Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MICHAEL SANDNER

MASTER THESIS

ENTWURF UND IMPLEMENTIERUNG EINER DSL FÜR BENUTZUNGSSCHNITTSTELLEN

Eingereicht am 19. Februar 2016

Betreuer:
Dipl.-Inf. Hannes Dohrn
Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 19. Februar 2016

License

Erlangen, 19. Februar 2016

Abstract

This master thesis designs and implements a domain specific language (dsl) for the sweble software. The domain of the dsl is the creation of user interfaces through domain experts without a specific qualification in computer science (so-called End-user-programmer). The main features of the dsl are functionality and layout of user interfaces, definition and preservation of constraints and the transformation in one or more domain objects. Therefore a specific syntax will be defined, a parser generator with an additional manually extension for the creation of an abstract syntax tree (AST) will be developed and an interpreter will be implemented which will process the AST and manipulate the wom tree which is the underlying data structure.

Zusammenfassung

Diese Masterarbeit entwirft und implementiert eine domänenspezifische Sprache (DSL) für die Sweble Software. Es handelt sich um eine DSL zur Erstellung von Benutzungsschnittstellen durch Fachexperten ohne spezifische Informatikerausbildung (sog. End-user-Programmierer). Wesentliche Eigenschaften der DSL sind Funktionalität für das Layout von Benutzungsschnittstellen, die Definition und Wahrung von Constraints, und die Abbildung in ein oder mehrere Domänenobjekte. Hierzu wird eine konkrete Syntax definiert, ein Parser-Generator mit zusätzlicher händischer Erweiterung zur Erzeugung eines abstrakten Syntaxbaumes (AST) entwickelt und ein Interpreter implementiert, welcher den AST ausführt und den WOM-Baum, die zugrundeliegende Datenstruktur, gemäß Quelltext manipuliert.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Heutzutage existieren im Internet sehr viele Wiki-Systeme um Wissen zu sammeln und zu vermitteln. Die Benutzung solcher Systeme ist beliebt, da es für jeden vom Browser aus möglich ist Inhalte zu verändern. Das wohl bekannteste davon ist Wikipedia mit MediaWiki als Wiki System.

> "By now the MediaWiki parser has become a complex software. Unfortunately, it converts Wikitext directly into HTML, so no higher-level representation is ever generated by the parser"(Dohrn & Riehle, 2011a, S. 2)

Als Folge dessen wird mit dem Sweble Wiki eine Art MediaWiki von der Open Research Group an der Friedrich-Alexander-Universität Erlangen-Nürnberg entwickelt, welches das Wiki Object Model (WOM) Format verwendet. Das Sweble Wiki soll um eine Funktion erweitert werden, die es den Benutzern ermöglicht mit einer Domain Specific Language (DSL) schnell und einfach Formulare zu erstellen und den Inhalt der ausgefüllten Inhalte zu speichern. Formulare dienen der Interaktion zwischen einem Anwender und einer Webseite. Sie sind ein wichtiger Bestandteil des World Wide Webs.

Ziel der Arbeit ist der Entwurf und die Implementierung einer "Sweble Form DSL". Zu den einzelnen Zielen gehört das Begutachten von existierenden DSLs, Rahmenwerken und Bibliotheken für vereinfachte Benutzungsschnittstellenprogrammierung zur Anforderungsermittlung. Aus den Ergebnissen soll eine domänenspezifische Sprache abgeleitet werden. Der Entwurf und die Implementierung von Parser (mittels Parsergenerator) und Interpreter, sowie eine entsprechend dokumentierte Sprachspezifikation sind ebenfalls Ziele dieser Arbeit. Die Funktionalität der Implementierung soll durch Textfälle nachgewiesen werden.

Nach (van Deursen, Klint & Visser, 2000, S. 3) involviert die Entwicklung einer DSL drei Schritte aus denen sich die Arbeit zusammensetzt.

Der erste Schritt der Arbeit befasst sich mit der Analyse der Problemdomäne. Dazu werden bereits existierende DSLs, sowie Rahmenwerke und Bibliotheken für vereinfachte Benutzerschnittstellenprogrammierung ausgewertet. Aus den dar-

aus gewonnen Aspekten werden die Anforderungen für die Sweble Form DSL (FormDSL) ermittelt.

Der zweite Teil befasst sich mit der Ausarbeitung der Anforderungen. Dies beinhaltet die Architektur und die Implementierung der domänenspezifischen Sprache. Des Weiteren wird auf die Integration in das Sweble Wiki und die Weiterentwicklung der Sprache eingegangen.

Im dritten Teil wird die Benutzung der DSL dargestellt und ein Überblick des Funktionsumfangs der entwickelten Sprache in Form einer Sprachspezifikation gegeben.

2 FormDsl

> „A *domain-specific language* (DSL) is a program- ming language or executable specification language that offers, through appropriate notations and ab- stractions, expressive power focused on, and usu- ally restricted to, a particular problem domain.“(van Deursen et al., 2000, S. 1)

Die Hauptaussage dieser Definition ist die fokussierte Ausdrucksstärke. DSLs werden für einen bestimmten Zweck entwickelt und verwendet. Viele davon werden unwissentlich benutzt, weil sie oft zum Tagesgeschäft in den verschiedenen Fachgebieten gehören. Das sogenannte End-User-Programming passiert, wenn Nutzer oder Domänen-Experten ohne größere Informatik-Ausbildung einfache Programmieraufgaben mit Hilfe von DSLs lösen. Beispiele hierfür sind TeX zum Generieren von Dokumenten, Matlab zum Berechnen von mathematischen Problemen oder das Benutzen von Excel-Makros.

Als Alternative zur Implementierung einer DSL von Grund auf, kann eine DSL auch als Erweiterung einer vorhandenen DSL implementiert werden. Der Hauptvorteil dieses Ansatzes ist es, dass alle Funktionen der Basissprache verfügbar sind und nicht neu implementiert werden müssen (van Deursen et al., 2000, S. 3). Im Gegensatz dazu, sind die Ziele einer DSL:

- „to make programming more accessible to end-users
- to improve correctness of the written programs, and
- to improve the program developing time“(Tomaz et al., 2009, S. 1)

Diese Ziele entsprechen dem, was den Sweble-Nutzern geboten werden soll.

Wie jede andere Programmiersprache besitzen domänenspezifische Sprachen eine bestimmte Syntax um eine Semantik auszudrücken. Eine DSL ist, wie in der oben genannten Definition erwähnt, auf einen bestimmten Bereich und auf bestimme Probleme begrenzt. Sie eignet sich nur für eine bestimmte Domäne. Sie besitzt demzufolge einen geringeren Funktionsumfang als eine komplexe Programmiersprache wie z.B. Java.

Eine DSL erlaubt die Darstellung einer Lösung in der Ausdrucksweise und dem Abstraktionslevel der Domäne. Dadurch können Domänen-Experten die Sprache verstehen, validieren, modifizieren und oft sogar DSL-Programme entwickeln (van Deursen et al., 2000). Für das Design der FormDSL steht das Verstehen und die Anwendung der DSL von End-Nutzern im Vordergrund.

Im folgenden Kapitel werden die Anforderungen an die zu entwickelnde FormDSL erarbeitet. Um die Domäne der Benutzungsschnittstellenerzeugung zu untersuchen, werden in Abschnitt 2.1 bereits existierende Sprachen und Bibliotheken analysiert und daraus im darauffolgenden Abschnitt 2.2 die Anforderungen erstellt.

2.1 Related Work

Für die Anforderungen einer Sprache werden die Sprachen in folgenden Punkten untersucht:

- Das Vokabular der Sprachen
- Der Funktionsumfang der Sprachen
- Die Anwendbarkeit der Sprachen
- Die Erlernbarkeit der Sprachen

2.1.1 HTML

Hypertext Markup Language (HTML) ist eine bekannte Sprache wenn es sich um Benutzungsschnittstellen handelt. HTML ist eine XML-basierte Programmiersprache, die es ermöglicht Webseiten zu erstellen, welche von Webbrowsern dargestellt werden. Bei vielen Webseiten im World Wide Web handelt es sich um HTML-Dokumente.

Ein HTML-Dokument ist in Form eines Baumes aufgebaut. Es besteht aus Elementen, welche wiederum Kindelemente und Attribute besitzen können. Die Elemente haben jeweils ein Anfangs- und ein End-Tag. Zwischen diesen beiden Tags wird das jeweilige Element beschrieben. In 2.1 sind zwei Elemente mit ihren beiden Tags zu sehen. Ein `DIV`-Element, dass ein `button`-Element als Kind-Element besitzt. Der Button besitzt wiederum ein type-Attribut.

```
1 <div>
2   <input type="button">Button
3   </input>
```

```
4 </div>
```

Listing 2.1: geschachtelte Elemente in einem HTML-Dokument

Mit dem Schachtelungsprinzip der einzelnen Elemente und dem Zuweisen von Attributen können einfache und komplexe Dokumente erstellt werden. HTML besitzt neben Elementen zur Formularerstellung einen großen Funktionsumfang zum Erstellen und Gestalten von Webseiten. Eine Auflistung kann in (Data, o. J.) gefunden werden. Zu Analysezwecken werden aber nur die relevanten Elemente für Benutzungsschnittstellen weiter betrachtet. Um ein HTML Formular zu erstellen, müssen alle entsprechenden dazugehörigen Elemente innerhalb der Formular-Tags definiert werden.

```
1 <form>
2    ... elements ...
3 </form>
```

Listing 2.2: Formulardefinition in HTML

Die Sprache besitzt viele Elemente zur Interaktion die aus Formularen bekannt sind:[1]

- Eingabe
  - Checkboxen
  - Radiobutton
  - Text
  - Nummer
  - E-Mail
  - Submit-Button
- Auswahlliste
- Textarea
- Button

Diese verschiedenen Form-Elemente sind wichtig, um mit einer Webseite oder einem Dokument interagieren zu können. Auf sie wird in der Sprachspezifikation der FormDSL 4 noch konkreter eingegangen.

Das Layout eines HTML-Dokuments wird mit *DIV*-Elementen und Cascading Style Sheets (CSS), einer weiteren Sprache neben HTML, beschrieben. *DIV*'s

[1] `http://www.w3schools.com/html/html_form_input_types.asp`

sind nichts anderes als Container, welche ein oder mehrere Elemente beherbergen. Die einzelnen *DIV*-Container werden mit CSS-Klassen zueinander angeordnet und beschreiben somit das Layout des Dokumentes. Ein zweispaltiges Seitenlayout kann beispielsweise mit zwei *DIV*-Elementen beschrieben werden, die nebeneinander angeordnet sind. Ein weiterer Container oberhalb der zwei Spalten, der die ganze Breite des Dokumentes einnimmt, kann eine darüber liegende Header-Sektion beschreiben. In den drei Containern können dann beliebige Elemente angeordnet werden. DIV-Container können auch geschachtelt werden, um komplexere Strukturen zu erstellen.

Fazit Der Funktionsumfang von HTML umfasst neben dem Erstellen von Formularen viele weitere Funktionen zur Erstellung und Gestaltung von Webseiten, die Nutzer des Sweble Wikis nicht benötigen. Zudem ist für die Gestaltung der Formulare mit CSS eine weitere Programmiersprache nötig. Die Sweble-Nutzer sollen schnell und einfach Formulare erstellen können, ohne sich über die Gestaltung der Webformulare Gedanken machen zu müssen. Für das Sweble wird demzufolge eine weniger umfangreiche Sprache benötigt, die eine schnelle Erlernbarkeit aufweist.

2.1.2 Frameworks

Für die einfachere und schnellere Erstellung von Webseiten gibt es zahlreiche, auf HTML aufbauende, Frameworks. In den Abschnitten 2.1.2 und 2.1.2 werden Zwei Frameworks kurz analysiert und vorgestellt. Da die Begutachtung aller in Frage kommenden Frameworks den Umfang dieser Arbeit überschreitet, werden zwei bekanntere Werke vorgestellt und darauf hingewiesen, dass es noch weitere Alternativen gibt.

Bootstrap Bootstrap ist ein beliebtes Framework für die Entwicklung mit HTML, CSS und Java Script. Es vereinfacht und verkürzt die Entwicklung für Webprojekte, durch integrierte Design-Templates und Form-Elemente, die für unseren Entwurf einer Programmiersprache wichtig sind. Bootstrap beinhaltet noch weitere Plugins, welche für diese Arbeit aber nicht weiter betrachtet werden.

Das Framework ermöglicht es Formulare zu erstellen, ohne sich vorher mit CSS auseinanderzusetzen. Das geschieht durch die Zuweisung von bereits in Bootstrap integrierte und vorgefertigte Design-Klassen, die den Elementen nur noch zugewiesen werden müssen.

Da Bootstrap auf HTML basiert, existieren dort die gleichen Elemente zur Formularerstellung. Es erweitert nicht den Funktionsumfang von HTML sondern

erleichtert auch die Benutzung z.B für das strukturieren der Elemente.

Das Layout eines mit Bootstrap erstellten Webformulars kann ohne viel Aufwand erstellt werden. Die Elemente könnten beispielsweise einfach untereinander angeordnet werden, wie in einem Stackpanel 2.1. Hierzu werden diese in einem *DIV*-Container platziert und untereinander angeordnet. Die Ausrichtung des Stapels kann auch horizontal statt vertikal erfolgen. Es besteht aber auch die Möglichkeit mit Hilfe eines `grids` die Elemente innerhalb eines Gitters in Reihen und Spalten anzuordnen. 2.3 zeigt die Definition eines Gitters mit Bootstrap. Dabei werden dem Attribut `class` vordefinierte CSS-Klassen zugeordnet, welche Reihen sowie Spaltenbreiten definieren. Alle Spalten einer Reihe müssen immer zusammen zwölf ergeben, wobei `col-*-*` die Spaltenbreite mit dem restlichen Platz einer Reihe auffüllt.

```
1  <div class="row">
2    <div class="col-sm-4"></div>
3  <div class="col-sm-8"></div>
4  <div class="col-*-*"></div>
5  </div>
```

Listing 2.3: Bootrap Beispiel

[2] Der Funktionsumfang der Layout-Funktionen bietet auch eine passende Skalierung für mobile Endgeräte, auf die aber nicht weiter eingegangen wird.

Foundation Foundation bietet als weiteres untersuchtes Framework ebenfalls eine gute Dokumentation und eine große Community [3]. Der Fokus von Foundation liegt bei der schnellen und einfachen Erzeugung von Webseiten, welche auf Geräten mit unterschiedlichen Auflösungen eine intuitive Nutzung erlauben. Genau wie Bootrap wird das Layout der Elemente anhand von einem `Grid` und `DIV`-Containern festgelegt. Listing 2.4 zeigt ein Beispiel zur Erstellung eines zweispaltigen Layouts mit Foundation. Die Größe der Spalten geschieht durch die Zuweisung von vordefinierten CSS-Klassen an das Attribut `class`. Genauso wie in Bootstrap ist die maximale Spaltenanzahl zwölf.

```
1 <div class="row">
2     <div class="eight columns">
3         <p>linke Spalte</p>
4     </div>
5     <div class="four columns">
6         <p>rechte Spalte</p>
```

[2] `http://www.w3schools.com/bootstrap/bootstrap_grid_system.asp`
[3] `http://foundation.zurb.com/sites.html`

```
7       </div>
8   </div>
```

Listing 2.4: Foundation Beispiel

Mit Formulare-Elementen verhält sich Foundation ähnlich wie andere Frameworks [4].

