sind, um aus dem Xaml in einer XML Datei einen passenden Objektgraphen zu erstellen. Schließlich wird in einer Teststellung die Umsetzbarkeit dieser Architektur geprüft.

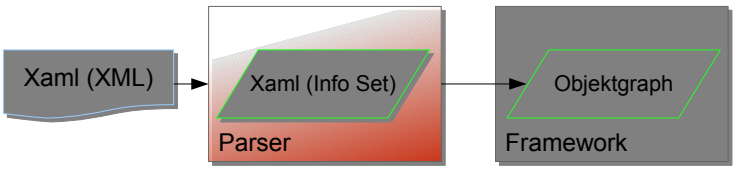

Abbildung 6.1: Übersicht

6.1 Architektur

6.1.1 Differenzen der Technologien

Zwischen Xaml und GWT gibt es einige Unstimmigkeiten. Ein wesentlicher Grund dafür ist, dass Xaml auf Eigenschaften ausgerichtet ist. Alle Merkmale des Objektbaumes, welcher mit Xaml abgebildet wird, finden sich in Eigenschaften von Objekten wieder. Namentlich als die Werte von Member Node Information Items, welche Object Node Information Items zugeordnet sind. In Java hat eine Klasse zwar auch Eigenschaften, diese sind aber üblicherweise *private* und werden über Getter und Setter Methoden angesprochen (vgl. [GJSB05], S 149). Auch c# kennt Methoden zur Validierung für eine Wertzuweisung. Diese Methodenaufrufe geschehen jedoch implizit bei einer einfachen Wertzuweisung mit dem '=' Operator (vlg. [Küh08], Kap. 5.1). Somit ist c# dem auf Eigenschaften basierten Modell von Xaml näher als Java. Sollen die Java Gepflogenheiten beibehalten werden, so kann in einer Xaml-GWT Datei also eine Eigenschaft *width* nicht wie in es in der WPF üblich wäre mit *Width* angesprochen werden. Stattdessen notiert man den Setter dieser Eigenschaft, also *setWidth*.

Eine weitere Konsequenz daraus zeigt sich bei Eigenschaften eines Objekts, welche Aufzählungen referenzieren. In c# sind auch diese sichtbar. GWT hingegen versteckt diese Aufzählungen hinter Methoden wie *add*. Es muss also aus der Liste von Widgets, welche dem Panel zu geordnet sind, eine Reihe von Methodenaufrufen erstellt werden. Bei einigen Panels, wie zum Beispiel StackPanel oder DockPanel, verlangt die add-Methode mehrere Parameter. Dieses Verhalten stimmt nicht mit jenem von Xaml überein, welches lediglich eine Liste an Kindelementen für einen MemberNode festlegen kann.

Parameter für eine add-Methode sind jedoch nicht vorgesehen.

Weiters wird in der WPF von Klassen, welche in einer Xaml Datei verwendet werden sollen, verlangt, dass sie einen Standardkonstruktor aufweisen. Es gibt in Xaml zwar das *Constructor Information Item*, dieses ist jedoch nur für Markuperweiterungen vorgesehen (vgl. [MS:08], S 18). Somit sind Widgets ohne Standardkonstruktor, wie zum Beispiel *RadioButton*, problematisch.

Auch bei der Behandlung von Ereignissen gibt es Unterschiede zwischen Java und .Net. C# kennt das Konstrukt der *Delegates*. Ein Ansatz, der sich gut für Xaml mit seinen Eigenschaften eignet (siehe 5.3.3). Java hingegen verwendet EventListener (siehe 4.8). Um die Methoden für die Ereignisbehandlung, also die Handler, mit eigener Logik zu füllen, werden sie in inneren Klassen überschrieben. Auch hier können wieder Methoden mit mehreren Parametern auftreten, wie zum Beispiel im Fall des *KeyboardListeners*.

Um Code-Behind Dateien zu referenzieren, wird in der WPF das C# Konstrukt der partiellen Klassen verwendet (siehe 4.3). In Java gibt es etwas derartiges nicht. Code-Behind muss sich also in einer Subklasse jener Klasse befinden, welche die grafische Oberfläche beschreibt. Somit können die grafischen Aspekte um die Logischen erweitert werden. Das geschieht entweder durch Überschreiben alter oder Einführen neuer Methoden.

Ebenfalls anders gehandhabt werden angefügte Eigenschaften. In der WPF registriert man angefügte Eigenschaften am Framework, welches diese verwaltet. Sie sind somit ein Sonderfall der Dependency Properties (siehe 4.5). In GWT ist etwas derartiges nicht vorgesehen, also kann auch keine Registrierung für angefügte Eigenschaften erfolgen. Sie müssen also auf andere Weise verarbeitet werden.

6.1.2 Überlegungen zur Umsetzung

Es sollen nun einige grundsätzliche Eckpunkte für den Einsatz von Xaml für GWT festgelegt werden. Diese dienen als Basis für das verwendete Klassenschema.

Ein dynamisches Einlesen der Xaml Dateien, wie in der WPF eingesetzt (siehe 4.2), wird aus mehreren Gründen nicht unterstützt. Zum einen ist eine derartige Funktion für herkömmlich beschriebene GWT Benutzeroberflächen auch nicht vorgesehen. Weiters würde sich auch das Laden der AJAX Applikation verlangsamen. In GWT wird außerdem dynamisches Laden von Ressourcen durch sogenanntes *Deferred Binding* vorweggenommen. Dazu müssen die Ressourcen jedoch zur Kompilierzeit bekannt sein. Schließlich kann im JavaScript Umfeld des Browsers die Klassenbibliothek nicht mehr über Reflection ausgelesen werden. Dies verhindert eine mögliche Variante zur Validation der Xaml.

Die Xaml Dateien im XML Format müssen also für den GWT-Kompiler aufbereitet werden, damit sie dieser zu JavaScript verarbeiten kann. Da der GWT-Kompiler Java Quelltexte benötigt, ist somit das Zielformat festgelegt. Es ist ein Verfahren vonnöten, welches XML zu Java Quelltext umwandelt. Somit scheiden Techniken zum Bearbeiten und Erstellen von Java Class Dateien, also Bytecode Tools wie zum Beispiel *BCEL* [1], zum Generieren der Klassen aus.

Die oben erwähnten Unstimmigkeiten zwischen der GWT Klassenbibliothek bzw. den Java Konventionen und Xaml bzw. .Net erfordern kleine Adaptierungen am Xaml Parser gegenüber seiner Beschreibung in [MS:08]. Gepaart mit der Problematik der verschiedenen Frameworks, .Net und Java, erscheint auch der Einsatz der .Net Klasse *XamlReader* als nicht zielführend. Der Autor kann zwar nicht definitiv ausschließen, dass der Einsatz des XamlReader möglich ist, aus genannten Gründen wurde jedoch auf eine Validierung dieser Option verzichtet.

Fazit

Als Ergebnis dieser Überlegungen entscheidet sich der Autor für eine eigene Implementierung eines XML zu Xaml Parsers. Dieser soll sich soweit als möglich an die Vorgaben in [MS:08] Abschnitt 6 halten. Dieser Parser liefert ein Xaml Information Set, welches die Daten aus der XML Datei repräsentiert.

Für diesen XML zu Xaml Parser ist die Validierung gegen ein Xaml Schema Information Set notwendig. In [Rel06] wird angedeutet, dass das Schema der WPF früher mit Hilfe von XML-Schema beschrieben wurde. Heute wird es dynamisch aus der Klassenbibliothek generiert. Für die vorliegende Arbeit werden diese beiden Möglichkeiten herangezogen, um die Schemainformationen bereitzustellen.

Als zweiter Schritt in der Bearbeitung muss das Xaml Information Set zu Quelltext, also in eine Zeichenkette umgewandelt werden. Dieser Schritt wird ebenfalls neu implementiert. Es handelt sich dabei um eine doch recht spezielle Lösung, für die dem Autor keine bestehenden Tools bekannt sind.

[1] http://jakarta.apache.org/bcel/

6.1.3 Konzept von Xaml2GWT

Für den Konvertierungsprozess von XML zu Java Quelltext wird die Struktur eingesetzt, wie sie in Abbildung 6.2 zu sehen ist. Ausgangspunkt stellt eine XML Datei dar, welche die deklarative Beschreibung der graphischen Oberfläche mit Xaml beinhaltet. Beispielhaft wird diese Datei *Foo.xaml* genannt.

Das Ergebnis der Konvertierung ist die Java Klasse *FooXaml.java*. Um die für die Darstellung erforderliche Logik implementieren zu können, leitet der Entwickler diese Klasse ab und erweitert sie in der Klasse *Foo.java*, welche die Code-Behind Datei darstellt. Hier können zum Beispiel die Methoden für die Eventbehandlung aus *FooXaml.java* überschrieben werden. Foo.java kann nun in die GWT Applikation eingebunden werden. Dazu kann zum Beispiel ein Objekt von Foo.java in jener Klasse, welche *EntryPoint* implementiert, instantiiert und an das RootPanel angefügt werden.

