



**THM**

TECHNISCHE HOCHSCHULE MITTELHESSEN

**CAMPUS  
GIESSEN**

**MNI**

Mathematik, Naturwissenschaften  
und Informatik

# Modelgetriebene Softwareentwicklung in der Praxis

Madelaine Tatjana Hestermann

18. Juli 2023

Dozent: Steffen Vaupel



## Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>1</b>
<b>2</b>	<b>Textuelle Beschreibung</b>	<b>1</b>
<b>3</b>	<b>Unterstützte HTML Elemente</b>	<b>2</b>
<b>4</b>	<b>Sprachspezifikation</b>	<b>2</b>
<b>5</b>	<b>Testen der Modellierungssprache</b>	<b>4</b>
<b>6</b>	<b>Testen des Generators</b>	<b>5</b>
<b>7</b>	<b>Architektur des Programms</b>	<b>6</b>
<b>8</b>	<b>Eingabeartefakte des Discoverers</b>	<b>7</b>
<b>9</b>	<b>Implementierung des Discoverers</b>	<b>7</b>
<b>10</b>	<b>Testen des Discoverers</b>	<b>7</b>
<b>11</b>	<b>Modellannotationen</b>	<b>8</b>
<b>12</b>	<b>Modellvorverarbeitung</b>	<b>8</b>
<b>13</b>	<b>Modellrestriktionen</b>	<b>8</b>
<b>14</b>	<b>Modell-Refactorings</b>	<b>9</b>
<b>15</b>	<b>Modell-Splitting</b>	<b>11</b>
<b>16</b>	<b>Spezifikationsloser Test</b>	<b>11</b>
<b>17</b>	<b>Erzeugung der Referenzseite</b>	<b>12</b>
<b>18</b>	<b>Ausblick</b>	<b>14</b>
<b>19</b>	<b>Programmausführung</b>	<b>15</b>



# 1 Einführung

Der vorliegende Bericht handelt von der Konzeption und Implementierung eines in der Programmiersprache Scala geschriebenen Generators zur automatischen Generierung von HTML Code. In den folgenden Abschnitten werden Umsetzung und Testung des Generators sowie der dazugehörigen Modellierungssprache und des dazugehörigen Discoverers vorgestellt. Anschließend werden Umfang und Eigenschaften des Generators diskutiert und zuletzt folgt ein Ausblick sowie ein kurzer Hinweis zur Nutzung des Programms.

## 2 Textuelle Beschreibung

Der Generator, der im Rahmen dieses Projektes entwickelt wurde, ist in der Lage Webseiten zu generieren, welche aus mindestens einer, andernfalls aber aus beliebig vielen Unterseiten besteht.

Diese Unterseiten, Pages genannt, bestehen aus jeweils einem Header, dann folgt jeweils ein Body und anschließend bestehen sie außerdem aus jeweils einem Footer.

Der Header besteht wiederum aus einem Bild und einer Navbar.

Der Body besteht aus einem oder mehreren frei wählbaren Body Elementen. Diese Elemente sind Bilder, geordnete, wie ungeordnete Listen, Formulare, Textbausteine, Links und Tabellen.

Der Footer besteht aus einem oder mehreren Links.

Die Navbar im Header setzt sich zusammen aus einer beliebigen Anordnung von Links und Dropdown-Menüs, die ihrerseits wieder einen oder mehrere Links enthalten können. Bilder verfügen über die Angabe ihres Dateipfades.

Sortierte und unsortierte Listen bestehen aus Listenelementen.

Formulare sind stets zusammengebaut aus einem Label und einer Textarea (mehrzeilig) oder einem Label und einem Input Feld (einzeilig). Des Weiteren verfügen Formulare standardmäßig über einen Submit Button.

Textbausteine setzen sich aus einer oder mehreren Überschriften und Paragraphen zusammen, die in beliebiger Anordnung und Anzahl auftreten können, solange stets mindestens eine Überschrift oder ein Paragraph angegeben ist.

Paragraphen verfügen über den Text, der als Inhalt dargestellt werden soll.

Überschriften verfügen sowohl über Text als auch über eine numerische Angabe in welcher Größe die Überschrift angegeben wird.

Links bestehen aus einer Textangabe, die auf der Website zu lesen sein wird, sowie einer Zieladresse, wohin der Link führt.

Tabellen setzen sich zusammen aus einem Tablehead und mehreren Tablerows. Der Tablehead besteht aus Tabellenspalten, welche die einzelnen Überschriften der Spalten darstellen sollen. Die übrigen Tablerows verfügen ebenfalls über Spalten, welche allerdings den Inhalt der Tabelle darstellen. Optisch unterscheiden sich die Spaltenelemente des Tableheads von den der Tablerows dadurch, dass die Spaltenelemente des Tableheads in einer Fettschrift dargestellt sind.