Fazit Im Bezug der Formularerstellung unterscheiden sich die Frameworks nur in kleinen Punkten, wie in den höheren Gestaltungsmöglichkeiten von Grids in Bootstrap. Die Hauptunterschiede liegen in anderen Punkten wie der Navigation und die intuitive Bedienbarkeit auf allen Geräten, welche für das Sweble Wiki nicht ausschlaggebend sind. Die einfache Erlernbarkeit und Handhabung der beiden Frameworks machen diese, trotz des großen Funktionsumfanges, zu möglichen Technologien für die Formularprogrammierung durch End-Nutzern. Durch den hohen Funktionsumfang, im Bezug auf das Erstellen von Benutzungsschnittstellen, eignen sie sich nicht direkt als DSL für das Sweble. Sie erweisen sich allerdings als Alternativen zur eigentlichen Formulargenerierung, die nach der Definition eines DSL-Nutzers, von der Sprache geleistet wird.

2.1.3 XAML

Eine weitere auf XML basierende Programmiersprache ist die von Microsoft entwickelte Extensible Application Markup Language (XAML). Sie dient wie HTML der Oberflächengestaltung und findet Anwendung in WPF- und Silverlight-Anwendungen.

XAML steht im Kontext dieser Arbeit beispielhaft für DSLs, die zur Entwicklung von Desktopanwendungen verwendet werden. Die sich dort ergebenden Möglichkeiten werden ebenfalls für die Anforderungsanalyse betrachtet.

Die einzelnen Elemente in XAML besitzen die für XML typischen Anfangs- und End-tags. Kind-Elemente und Attribute sind ebenfalls, wie in beiden vorhergehenden Sprachen, möglich.

Layout

Für das Erstellen eines passenden Layouts gibt es in XAML viele Möglichkeiten. Die zu strukturierenden Elemente werden in einem umschließenden Panel angeordnet. Je nach Panel ergeben sich unterschiedliche Möglichkeiten zum Anordnen der beherbergten Elemente. Zum einen existiert die Möglichkeit, wie bei

[4] `http://foundation.zurb.com/sites/docs/forms.html`

Bootstrap, die Elemente innerhalb eines Gitters (Grid) auszurichten. Weitere Möglichkeiten sind das *Canvas*, das *Stackpanel* oder das *Dockpanel*.

Stackpanel In einem Stackpanel werden alle Elemente wie bei einem Stapel nacheinander aufgebaut. In welcher Richtung die Elemente angeordnet werden, muss dabei dem Stackpanel mitgeteilt werden. So gibt es die Möglichkeit die Kindelemente vertikal oder horizontal in einem Stack anzuordnen. Bei vertikaler Ausrichtung können die Elemente von unten nach oben oder umgekehrt angeordnet werden. Bei horizontaler Ausrichtung von rechts nach links oder andersherum. In 2.1 werden die Kindelemente im vertikalen Stackpanel von unten nach oben und im horizontalen Stackpanel von rechts nach links angeordnet.

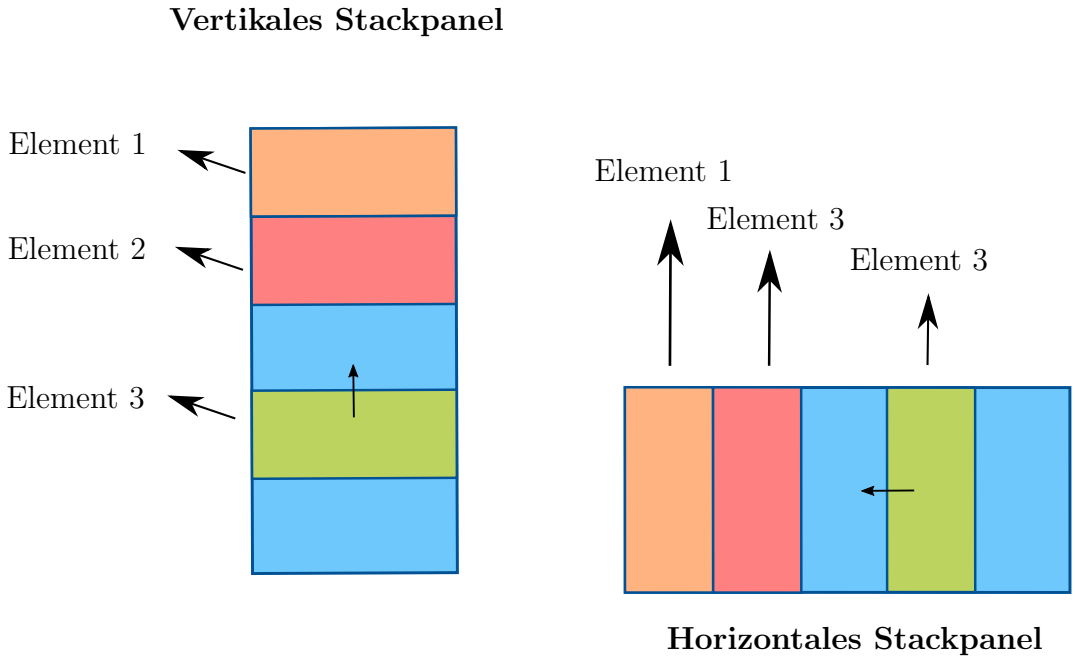

Abbildung 2.1: Vertikales und horizontales Stackpanel

Dockpanel Eine weitere Möglichkeit ist das Dockpanel. Im Gegensatz zum Stackpanel wird nicht alles der Reihe nach angeordnet, sondern es steht dem Benutzer bei jedem Element frei, in welche Richtung er das Element anheften will. Ihm stehen hierbei vier Richtungen zur Verfügung. Er kann das Element entweder rechts, links oben oder unten im zur Verfügung stehenden Bereich des Panels anheften. Ist ein Kind im Dockpanel angebracht, steht für das nächste Kind nur noch ein kleinerer Teil des Dockpanels zur Verfügung, welcher nicht von

den vorherigen Kindern eingenommen wird. Abbildung 2.2 zeigt die Füllung eines Dockpanels mit zwei Elemente. Das ursprüngliche Dockpanel besitzt einen blauen Rahmen. In Schritt eins auf der linken Seite wird das in hellgrün dargestellte neue `Element 1` unten im Dockpanel angeordnet. Der rote Rahmen beschreibt den Bereich, der den restlichen Elementen noch zur Verfügung steht. In Schritt zwei wird ein weiteres neues `Element 2` auf der rechten Seite angeordnet. Der kleiner werdende restliche Bereich wird wieder mit einem roten Rahmen abgebildet.

In jedem Element muss demzufolge angegeben werden, an welcher Seite des Dockpanels es angedockt werden soll.

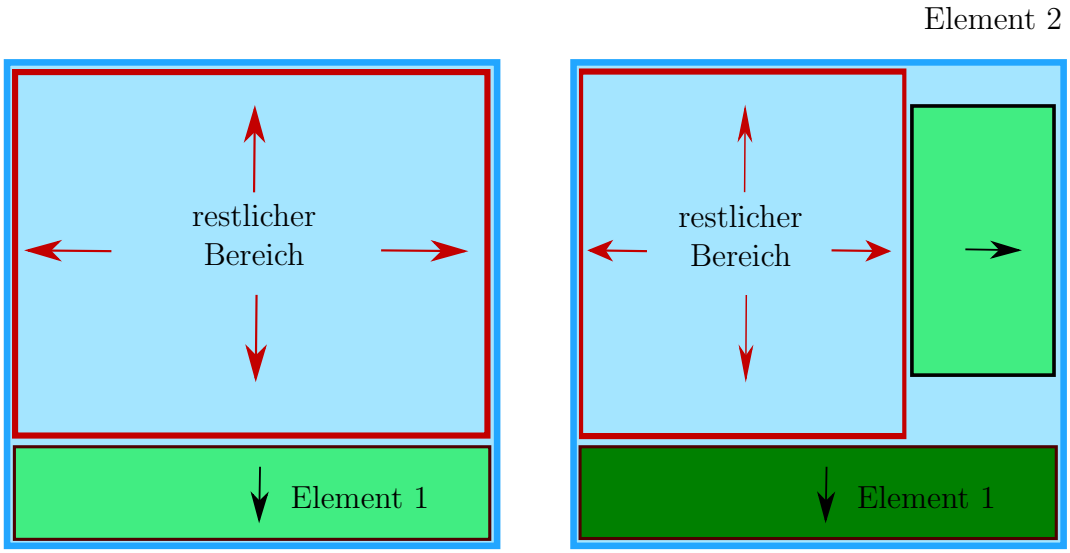

Abbildung 2.2: Dockpanel

Canvas In einem Canvas können die Elemente völlig frei platziert werden. Hierzu werden den Elementen ein fester Abstand zu den Rändern zugeordnet, welcher dann den Elementen einen festen Platz im Canvas zuordnet.

Fazit XAML ist als Teil von Microsoft-Produkten nicht für die Integration in das Sweble Wiki geeignet. Die Sprache bietet allerdings verschiedenen Möglichkeiten für die Anordnung von Elementen, welche für das Design einer DSL aufgegriffen werden können.

2.1.4 „Good Form“ DSL

Good Form ist ein Grails Plugin, das es ermöglicht leicht zu nutzende, gut aussehende, komplexe Web Formen zu erstellen. Der Fokus in Good Form liegt auf

einem durchdachten logischen Ablauf. Wird beispielsweise die Frage nach dem Besitz eines Hundes gestellt, könnte die Frage nach der Hundelizenz als nächstes abgefragt werden. In dieser Sprache werden die Elemente der Formulare als Fragen gestellt. Zu einer Frage gehören jeweils eine Frage und die zugehörige Antwortmöglichkeit als interaktives Elemente des Webformulars. Die einzelnen Fragen sind in der Programmiersprache mit Hilfe einer *group* verschachtelbar. Dies ermöglicht die Zuordnung von Fragen zu einer übergeordneten Frage.

2.5 zeigt die Erstellung einer Webform mithilfe der Sprache Good Form. Zeile 1 definiert eine Frage. In Zeile 2 wird eine Frage erstellt, die als Gruppe definiert wird. In den Zeilen 3, 4 und 5 werden weitere Fragen erstellt, welche Texteingaben mit limitierter Zeichenanzahl besitzen. Zeile 6 definiert eine Frage mit einer Checkbox als Antwortmöglichkeit. Die untergeordneten Elemente der Frage, die wie bei einer Gruppe in geschweiften Klammern definiert sind, werden nur bei aktivierter Checkboxauswahl angezeigt. Zeile 7 definiert eine Frage mit einer Liste als Formelement. In die Liste kann der Benutzer beliebig viele Einträge mit den vorgegebenen Textfeldern hinzufügen.

```
1  question("G2") {
2      "What is your name?"
3    group: "names", hint: "Name of the person requiring
         assistance", {
4          "Title" text: 10, hint: "e.g. Mr, Mrs, Ms, Miss,
             Dr", suggest: "title", map: 'title'
5          "Given Names" text: 50, required: true, map: '
             givenNames'
6          "Last or Family Name" text: 50, required: true,
             map: 'lastName'
7          "Have you been or are you known by any other names
             ?" hint: "e.g. maiden name, previous married
             name, alias, name at birth", map: '
             knownByOtherNames',{
8          "List your other names" listOf: "aliases", {
9                  "Other name" text: 50, map: 'otherName'
10                 "Type of name" text: 40, hint: "e.g maiden
                       name", suggest: "nameType", map: '
                     otherNameType'
11             }
12         }
13     }
14 }
```

Listing 2.5: Good Form Beispiel (http://nerderg.com/GoodForm+DSL)

Good Form ermöglicht viele Elemente für seine Webformulare [5]. Die möglichen Form-Elemente werden aufgezählt und die wichtigsten Eigenschaften beschrieben. Bei jedem Element ist es möglich einen Anzeigentext und einen Hinweis *hint* anzugeben.

- **text**
  Definiert ein Textfeld, dessen mögliche Zeichenanzahl begrenzt werden kann.
- **date**
  Definiert ein Datum-Auswahlfeld. Ein maximales und ein minimales Datum können definiert werden.
- **datetime**
  Definiert ein Datum und Zeit-Auswahlfeld.
- **bool**
  Definiert eine Ja-Nein-Antwortmöglichkeit in Form einer Checkbox.
- **pick** Definiert eine Gruppe von Ja-Nein-Antwortmöglichkeiten. Es kann zwischen Radiobuttons und Checkboxen gewählt werden.
- **group** Definiert eine Menge von Elementen zu einer Gruppe.
- **listOf**
  Definiert eine Menge von Formelementen, die beliebig oft hinzugefügt werden können.
- **money**
  Definiert ein numerisches Textfeld. Die Anzahl an Ziffern kann gewählt werden.
- **number**
  Definiert ein numerisches Textfeld. Die Anzahl an Ziffern kann gewählt werden. Ein Auswahlbereich der möglichen Nummern kann ebenfalls gewählt werden.
- **phone**
  Definiert ein numerisches Textfeld. Die Anzahl an Ziffern kann gewählt werden.
- **attachment**
  Definiert ein Upload-Feld.
- **each**
  Definiert eine Menge von Formelementen, die sich für jedes Item einer Liste wiederholen.

[5] `http://nerderg.com/GoodForm+DSL`

- **heading**
  Definiert eine Überschrift.
- **select**
  Definiert eine Dropdown-Liste und deren Auswahlmöglichkeiten. Eines der Auswahlmöglichkeiten kann voreingestellt angegeben werden.

Das Layout bei dieser Sprache ist fest vorgegeben. Mit Good Form können nur die einzelnen Elemente des Webformulars definiert werden. Diese als Fragen definierten Formelemente bestehen aus der Frage und dem dazu gehörenden Antwortfeld. Die Fragen werden alle untereinander, wie in 2.1.3 beschrieben, in einem festen Layout angeordnet und bilden damit das Webformular.

2.1.5 Jim Duey

Jim Duey beschreibt in seinem Blog eine Sprache „A Dsl For Web Forms“(Duey, o. J.), in denen die Inhalte der Form direkt durch Datenbankinhalte gefüllt werden. Sie unterscheidet sich in ihrem Syntax der bisher analysierten Sprachen.

Die Idee der Sprache besteht darin, dass man HTML ähnlichen Code schreibt, in diesen aber beschreibende Wörter, dort platziert, wo Daten eingefügt werden sollen (Duey, o. J.).