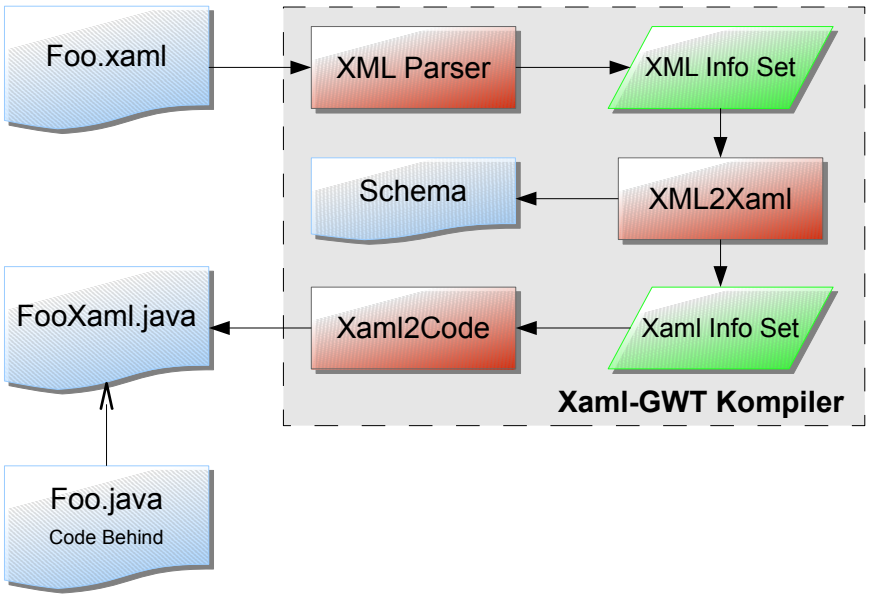

Abbildung 6.2: Xaml zu GWT Kompiler - Schema

Die eigentliche Transformation geschieht dann in den in Rot gehaltenen Komponenten in Abbildung 6.2. Grün dargestellt sind die Zwischenergebnisse der einzelnen Schritte zu sehen.

XML Parser In diesem Schritt wird die XML Datei eingelesen. In der Java Standard Edition ist eine API für die Verarbeitung von XML Dateien, namens Java API for XML Processing (JAXP)[2], enthalten. Diese wird für den XML Parser herangezogen.

XML Info Set Das Ergebnis des XML Parsers ist ein XML Information Set (vgl. [W3C04], welches alle Daten aus der Xaml Datei enthält.

XML2Xaml Als nächstes wird aus dem XML Information Set ein Xaml Information Set (siehe 3.6) generiert. Die Logik für die Umwandlung basiert dabei auf dem in [MS:08], Abschnitt 6 beschriebenen Parser.

Xaml Info Set Das aus XML2Xaml resultierende Xaml Information Set.

Xaml2Code Schließlich wird aus Xaml Java-Quellcode erzeugt. Die in diesem Schritt erstellte Klasse enthält alle Widgets, welche in der Xaml Datei angegeben wurden, samt den dazu festgelegten Eventhandlern.

Um die beiden Information Sets von XML und Xaml in Java verwenden zu können, müssen deren Elemente als Java-Klassen nachgebildet werden. Für das XML Information Set ist das bereits geschehen. Die Java Standardbiblothek enthält im Paket *org.w3c.dom* diese Klassen. Für das Xaml Information Set müssen diese Klassen erst erstellt werden. Ebenso werden Klassen für das Xaml Schema Information Set benötigt.

6.1.4 Xaml Schema Information Set

Um aus einem XML Information Set ein Xaml Information Set erstellen zu können, muss die Validität der Daten im XML Information Set geprüft werden. Als Referenz dient ein Xaml Schema Information Set. Darüber hinaus werden alle Objekte und Member eines Xaml Information Sets mit den korrespondierenden Typen aus dem Schema verknüpft. Ist also im XML Information Set ein Element mit dem Namen Button vorhanden, so wird es im Xaml Information Set mit dem XamlType Information Item Widget verknüpft. Für GWT existiert ein solches Xaml Schema noch nicht, und

[2] https://jaxp.dev.java.net/

muss daher erst erstellt werden. Wie bereits in Abschnitt 6.1.2 erwähnt, kommen bei Microsoft die beiden Varianten XML-Schema und dynamisches Auslesen der Klassenbibliothek zum Einsatz (vgl. [Rel06]). Im Folgenden werden diese beiden Möglichkeiten im Hinblick auf GWT besprochen.

Um die Implementierung für verschiedene Varianten der Validierung offen zu halten, kann ein Interface als Schnittstelle dienen. Dieses definiert die für den Parser notwendigen Methodenaufrufe. Alle konkreten Valdiatoren implementieren dieses Interface und sind somit austauschbar. Durch den Einsatz der Schnittstelle erreicht man eine losere Kopplung der Klassen (vgl. [Bac09], S 138f).

XML-Schema

Mit XML-Schema kann festgelegt werden, welche XML Elemente und Attribute in einer XML Datei vorkommen dürfen. Außerdem kann definiert werden, innerhalb welcher Elemente ein Element bzw. ein Attribut angegeben werden kann. Auch die Anzahl der Elemente kann festgelegt werden. Ebenso ist es möglich Aussagen über den möglichen Wertebereich zu treffen (vgl. [XML04b]; [Hol01], S 242f).

XML-Schema bietet somit zwei Möglichkeiten. Entweder man beschreibt direkt das Xaml Schema, oder man beschreibt den möglichen Inhalt der Xaml Datei. Wird das Xaml Schema im Hinblick auf die XML bzw. Xaml Datei geschrieben, gewinnt man dadurch in geeigneten Editoren Syntax-Highlighting und oft auch Auto-Vervollständigung, da der Editor dann weiß, was alles im Rahmen der Datei erlaubt und möglich ist. Ein solches XML-Schema muss jedoch für die Umwandlung des XML Information Sets in ein Xaml Information Set aufbereitet werden. Die für diesen Prozess erforderlichen Informationen sind zwar vorhanden jedoch nicht in der gewünschten Form, also nicht in Form von XamlType Information Items und so weiter.

Ein weiterer Vorteil von XML Schema ist die Möglichkeit wichtige Zusatzinformationen, wie die Angabe der Inhaltseigenschaft, festlegen zu können.

Nachteilig ist, dass alle Klassen welche graphische Komponenten repräsentieren, in der Xaml Schema Definition (XSD) samt ihren Abhängigkeiten enthalten sein müssen. Dazu muss die gesamte Vererbungshierarchie aufgelöst werden, um die zuweisbaren

Elemente festlegen zu können. Möchte man zusätzliche Widget-Bibliotheken von Dritt-Anbietern einsetzen, muss auch für diese eine XSD erstellt werden. Eine gewissermaßen redundante Aufgabe, da alle Informationen bereits in der Klassenbibliothek enthalten sind.

Validation gegen die Klassenbibliothek

Um die für das Schema relevanten Informationen aus der Klassenbibliothek zu erhalten, kann Javas Reflection API[3] verwendet werden. Reflection ermöglicht es anhand eines Objektes oder eines Klassennamens,, Informationen über die zugehörige Klasse zu ermitteln. Dazu zählen unter anderem die Eigenschaften und Methoden samt Modifizierer, Typen und Parameter. Aus diesen Daten kann das benötigte Schema Element erstellt werden.

Es bietet sich hier eine dynamische Validation an. Immer wenn der Parser ein Schema Element für ein Element aus dem XML Information Set benötigt, erfolgt die Abfrage der zugehörigen Klasse und das Schemaelement wird erstellt. Die Zuordnung von XML Element zu Java Klasse erfolgt über den Namespace. Wie bereits erwähnt soll ja als Namespace der Java Paketname verwendet werden.

Mit diesem Ansatz lässt sich das entsprechende Schema für alle Klassen, die sich im Classpath befinden, erstellen. Darunter fallen auch eventuell verwendete zusätzliche Bibliotheken. Die einfache Integration von Drittbibliotheken und das Wegfallen der Nachbildung bereits bestehender Information stellt den Vorteil dieses Ansatzes dar.

Von Nachteil ist die fehlende Unterstützung für Editoren, also das zuvor angesprochene Syntax Highlighting. Weiters können keine Angaben über die Inhaltseigenschaft getroffen werden. In der WPF werden derartige Informationen mit Annotationen in den Quelltext eingearbeitet. Für GWT müsste man dafür die gesamte Klassenbibliothek mit Annotationen versehen. Eine einfache Möglichkeit das zu umgehen ist die Definition von Standardwerten für die Inhaltseigenschaft. Das bedeutet, es werden jene Methodennamen, die für die meisten Klassen sinnvoll als Inhaltseigenschaft verwendet werden können hinterlegt. Wird für ein Element die Inhaltseigenschaft benötigt,

[3] http://java.sun.com/developer/technicalArticles/ALT/Reflection/index.html

wird die zugehörige Klasse nach Methoden durchsucht, deren Name mit einem dieser Standardwerte übereinstimmt.

Es ergibt sich jedoch noch eine zusätzliche Schwierigkeit. Für GWT Klassen, welche für die grafische Benutzerschnittstelle eingesetzt werden, ist eine Instantiierung nur in GWTs Hosted Mode vorgesehen. Für den Web Mode kommt ja bereits das kompilierte JavaScript zum Tragen. Das Instantiieren eines Objektes erfolgt durch die statische Methode *GWT.create()*. Dabei erkennt diese Methode, ob sie im Zuge des Hosted Mode aufgerufen wurde, und bricht andernfalls ab. Für eine dynamische Validierung, wie sie zuvor beschrieben wurde, muss die Klasse über eines ihrer Objekte gefunden werden. Es müssen also Objekte der GWT Klassen erstellt, und somit GWT.create umgangen werden. Dazu werden alle Aufrufe von GWT.create aus der Klassenbibliothek entfernt.