In Listing 2.6 wird eine Form *person* mit zwei DIV-Containern beschrieben, die jeweils ein Label und ein Textfeld enthalten. Die Form ist mit einer Datenbanktabelle verlinkt, welche die zwei Spalten *first_name* und *last_name* besitzt. Für die Generierung soll eine Funktion existieren, welche die Textfelder mit den Daten aus der Datenbank vorausgefüllt rendert.

```
1 (defform person
2    (div
3       (label :first_name "First Name:")
4       (string :first_name))
5    (div
6       (label :last_name "Last Name:")
7       (string :last_name)))
```

Listing 2.6: Codebeispiel zum Erzeugen von HTML mit eingefügten Datenbankinhalten (Duey, o. J.)

Fazit Auf den ersten Blick entspricht der Ansatz von Jim Duey's DSL zur Generierung vorgefüllter Formulare nicht den Anforderungen der FormDSL. Benutzt man die Idee der Definition von Datenfeldern zur Erzeugung von vor ausgefüllten

Datenstrukturen anstelle von Formularen, kann diese für das Design der zu entwickelnden Sprache aufgegriffen werden.

2.2 Anforderungen

Die DSL für das Sweble Wiki soll folgende Funktionen besitzen:

- Definition von Benutzungsschnittstellen
- Definition von Constraints
- Definition einer Datenstruktur

Keine der im vorhergehenden Abschnitt analysierten Sprachen und Bibliotheken bietet den komplett benötigten Funktionsumgang.

Um alle diese Punkte zu erfüllen, wird im folgenden Teil der Arbeit mit der „Sweble Form DSL“eine Sprache entwickelt, die den geforderten Funktionsumfang besitzt und trotzdem für den Sweble-Nutzer als Fachexperten, schnell und einfach anwendbar ist.

Um die Sweble FormDSL zu entwerfen, werden in den folgenden Abschnitten konkrete funktionale Anforderungen aus den in 2.1 analysierten Sprachen ermittelt.

2.2.1 Formular-Elemente

Die analysierten Sprachen besitzen zum Großteil dieselben Elemente zum Aufbau eines Formulars. Da diese Elemente in jeder dieser Sprachen existieren, sind sie auch für die FormDSL von elementarer Bedeutung und werden in diese integriert. Die Erstellung folgender Formulare-Elemente soll in der FormDSL möglich sein:

- **header**
  Es soll möglich sein Überschriften zu definieren. Sie dienen dazu die Formulare zu strukturieren und in logische Abschnitte einzuteilen.
- **label**
  Es soll möglich sein Labels zu definieren. Labels sind einfache Textfelder die Text anzeigen und mit denen keine Interaktion möglich ist.
- **textbox**
  Es soll möglich sein Textboxen zu definieren. In sie kann Text eingegeben werden.
- **checkbox**
  Es soll möglich sein Checkboxen zu definieren. Eine Checkbox kann für

die Werte **true** oder **false** stehen. Durch einen Klick kann der Status einer Checkbox verändert werden. Der Standartwert soll durch ein Attribut definiert werden können. Checkboxen sollen gruppiert werden können.

- **radio** Es soll möglich sein Radiobuttons zu definieren. Ein Radiobutton kann für die Werte **true** oder **false** stehen. Durch einen Klick kann der Status eines Radiobuttons verändert werden. Der Standartwert soll durch ein Attribut definiert werden können. Radiobuttons sollen gruppiert werden können. Innerhalb einer Gruppe von Radiobuttons kann immer nur ein Button den Wert **true** besitzen.
- **group** Es soll möglich sein mehrere Elemente einer Gruppe zuzuordnen.
- **number**
  Definiert ein numerisches Textfeld. Die Anzahl an Ziffern kann gewählt werden. Ein Auswahlbereich der möglichen Nummern kann ebenfalls gewählt werden.
- **list**
  Definiert eine Dropdown-Liste und deren Auswahlmöglichkeiten. Eines der Auswahlmöglichkeiten kann als Standartauswahl durch ein Attribut definiert werden.

Bei den oben genannten Elementen handelt es sich um die in Webformen häufigsten Elemente. Die Sprache soll so entworfen werden, dass im weiteren Entwicklungsverlauf weitere Elemente integriert werden können. Ein Beispiel hierfür wäre ein **Upload-Element**, welches eine Datei durch die Auswahl eines Pfades hoch lädt. In HTML gibt es deutlich mehr Form-Elemente, die durch **Input Types** definiert werden [6]. Die Definition solcher Typen soll in der FormDSL durch Constraints geschehen, da diese keine Auswirkungen auf das Layout haben.

2.2.2 Layout

In Abschnitt 2.1 wurden uns mit dem `Dockpanel`, dem `Stackpanel` und dem `Grid` verschiedene Layout-Möglichkeiten vorgestellt. Ein wichtiger Punkt für das Design unserer Sprache ist die einfache Erlern- und Handhabbarkeit. Der Anwender soll schnell in der Lage sein den Funktionsumfang zu erlernen und diesen anzuwenden. Ist es dem Nutzer möglich ein komplexes Layout zu erstellen, erfordert das einen größeren Funktionsumfang und demzufolge einen Mehraufwand die Sprache zu erlernen. Für die DSL ist es wichtig Formulare zu erstellen und nicht deren Gestaltung. Die Sprache soll die einfachste Möglichkeit bieten Formulare zu erstellen. Dadurch muss sich der Anwender nicht mit den Gestal-

[6] `http://www.w3schools.com/html/html_form_input_types.asp`

tungsmöglichkeiten eines Formulars beschäftigen und kann seinen Fokus auf die benötigten Formularelemente richten.

Ein Dockpanel bietet die Möglichkeit Elemente innerhalb des Panels beliebig an den vier Seiten auszurichten. Die dadurch entstehenden Freiheiten eröffnen die Gefahr der Gestaltung von benutzerunfreundlichen Formularen.

In der FormDSL wird für die Layout-Generierung ein Stackpanel verwendet. Durch das Stackpanel werden die Bestandteile klar und deutlich unter einander angeordnet, was für eine einfache und benutzerfreundliche Struktur sorgt. Einzelne kleine Formular-Elemente sollen bedingt platzsparender angeordnet werden können.

Ein Gitter zur Strukturierung der Formelemente ist eine weitere Möglichkeit. Die spaltenweise Anordnung ist übersichtlich und für den Nutzer der FormDSL einfach zu erlernen. Da die Hauptaufgabe der DSL die Erzeugung von Formularen und nicht die gestalterische Anordnung der Elemente innerhalb dieser ist, wird sich bei der Gestaltung der Sprache gegen ein Gitter entschieden. Ein weiterer Punkt sind die Einschränkungen auf die bei der Implementation geachtet werden muss. Wie viele Spalten sind möglich? Und wie geschieht die Zuordnung zwischen Elementen und Spalten? Diese Punkte führen dazu, dass beim Layout kein Gitter sondern ein Stackpanel verwendet wird.

2.2.3 Constraints

Nicht nur das Erstellen von Formularen mit deren Elemente ist Anforderung an die DSL, sondern auch das Definieren von Einschränkungen für das Formular. Durch die Constraints soll es möglich sein Eingaben einzuschränken und funktionale Einschränkungen für die Benutzungsschnittstellen zu definieren.

Durch das Definieren von Bedingungen soll es möglich sein die einzugebenden Daten in das Formular einzuschränken. Die Einschränkungen für erzeugte Formelemente können ebenfalls aus der Analyse der verschiedenen Sprachen abgeleitet werden.

Eingabetype von Textfeldern Der Eingabetext von Textfeldern soll durch Constraints eingeschränkt werden können. HTML bietet verschiedene Input-Typen wie *password*, *date*, *email* oder *number* als Formular-Elemente an[7]. Die in 2.1.4 beschriebene DSL bietet ebenfalls verschiedene Eingabetypen, welche unterschiedliche Eingabemöglichkeiten zulassen. Diese Eingabetypen haben keine Auswirkung auf das Layout und sollen deshalb innerhalb der Constraints definiert werden können.

[7] `http://www.w3schools.com/html/html_form_input_types.asp`

Versteckte Elemente In 2.1.4 existiert mit *bool* ein Element, welches es ermöglicht durch Aktivieren einer Checkbox weitere Fragen eines Formulars zu aktivieren. Diese Gruppe von Elementen erscheint erst, wenn die Checkbox aktiviert ist. Auch in 2.1.1 gibt es Attribute, die Elemente verbergen oder inaktiv schalten können. Es soll also möglich sein, die Sichtbarkeit von Elemente an bestimmte Constraints zu binden.

Im Laufe der Zeit können sich die Anforderungen der DSL und damit auch an die möglichen Constraints ändern. Die Möglichkeit weiterer Bedingungen kann gewünscht werden. Deshalb sollen die Constraints in der Sprache so implementiert werden, dass eine Erweiterung der festzulegenden Bedingungen einfach realisierbar ist.

Die Definition eines Constraints wird folgendermaßen festgelegt: Ein Constraint besteht aus einer oder mehreren Bedingungen, sowie der daraus erfolgenden Einschränkung. Die Bedingungen können durch logische Operatoren miteinander verbunden werden. Innerhalb der Bedingungen und Einschränkungen muss auf Formularelemente verwiesen werden, da die Constraints Formularelemente einschränken sollen.

2.2.4 Datenstruktur für Eingegebene Inhalten

Die durch den Formular-Nutzer eingegebenen Daten sollen gespeichert werden. Dem DSL-Nutzer soll es möglich sein, eine Struktur zu definieren, wie die eingegebenen Daten abgespeichert werden. Die in 2.1.5 vorgestellte Sprache ermöglicht es mit sogenannten „fields"Daten in eine vorgegebene Struktur zu integrieren. Anstatt Datenbankfelder soll auf Formulareingaben referenziert werden. Dem DSL-Nutzer soll es also möglich sein eine Datenstruktur mit Verweisen auf Formulareingaben zu definieren. Die Definition einer Datenstruktur soll möglich sein, muss vom DSL-Nutzer aber nicht zwangsläufig angewendet werden. Wird keine Struktur definiert, so sollen die Daten standardmäßig in einer vordefinierten Struktur gespeichert werden.

3 Design und Implementierung

Das folgende Kapitel befasst sich mit dem Design und der Implementierung der FormDSL. Grafik 3.1 zeigt die Architektur der Implementierung. Sie besteht aus drei Modulen, dem `Parser`, dem `HTML-Creator` und dem `Filter`.

Im Parser sind die Sprachdefinition, sowie die Abbildung in die zugrunde liegende Datenstruktur implementiert. Der Aufbau und die Funktionsweise des Parsers wird in Abschnitt 3.1 beschrieben.

Die erzeugte Datenstruktur wird zum einen vom HTML-Creator benutzt um HTML-Formulare zu generieren, zum anderen wird sie vom Filter zusammen mit den abgeschickten Daten des Formulare dazu verwendet, um die vom DSL-Nutzer definierten Constraints zu prüfen und die gewünschte Datenstruktur der Inhalten zu erzeugen 3.3.

Im Anschluss wird in Abschnitt 3.4 noch auf die Fehlerbehandlung innerhalb der FormDSL eingegangen.

3.1 Parser-Modul

Für die Implementation des Parsers wird mit ANTLR ein Parser-Generator verwendet. Durch die Definition einer Grammatik, bestehend aus einem Lexer 3.1.1 und den Grammatikregeln 3.1.2, generiert ANTLR einen Parser, der einen Syntaxbaum erzeugt und mit dem Interpreter 3.1.3 durchlaufen werden kann[1].

3.1.1 Lexer

Die lexikalische Analyse bezeichnet die Zerlegung eines Eingabestromes in verschiedene Symbole, die sogenannten *Tokens*. In der Scanphase wird eine Sequenz

[1] `http://www.antlr.org/`

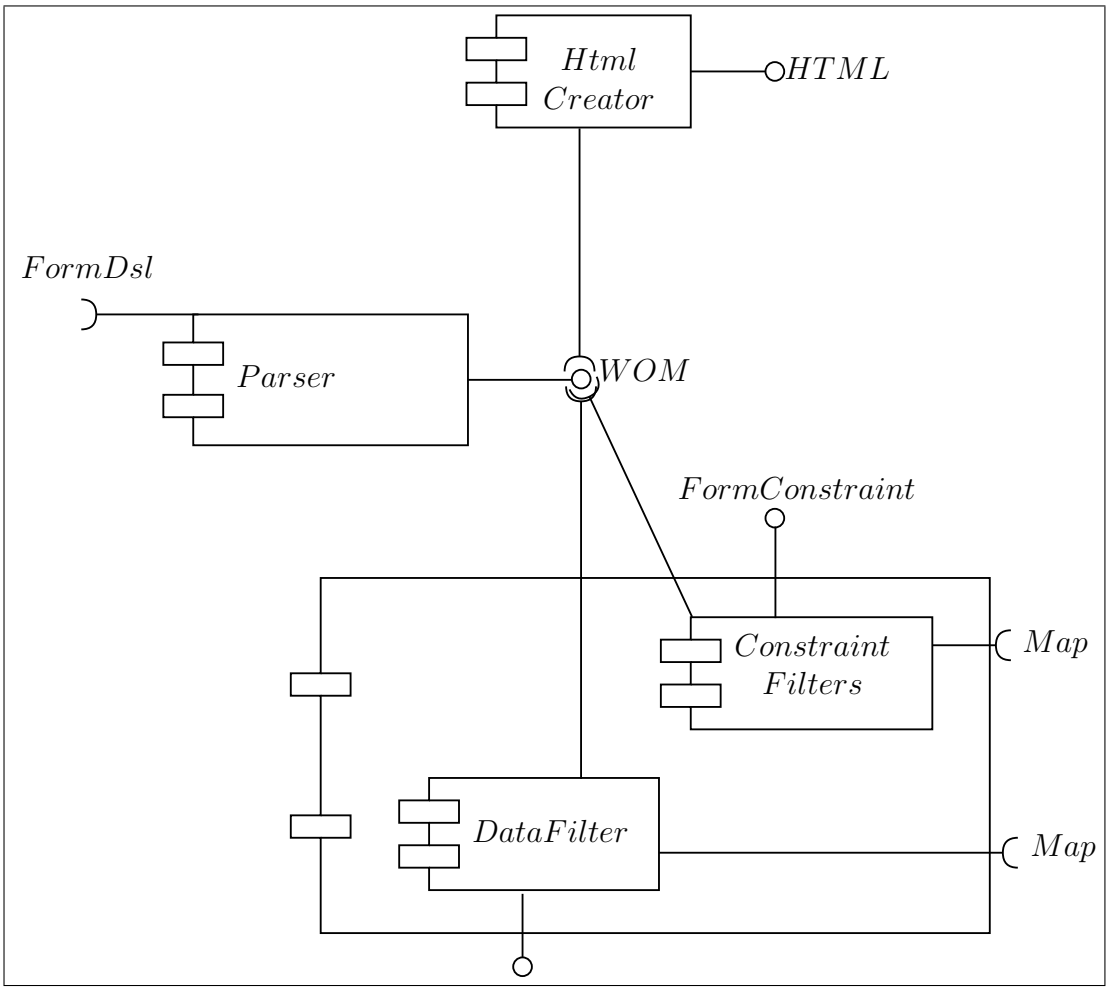

Abbildung 3.1: Architektur der FormDSL-Implementierung

von Zeichen eingelesen und diese in Tokens umgewandelt. Dies dient als Vorbereitung für die anschließende syntaktische Analyse (Louden, 1993, S. 64).

Im Lexer A werden die Bezeichnungen der Tokens auf der linken Seite und die Zuordnung der jeweiligen Zeichensequenzen auf der rechten Seite, getrennt durch einen Doppelpunkt : dargestellt. Das Semikolon ; beendet die Definition eines Tokens.

Eine typische Art von Tokens sind die *keywords* oder reservierten Wörter. Wird exakt ein solches Keyword im Inputstream gescannt, so wird es dem Token des Keywords zugeordnet. Im Anhang Lexer A befinden sich die Definitionen der Schlüsselwörter in den Zeilen 23 - 32.

Weitere wichtige Tokens sind die Spezialsymbole wie Klammern, Punkte oder andere Sonderzeichen. Diese werden in den Zeilen 6 - 17 definiert.

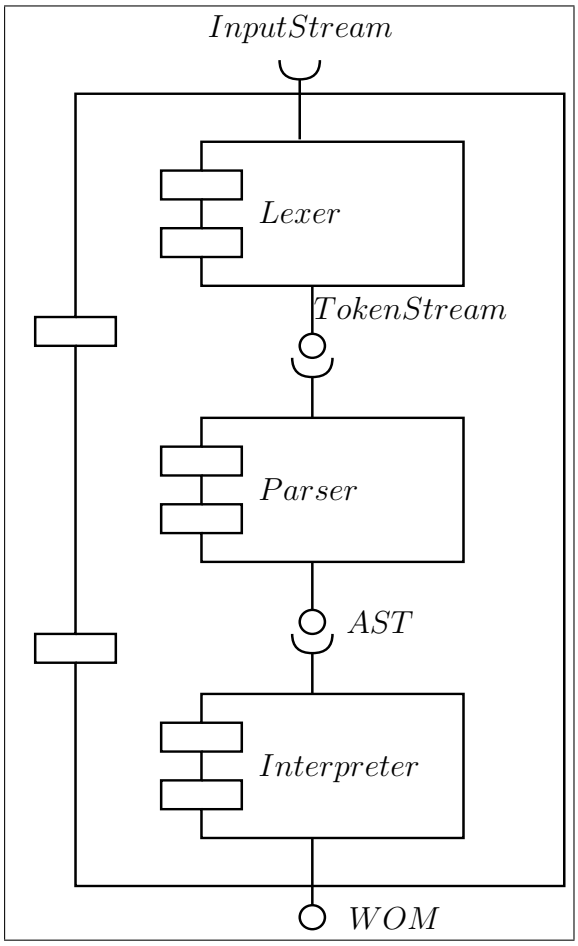

Abbildung 3.2: Überblick der FormDsl-Parser-Implementierung

Eine weitere Art von Tokens sind die *Identifier*, welche in den Zeilen 62 - 73 definiert sind. Diesen Tokens können mehreren Zeichensequenzen zugeordnet werden. Die oben genannten reservierten Wörter werden so genannt, da ein Identifier nicht die selbe Zeichenfolge besitzen kann wie ein Schlüsselwort. Ein String *form* wird also dem Token `Form` zugeordnet und nicht dem Token `Characters`, obwohl die Zeichenkette nur aus `JavaLettern` besteht und ebenfalls ein `Characters` Token sein könnte.

Die Schwierigkeit bei der Definition von Tokens besteht darin, dass eine Zeichenfolge nur einem Token zugeordnet werden kann. Besteht die Möglichkeit eine Zeichensequenz mehreren Tokens zuzuordnen, so muss im Parser darauf Rücksicht genommen werden, indem alle Möglichkeiten für die Regel in Betracht gezogen werden. In 3.1 ist ein Teilausschnitt des für die FormDsl verwendeten Lexers abgebildet. Er zeigt die Definition von `Equals`-Token sowie zwei verschiedene Identifier-Möglichkeiten. Während `Identifier` zu Beginn der Sequenz einen Buchstaben oder Unterstrich besitzen muss und anschließend aus beliebig vielen

Buchstaben und Ziffern bestehen kann, besteht ein `Character` nur aus beliebig vielen Buchstaben.

```
1  Equals:      '=' ;
2
3  Characters: JavaLetter+ ;
4
5  Identifier: IdentifierChars ;
6
7
8  fragment IdentifierChars: JavaIdLetter JavaLetterOrDigit*
       ;
9
10 fragment JavaLetter: [A-Z] | [a-z] ;
11
12 fragment JavaIdLetter: JavaLetter | '_' ;
13
14 fragment JavaLetterOrDigit: JavaIdLetter | [0-9] ;
```

Listing 3.1: Ausschnitt eines Lexer-Programmes

Die Lexer-Eingabe 3.2 kann dadurch mehrere Ausgaben erzeugen, die in 3.3 gezeigt werden. Während die zwei letzten Zeichensequenzen eindeutig einem Token zugeordnet werden, ist *name* laut Definition sowohl ein Character als auch ein Identifier.

```
1  name = button1
```

Listing 3.2: Zeichensequenz als Lexer-Input

```
1  Characters Equals Identifier
2
3  Identifier Equals Identifier
```

Listing 3.3: Tokensequenz als Lexer-Output

3.1.2 Parser

Nach dem Umwandeln der eingegebenen Zeichen in Tokens, wird der Syntax *Tokenstream* durch den Parser und die dort definierten Grammatikregeln geprüft. Die einzelnen Regeln werden im Parser in Form von Ableitungen festgelegt. Auf der linken Seiten stehen die sogenannten *Symbole*. Diese können durch die Alternativen auf der rechten Seite ersetzt werden. 3.1 zeigt uns eine mögliche Ableitung

der DSL. Das Symbol `panelElement` kann entweder durch `element` oder durch `group` ersetzt werden.

$$< panelElement > \longrightarrow < element > \\ \longrightarrow < group > \tag{3.1}$$

Listing 3.4 zeigt eine Ableitungsregel der FormDsl im ANTLR-Parser. Das zu ersetzende *panelElement* wird in der ersten Zeile definiert. In Zeile zwei und drei werden die Alternativen *groupDefWithMaybeWs* und *elementDefWithMaybeWs* zugeordnet und durch das Metazeichen | getrennt, das für ein logisches **OR** darstellt. Die zwei Alternativen werden Variablen zugeordnet, um bei der weiteren Verarbeitung durch den Interpreter darauf zugreifen zu können. Das gleiche gilt für den Rückgabetype *Wom3Node* und die Rückgabevariable *result*.

```
1 panelElementDef returns [ Wom3Node result]:
2   group = groupDefWithMaybeWs
3 | element = elementDefWithMaybeWs
4 ;
```

Listing 3.4: FormDsl Parserregel

Dieser Vorgang wird **Ableiten** genannt. Eine Ableitung ist ein Vorgang, bei dem das Symbol auf der linken Seite einer Regel durch die Alternative auf der rechten Seite der Regel ersetzt wird. Die Ableitungen können durch einen Abstract Syntax Tree (AST) dargestellt werde.

Kann ein Symbol so lange abgeleitet werden bis am Ende nur noch Tokens anstelle der Symbole stehen, so ist es eine legale Eingabe der Sprache. Dadurch entsteht ein Abstract Syntax Tree (AST). Der Syntaxbaum einer Eingabe der FormDsl 3.5 lässt sich wie in Abbildung 3.3 darstellen. Das Symbol `panelElement` wird in die Symbole `element` und `semicolon` abgeleitet. Dies geschieht so lange, bis am Ende des Baumes als Blätter nur noch Tokens übrig sind.

```
1 header(name = header0);
```

Listing 3.5: Befehl zum Erzeugen eines Elements in der FormDsl

3.1.3 Interpreter

Nach dem Erzeugen des Syntaxbaumes wird dieser vom Interpreter in ein Wiki Object Model (WOM) übertragen. WOM ist im Gegensatz zum AST der

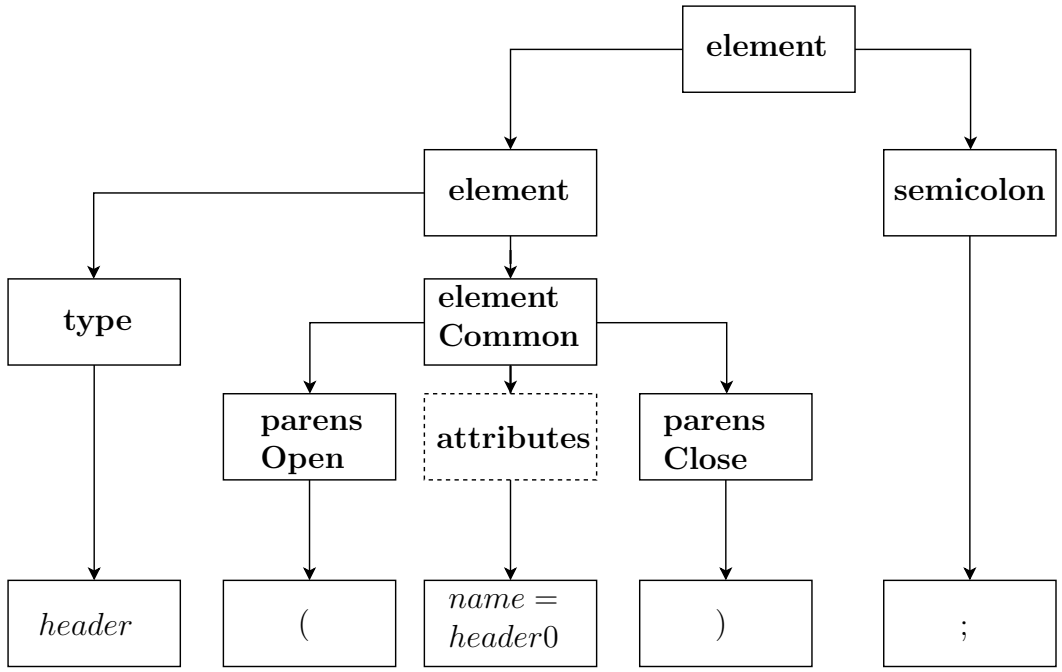

Abbildung 3.3: AST Beispiel

Abstrakte-Syntax-Baum des Sweble Wiki und die zugrunde liegende Datenstruktur (Dohrn & Riehle, 2011b). Durch den Interpreter wird dieser manipuliert und für die weitere Verarbeitung oder Datenspeicherung aufgebaut. Listing 3.6 zeigt den Teilbaum aus dem oben genannten Sprachbefehl 3.5. Der WOM-Baum besteht aus einem Formelement vom Type `header`, der als Attribut angegeben ist. Die Attribute des Elementes im DSL-Befehl sind Kinder *form:attribute* in der Datenstruktur. Des Weiteren existieren noch `rtd`-Kinder, welche die Eingabezeichen der DSL speichern, um die Eingabe des Benutzers wieder herstellen zu können.

Der Vorteil die Attribute der DSL-Eingabe als Kindknoten des Oberelementes zu speichern und nicht als Attribute von diesem ist, dass die Attribut-Knoten wieder extra rtd-Knoten als Kinder besitzen können, auf die später getrennt zugegriffen werden kann. Wird durch Manipulation des WOM ein Attribut gelöscht, so wird auch sein rtd-Kindknoten gelöscht und das Attribut ist auch in der Wiederherstellung der DSL-Eingabe nicht mehr vorhanden.

```
1 <form:element xmlns:form="http://sweble.org/schema/form"
      type="header">
2   <rtd xmlns="http://sweble.org/schema/wom30">header (</
        rtd>
3   <form:attribute name="header0">
4     <rtd xmlns="http://sweble.org/schema/wom30">name =
```

```
        header0</rtd>
  </form:attribute>
  <rtd xmlns="http://sweble.org/schema/wom30">);&amp;#xD;
    &amp;#xD;
  </rtd>
</form:element>
```

Listing 3.6: Wom-Baum der FormDsl

3.2 Formular-Generierung

Nach dem Aufbau der WOM-Struktur durch den Interpreter wird mittels HTML-Creator-Klassen das Formular generiert. Dazu iteriert ein Handler-Objekt über alle Elementknoten und erzeugt von jedem HTML-Code. Diese Codefragmente und somit auch die einzelnen Elemente werden in ein HTML-Template eingefügt und erzeugen das HTML-Formular. Das Code-Template benutzt Bootstrap für die HTML-Generierung. Das vereinfacht die Anordnung der einzelnen Code-Fragmente beim Einfügen in das Template. Für jeden Elementtyp muss eine eigene Creator-Klasse bestehen. Werden neue Elementtypen zur Sprache hinzugefügt, so muss auch eine neue Klasse zur HTML-Generierung für genau dieses Element erstellt werden. Das ist notwendig, da die verschiedenen Elementtypen unterschiedliche Attribute und Eigenschaften besitzen, was zur Folge hat, dass jeder Typ unterschiedlichen HTML-Code erzeugt. Wie Abbildung 3.4 zeigt, muss jedes konkrete `HtmlFormElement` von der abstrakten Oberklasse `HtmlFormElement` abgeleitet sein, die das Interface `IHtmlElement` implementiert und eine `getHtml`-Methode besitzt. Diese Methode gibt den spezifischen HTML-Code zurück und ist notwendig, um das Element in der Webform zu erstellen.

3.3 Filter

Nach dem Erstellen und Bereitstellung der Formulare können diese von einem Wiki-Nutzer ausgefüllt und abgeschickt werden. Die abgesendeten Formulardaten sind als Key-Value-Paare implementiert. Da die Sprache noch nicht in das Wiki integriert wurde und der Prozessablauf vom Anzeigen der Webformulare bis hin zum Zurücksenden der Formulardaten an das System nicht implementiert ist, werden die Key-Value-Paare zum Text manuell erzeugt und weiterverarbeitet.