6.1.5 Anforderungen an den Quelltext

Der vom Xaml-GWT Kompiler erzeugte Quelltext, muss eine Klasse sein, da Java eine objektorientierte Sprache ist. Damit sich die neue Klasse in die Oberfläche von GWT integrieren lässt, muss sie in irgendeiner Form von der Klasse *Widget* erben.

Andernfalls könnte man Objekte der neuen Klasse nicht an das *RootPanel* anfügen. Da es durch die Deklaration mit Xaml ohnehin ein definiertes Wurzelobjekt gibt, erbt die Klasse von diesem Widget. Im übrigen weist die WPF in diesem Punkt das selbe Verhalten auf.

Alle Widgets, die unterhalb des Wurzelobjekts liegen, werden als private Eigenschaften deklariert und im Konstruktor der Klasse instantiiert. Weiters werden für sie öffentliche Getter und Setter erstellt, um sie von außerhalb der Klasse ansprechen zu können. Das Konzept der Composites (siehe 5.3.2) wird nicht herangezogen, da alle Widgets außer des Obersten versteckt werden. Dieses Verhalten ist nicht immer gewünscht.

Xaml bietet durch Elemente des x Namespace einige Möglichkeiten, die zu erstellende Klasse zu beeinflussen. Namentlich sind das die XamlMemberInformationItems x:Class, x:Subclass sowie x:ClassModifier. In der Xaml Spezifikation ist das Verhalten dieser Direktiven nicht festgelegt. Es steht jeder Implementierung also frei, wie die Angaben dieser Direktiven behandelt werden (vgl. [MS:08] S 49). Für den vorliegenden Fall wird

die x:Class Direktive dazu verwendet, den vollqualifizierten Namen der zu erstellenden Klasse, das heißt den Klassennamen inklusive Paketnamen, anzugeben. Mit Hilfe von x:ClassModifier können Interfaces angegeben werden, die implementiert werden sollen. Als Syntax scheint jene, die in HTML für CSS Eigenschaften eingesetzt wird, als sinnvoll. Das bedeutet, Bezeichner werden von ihren Werten durch Doppelpunkt getrennt. Mehrere Werte trennt man mit Leerzeichen. Auf die Angabe der Code Behind Klasse mit x:Subclass wird verzichtet. In einer Vererbungshierarchie hat eine Super-Klasse keine Kenntnis über ihre Sub-Klassen. Die Angabe der Sub-Klasse ist somit nicht notwendig. Entwickler können selbst entscheiden, ob sie eine Code Behind Klasse benötigen, und diese gegebenenfalls anlegen.

Listing 6.1 zeigt das Wurzelelement einer Xaml Datei mit den Attributen Class und ClassModifier. In diesem Beispiel werden mit Hilfe des ClassModifier die Eventlistener *ClickListener* und *KeyboardListener* implementiert. Weiters sieht man die Angabe der verwendeten Namespaces. Alle Java Klassen sind einem Paket zugeordnet. Verwendet man den Paketnamen als Namespace, so sind alle Klassen eindeutig zuordenbar. Demnach können alle GWT Widgets verwendet werden, wenn *com.google.gwt.user.client.ui* als allgemeiner Namespace festgelegt wird. Weiters wird Xamls *x* Namespace angegeben, um Vokabular wie x:Name oder x:Class nutzen zu können. Schließlich wird noch *at.ac.fhstp.xaml2gwt* deklariert. Dieser Namespace wird für die Angaben von Eventhandlern verwendet (siehe 6.1.7).

```
<DockPanel
    x:Class="at.ac.fhstp.schauer.foo.client.SampleUiXaml"
    x:ClassModifier=
        "implements:ClickListener KeyboardListener;"
    xmlns="com.google.gwt.user.client.ui"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:y="at.ac.fhstp.xaml2gwt" >
    [...]
</DockPanel>
```

Listing 6.1: GWT-Xaml Klassendefinition

6.1.6 Hinzufügen von Widgets zu Panels

Es wurde bereits angesprochen, dass Methodenaufrufe mit mehreren Parametern nur schlecht in Xaml abgebildet werden können. Zum Tragen kommt dieser Umstand unter anderem beim Anfügen von Widgets an Panels. Wenn die add Methode des Panels mehrere Parameter verlangt, so reicht nicht mehr das Widget selbst als Parameter. Es müssen weitere angegeben werden. Dafür bieten sich angefügte Eigenschaften an. Weil eine Eigenschaft nur einmal vorkommen darf, muss die angefügte Eigenschaft im Fall von mehr als zwei Parametern mehrere Werte enthalten. Listing 6.2 zeigt wie ein Label an ein StackPanel angefügt werden kann. Als zusätzlicher Parameter wird der Titel unter dem das Widget im StackPanel angezeigt wird als Zeichenkette angegeben (vgl. [GWT08a]). Für den Xaml-GWT Kompiler bedeutet das Verwenden von angefügten Eigenschaften, dass bevor ein Element behandelt werden kann, seine Eigenschaften ausgewertet werden müssen.

```
<StackPanel x:Name="SampleUiXaml" >
    <Label x:Name="lbl" setText="Der Text im Label."
        StackPanel.add="stackText" />
</StackPanel>
```

Listing 6.2: GWT-Xaml StackPanel

6.1.7 Eventhandler

GWTs EventListener-Modell (siehe 5.3.3) stellt nicht konkrete Ereignisse als Eigenschaft eines Widgets bereit. Vielmehr verfügen die Widgets über Methoden zum Anfügen eines Listeners. Dieser Listener wiederum bietet Eventhandler die mit eigener Logik überschrieben werden. In Xaml müssen also bei einem Widget, welches ein Ereignis behandeln möchte, folgende Angaben abgebildet werden.

- Die Methode zum Anfügen des Listeners.
- Methodenpaare bestehend aus der Handlermethode des Listeners und der Methode, welche die Logik für die Behandlung des Ereignisses enthält.

Die zweite Methode eines Paares kann in der Code-Behind Klasse, in Abbildung 6.2 ist das Foo.java, überschrieben, und also mit Logik befüllt werden.

Um alle notwendigen Angaben im Xaml Information Set abbilden zu können, welches beim Kompilieren entsteht, werden die Objekte *Event* und *ParameterList* eingeführt. Sie befinden sich im Namespace *at.ac.fhstp.xaml2gwt* und können mit jenen in Xamls x Namespace verglichen werden. Als Präfix wird y vorgeschlagen. In Abbildung 6.3 wird ihr Einsatz am Beispiel des EventListeners ClickListener gezeigt.

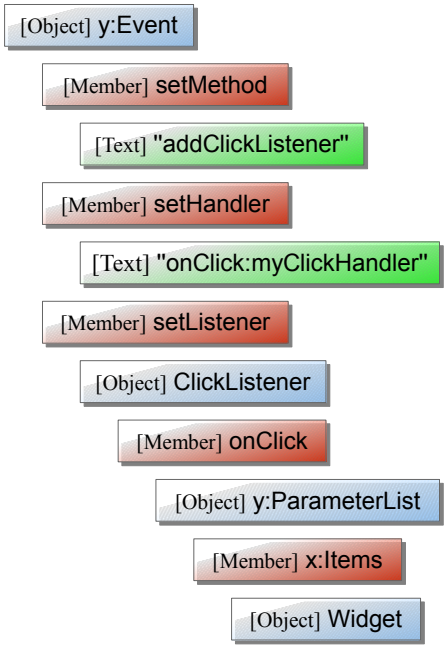

Abbildung 6.3: Xaml Objektgraph - Event

Die Klasse *Event* hat drei Eigenschaften: method, handler und listener. In *method* wird der Name der Methode hinterlegt, die zum Anfügen des EventListeners verwendet wird. Für den Fall des Widgets Button und des EventListeners ClickListener ist das *addClickListener*. Wie Abbildung 6.3 zeigt, findet sich diese Zeichenkette in einem Text Node Information Item das am Member Node Information Item setMethod hängt wieder. Der Member heißt setMethod, da die Klasse *Event* Getter und Setter für ihre Eigenschaften veröffentlicht. In der Xaml Datei können dann nur diese öffentlichen Setter referenziert werden, und nicht die privaten Eigenschaften selbst.

Alle Handler-Methoden Paare werden in der Eigenschaft bzw. dem MemberNodeInformationItem handler bzw. setHandler definiert. In Beispielfall lautet die eigene Methode

auf den Namen *myClickHandler*, die ihr zugeordnete Methode des Interfaces ClickListener ist *onClick*. Als Syntax wird auch in diesem Fall wieder jene aus HTML entlehnt. Zwischen den Methodennamen eines Paares steht ein Doppelpunkt. Wertpaare sind durch Strichpunkte von einander getrennt.

listener dient schließlich dazu, die Listener Klasse, ihre Handlermethoden sowie deren Parameter vorzuhalten. Diese Informationen werden für die Erstellung des Java Quelltextes benötigt. Bei einigen EventListenern, zum Beispiel bei KeyboardListener, kommen auch mehrere Parameter für manche Handlermethoden vor. Somit ist ein Objekt notwendig, welches eine Liste von Parameter-Objekten aufnehmen kann. Dafür wird die bereits erwähnte Klasse *ParameterList* verwendet. Wie das Beispiel in der Abbildung zeigt, hat jede Handler Methode eine ParameterList mit einem Member x:Items, der für Aufzählungen in Xaml verwendet wird.