Die Weiterverarbeitung geschieht im Filter-Modul. In 3.1 wird im unteren Teil das Filter-Modul dargestellt, welches aus `Constraint-Filter` und `Daten-Filter` besteht. Diese bekommen als Input die oben genannten Key-Value-Paare als Map

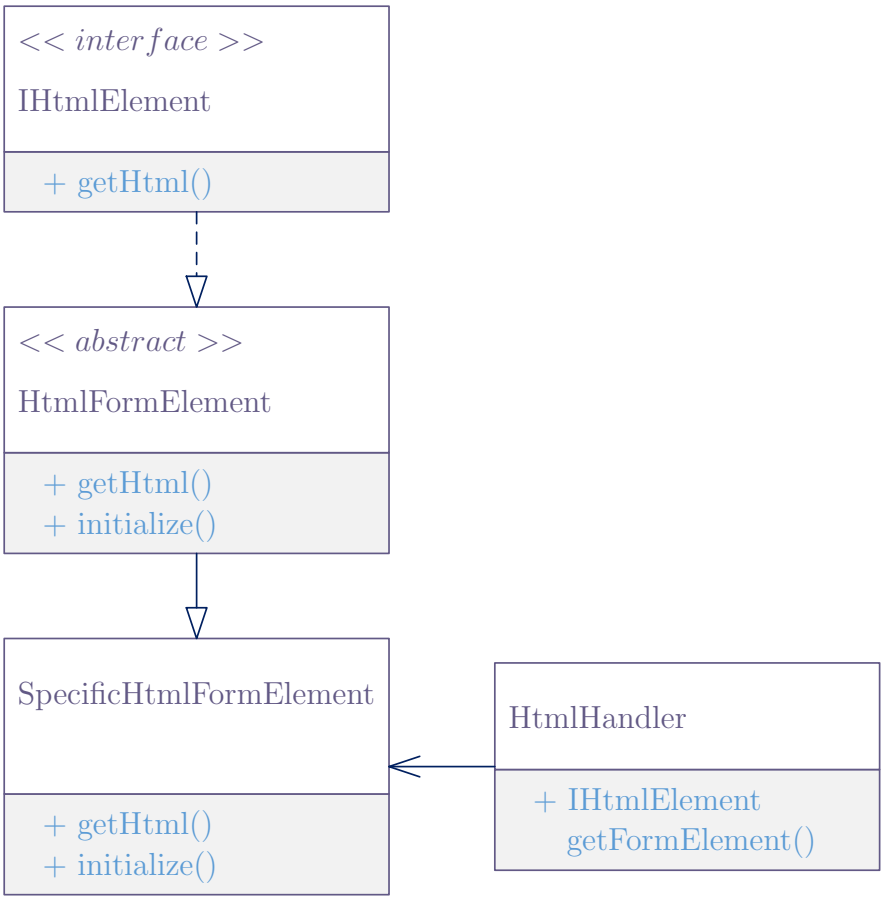

Abbildung 3.4: HTML-Formular-Erzeugung in der FormDsl-Implementierung

und die vom Interpreter aufgebaute WOM-Datenstruktur. Die beiden Filter werden in 3.3 und 3.3 genauer beschrieben.

Die Implementierung der Filter zeigt 3.5. `FormDslConstraintFilter` und `FormDslDataFilter` sind konkrete Klassen einer abstrakten Oberklasse `FormDslFilter`, die das Interface `IFormDslFilter` implementiert. Beide Unterklassen erben von ihrer Oberklasse eine `initialize`-Methode, mit dem erzeugten WOM-Baum und der Daten-Map (Key-Value-Paare) als Übergabeparameter. Die Filter müssen eine Methode `apply` implementierten, mit der es möglich ist den Filter auf den initialisierten Daten anzuwenden.

Constraint-Filter

Im Constraint-Filter werden die Formulareingaben auf ihre Einschränkungen überprüft. Das geschieht durch den Vergleich der zurück gelieferten Daten und mit den vom DSL-Nutzer festgelegten Einschränkungen, welche im WOM-Baum

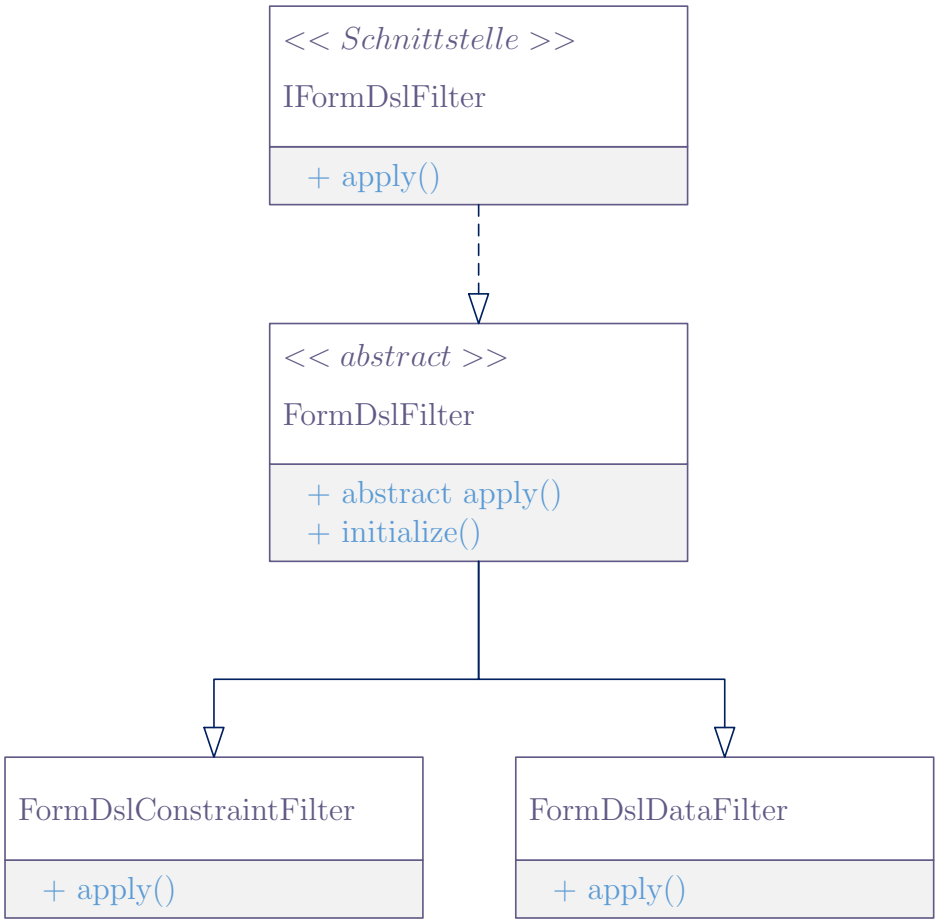

Abbildung 3.5: Filter-Implementation der FormDsl

angegeben sind. Die Constraints und die Formulareingaben können mittels Element-Id zugeordnet werden. Dadurch können die vom Nutzer getätigten Eingaben auf ihre Gültigkeit untersucht werden.

Abbildung 3.6 zeigt die Implementierung des Constraints-Filter und der damit stattfindenden Überprüfung der Bedingungen. Eine *Condition* stellt eine konkrete Bedingung dar, auf die eine Eingabe überprüft werden kann. Ein Beispiel ist die Condition *min*, die eine Eingabe auf ihre Mindestwert oder Mindestlänge überprüft. Die genaue Validierung ist in konkreten Validierungsklassen implementiert. Die Conditions sind für die spätere Erweiterung als *Enum* implementiert. Jede Condition besitzt eine Validierungsklasse `SpecificDslCondition`, welche von einer Oberklasse `FormDslCondition` erbt und das Interface `IFormDslCondition` implementiert. Die Unterklassen erben von ihrer Oberklasse eine `initialize`-Methode, welche die internen Datenfelder bei der Objekterzeugung initialisiert. Dafür wird die erzeugte WOM-Datenstruktur und eine Key-Value-Map benötigt. Durch die `checkCondition`-Methode wird eine Eingabe auf ihre Gültigkeit vali-

diert.

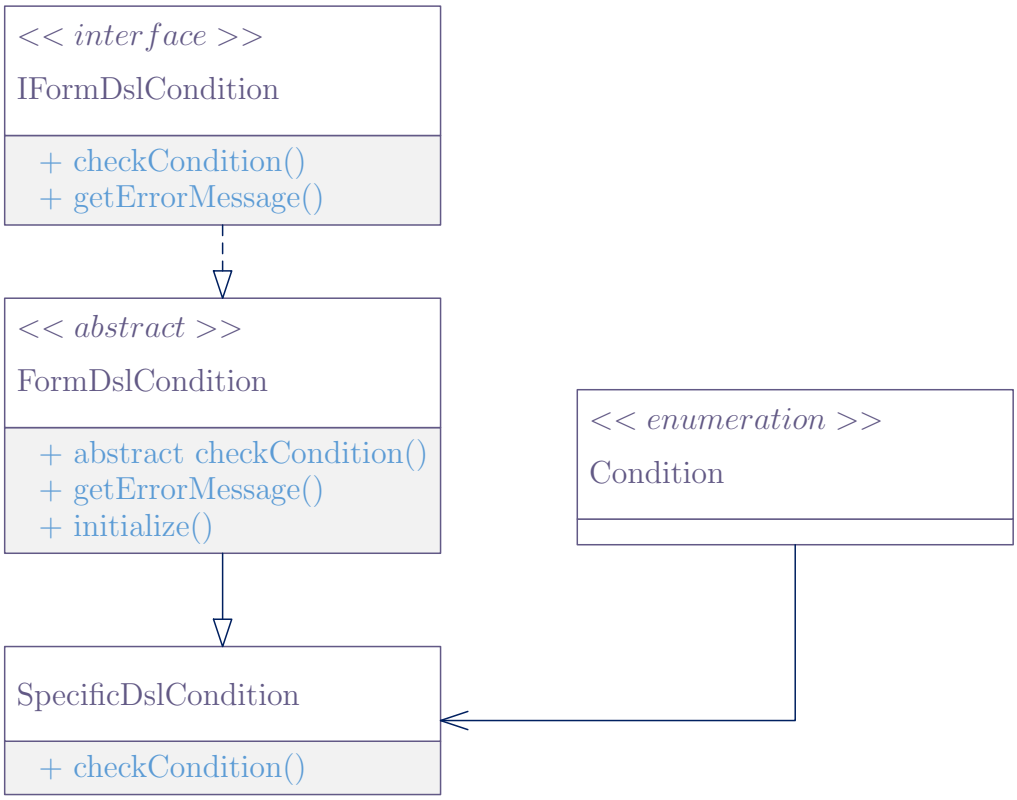

Abbildung 3.6: Constraint-Implementierung der FormDsl

Daten-Filter

Im Daten-Filter wird bei der Erzeugung aus den Formulareingaben und dem WOM-Baum eine vom DSL-Nutzer bestimmte Datenstruktur aufgebaut. Die im WOM festgelegte Struktur wird im Daten-Filter erzeugt und die durch Referenzen festgelegten Werte in der Key-Value-Map gesucht und in die Datenstruktur übernommen.

Die nach der `apply`-Methode erzeugte Datenstruktur wird durch `getDataWom` abgerufen und weiter verarbeitet.

Listing 4.6 zeigt die Definition einer Datenstruktur innerhalb der FormDSL. Zusammen mit der vom Interpreter erzeugten WOM-Datenstruktur und den Formulareingaben, erstellt der Datenfilter beim Anwenden die in 3.8 dargestellte XML-Struktur.

```
1 data
2 {
```

```
3     section[ title = "city" ]
4     {
5       field( name = "hometown", value = @textbox0);
6       field( name = "specialities", value = @textarea0);
7     }
8   }
```

Listing 3.7: Definition einer Datenstruktur in der FormDSL

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <data>
3    <data:section>
4      <data:attribute title="city"/>
5      <data:field>
6        <data:attribute name="hometown"/>
7        <data:attribute value="Erlangen"/>
8      </data:field>
9      <data:field>
10       <data:attribute name="specialities"/>
11       <data:attribute value=""/>
12     </data:field>
13   </data:section>
14 </data>
```

Listing 3.8: FormDsl Parserregel

3.4 Fehlerbehandlung

In diesem Abschnitt wird auf die Fehlerbehandlung in der Implementation der FormDsl eingegangen. Diese ist ein wichtiger Punkt für die Nutzbarkeit der Sprache. Um den Sprach-Nutzer nicht zu frustrieren, müssen ihm Fehler, die bei der Eingabe entstehen mitgeteilt werden. Die Mitteilung der Eingabefehler muss für den Benutzer klar verständlich sein, da dieser die Eingabe sonst nicht berichtigen kann. Zu unterscheiden sind dabei:

- Fehler, die aufgrund der Grammatik keine gültige Sprache erzeugen
- Fehler, die eine gültige Sprache erzeugen, aber nicht interpretiert werden

Erstere erzeugen keine gültige Eingabe und können vom System nicht korrekt weiter verarbeitet werden, da die Eingabe keine gültige Sprache der definierten Grammatik ist. Bei der zweiten Art von Fehlern wird eine gültige Sprache

erzeugt. Diese könnte aber eine andere Ausgabe erzeugen als der Benutzer erwartet, da Teile der Eingabe nicht interpretiert wurden. Die Eingabefehler werden zwischengespeichert um dem Nutzer darüber informieren zu können. Das erfolgt mittels Fehler-Knoten innerhalb der WOM-Datenstruktur. Wird beispielsweise ein nicht existenter Elementtyp mit einer Eingabe erzeugt, so wird anstatt des Element-Knoten ein Fehler-Knoten erzeugt. Dieser enthält als Attribut eine Fehlernachricht auf die zugegriffen werden kann. Das gleiche gilt für falsche Attribute oder Attribut-Werte. In Listing 3.9 wird ein fehlerhaftes Attribut und ein fehlerhaftes Element vom Benutzer definiert. Der erzeugte WOM-Baum 3.10 enthält statt eines Attributknoten einen Fehlerknoten mit Fehlermeldung (Zeile 6-8). Gleiches gilt für den Fehlerkonten der das fehlerhafte Element definiert (Zeile 19-25). Beim Erstellen der Fehlerknoten ist es wichtig die `rtd`-Knoten zu erzeugen, um die ursprüngliche FormDSL-Eingabe wiederherstellen zu können.

```
1  <form:element xmlns:form="http://sweble.org/schema/form"
       type="header">
2    <rtd xmlns="http://sweble.org/schema/wom30">header (</
         rtd>
3    <form:attribute name="header0">
4      <rtd xmlns="http://sweble.org/schema/wom30">name =
           header0</rtd>
5    </form:attribute>
6    <rtd xmlns="http://sweble.org/schema/wom30">);&#xD;
7      &#xD;
8    </rtd>
9  </form:element>
```

Listing 3.9: FormDSL Eingabe mit Fehlern

```
1  <form:element type="header">
2    <rtd>header (</rtd>
3    <form:attribute name="header0">
4      <rtd>name = header0</rtd>
5    </form:attribute>
6    <form:error msg="namee is no valid attribute name">
7      <rtd>, namee = header1</rtd>
8    </form:error>
9    <form:attribute text="My first form" lang="en">
10     <rtd>, text:en = "My first form"</rtd>
11   </form:attribute>
12   <form:attribute text="Mein erstes Formular" lang="de">
13     <rtd>, text:de = "Mein erstes Formular"</rtd>
14   </form:attribute>
```

```
15    <rtd>);&amp;#xD;
16      &amp;#xD;
17    </rtd>
18  </form:element>
19  <form:error msg="label element misses required attribute
        text">
20    <rtd>label (</rtd>
21    <form:attribute name="label1">
22      <rtd>name = label1</rtd>
23    </form:attribute>
24    <rtd>);</rtd>
25  </form:error>
26  <rtd>&amp;#xD;
27  </rtd>
```

Listing 3.10: WOM-Baum fehlerhaften FormDSL Eingabe

4 Sprachspezifikation

Das folgende Kapitel beinhaltet die Sprachspezifikation der FormDSL. Die FormDSL ist eine Programmiersprache zur Erstellung von Formularen, zum Definieren von Randbedingungen für deren Elemente und zum Festlegen einer Datenstruktur zum Abspeichern des Inputdatensatzes.

4.1 Syntax

Der folgende Abschnitt beschreibt den Syntax der FormDSL. Dieser kann in drei Teilabschnitte gegliedert werden, welche in den Sektionen weiter ausgeführt sind. Jeder von ihnen wird durch ein Signalwort eingeleitet:

- **form** definiert die Formularbeschreibung
- **constraints** definiert die Constraints
- **data** definiert die Datenstruktur

Listing 4.1 zeigt die Definitionen dieser drei Abschnitte innerhalb der FormDSL. Die geschweiften Klammern bilden den Raum für die Anweisungen innerhalb des Abschnittes. Um ein Formular mit der DSL zu erstellen muss ein form-Abschnitt definiert sein. Die beiden anderen Teile sind nicht zwingend erforderlich.

```
1 form{
2 }
3 constraints{
4 }
5 data{
6 }
```

Listing 4.1: Grunddefinitionen in der FormDSL

4.1.1 Form

Im **form-Block** wird das Formular mit seinen Elementen erstellt. Die Definition der einzelnen Formelemente erfolgt mit dem Befehl 4.2. Anstelle von *element* steht das Formelement, welches erzeugt werden soll. Die möglichen Formelemente sind in Abschnitt 4.2 aufgelistet. Innerhalb der Klammern werden die jeweiligen Attribute eines Elementes durch Komma getrennt definiert. Ein Attribut besitzt einen Attributnamen und einen Attributwert. Die existierenden Attribute und ihre gültigen Werte sind in 4.3 aufgelistet. Ein Semikolon hinter der schließenden Klammer beendet das Element. 4.3 zeigt die Auflistung von mehreren Elementen in einer Form-Definition. Die drei Punkte stehen stellvertretend für die Attribute des Elementes.

```
1 element (attribut1 = wert1 , attribut2 = wert2);
```

Listing 4.2: Liste in der FormDSL

```
1 form
2 {
3   element ( ... );
4   element ( ... );
5   element ( ... );
6 }
```

Listing 4.3: Formelemente der FormDSL

Spezielle Elemente, die Unterelemente zwischen der geschlossenen Klammer besitzen und dem Semikolon noch einen weiteren Abschnitt, der durch eckige Klammern definiert wird. In diesem werden die Unterelemente in der gleichen Art beschrieben, wie Oberelemente in ihrem Formular. Listing 4.13 zeigt das List-Element als Beispiel.

4.1.2 Constraints

Im Constraints-Block werden Einschränkungen für das Formular definiert. Listing 4.4 zeigt die Definition von zwei Constraints innerhalb des Constraints-Abschnittes. Die Einschränkungen werden als Anweisungen formuliert, die durch Klammern und einem Pfeil dargestellt werden. Abgeschlossen wird eine Anweisung durch ein Semikolon.

Vor dem Pfeil stehen die einzelnen Bedingungen, die mit **UND** & oder **ODER** | verknüpft werden können. Eine Bedingung besteht aus einem wie in 4.3 spezifizierten Verweis auf einen Elementwert, eine durch einen . angehängten Eigen-

schaft, einen Operator, sowie einen angegebenen Wert. Operatoren sind <, >, =, <=, >=, und ! =.

Nach dem Pfeil steht die Folgerung, welche durch einen Verweis und eine Eigenschaft festgelegt wird. Operator und eine Wertzuweisung ist in der Folgerung nicht möglich.

In 4.4 darf der in *textarea0* eingegebenen Text mindestens fünf und höchstens neun Zeichen besitzen.

Die Einschränkungen können in einzelne Constraints gefasst werden oder wie in 4.5 mit **UND** & oder **ODER** | verknüpft werden. Bedingungen können auch geschachtelt werden.

```
1 constraints
2 {
3     (@textarea0.min = 5) --> (@textbox0.valid);
4     (@textarea0.max = 9) --> (@textbox0.valid);
5 }
```

Listing 4.4: Constraints in der FormDSL

```
1 constraints
2 {
3     (@textarea0.min = 5) & (@textarea0.min = 9) --> (
          @textbox0.valid);
4     (@textarea1.min = 5) | (@textarea1.min = 9) --> (
          @textbox1.valid);
5 }
```

Listing 4.5: Constraints in der FormDSL

4.1.3 Data

Im data-Block wird das Datenmodell, in dem die vom Nutzer eingegebenen Daten gespeichert werden, definiert.

Durch das Signalwort *section* werden logische Abschnitte definiert. Innerhalb von eckigen Klammern können Attribute festgelegt werden, die von Kommas getrennt werden. Folgende geschweifte Klammern bilden den Block für die Definition der Datenfelder.

Die Datenfelder werden durch das Signalwort *field* eingeleitet und durch ein Semikolon beendet. Dazwischen werden innerhalb von Klammern die Attribute fest-

gelegt.4.6 zeigt die Definition von zwei Datenfeldern innerhalb einer *section* im Datenmodell.

Die Attributwerte können mit Hilfe eines Verweises auf Eigenschaften eines Formularelementes verweisen. Dadurch wird der Attributwert automatisch belegt. Zeile sechs in 4.6 zeigt diesen Fall. Dort gleicht der Wert des Datenfeldes dem Wert, welcher vom Benutzer in die Textbox mit der Id `textbox1` eingetragen hat.

```
1 data
2 {
3   section[title = Stadt]
4   {
5     field(name = attributeName, value = attributeValue);
6     field(name = attributeName, value = @textbox1);
7   }
8 }
```

Listing 4.6: Definition einer Datenstruktur in der FormDSL

4.2 Elemente

Die folgende Auflistung beinhaltet die Elemente der FormDSL. **Fett** geschriebenen Attribute müssen für die jeweiligen Elemente definiert werden. Alle anderen Elemente sind optional. Die verschiedenen Attribute werden in 4.3 beschrieben.

Header Das *Header*-Element ist eine Überschrift in einem Formular. Listing 4.7 zeigt die Definition eines Headers mit seinen möglichen Attributen:

- **name**
- **text**

```
1 header (name = header0, text:en = "This is a header", text
    :de = "Das ist ein Header");
```

Listing 4.7: Header in der FormDSL

Label Das *Label*-Element ist ein unveränderliches Textelement in einem Formular. Listing 4.8 zeigt die Definition eines Labels mit seinen möglichen Attributen:

- **name**

- **text**

```
1 label (name = label0 , text:en = "is this interesting?",
    text:de = "ist das interessant?");
```

Listing 4.8: Label in der FormDSL

Abbildung 4.1: Darstellung von Label und Header im Vergleich

Textbox Das *Textbox*-Element ist Formularelement, mit welchem der Endnutzer durch die Eingabe von Text interagieren kann. Listing 4.9 zeigt die Definition einer Textbox mit seinen möglichen Attributen:

- **name**
- text

Der im Textattribut definierte Text wird in der Textbox als vorgeschlagener Text angezeigt.

```
1 textbox (name = textfield0 , text:en = "What's your
    hometown?", text:de = "Was ist deine Heimatstadt");
```

Listing 4.9: Textbox in der FormDSL

Textarea Das *Textarea*-Element ist Formularelement, mit welchem der Endnutzer durch die Eingabe von Text interagieren kann. Listing 4.10 zeigt die Definition einer Textbox mit seinen möglichen Attributen:

- **name**
- text
- rows

```
1 textarea (name = textarea0 , text:en = "write a text", text
    :de = "Schreibe einen Text", rows = 5);
```

Listing 4.10: Textarea in der FormDSL

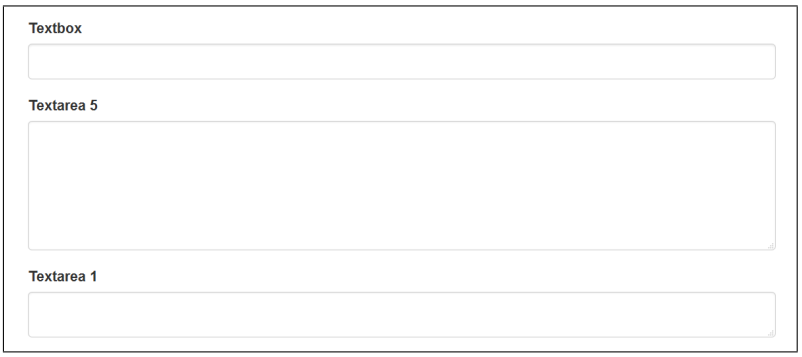

Abbildung 4.2: Darstellung einer Textbox und zweier Textareas im Vergleich

Checkbox Das *Checkbox*-Element ist Formularelement, mit welchem der Endnutzer durch Auswahl interagieren kann. Eine Checkbox kann folgende Attribute besitzen:

- **name**
- **text**
- orientation
- checked

Grafik 4.3 zeigt die formulare Darstellung der in Listing 4.11 definierten Checkboxen.

```
1 checkbox (name = checkbox0, orientation = horizontal, text
    :en = "basketball", text:de = "Basketball");
2 checkbox (name = checkbox0, orientation = horizontal, text
    :en = "football", text:de = "Football");
3 checkbox (name = checkbox0, orientation = horizontal,
    checked = true, text:en = "swimming", text:de = "
    Schwimmen");
4
5 checkbox (name = checkbox1, text:en = "basketball", text:
    de = "Basketball");
6 checkbox (name = checkbox1, text:en = "football", text:de
    = "Football");
7 checkbox (name = checkbox1, checked = true, text:en = "
    swimming", text:de = "Schwimmen");
```

Listing 4.11: Checkboxen in der FormDSL

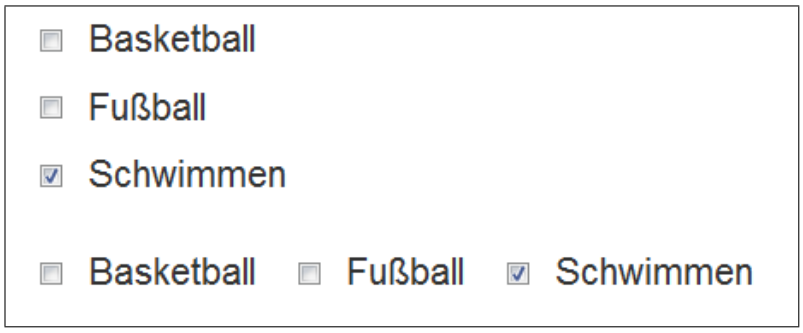

Abbildung 4.3: Darstellung von drei vertikalen und drei horizontalen Checkboxen

Radiobutton Das *Radio*-Element ist Formularelement, mit welchem der Endnutzer durch Auswahl interagieren kann. Bei zusammengehörenden Radiobuttons kann nur einer von ihnen aktiv sein. Ein Radiobutton kann folgende Attribute besitzen:

- **name**
- **text**
- orientation
- checked

Ist mehr als ein Radiobutton einer Gruppe mit *checked* vordefiniert, so werden alle weiteren Attribute dieser Art in der Gruppe ignoriert. Grafik 4.4 zeigt die formulare Darstellung der in Listing 4.12 definierten Radiobuttons.

```
1 checkbox (name = radio0, orientation = horizontal, text:en
      = "summer", text:de = "Sommer");
2 checkbox (name = radio0, orientation = horizontal, text:en
      = "winter", text:de = "Winter");
3
4 checkbox (name = radio1, text:en = "summer", text:de = "
    Sommer");
5 checkbox (name = radio1, checked = true, text:en = "winter
    ", text:de = "Winter");
```

Listing 4.12: Radiobuttons in der FormDSL

List Das *List*-Element ist ein Formularelement, mit welchem der Endnutzer durch Auswahl interagieren kann. Eine Liste kann folgende Attribute besitzen:

- **name**

Abbildung 4.4: Darstellung von drei vertikalen und drei horizontalen Radiobuttons

- select

Die verschiedenen Auswahlmöglichkeiten einer Liste werden mit Listenelementen definiert 4.2. Grafik 4.5 zeigt die in Listing 4.13 definierte Liste mit ihren Listenelementen.

```
1  List(name = alter)
2  [
3    item (text:en = "below 16", text:de = "unter 16");
4    item (text:en = "above 16", text:de = "ueber 16");
5    item (text:en = "above 18", text:de = "ueber 18");
6  ];
7
8  List(name = alter, select = multiple)
9  [
10   item (text:en = "below 16", text:de = "unter 16");
11   item (text:en = "above 16", text:de = "ueber 16");
12   item (text:en = "above 18", text:de = "ueber 18");
13 ];
```

Listing 4.13: Liste in der FormDSL

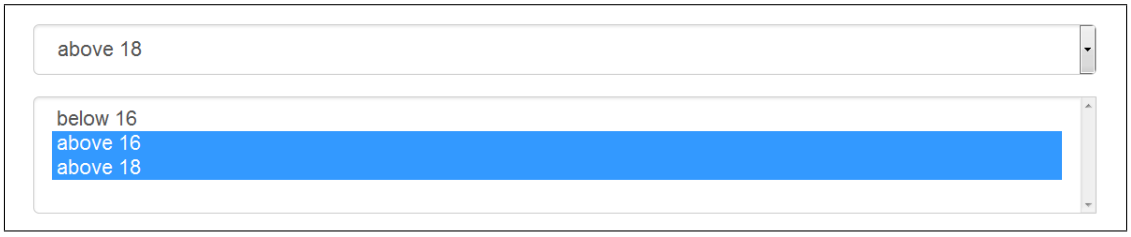

Abbildung 4.5: Darstellung von zwei Listen ohne und mit Mehrfachauswahl

Listenelement Das *item*-Element ist ein Formularelement, welches die Listenelemente einer Liste 4.2 definiert. Eine Liste kann folgende Attribute besitzen:

- **text**

Button Das *Button*-Element ist ein Formularelement, mit welchem der Endnutzer durch Klicken interagieren kann. Listing 4.14 zeigt die Definition eines Buttons mit seinen möglichen Attributen:

- **name**
- **text**

```
1 button ( name = button0, text:en = "decline", text:de = "
    abbrechen");
```

Listing 4.14: Button in der FormDSL

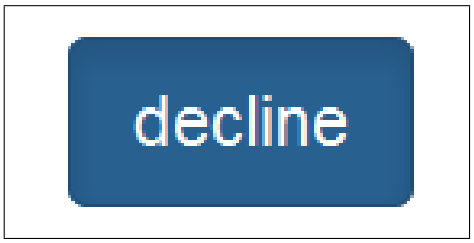

Abbildung 4.6: Darstellung eines Buttons in einem Webformular

Datenfeld Das *field*-Element ist ein Datenelement und dient zur Erzeugung von Datenfeldern in der Datenstruktur. Es kann folgende Attribute besitzen:

- **name**
- **value**

4.3 Attribute

name Das *name*-Attribut ist die eindeutige Id eines Elementes oder einer zusammengehörigen Gruppe von Elementen. Listing 4.15 zeigt die Definition des name-Attributes.