Listing 6.3 zeigt die zu Abbildung 6.3 korrespondierende Angabe in einer Xaml Datei. Als Widget wird Button angenommen. Der ClickListener wird an den Button mit der Eigenschaftenelementsyntax angefügt. Diese erlaubt es das Element y:Event samt Attributen anzuschreiben.

```
<Button x:Name="myButton"
        setText="Click Mich" >
    <Button.addClickListener>
        <y:Event setMethod="Button.addClickListener"
            setHandler="onClick:btnClickHandler" />
    </Button.addClickListener>
</Button>
```

Listing 6.3: GWT-Xaml EventListener in Xaml

6.1.8 Widgets ohne Standard-Konstruktor

Da Xaml keine Konstruktorargumente, die Ausnahme stellen Markuperweiterungen dar, unterstützt, müssen manche Widgets adaptiert werden (vgl. [MS:08], S 18). Um den benötigten Standardkonstruktor zu erhalten, kann die GWT Klasse um einen solchen erweitert werden. Möchte man die GWT Klassenbibliothek unangetastet lassen,

muss man das entsprechende Widget ableiten. Der Standardkonstruktor der abgeleiteten Klasse muss dabei aber trotzdem den Super-Konstruktor aufrufen. Es ist also notwendig Standardwerte für die Konstruktorargumente zu vergeben. Diese Werte überschreibt man anschließend durch Aufruf der entsprechenden Setter.

6.2 Implementierung

Im Folgenden wird nun eine Testimplementierung beschrieben, welche die beschriebenen Konzepte in die Praxis umsetzt. Als Lösung für die Validierung wird dabei das Auslesen der Klassenbibliothek mit Reflection verwendet. Die Abweichungen gegenüber der Variante mit XML-Schema beschränken sich auf die Implementierung des Validators. Alle übrigen Teile sind in beiden Varianten gleich.

Da, wie bereits erwähnt wurde, die Widget Klassen vom Aufruf GWT.create() befreit werden müssen um Reflection nutzen zu können, wurde eine präparierte Version des Jar Archives gwt-user.jar erstellt. Dabei wurde sämtliche Logik aus der Klasse *UIObject* entfernt, so dass nur noch die Methodensiganturen übrig sind. Es gehen dabei keine Informationen verloren, welche über Reflection ausgelesen werden können, und der GWT.create Aufruf verschwindet. Diese präparierte Bibliothek muss sich im Classpath des Xaml-GWT Kompilers befinden.

Um den zeitlichen Rahmen der Arbeit nicht zu sprengen, werden bei der Umsetzung einige Einschränkungen vorgenommen. In der Testimplementierung gibt es keine Unterstützung für folgende Elemente des x Namespace: x:ClassModifier, x:Code, x:XData, x:UID, x:DirectiveChildren, xml:space, xml:lang. Das bedeutet, es sind zum Beispiel keine Code Schnipsel im Xaml möglich. Es können nur Widgets mit Standardkonstruktoren verwendet werden. Für Panels, welche mehrere Parameter zum Anfügen der Widgets erwarten wird nur die Attributsyntax unterstützt. Es können also nur Referenzen auf komplexe Datentypen und primitive Datentypen übergeben werden. Um komplexe Typen neu anzulegen, wäre die Eigenschaftenelementsyntax notwendig. Auch werden angefügte Eigenschaften nur für dieses Anfügen von Widgets zu Panels unterstützt. Schließlich gibt es keine Unterstützung für Inhaltseigenschaften, Markuperweiterungen, Initialisierungstexte und Dictionaries.

Vorweg sei eine beispielhafte Xaml Datei sowie ihr Ergebnis in GWTs Hosted Browser gezeigt. Wie in Listing 6.4 zu sehen ist, enthält das Beispiel als Wurzelelement ein DockPanel. In dessen oberen Bereich befindet sich ein Label, welches eine Überschrift beinhaltet. Links wird ein Menü bestehend aus Hyperlinks innerhalb eines VerticalPanels platziert. Im zentralen Bereich findet sich ein Button, für den ein onClick Eventhandler definiert wird. Abbildung 6.4 zeigt diese Oberfläche im Hostet Browser. Um die Struktur der Panels besser erkennen zu können, wird in der CSS Datei *Foo.css* die Definition eines Rahmen als Stilklasse vorgenommen. Im DockPanel dient *setStyleName* dazu, die Stilklasse zu referenzieren. Die Existenz der CSS-Datei wird in der Modul Datei bekannt gemacht.

```
<DockPanel
    x:Class="at.ac.fhstp.schauer.foo.client.SampleUiXaml"
    xmlns="com.google.gwt.user.client.ui"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:y="at.ac.fhstp.xaml2gwt"
    x:Name="this"
    setStyleName="cw-DockPanel" >
    <Label x:Name="lbl" setText="Hier ist der Titel!"
        DockPanel.add="com.google.gwt.user.client.ui.
            DockPanel.NORTH" />
    <VerticalPanel x:Name="vp"
            DockPanel.add="com.google.gwt.user.client.ui.
                DockPanel.WEST" >
        <Label x:Name="lblMenu" setText="Menu" />
        <Hyperlink x:Name="linkHome" setText="Home"
            setTargetHistoryToken="Home"  />
        <Hyperlink x:Name="linkImpr" setText="Impressum"
            setTargetHistoryToken="Impr" />
    </VerticalPanel>
    <Button x:Name="myButton" setText="Click Mich"
            DockPanel.add="com.google.gwt.user.client.ui.
                DockPanel.CENTER" >
        <Button.addClickListener>
            <y:Event setMethod="Button.addClickListener"
                setHandler="onClick:btnClickHandler" />
        </Button.addClickListener>
```

```
    </Button>
</DockPanel>
```

Listing 6.4: GWT-Xaml Beispiel Xaml

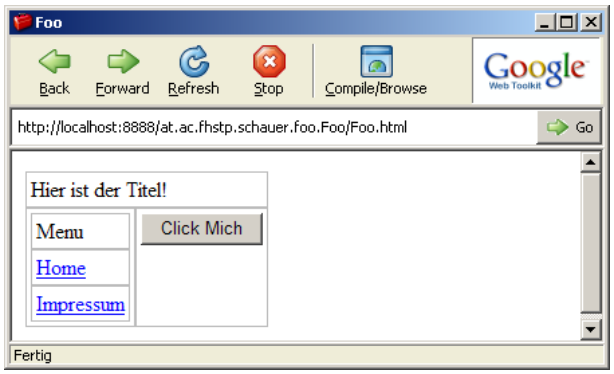

Abbildung 6.4: Ergebnis von Listing 6.4 im Hosted Browser

6.2.1 Übersicht über die verwendeten Klassen

Die Hauptklasse der Implementierung ist *Main*. Sie ist für die Ablaufsteuerung verantwortlich. Von ihr aus werden die Methoden für die einzelnen Bearbeitungsschritte aufgerufen. Abbildung 6.5 zeigt nur zwei von Main verwendete Klassen, *XML2Xaml* und Xaml2Code, da der Aufwand zum Erstellen des XML Information Sets so gering ist, dass diese Logik direkt in der Klasse Main enthalten ist.

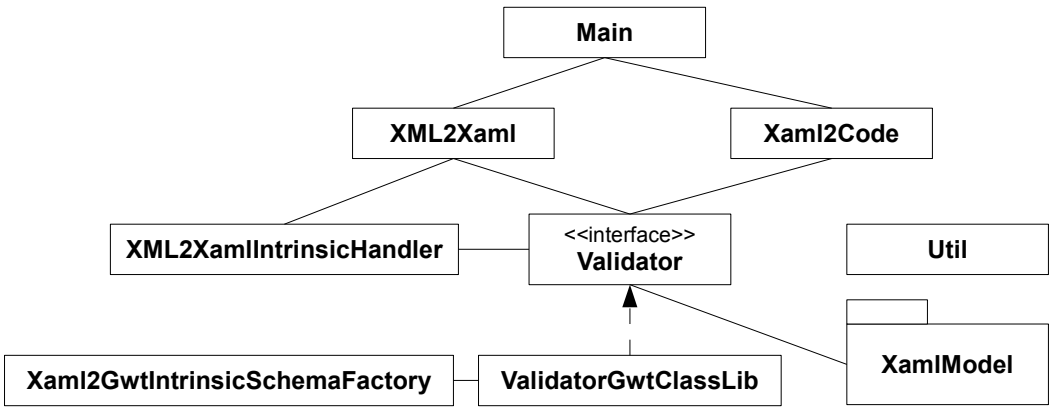

Abbildung 6.5: Übersicht über die Klassen von Xaml2GWT

Die Aufgaben der Klassen sehen für diesen Schritt wie folgt aus:

XML2Xaml Durchgehen der Elemente des XML Information Set. Aufruf des Validators für jedes dieser Elemente. Verpacken der Information in die passenden Elemente eines Xaml Information Sets.