```
1   name = elementName
```

Listing 4.15: name-Attribut

text Das *text*-Attribut definiert den von einem Element in der Form angezeigten Text. Der Text kann in mehreren Sprachen angegeben werden. Der dargestellte Text muss zwischen Anführungszeichen und Schlusszeichen definiert werden. Listing 4.16 zeigt eine englische Textdefinition, während Listing 4.17 eine deutsche Textdefinition zeigt.

```
1 text:en = "Hello World"
```

Listing 4.16: text-Attribut

```
1 text:de = "Hello World"
```

Listing 4.17: text-Attribut

orientation Das *orientation*-Attribut definiert die Orientierung eines einzelnen Elementes. Es ist nicht bei allen Elementtypen erlaubt. Listings 4.18 und 4.19 zeigen das Attribute mit den zwei möglichen Werten *horizontal* und *vertical*. Wird das Attribut nicht verwendet, so wird der Standardwert *vertical* verwendet.

```
1 orientation = horizontal
```

Listing 4.18: Horizontale Elementausrichtung

```
1 orientation = vertical
```

Listing 4.19: Vertikale Elementausrichtung

checked Das *checked*-Attribut definiert die Standardauswahl eines Elementes. Es ist nicht bei allen Elementtypen erlaubt. 4.20 und 4.21 zeigt das Attribute mit den zwei möglichen Werten *true* und *false*. Wird das Attribut nicht verwendet, so wird der Standardwert *false* verwendet.

```
1 checked = true
```

Listing 4.20: Auswahl eines Elementes aktiviert

```
1 checked = false
```

Listing 4.21: Auswahl eines Elementes nicht aktiviert

select Das *select*-Attribut definiert die Auswahlmöglichkeiten eines Elementes. Es ist nicht bei allen Elementtypen erlaubt. 4.22 und 4.23 zeigen das Attribut mit den zwei möglichen Werten *single* und *multi*. Wird das Attribut nicht verwendet, so wird der Standardwert *single* verwendet.

```
1 select = single
```

Listing 4.22: Einfachauswahl eines Elementes

```
1 select = multi
```

Listing 4.23: Mehrfachauswahl eines Elementes

value Das *value*-Attribute definiert einen Wert oder Text der in der Datenstruktur gespeichert wird. Ein Attributverweis für den Wert des Attributes ist möglich.

Verweis Verweise können benutzt werden, um den Wert eines Attributes mit einem Verweis anstatt mit einem festen Wert zu belegen. Verweise werden immer durch ein @ und einer darauf folgenden Element-id festgelegt. Das Attribut bekommt dann den Wert der Eingabe, des ihm zugewiesenen Elementes. Attributverweise sind nicht bei allen Attributen erlaubt.

5 Diskussion und Ausblick

Ziel der Arbeit war es, eine domänenspezifische Programmiersprache zu entwickeln welche es ermöglicht, Benutzungsschnittstellen zu erzeugen, Constraints festzulegen und die Abbildung in Domänenobjekte festzulegen. Im abschließenden Kapitel wird die Erfüllung dieser Ziele analysiert.

In Kapitel 2 wurden bereits existierende DSLs, Rahmenwerke und Bibliotheken für vereinfachte Benutzungsschnittstellen zum Zweck der Anforderungsermittlung untersucht. Es wurden das Vokabular und der Funktionsumfang von fünf Werken untersucht. Da die Syntax von verschiedenen Sprachen unterschiedlich ist und bei der zu entwickelnden Sprache nicht einfach mehrere Syntax vermischt werden sollten, wurde das Hauptaugenmerk auf den Funktionsumfang gelegt. Durch die unterschiedlichen Schwerpunkte, in deren Anwendungsmöglichkeiten, konnte für jeden der drei Punkte (Erstellung von Formularen, Festlegung von Constraints und Abbildung in Domänenobjekte) Aspekte für das Design und die Entwicklung einer DSL entnommen werden. Obwohl nur fünf Sprachen analysiert wurden, wird der Punkt als erfüllt betrachtet, da diese für die Anforderungsermittlung ausreichend waren.

Aus den Anforderungen 2.2 wurde eine DSL abgeleitet, dessen Syntax und Funktionsumfang im Abschnitt 4 aufgelistet ist. Die FormDSL enthält Funktionen zur Formularerstellung 4.1.1, zur Definition von Constraints 4.1.2, sowie zur Festlegung einer Abbildungsstruktur 4.1.3. Das Ziel der Ableitung einer DSL aus den ermittelten Anforderungen, sowie eine dokumentierte Sprachspezifikation wird als erreicht betrachtet.

Für die Entwicklung der FormDSL wurde ein Parser (mittels Parser-Generator) und ein Interpreter entworfen und implementiert 3.1.2. Die Übertragung in die zugrunde liegenden Datenstruktur (WOM) wird durch Testfälle überprüft, welche die erzeugte Datenstruktur mit der gewünschten Datenstruktur vergleichen. Des Weiteren wurde die Erzeugung von HTML-Formularen 3.2, die Validierung von Constraints und die Erzeugung von Domänenobjekten 3.3 aufgrund der erzeugten WOM-Datenstruktur implementiert. Die Implementierung wird hier ebenfalls mit Testfällen nachgewiesen, welche das gewünschte Ergebnis mit dem erzeugten

Ergebnis vergleicht. Durch die vielen Gestaltungsmöglichkeiten von Formularen und Constraint-Definitionen, wurde für dieser Arbeit nur ein Teil des möglichen Funktionsumfanges der FormDSL implementiert. Das Ziel des Entwurfs und der Implementation mit Nachweis durch Testfälle wird trotzdem als erfüllt angesehen, da die wichtigsten Grundfunktionen wie das Erstellen von einfachen Formularen, die Überprüfung von bestimmten Einschränkungen und das Erstellen einer in der FormDSL festgelegten Datenstruktur implementiert wurden.

In den abschließenden Abschnitten wird ein Ausblick auf weitere Schritte für die Weiterentwicklung und Integration der FormDSL geworfen. Die Schritte werden unterteilt in Integration 5.1, was nicht Teil dieser Arbeit war und Erweiterungen 5.2, welche aufgrund der vielzähligen Möglichkeiten nicht vollständig implementiert wurden.

5.1 Integration in Sweble

Die FormDSL ist bisher nur innerhalb des Quellcodes in das bestehende System integriert. Ein Nutzer des Sweble Wikis ist noch nicht in der Lage den Funktionsumfang der Sprache zu nutzen und diese anzuwenden. Um das möglich zu machen sind weitere Schritte für die Integration nötig, welche im Folgenden beschrieben sind.

Erzeugung der HTML-Seiten Im bisherigen Architekturaufbau werden die Formulare mittels Creator-Klasse erzeugt. Diese dienen zum Testen der Formularerstellung und sind nicht der gebräuchliche Weg zur HTML-Seitenerstellung im Sweble Wiki. Für die weitere Integration der DSL wird die Erzeugung der Formulare an die Prozedur innerhalb des bestehenden Systems angepasst.

Rückgabe von Formulareingaben Durch die bisher noch nicht erfolgte Integration der DSL in das System, werden die vom Endnutzer getätigten Formulareingaben bisher als Map von Key-Value-Paaren simuliert. Dies ist nötig um die Eingaben innerhalb der Anwendung auf Constraints zu prüfen und eine Datenstruktur aus den Werten zu erzeugen. Das Abschicken eines Formulares und die Umwandlung in Key-Value-Paare oder ein anderes Format ist ein weiterer Schritt für die Integration in das Sweble Wiki.

Benutzerinterface für die DSL Um die FormDSL innerhalb des Wikis auch anwenden zu können, ist es nötig eine Benutzerschnittstelle oder Quellcode-Editors zur Eingabe der Anweisungen und Befehle in die Wiki-Oberfläche zu

integrieren. Eine weitergehende Möglichkeit wäre die Erweiterung oder Integration eines Visual-Editor (Wenzel, 2015) zur noch einfacheren Anwendung der FormDSL.

Sprachauswahl Formulartexte können innerhalb der FormDSL in mehreren Sprachen definiert werden. In welcher Sprache das Formular angezeigt wird, bestimmt eine Konstante die innerhalb des Quellcodes definiert ist. Um die Auswahl einer Sprache nach der Integration in die Wiki Oberfläche zu übernehmen, muss die Konstante durch eine geeignete Schnittstelle ersetzt werden.

Fehlerbenachrichtigung Falsche Eingaben eines DSL-Nutzers müssen diesem mitgeteilt werden. Dafür muss bei der Integration eine Fehlerbenachrichtigung implementiert werden, die die Fehler-Knoten im zugrunde liegenden WOM-Baum dem Nutzer anzeigt.

5.2 Erweiterung

Die DSL ist in manchen Bestandteilen so implementiert, dass sie erweiterbar ist. Dies ist nötig, da in Zukunft noch weitere Aspekte für den Funktionsumfang hinzukommen können oder nicht alle berücksichtigt wurden. In den folgenden Punkten wird eine mögliche Erweiterung aufgezeigt.

Erweiterung der Formularelemente Die verschieden Typen von Formularelemente sind als Enum implementiert und können durch weitere Typen erweitert werden. Listing 5.1 zeigt die Elementtypen. Jeder Typ enthält einen Namen, wie er auch in der FormDSL erzeugt wird, zwingend erforderliche Attribute und optionale Attribute. Für eine Erweiterung muss der gewünschte Typ mit diesen Angaben in das Enum integriert werden. Die Validierung von gültigen Attributen geschieht über die Angabe der benötigten und optionalen Attribute und muss nicht ergänzt oder überschrieben werden.

```
1 public enum ElementType
2 {
3   BUTTON("button",
4         new Attribute[] { Attribute.NAME, Attribute.TEXT },
5         new Attribute[] {}),
6   LABEL("label",
7         new Attribute[] { Attribute.NAME, Attribute.TEXT },
8         new Attribute[] {}),
```

```
9     CHECKBOX("checkbox",
10        new Attribute[] { Attribute.NAME, Attribute.TEXT },
11        new Attribute[] { Attribute.CHECKED, Attribute.
              ORIENTATION}),
12    HEADER("header",
13        new Attribute[] { Attribute.NAME, Attribute.TEXT },
14        new Attribute[] {}),
15    RADIO("radio",
16        new Attribute[] { Attribute.NAME, Attribute.TEXT },
17        new Attribute[] { Attribute.CHECKED, Attribute.
              ORIENTATION}),
18    TEXTBOX("textbox",
19        new Attribute[] { Attribute.NAME },
20        new Attribute[] { Attribute.TEXT, Attribute.TYPE }),
21    TEXTAREA("textarea",
22        new Attribute[] { Attribute.NAME },
23        new Attribute[] { Attribute.TEXT }),
24    GROUP("group",
25        new Attribute[] {},
26        new Attribute[] {}),
27    LIST("list",
28        new Attribute[] { Attribute.NAME },
29        new Attribute[] {}),
30    ITEM("item",
31        new Attribute[] { Attribute.TEXT },
32        new Attribute[] {});
33
34    private final String elementType;
35    private final Attribute[] requiredAttributes;
36    private final Attribute[] attributes;
37
38    ElementType(
39        String elementType,
40        Attribute[] requiredAttributes,
41        Attribute[] attributes)
42    {
43      this.elementType = elementType;
44      this.requiredAttributes = requiredAttributes;
45      this.attributes = attributes;
46    }
47  }
```

Listing 5.1: Elementtypen eines Formulars in der FormDSL

Erweiterung der Attribute Die verschiedenen Attribute sind nicht fix, sondern können durch Erweiterung des Enums 5.2 ergänzt werden. Dadurch ist es möglich auch zu späteren Zeitpunkten weitere Attribute hinzuzufügen, die von neu hinzugefügten Elementen benötigt werden oder durch neue Anforderungen entstehen. Attribute besitzen einen Namen, wie er auch in der DSL angegeben wird und ein Feld von möglichen Werten, die das Attribut annehmen kann.

```
1  public enum Attribute
2  {
3    TEXT("text", null),
4    STATUS("status", new String[]{"active", "disabled"}),
5    TYPE("type", null),
6    CHECKED("checked", new String[]{"true", "false"}),
7    NAME("name", null),
8    ITEM("item", null),
9    LANG(Language.getKeyName(), Language.getValues()),
10   SELECT("select", new String[]{"single", "multi"}),
11   ORIENTATION("orientation", new String[]{"horizontal", "
       vertical"});
12
13   private final String keyName;
14   private final String[] attributeValues;
15
16   Attribute(String keyName, String[] attributeValues)
17   {
18     this.keyName = keyName;
19     this.attributeValues = attributeValues;
20   }
21 }
```

Listing 5.2: Attribute eines Elementes in der FormDSL

Spracherweiterung Formulare, welche durch die FormDSL erstellt wurden, können in mehreren Sprachen angezeigt werden, falls die Texte des Formulars in mehreren Sprachen definiert wurden. Die Anzahl der Sprachen ist als Enum implementiert und erweiterbar. Listing 5.3 zeigt einen Ausschnitt des Enums. Um eine Sprache zu erweitern muss die Sprache mit ihrem Namen und verschiedenen Namenskürzeln angegeben werden.