ValidatorGetClassLib Suchen nach jener Java Klasse, welche mit dem jeweiligen Element korrespondiert.

XamlModel Dieses Paket enthält Java Klassen für die Elemente, welche in einem Xaml Information Set und in einem Xaml Schema Information Set enthalten sein können.

Xaml2GwtIntrinsicSchemaFactory Dient zum Erstellen von Objekten des Typs ParameterList, der für die Abbildung von Events benötigt wird.

XML2XamlIntrinsicHandler Erstellen von Object Node Information Items vom Typ *Event*.

Unerwähnt geblieben ist bis jetzt Util. Diese Klasse bietet einige statische Methoden, welche von mehreren Klassen benötigt werden.

Im weiteren werden nun die einzelnen Schritte, wie sie in Abbildung 6.2 dargestellt sind, beschrieben. Zu Beginn wird noch auf die Hauptklasse eingegangen, und wie der Kompiler aufgerufen wird.

6.2.2 Aufruf des Xaml-GWT Komilers

Die Klasse *Main* stellt den Einstiegspunkt in den Kompiler dar. In ihrer statischen main Methode ruft sie der Reihe nach die Methoden zur Transformation der XML Daten auf. Die resultierende Zeichenkette, welche den Java Quelltext enthält, wird anschließend in eine Datei geschrieben. Als Parameter erwartet Main den Pfad zu der Xaml Datei, die kompiliert werden soll. Die resultierende Datei wird im selben Verzeichnis erstellt. Als Dateiname dient der Namen der Xaml Datei mit der Endung *Xaml.java*. Ist der Name der Xaml Datei also Foo.xaml, so heißt die generierte Java Datei FooXaml.java

Da sich die bereits erwähnte, präparierte Version von gwt-user.jar im Classpath des Kompilers befinden muss, und zusätzlich auch die übrigen vom User verwendeten Klassen ebenso, ist ein Aufruf des Kompilers von der Kommandozeile unkomfortabel. Ein Ant [4] Skript stellt eine bessere Möglichkeit des Aufrufs dar. Um mehrere Xaml Dateien mit einem Aufruf von Ant zu kompiliern, kann der *for* Task von ant-contrib [5] verwendet werden. Damit kann eine Aktion für jedes Element einer Liste ausgeführt werden. In diesem Fall also der Aufruf des Kompilers für jede Xaml Datei in einem FileSet.

6.2.3 XML Parser

Für das Einlesen der XML Datei wird JAXP verwendet. JAXP ist Teil der Java Klassenbibliothek, und stellt eine API zur Behandlung von XML Daten dar. Im Ramen dieser API werden drei Varianten angeboten: SAX, DOM und StAX (vgl. [JAX08]). Gewählt wird aus diesen Möglichkeiten die Variante DOM. Damit wird eine XML Datei eingelesen und daraus ein DOM Baum generiert. Dem DOM Ansatz wird entgegengehalten, dass er speicherintensiver ist als SAX oder StAX. Da der Kompiler aber nur für die Entwicklung verwendet wird, kann dieser Nachteil in Kauf genommen werden. Vorteile an JAXP/DOM sind die einfache Erstellung des DOM Baums sowie die ständige Verfügbarkeit aller Daten.

Der Aufruf von JAXP/DOM erfolgt in der Methode *xml2Dom* in der Klasse Main. JAXP/DOM erlaubt mehrere Einstellungen. Genutzt werden in diesem Fall das Ignorieren von Kommentaren, sowie die Erkennung von Namespaces. Dadurch kommen im generierten DOM Baum keine Kommentare vor, und alle Elemente wissen über den Namespace, in dem sie erstellt wurden Bescheid.

6.2.4 Transformation XML Information Set - Xaml Information Set

Für die Transformation des XML Information Set in ein Xaml Information Set wird die Methode *convertXML2Xaml* der Klasse *XML2Xaml* aufgerufen. XML2Xaml hält sich

[4] http://ant.apache.org/
[5] http://ant-contrib.sourceforge.net/

weitest gehend an die Vorschriften für einen derartigen Parser aus [MS:08], S 58f. Für die dort festgelegten Regeln existiert je eine private Methode, welche diese Regel nachbildet. Abweichungen treten auf, wenn manche Teile nicht implementiert werden, da diese aus der Testimplementierung ausgeklammert werden. Dazu zählen zum Beispiel Markuperweiterungen und Codeschnipsel in Xaml. Ein weiterer Unterschied ergibt sich in der Behandlung der Namespaces. Jedes Element im DOM Baum, welcher von JAXP/DOM generiert wird, kennt seinen Namensraum. Es ist also nicht notwendig alle Namensräume zu verwalten.

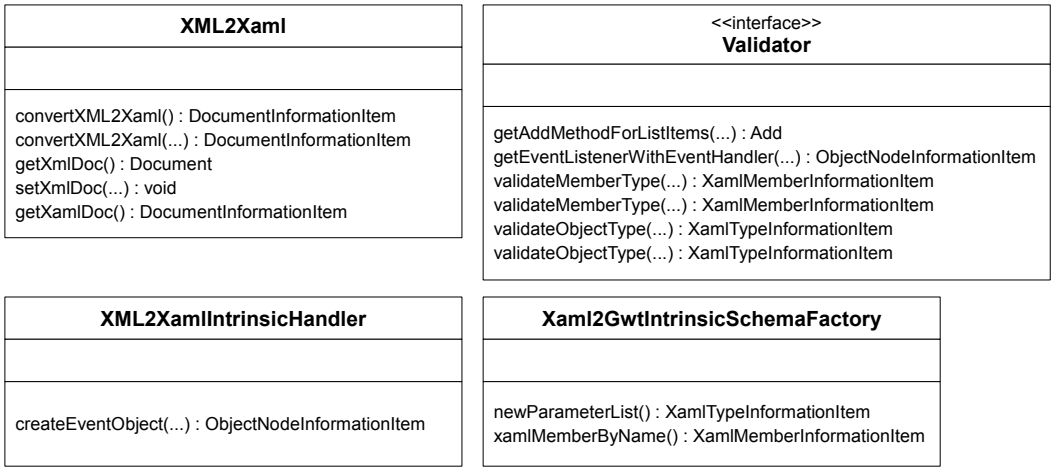

Abbildung 6.6: Klassen für die Xaml Info Set Erstellung

Um die Schema Informationen zu erhalten, werden Methoden der *Validator* Implementation *ValidatorGwtClassLib* aufgerufen. Diese Methoden, zu sehen sind sie in Abbildung 6.6, geben entweder ein XamlType Information Item oder ein XamlMember Information Item zurück. Davon gibt es zwei Ausnahmen. Die Methode *getEventListenerWithEventHandler* liefert ein Object Node Information Item vom Typ y:Event. Events werden direkt im Validator erstellt, um den Listener mit seinen Handlern und deren Parametern setzen zu können (siehe Abbildung 6.3). Die zweite Ausnahme ist *getAddMethodForListItems*, welche von Xaml2Code verwendet und dort besprochen wird.

Damit der Validator Xaml Information Items zurückgeben kann, müssen diese als Java Klassen existieren. Das Paket *at.ac.fhstp.xaml2gwt.XamlModel* enthält die Klassen für das Xaml Information Item, das Subpacket *Schema* jene für Xaml Schema Information Item. Zu erwähnen wäre hier die Klasse *IntrinsicSchemaFactory*, die Objekte aus dem

x Namespace ausliefert und mit Standardwerten versieht.

Schließlich gibt es noch die beiden Klassen *XML2XamlIntrinsicHandler* und *Xaml2GwtIntrinsicSchemaFactory*, welche die für GWT spezifischen Elemente des *y* Namespace behandeln. Xaml2GwtIntrinsicSchemaFactory dient zum Erstellen von ParameterList Objekten. XML2XamlIntrinsicHandler wird vom Validator aufgerufen um das Event Objekt mit den erforderlichen Werten zu bauen. Beide Klassen können bei Bedarf für zusätzlich Typen erweitert werden.

Für die Fehlerbehandlung finden sich im Paket *at.ac.fhstp.xaml2gwt.parserExceptions* Exception Klassen. Alle in diesem Pakte befindlichen Klassen erben von *XamlParserException*.

6.2.5 Transformation Xaml Information Set - Java Quelltext

Um schließlich den Java Quelltext zu erstellen, wird die Methode *convertXaml2Code* der Klasse *Xaml2Code* aufgerufen. Die Funktionalität dieser Klasse gliedert sich in drei Bereiche. Zuerst wird die Klassensignatur erstellt. Dazu wird wie bereits besprochen das oberste Objekt der Hierarchie als Basistyp herangezogen. Wie Listing 6.5 zeigt, sind die generierten Klassen public. Um Typen zu referenzieren, werden vollqualifizierte Namen verwendet. Im Vergleich zu *import* Anweisungen hat das den Vorteil, dass Konflikte bei gleichnamigen Klassen in verschiedenen Pakten vermieden werden.

```
package at.ac.fhstp.schauer.foo.client;
public class SampleUiXaml extends
    com.google.gwt.user.client.ui.DockPanel {
}
```

Listing 6.5: GWT-Xaml generierter Quelltext - Klassensignatur

Im zweiten Schritt werden die vorkommenden Widgets als private Eigenschaften mit öffentlichen Gettern und Settern erstellt. Genaugenommen werden für alle Object Node Information Items, welche nicht aus dem Namespace *at.ac.fhstp.xaml2gwt* sind, Eigenschaften und Zugriffsmethoden erstellt. Objekte im genannten Namespace gelten als Intrinsic Types und dienen nur für den Kompiler selbst.

Als dritter Schritt wird der Konstruktor erstellt. Im Konstruktor werden alle Objekte instantiiert und an ihr übergeordnetes Panel angefügt (siehe Listing 6.6). Jene Methode des Panels, die das Anfügen von Widgets ermöglicht, wird mit Hilfe der Methode *getAddMethodForListItems* aus dem Interface *Validator*, bzw. einer Implementierung dessen, ermittelt. Rückgabewert dieser Methode ist ein Objekt vom Typ *Add*. Die Klasse Add kapselt Name, sowie Parametertypen einer Methode. In diesem Fall also der Methode zum Anfügen eines Widgets an das Panel. Als Parameter werden beim Anfügen zum einen das Widget selbst sowie der Inhalt einer allenfalls vorhandenen angefügten Eigenschaft verwendet. Dabei gilt die Konvention, dass das Widget der erste Parameter sein muss. Andernfalls wäre es schwierig die richtige Reihenfolge der Parameter festzustellen, wenn zwei oder mehrere Parameter den gleichen Typ aufweisen. Durch das Hilfsobjekt Add können Aufrufe der Reflection API im Validator bleiben.

```
public SampleUiXaml() {
    [...]
    myButton = new com.google.gwt.user.client.ui.Button();
    myButton.setText("Click Mich");
    myButton.addClickListener(
            new com.google.gwt.user.client.ui.ClickListener() {
        public void onClick(
            com.google.gwt.user.client.ui.Widget myWidget0) {
            btnClickHandler(myWidget0);
        }
    });
    this.add(myButton,
        com.google.gwt.user.client.ui.DockPanel.CENTER);
}
```

Listing 6.6: GWT-Xaml generierter Quelltext - Konstruktor

Ist für ein Widget ein EventListener definiert, wird eine innere Klasse für den Listener erzeugt, in der seine Handlermethoden überschrieben werden. In diesen Methoden wird die eigene Handlermethode aufgerufen, die der Entwickler später in der Code Behind Klasse implementieren kann. Erkannt wird ein EventListener an einem Object Node Information Item vom Typ Event. Dieses Objekt verfügt über alle nötigen Informationen um den Code für die innere Klasse und den darin befindlichen Methoden zu erzeugen.

Die eigenen Eventhandler Methoden bilden schließlich den Abschluss der Klasse.

6.3 Fazit

Wie die entwickelte Implementierung zeigt, ist die Deklaration von GWT Benutzeroberflächen mit Xaml durchaus möglich. Der Ansatz, die Xaml Dateien zu kompilieren, bevor sie an GWT übergeben werden, fügt sich gut in den Entwicklungsablauf ein. Ist die Oberfläche bereits im Hosted Browser zu sehen, reicht ein Aufruf des Ant-Tasks und die Änderungen können per Refresh in den Hosted Browser geladen werden. Bei einer entsprechenden Unterstützung der verwendeten IDE für Ant reduziert sich der zusätzliche Aufwand auf einen Mausklick. Weiters bietet die Konvention, wonach Javas Paketnamen als Namespaces dienen, eine einfache und flexible Lösung, beliebige Klassen einzubinden.

Von den Problemen, welche die unterschiedlichen Ansätze der beiden verwendeten Technologien bezüglich Eigenschaften mit sich bringen, können einige zufrieden stellend gelöst werden. Allen voran ist die Unterstützung für Events umfassend gelungen. Ein Mangel ist die fehlende Möglichkeit, Adapter statt des Listeners selbst zu verwenden. Aber auch diese könnten über einen booleschen Wert integriert werden. So würde, je nach Zustand dieses Wertes, nach dem EventListener selbst oder seinem Adapter gesucht. Diese Suche kann nach der gängigen Namenskonvention erfolgen.

Womit ein weiterer Punkt angesprochen ist. In der entwickelten Implementierung wird viel über Konventionen geregelt. Zum Beispiel muss beim Anfügen eines Widgets an ein Panel dieses Widget der erste Parameter der Methode sein. Auch die Inhaltseigenschaft wird über Standardwerte gefunden, was also auch gewisse Konventionen verlangt - andernfalls würden brauchbare Methoden jeder Klasse anders heißen.

Positive Aspekte der Implementierung sind die saubere Trennung der Validation des Schemas von der übrigen Funktionalität, wodurch auch andere Ansätze für die Validierung leicht angewendet werden können. JAXP/DOM entspricht bestens den Anforderungen des XML-XAML Konverters, wie er in [MS:08] beschrieben ist. Schließlich lässt sich ein Xaml Information Set gut in Java Quelltext umwandeln, da darin bereits

die Unterscheidung von Objekten und deren Member vorhanden ist, die in XML noch fehlt.

Aber auch einige Nachteile haben sich im Laufe der Umsetzung gezeigt. Klassen, die keinen Standardkonstruktor aufweisen, können nicht eingesetzt werden, wovon auch einige Widgets der GWT Bibliothek betroffen sind. Der gewählte Ansatz, das Schema direkt gegen die Klassenbibliothek zu validieren, bringt keine Unterstützung für Editoren in den Bereichen Syntax-Highlighting und Auto-Vervollständigung. Dafür ist er offen für Bibliotheken von Drittherstellern. Es ist aber auch die Präparierung der GWT Bibliothek gwt-user.jar erforderlich, um den Kompiler außerhalb des Hosted Mode laufen zu lassen.

Deklarative Beschreibung von GWT Benutzeroberflächen mit Xaml ist also möglich, aber nicht problemlos.

Kapitel 7

Zusammenfassung

7.1 Zusammenfassung der Arbeit

Die Beschreibung von Benutzerschnittstellen mit Programmiersprachen wie c# oder Java erlaubt unstrukturierten Quelltext und die Vermischung von Oberflächenelementen und Darstellungslogik. Verwendet man hingegen deklarative Sprachen für die Beschreibung einer grafischen Oberfläche, so spiegelt sich die Struktur der Oberfläche im Quelltext wieder. Darüberhinaus wird die Trennung von Grafik und Logik forciert.

Es gibt unterschiedliche solche deklarative Sprachen, von denen einige auf XML basieren. Zu nennen sind hier unter anderem Xaml, XUL und MXML. Xaml unterscheidet sich von den beiden anderen durch seine Orientierung an einer Objektstruktur.

Xaml wurde zwar im Rahmen der WPF entwickelt, kann aber dazu verwendet werden, Objektgraphen jeglicher Art abzubilden. Um diese Aufgabe erfüllen zu können, bildet Xaml die Eigenschaften von Objekten ab. Umgekehrt ergibt sich daraus die Forderung an die Objekte, ihre Eigenschaften preis zu geben. Als Datenstruktur verwendet Xaml ein eigenes Konzept, Xaml Information Set genannt, aus Object, Member und Text Nodes. Dieses ist zwar an das von XML angelehnt, stellt aber doch eine deutliche Veränderung gegenüber einem XML Information Set dar.

Der praktische Teil der Arbeit befasst sich mit dem Zusammenspiel von Xaml und GWT. GWT ist ein Framework, das es erlaubt, AJAX Applikationen komplett in Java

zu programmieren. Eine graphische Oberfläche wird mit Panels und Widgets beschrieben. Auf Ereignisse wird mit EventListenern reagiert, die an diese Panels und Widgets angefügt werden können. Während der Entwicklung kann sich der Programmierer das Ergebnis seines Quelltextes im so genannten Hosted Browser ansehen. Dabei wird ein Browser in einer verwalteten Umgebung gestartet, der die Grafik direkt aus den Java Dateien erstellen kann. Für das Deployment einer GWT Applikation werden die Java Dateien zu JavaScript kompiliert. Dazu hat Google Teile der Java Klassenbibliothek in JavaScript nachgebildet. Dieses Feature gehört zu den Hauptmerkmalen von GWT.

In Java ist es üblich, Eigenschaften hinter Getter und Setter Methoden zu verstecken. Das widerspricht dem zuvor erwähnten Ansatz von Xaml, Objekte über ihre Eigenschaften abzubilden. Weitere Unstimmigkeiten ergeben sich bei Xamls Forderung nach Standardkonstruktoren. Die GWT Klassenbibliothek weist einige Widgets auf, die keinen parameterlosen Konstruktor aufweisen. Objekte dieser Klassen können mit Xaml nicht abgebildet werden. GWT zeigt auch bei Widgets innerhalb eines Panels ein anderes Verhalten als Xaml. Xaml bildet mehrere Objekte, die sich auf der selben Ebene befinden, in Auflistungstypen ab. GWT tut das zwar auch, diese Listen sind aber hinter Methoden zum Anfügen von Listenelementen verborgen. Schließlich behandelt Java Ereignisse mit EventListenern in inneren Klassen. Xaml hingegen möchte Methodennamen an Eigenschaften zuweisen.