```
1  public enum Language
2  {
3    GERMAN("german", "de", "deu"),
```

```
4     ENGLISH("english", "en", "eng"),
5     FRENCH("french", "fr", "fra");
6
7     private final String languageName;
8     private final String token2;
9     private final String token3;
10
11    private static final String keyName = "lang";
12    private static final String defaultToken = "token2";
13
14    Language(String languageName, String token2, String
          token3)
15    {
16       this.languageName = languageName;
17       this.token2 = token2;
18       this.token3 = token3;
19    }
20 }
```

Listing 5.3: Sprachen in der FormDSL

Anahng

Anhang A Lexer

```
1
2  lexer grammar FormDslLexer;
3
4  // ============================================
5
6  Comma:          ',' ;
7  Colon:          ':' ;
8  Semicolon:      ';' ;
9  Dot:        '.' ;
10 At:         '@' ;
11 BraceOpen:      '{' ;
12 BraceClose:     '}' ;
13 BracketOpen:    '[' ;
14 BracketClose:   ']' ;
15 ParensOpen:     '(' ;
16 ParensClose:    ')' ;
17 Equals:       '=' ;
18
19
20 // ============== keywords ====================
21
22
23 KwFormDsl:      'formDsl'    ;
24 KwForm:       'form'       ;
25 KwConstraints:   'constraints' ;
26 KwData:       'data'       ;
27
28 KwGroup:      'group'      ;
29 KwSection:      'section'    ;
```

```
30 KwField:      'field'      ;
31
32 KwItem:       'item'       ;
33
34
35 // ========== operators ======================
36
37 ComparisonOperatorWithoutEquals:
       ComparisonOperatorCharWithoutEquals ;
38
39 fragment ComparisonOperatorCharWithoutEquals:   '!=' | '>'
       | '>=' | '<' | '<=' ;
40
41 OrOperator: '|' ;
42
43 AndOperator: '&' ;
44
45
46 ThenOperator: ThenOperatorSign ;
47
48 fragment ThenOperatorSign: 'dann' | '-->';
49
50
51 // ============== numeral ====================
52
53 DecimalNumeral: Digits ;
54
55
56 fragment Digits: Integer+ ;
57
58 fragment Integer: [0-9] ;
59
60 // ============================================
61
62 Characters: JavaLetter+ ;
63
64 Identifier: IdentifierChars ;
65
66
67 fragment IdentifierChars: JavaIdLetter JavaLetterOrDigit*
       ;
68
69 fragment JavaLetter: [A-Z] | [a-z] ;
```

```
70
71 fragment JavaIdLetter: JavaLetter | '_' ;
72
73 fragment JavaLetterOrDigit: JavaIdLetter | [0-9] ;
74
75
76 // ================== text ====================
77
78
79 Text: Quote TextChars Quote;
80
81 fragment Quote: '"' ;
82
83 fragment TextChars: ~('"')* ;
84
85
86 // ============================================
87
88
89 Dummy: '$$$' ;
90
91 // ============================================
92
93 WS: [ \t\n\r]+ ;
94
95
96
97 // EOF.
```

Listing 5.4: FormDsl Lexer

Anhang B Parser

```
1
2  parser grammar FormDslParser;
3
4  @parser::header {
5    import org.sweble.wom3.Wom3Node;
6  }
7
8  options {
9    tokenVocab=FormDslLexer;
10 }
11
12 // ================================================
13 //           form dsl
14 // ================================================
15 // the form dsl and the three main sections of the dsl are
       defined in this section
16
17
18 // formDsl { ... }
19 formDslDef returns [ Wom3Node result]:
20   form = formDef wsFrom_ = WS?
21   (constraint = constraintsDef wsConstraints_ = WS?)?
22   (dataStructure = dataStructureDef wsDataStructure_ = WS
       ?)?
23 ;
24
25 // form { ... }
26 formDef returns [ Wom3Node result ]:
27   form = KwForm ws_BraceOpen = WS? braceOpen = BraceOpen
       wsBraceOpen_ = WS?
28   (panelElements += panelElementDef)+
29   ws_BraceClose = WS? braceClose = BraceClose
       wsBraceClose_ = WS?
30 ;
31
32 // constraints { ... }
33 constraintsDef returns [ Wom3Node result ]:
34   kwConstraint = KwConstraints ws_BraceOpen = WS?
       braceOpen = BraceOpen wsBraceOpen_ = WS?
35   (constraints += constraintFieldDefWithMaybeWs )*
```

```
36    ws_BraceClose = WS? braceClose = BraceClose
          wsBraceClose_ = WS?
37  ;
38
39  // data { ... }
40  dataStructureDef returns [ Wom3Node result ]:
41    kwData = KwData
42    ws_BraceOpen = WS? braceOpen = BraceOpen wsBraceOpen_ =
          WS?
43    (data += dataSectionDefWithMaybeWs)*
44    braceClose = BraceClose wsBraceClose_ = WS?
45  ;
46
47  // ================================================
48  //           data structure
49  // ================================================
50  // the elements of the data structure are defined in this
        section
51
52
53  dataSectionDefWithMaybeWs returns [ Wom3Node result ]:
54    dataSection = dataSectionDef wsDataSection_ = WS?
55  ;
56
57  dataSectionDef returns [ Wom3Node result ]:
58    kwSection = KwSection
59    ws_BracketOpen = WS? bracketOpen = BracketOpen
          wsBracketOpen_ = WS?
60    attributes = dataAttributesDef
61    ws_BracketClose = WS? bracketClose = BracketClose
          wsBracketClose_ = WS?
62    braceOpen = BraceOpen wsBraceOpen_ = WS?
63    dataNested = dataNestedDef?
64    braceClose = BraceClose
65  ;
66
67  dataNestedDef returns [ List<Wom3Node> result ]:
68    ( (dataSection += dataSectionDefWithMaybeWs)* )
69  | ( (dataField += dataFieldDefWithMaybeWs)* )
70  ;
71
72  dataFieldDefWithMaybeWs returns [ Wom3Node result ]:
73    dataField = dataFieldDef wsDataField_ = WS?
```

```
74 ;
75
76 dataFieldDef returns [ Wom3Node result ]:
77   kwField = KwField
78   ws_ParensOpen = WS? parensOpen = ParensOpen
       wsParensOpen_ = WS?
79   attributes = dataAttributesDef
80   ws_ParensClose = WS? parensClose = ParensClose
       wsParensClose_ = WS?
81   semicolon = Semicolon
82 ;
83
84 dataAttributesDef returns [ List<Wom3Node> result ]:
85   attributes += attributeDataStructureDef
86   (comma += commaWithMaybeWs attributes +=
       attributeDataStructureDef)*
87 ;
88
89 // ================================================
90 //          constraints
91 // ================================================
92 // the elements of the constraints are defined in this
     section
93
94
95 constraintFieldDefWithMaybeWs returns [ Object[] result ]:
96   constraint = constraintFieldDef wsElement_ = WS?
97 ;
98
99 constraintFieldDef returns [ Wom3Node result ]:
100  conditions = outerConditionDef
101  thenOp = thenOperatorWithMaybeWs
102  constraint = constraintDef
103  ws_Semicolon = WS? semicolon = Semicolon
104 ;
105
106 constraintDef returns [ String result ]:
107  parensOpen = ParensOpen wsParensOpen_ = WS?
108  access = accessDef
109  ws_ParensClose = WS? parensClose = ParensClose
110 ;
111
112 outerConditionDef returns [ Wom3Node result ]:
```

```
113     parensOpen = ParensOpen wsParensOpen_ = WS?
114     cond = innerConditionDef
115     ws_ParensClose = WS? parensClose = ParensClose
116 ;
117
118 innerConditionDef returns [ Wom3Node result ]:
119     singleCond = singleConditionDef
120   | multiCond = multiConditionDef
121 ;
122
123 singleConditionDef returns [ Wom3Node result ]:
124     access = accessDef comOp = comparisonOperatorWithMaybeWs
          value = idOrNummeral
125 ;
126
127 multiConditionDef returns [ Wom3Node result ]:
128     andCond = andConditionDef
129   | orCond = orConditionDef
130 ;
131
132 andConditionDef returns [ Wom3Node result ]:
133     andCond += outerConditionDef ( andOp +=
          andOperatorWithMaybeWs andCond += outerConditionDef)+
134 ;
135
136 orConditionDef returns [ Wom3Node result ]:
137     orCond += outerConditionDef ( orOp +=
          orOperatorWithMaybeWs orCond += outerConditionDef)+
138 ;
139
140 accessDef returns [ String result ]:
141     at = At name = identifierValue dot = Dot attr =
          identifierValue
142 ;
143
144
145 // ================================================
146 //              form
147 // ================================================
148
149
150 // ================================================
151
```

```
152 panelElementDef returns [ Wom3Node result]:
153   group = groupDefWithMaybeWs
154 | element = elementDefWithMaybeWs
155 ;
156
157 groupDefWithMaybeWs returns [ Object[] result ]:
158   group = groupDef wsGroup_ = WS?
159 ;
160
161 elementDefWithMaybeWs returns [ Object[] result ]:
162   element = elementDefWithSemicolon wsElement_ = WS?
163 ;
164
165 // ================================================
166
167 groupDef returns [ Wom3Node result]:
168   group = KwGroup ws_BraceOpen = WS?
169   braceOpen = BraceOpen wsBraceOpen_ = WS?
170   (elements += elementDefWithMaybeWs)+
171   braceClose = BraceClose
172 ;
173
174 // panelElement: $element$ ;
175 elementDefWithSemicolon returns [ Wom3Node result]:
176   element = elementDef ws_Semicolon = WS? semicolon =
        Semicolon
177 ;
178
179 elementDef returns [ Wom3Node result ]:
180   defaultE = defaultElementDef
181 | listE = listElementDef
182 ;
183
184 defaultElementDef returns [ Wom3Node result ]:
185   type = attributeValue element = elementDefCommon
186 ;
187
188
189 listElementDef returns [ Wom3Node result ]:
190   type = attributeValue element = elementDefCommon
191   ws_BracketOpen = WS? bracketOpen = BracketOpen
        wsBracketOpen_ = WS?
192   (items += itemDefWithSemicolon wsItem_ = WS?)*
```

```
193     bracketClose = BracketClose wsBracketClose_ = WS?
194 ;
195
196 elementDefCommon returns [ String[] result ]:
197     wsType_ = WS?
198     parensOpen = ParensOpen wsParensOpen_ = WS?
199     attributes = attributesDef
200     ws_ParensClose = WS? parensClose = ParensClose
201 ;
202
203 itemDefWithSemicolon returns [ Wom3Node result ]:
204     item = itemDef ws_Semicolon = WS? semicolon = Semicolon
205 ;
206
207 itemDef returns [ Wom3Node result ]:
208     type = KwItem element = elementDefCommon
209 ;
210
211
212 // ================================================
213
214
215 attributesDef returns [ List<Wom3Node> result ]:
216     attributes += attributeDef
217     (comma += commaWithMaybeWs attributes += attributeDef)*
218 ;
219
220 attributeDef returns [ Wom3Node result ]:
221     attributeLabel = attributeLabelDef
222   | attributeId = attributeIdDef
223 ;
224
225 attributeLabelDef returns [ Wom3Node result ]:
226     attributeInternat = attributeLabelDefInternat
227   | attributeNormal = attributeLabelDefNormal
228 ;
229
230 attributeLabelDefInternat returns [ Wom3Node result ]:
231     keyText = attributeValue colon = colonWithMaybeWs
          keyLang = attributeValue
232     equals = equalsWithMaybeWs value = textValue
233 ;
234
```

```
235 attributeLabelDefNormal returns [ Wom3Node result ]:
236   key = attributeValue equals = equalsWithMaybeWs value =
        attributeValue
237 ;
238
239 attributeIdDef returns [ Wom3Node result ]:
240   key = attributeValue equals = equalsWithMaybeWs value =
        identifierValue
241 ;
242
243 attributeDataStructureDef returns [ String result ]:
244   key = attributeValue equals = equalsWithMaybeWs value =
        dataStructureValue
245 ;
246
247 dataStructureValue returns [ String result ]:
248   id = conditionValue
249 | access = accessDef
250 ;
251
252 // ================================================
253 //           value types
254 // ================================================
255
256
257 attributeValue returns [ String result ]:
258   attr = Characters
259 ;
260
261 identifierValue returns [ String result ]:
262   chars = attributeValue
263 | id = Identifier
264 ;
265
266 decimalValue returns [ String result ]:
267   num = DecimalNumeral
268 ;
269
270 conditionValue returns [ String result ]:
271   id = identifierValue
272 | num = decimalValue
273 ;
274
```

```
275 textValue returns [ String result ]:
276   text = Text
277 ;
278
279 comparisonOperator returns [ String result ]:
280   op = ComparisonOperatorWithoutEquals
281 | eq = Equals
282 ;
283
284 idOrNummeral returns [ String result ]:
285   num = DecimalNumeral
286 | id = Identifier
287 ;
288
289
290 // ============================================================
291 //           signs with whitespaces
292 // ============================================================
293
294 commaWithMaybeWs returns [ String result ]:
295   ws_Comma = WS? comma = Comma wsComma_ = WS?
296 ;
297
298 colonWithMaybeWs returns [ String result ]:
299   ws_Colon = WS? colon = Colon wsColon_ = WS?
300 ;
301
302 equalsWithMaybeWs returns [ String result ]:
303   ws_Equals = WS? equals = Equals wsEquals_ = WS?
304 ;
305
306 pipeWithMaybeWs returns [ String result ]:
307   ws_Pipe = WS? pipe = Pipe wsPipe_ = WS?
308 ;
309
310 comparisonOperatorWithMaybeWs returns [ String result ]:
311   ws_operator = WS? comOperator = comparisonOperator
        wsOperator_ = WS?
312 ;
313
314 orOperatorWithMaybeWs returns [ String result ]:
315   ws_operator = WS? orOperator = OrOperator wsOperator_ =
        WS?
```

```
316 ;
317
318 andOperatorWithMaybeWs returns [ String result ]:
319   ws_operator = WS? andOperator = AndOperator wsOperator_
        = WS?
320 ;
321
322 thenOperatorWithMaybeWs returns [ String result ]:
323   ws_ThenOp = WS? thenOperator = ThenOperator wsThenOp_ =
        WS?
324 ;
325
326
327 // ================================================
328
329 //expression returns [ Wom3Node result ]:
330 //   dummy = Dummy
331 //;
332
333 // EOF.
```

Listing 5.5: FormDsl Parser

Licenses

ANTLR 4:
BSD License, `https://opensource.org/licenses/BSD-3-Clause`

Bootstrap:
MIT License, `https://opensource.org/licenses/MIT`

References

Data, R. (o. J.). *w3schools.com.* http://www.w3schools.com. (Access: 2016-01-21)

Dohrn, H. & Riehle, D. (2011a). Design and implementation of the sweble wikitext parser: Unlocking the structured data of wikipedia. In *The 7th international symposium on wikis and open collaboration.*

Dohrn, H. & Riehle, D. (2011b, July). Wom: An object model for wikitext. *Technical Reports, CS-2011-05* (5). Zugriff auf `sweble.org/downloads/wom-tr.pdf`

Duey, J. (o. J.). *A clojure story.* Zugriff auf `http://www.clojure.net/` (Access: 2016-01-18)

Louden, K. C. (1993). *Programming languages.* Brooks Cole.

Tomaz, K., Mernik, M., Crepinsek, M., Henriques, P. R., da Cruz, D., Pereira, M. J. V. & Oliveira, N. (2009). Influence of domain-specific notation to program understanding. In *International multiconference on computer science and information technology* (Bd. 4). IEEE.

van Deursen, A., Klint, P. & Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, *35* (6), 26-36.

Wenzel, M. (2015). *Extend and integrate a visual editor into the sweble wiki* (Unveröffentlichte Diplomarbeit). Friedrich-Alexander University Erlangen-Nürnberg.

Lebenslauf