Im Rahmen einer Implementierung wurde versucht Xaml und GWT auf praktikable Weise zusammen zu führen. Dabei hat sich folgender Ablauf für das Einbinden der Xaml Datei als sinnvoll herausgestellt: Zuerst wird die Xaml Datei mit einem XML Parser in einen DOM Baum eingelesen. Danach erfolgt eine Konvertierung zu einem Xaml Schema Information Set. Aus diesem wird schließlich Java Quelltext generiert. Dieser Quelltext enthält die mit Xaml beschriebene Oberfläche und kann in weiterer Folge von GWT in den Hosted Mode eingebunden bzw. vom GWT Kompiler zu JavaScript umgewandelt werden. Die Umwandlung des XML DOM Baumes in ein Xaml Schema Information Set folgt dafür erstellten Regeln der Xaml Spezifikation.

Für die Validierung der Xaml Datei bzw. zum Erstellen des nötigen Xaml Schemas wurden zwei Möglichkeiten in Betracht gezogen. Zum einen kann mit XML Schema die GWT Klassenbibliothek modelliert werden. Zum anderen kann die Schemainformation

direkt aus der Klassenbibliothek gewonnen werden. Für die Umsetzung wurde der zweite Ansatz gewählt. Dazu muss jedoch die GWT Klassenbibliothek präpariert werden. GWT Widgets werden durch die statische Methode GWT.create instantiiert. Diese Methode prüft, ob sich das erstellte Objekt im Hosted Mode befindet. Ist dies nicht der Fall, so tritt ein Fehler auf. Es muss also eine Version der GWT Klassenbibliothek erstellt werden, die keine Aufrufe von GWT.create enthält.

Es konnte anhand einer einfachen graphischen Benutzeroberfläche gezeigt werden, dass Xaml und GWT zusammen einsetzbar sind. Verwendet man ein Buildtool wie zum Beispiel Ant, so ist auch der Aufwand zur Quelltexterstellung aus der Xaml Datei gering.

Bei der Implementierung konnten die beschriebenen Unstimmigkeiten zwischen GWT und Xaml für die Bereiche Events und Anfügen von Widgets an Panels gelöst werden. Widgets ohne Standardkonstruktor können jedoch nicht verwendet werden. Darüber hinaus können bestimmte für Xaml benötigte Informationen, wie die Inhaltseigenschaft eines Widgets, nicht aus der Klassenbibliothek abgeleitet werden. Die entwickelte Lösung ist also in dieser Form noch nicht reif für die Praxis.

7.2 Ausblick

Damit Xaml fixer Bestandteil der Arbeit von GWT Entwicklern werden kann, müssten einige Veränderungen vorgenommen werden. Eine tiefere Integration von Xaml in GWT könnte dabei einige Probleme lösen. Wenn alle Widgets Standardkonstruktoren erhalten, stehen sie sämtlich für Xaml zu Verfügung. Eigenschaften, die jetzt durch den Konstruktor gesetzt werden, würden dann über Setter definiert. Das Vermeiden von mehreren Parametern in Methoden, die Widgets an Panels anfügen, würde zu einer klareren Struktur des Xaml-GWT Kompilers führen. Die dort übergebenen Parameter können in den allermeisten Fällen auch nachträglich gesetzt werden. Durch eine Adaption der GWT.create Methode könnte das instantiieren von GWT Widgets während des Kompilierens einer Xaml Datei erlaubt werden. Dadurch würde das Präparieren der GWT Klassenbibliothek entfallen. Wird GWT in diesem Maße umgebaut, so können

auch gleich für Xaml wichtige Informationen einfließen. So wäre es denkbar, die Inhaltseigenschaft jedes Widgets mit einer Annotation zu versehen.

Noch in der Testimplementierung enthalten ist eine Unterstützung für Composites. Diese sowie ein automatisiertes Anlegen der Code-Behind Klasse wären denkbare Erweiterungen. Schließlich fehlt Unterstützung für Editoren oder gar graphische Tools. Möglich wäre hier zum Beispiel das Generieren von XSD aus der GWT Klassenbibliothek. Dadurch würden geeignete Editoren Syntax-Highlighting und Auto-Vervollständigung für GWT-Xaml Dateien bieten. Und darf man [Rela] glauben, so ist das Erstellen von grafischen Tools zur Oberflächengestaltung einfacher, wenn Xaml als Schnittstelle dient. Vielleicht kann hier aber auch eine Brücke zu Microsofts Designtools für Xaml geschlagen werden.

Literaturverzeichnis

[Ado] *Flex overview*. Adobe. http://www.adobe.com/products/flex/overview/. – 10.02.2009

[Bac09] Bach, Markus: Mon General - Typgeneralisierung per Eclipse-Plug-in. In: *iX* (2009), Jänner, Nr. 1/2009

[Boj07] Bojanic, Peter: *The Joy of XUL*. Mozilla. developer.mozilla.org/en/The_Joy_of_XUL. Version: September 2007. – 15.01.2009

[Bro08] Brown, Charles E.: *The Essential Guide to Flex 3*. o.O. : Friends of ed, 2008

[Coe04] Coenraets, Christophe: *An overview of MXML: The Flex markup language*. Adobe. www.adobe.com/devnet/flex/articles/paradigm.html. Version: März 2004. – 15.01.2009

[Dea07a] Deakin, Neil: *XUL Tutorial - Adding Event Handlers*. Mozilla. developer.mozilla.org/en/XUL_Tutorial/Adding_Event_Handlers. Version: August 2007. – 11.02.2009

[Dea07b] Deakin, Neil: *XUL Tutorial - Creating a Window*. Mozilla. developer.mozilla.org/en/XUL_Tutorial/Creating_a_Window. Version: August 2007. – 11.02.2009

[Dea07c] Deakin, Neil: *XUL Tutorial - Introduction*. Mozilla. developer.mozilla.org/en/XUL_Tutorial/Introduction. Version: August 2007. – 11.02.2009

[Dea07d] Deakin, Neil: *XUL Tutorial - Introduction to XBL*. Mozilla. developer.mozilla.org/en/XUL_Tutorial/Introduction_to_XBL. Version: August 2007. – 11.02.2009

[Dea07e] Deakin, Neil: *XUL Tutorial - XUL Structure*. Mozilla. developer.mozilla.org/en/XUL_Tutorial/XUL_Structure. Version: August 2007. – 11.02.2009

[Fow03] Fowler, Martin: *Pattern für Enterprise Application-Architekturen*. 1. Bonn : mitp, 2003

[Fri07] Frischalowski, Dirk: *Windows Presentation Foundation : Grafische Oberflächen entwickeln mit .NET 3.0*. München : Addison-Wesley, 2007

[GJSB05] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *The Java Language Specification*. 3. München : Addison-Wesley, 2005 java.sun.com/docs/books/jls/download/langspec-3.0.pdf

[Glo07] Glover, Andrew: *Unit testing Ajax applications*. IBM. www.ibm.com/developerworks/java/library/j-cq07247/index.html. Version: Juli 2007. – 09.12.2008

[GWT07a] *What's with all the cache/nocache stuff and weird filenames?* Google. code.google.com/support/bin/answer.py?answer=77858&topic=13006. Version: 2007. – 19.01.2009

[GWT07b] *Google Web Toolkit Terms and Conditions*. Google. code.google.com/intl/de-DE/webtoolkit/terms.html. Version: 23.02.2007. – 10.01.2009

[GWT08a] Google (Hrsg.): *GWT 1.5 API Reference*. Google, 2008. google-web-toolkit.googlecode.com/svn/javadoc/1.5/index.html. – 20.01.2009

[GWT08b] *What's ahead for Google Web Toolkit*. Google. googlewebtoolkit.blogspot.com/2008/12/whats-ahead-for-google-web-toolkit_10.html. Version: Dezember 2008. – 17.12.2008

[GWT08c] *Compile Script*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideModuleCompileScript. Version: 2008. – 19.01.2009

[GWT08d] *Creating Custom Widgets*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideCreatingCustomWidgets. Version: 2008. – 19.01.2009

[GWT08e] *Deferred Binding*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideDeferredBinding. Version: 2008. – 19.01.2009

[GWT08f] *Defining a module: format of module XML files*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideModuleXml. Version: 2008. – 16.01.2009

[GWT08g] *Deployment in Web Mode*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideWebMode. Version: 2008. – 19.01.2009

[GWT08h] *Directories/Package conventions*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideDirectoriesPackageConventions. Version: 2008. – 12.01.2009

[GWT08i] *Events and Listeners*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideEventsAndListeners. Version: 2008. – 19.01.2009

[GWT08j] *Filtering Public and Source Packages*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuidePublicPackageFiltering. Version: 2008. – 16.01.2009

[GWT08k] *Hosted Mode Shell Script*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideModuleHostedModeScript. Version: 2008. – 16.01.2009

[GWT08l] *Debugging in Hosted Mode*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideHostedMode. Version: 2008. – 16.01.2009

[GWT08m] *Making HTTP requests*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideHttpRequests. Version: 2008. – 04.01.2009

[GWT08n] *JRE Emulation*. Google. code.google.com/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=RefJreEmulation. Version: 2008. – 26.11.2008

[GWT08o] *Working with JSON*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideJSON. Version: 2008. – 11.01.2009

[GWT08p] *Moduls*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideModules. Version: 2008. – 16.01.2009

[GWT08q] *Product Overview*. Google. code.google.com/intl/de-DE/webtoolkit/overview.html. Version: 2008. – 22.12.2008

[GWT08r] *Google Web Toolkit Release Archive*. Google. code.google.com/intl/de-DE/webtoolkit/versions.html. Version: 2008. – 10.01.2009

[GWT08s] *Automatic Resource Inclusion*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideAutomaticResourceInjection. Version: 2008. – 16.01.2009

[GWT08t] *RPC Plumbing Diagram*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuidePlumbingDiagram. Version: 2008. – 05.01.2009

[GWT08u] *Command-line Tools*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideCommandLineTools. Version: 2008. – 14.01.2009

[GWT08v] *Working with XML*. Google. code.google.com/intl/de-DE/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideXML. Version: 2008. – 11.01.2009

[Hei08a] *Neue Version des Google Web Toolkit angekündigt*. Heise-Online. www.heise.de/newsticker/Neue-Version-des-Google-Web-Toolkit-angekuendigt--/meldung/120313. Version: 12.12.2008. – 12.12.2008

[Hei08b] *Microsoft bringt neue Programmiersprache M*. Heise-Online. www.heise.de/newsticker/Microsoft-bringt-neue-Programmiersprache-M--/meldung/117263. Version: 13.10.2008. – 03.11.2008

[Hei08c] *PDC: Microsoft konkretisiert Modellierung mit "Oslo"*. Heise-Online. www.heise.de/newsticker/PDC-Microsoft-konkretisiert-Modellierung-mit-Oslo--/meldung/118134. Version: 29.10.2008. – 03.11.2008

[Hol01] Holzner, Steven: *Insider XML*. München : Markt+Technik, 2001

[HT07] Hanson, Robert ; Tracy, Adam: *GWT in action: easy Ajax with the Google Web Toolkit*. Greenwich, Conn. : Manning, 2007

[JAX08] *Java API for XML Processing (JAXP) Tutorial - Chapter 1*. http://java.sun.com/webservices/reference/tutorials/jaxp/html/intro.html. Version: Juli 2008. – 23.12.2008

[Joh08] Johnson, Bruce: *GWT 1.5 Now Available*. Google. googlewebtoolkit.blogspot.com/2008/08/gwt-15-now-available.html. Version: August 2008. – 22.12.2008

[Küh08] KÜHNEL, Andreas: *Visual C# 2008: Das umfassende Handbuch*. 4. Bonn : Galileo Press, 2008

[Mac06] MACVITTIE, Lori A.: *XAML in a Nutshell*. 1. O'Reilly Media, 2006 my.safaribooksonline.com/0596526733

[MS:08] MICROSOFT (Hrsg.): *Xaml Object Mapping Specification 2006*. 1.0. Microsoft, Juni 2008. www.microsoft.com/downloads/details.aspx?FamilyID=52a193d1-d14f-4335-aa86-c53193e1885d&DisplayLang=en. – 13.01.2009

[MSD07a] *XAML und benutzerdefinierte Klassen*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/ms753379.aspx. Version: November 2007. – 05.12.2008

[MSD07b] *Code-Behind und XAML*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/aa970568.aspx. Version: November 2007. – 03.12.2008

[MSD07c] *DockPanel.Dock (angefügte Eigenschaft)*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/system.windows.controls.dockpanel.dock.aspx. Version: November 2007. – 09.11.2008

[MSD07d] *Gewusst wie: Veröffentlichen von Ereignissen, die den .NET Framework-Richtlinien entsprechen (C#-Programmierhandbuch)*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/w369ty8x.aspx. Version: November 2007. – 11.02.2009

[MSD07e] *ContentPropertyAttribute-Klasse*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/system.windows.markup.contentpropertyattribute.aspx. Version: November 2007. – 05.12.2008

[MSD07f] *MSBuild-Referenz*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/0k6kkbsd.aspx. Version: November 2007. – 01.12.2008

[MSD07g] *Partielle Klassen und Methoden (C#-Programmierhandbuch)*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/wa80x488.aspx. Version: November 2007. – 02.12.2008

[MSD07h] *MarkupExtension.ProvideValue-Methode*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/system.windows.markup.markupextension.providevalue.aspx. Version: November 2007. – 08.12.2008

[MSD07i] *Terminologie der XAML-Syntax*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/ms788723.aspx. Version: November 2007. – 08.11.2008

[MSD07j] *TypeConverter und XAML*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/aa970913.aspx. Version: November 2007. – 06.12.2008

[MSD07k] *Verwenden von Workflowmarkup*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/ms735921.aspx. Version: November 2007. – 03.11.2008

[MSD07l] *WPF-Architektur*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/ms750441.aspx. Version: November 2007. – 05.12.2008

[MSD07m] *Übersicht über XAML*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/ms752059.aspx. Version: November 2007. – 23.10.2008

[MSD08] *Erstellen einer WPF-Anwendung (WPF)*. Microsoft Developer Network. msdn.microsoft.com/de-de/library/aa970678.aspx. Version: Juli 2008. – 01.12.2008

[Nee08] Neethling, Schalk: *Understanding the GWT compiler*. java.dzone.com/news/understanding-gwt-compiler. Version: Juni 2008. – 19.01.2009

[Ram08] Raman, T.V.: *ARIA For GWT: My health Is feeling accessible*. Google. googlewebtoolkit.blogspot.com/2008/10/by-t.html. Version: Oktober 2008. – 15.10.2008

[Rela] Relyea, Rob: *Acropolis uses Xaml to define business logic components*. rrelyea.spaces.live.com/blog/cns!167AD7A5AB58D5FE!2056.entry. – 21.10.2008

[Relb] Relyea, Rob: *Case Sensitivity of Xaml*. rrelyea.spaces.live.com/blog/cns!167AD7A5AB58D5FE!2055.entry. – 22.10.2008

[Rel06] Relyea, Rob: *Intellisense in XAML for your types too!* rrelyea.spaces.live.com/blog/cns!167AD7A5AB58D5FE!695.entry. Version: September 2006. – 16.11.2008

[Sch07] Schwichtenberg, Holger: Handhabungssache. In: *iX* (2007), Nr. 3/2007

[SH07] Stropek, Rainer ; Huber, Karin: *XAML*. 1. Frankfurt/M. : Entwickler.press, 2007

[Vol07] Volkmann, Mark: *Google Web Toolkit (GWT)*. Object Computing, Inc. www.ociweb.com/mark/programming/GWT.html. Version: 2007. – 07.01.2009

[W3C00] *Document Object Model Events*. W3C. www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113/events.html. Version: 13.11.2000. – 20.01.2009

[W3C04] *XML Information Set*. W3C. www.w3.org/TR/xml-infoset/. Version: 2, Februar 2004. – 05.02.2009

[WJ08] Webber, Joel ; Johnson, Bruce: *Explicit DOM Classes in GWT*. Google. code.google.com/p/google-web-toolkit/wiki/DomClassHierarchyDesign. Version: März 2008. – 06.01.2009

[XML04a] *XML Schema Part 2: Datatypes Second Edition*. W3C. www.w3.org/TR/xmlschema-2/. Version: 28.04.2004. – 24.11.2008

[XML04b] *XML Schema Part 0: Primer Second Edition*. W3C. www.w3.org/TR/xmlschema-0/. Version: Oktober 2004. – 06.02.2009

[XML06] *Namespaces in XML 1.0 (Second Edition)*. W3C. www.w3.org/TR/REC-xml-names/. Version: 16.08.2006. – 05.11.2008

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Anhang A

Beiliegende CD

Beiliegend befindet sich eine CD, welche die Ergebnisse des praktischen Teils dieser Arbeit sowie alle verwendeten Online-Quellen enthält.

A.1 GWT-Xaml Kompiler

Um die GWT-Beispielapplikation zu verwenden, und Xaml Dateien zu Java-Quelltext kompilieren zu können, müssen Sie das Java SDK installieren. Für die Entwicklung wurde der JDK in der Version 6 Update 11 verwendet. Für Windows-Plattformen befindet sich der JDK auf dieser CD. JDKs für andere Plattformen können von der Website http://java.sun.com/ bezogen werden.

Sowohl die Dateien des Xaml-GWT Kompilers als auch eine GWT-Beispielapplikation befinden sich im Verzeichnis 'XAML2GWT'. Um die Beispielapplikation zu starten, verwenden Sie die Kommandozeilenskripte im Verzeichnis der Applikation 'GwtBspApp'. Diese Skripte verwenden die GWT-Bibliotheken aus dem Verzeichnis 'gwt-windows-1.5.3', welches sich ebenfalls im Verzeichnis 'XAML2GWT' befindet. Zum Kompilieren der Xaml Dateien können Sie die Ant-Datei aus dem Verzeichnis der Beispielapplikation benutzten. Die Quelltexte des GWT-Xaml Kompilers befinden sich im Verzeichnis 'XAML2GWT/Xaml2Gwt'. Die zugehörige Javadoc liegt im Unterverzeichnis 'dist/javadoc'.

A.2 Online-Quellen

Die in der Arbeit verwendeten Online-Quellen finden sich im Verzeichnis 'online-quellen'. Für jede Quelle befindet sich in diesem Verzeichnis ein Unterverzeichnis das auf den Namen des Kürzels aus dem Literaturverzeichnis der Diplomarbeit lautet.