### 3 Unterstützte HTML Elemente

Die entwickelte Modellierungssprache ist in der Lage alle Elemente, welche in der textuellen Beschreibung genannt werden umzusetzen. In Abbildung 1 können alle Elemente und ihre Beziehung zueinander in Form eines Abstrakten Syntaxbaums betrachtet werden.

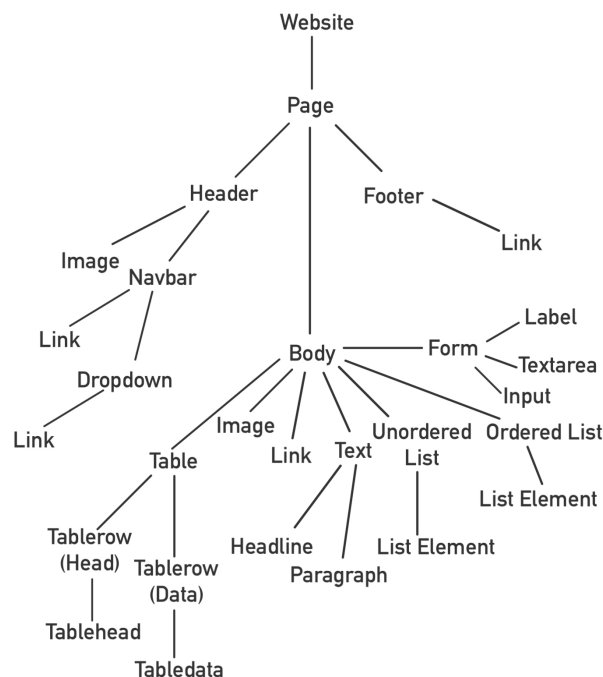


Abbildung 1: Abstrakter Syntaxbaum der Modellierungssprache

### 4 Sprachspezifikation

Im diesem Abschnitt wird die Modellierungssprache in ihrer Syntax vorgestellt und erläutert. Das folgende Listing 1 dient als Beispiel für eine mögliche Website, welche mit der Modellierungssprache abgebildet wird.

Listing 1: Beispiel der Modellierungssprache

1 Website :

```

3      (Page:
4        (Header:
5          (Image:( misc/Logo_THM_MNI.png)),
6          (Navbar:
7            (Link: (Startseite), (index.html)),
8            (Dropdown: (Veranstaltungen),
9              (Link: (Algorithmen), (algorithmen.html)),
10             (Link: (Betriebssysteme), (betriebssysteme.html)),
11             (Link: (Computergrafik), (computergrafik.html)),
12             (Link: (Archiv), (archiv.html))
13           ),
14           (Link: (Literaturempfehlungen), (literaturempfehlungen.html)),
15           (Link: (Stundenplan), (stundenplan.html))
16         ),
17       ),
18       (Body:
19         (Text:
20           (Headline 1: (Betriebssysteme)),
21           (Headline 4: (Kurzbeschreibung:)),
22           (Paragraph: (In der Veranstaltung werden Grundlagen vertieft.)),
23           (Headline 4: (Studiengang:)),
24           (Paragraph: (Informatik B.Sc., Ingenieur-Informatik B.Sc.)),
25           (Headline 4: (Nächste Veranstaltung:)),
26           (Paragraph: (Montag, 24.04.2023: 08:00 Uhr – 09:30 Uhr))
27         ),
28       ),
29       (Footer:
30         (Link: (Kontakt), (kontakt.html)),
31         (Link: (Impressum), (impressum.html))
32       ),
33     ),
34     (Page:
35       (Header:
36         (Image:( misc/Logo_THM_MNI.png)),
37         (Navbar:
38           (Link: (Startseite), (index.html)),
39           (Dropdown: (Veranstaltungen),
40             (Link: (Algorithmen), (algorithmen.html)),
41             (Link: (Betriebssysteme), (betriebssysteme.html)),
42             (Link: (Computergrafik), (computergrafik.html)),
43             (Link: (Archiv), (archiv.html))
44           ),
45           (Link: (Literaturempfehlungen), (literaturempfehlungen.html)),
46           (Link: (Stundenplan), (stundenplan.html))
47         ),
48       ),
49       (Body: (List ordered:
50         (Gogol-Döring, A.; Letschert, T.: Algorithmen für Dummies. Wiley.),
51         (Cormen, T. H.: Algorithmen. Eine Einführung. De Gruyter.),
52         (Skiena; S.: The Algorithm Design Manual. Springer.),

```

```

53         (Robert , Y.; Benoit , A.; Vivien , F.: A Guide To Algorithm Design.)
        )),
        (Footer :
55         (Link: (Kontakt) , (kontakt.html)) ,
          (Link: (Impressum) , (impressum.html))
57     )
    )

```

Die Modellierungssprache besteht aus einigen Schlüsselwörtern, sowie semantisch sensitiven Zeichen. Alle in der Sprache definierten Schlüsselwörter bilden ihrerseits ein Element der Sprache ab. Die folgende Liste umfasst alle dieser Schlüsselwörter:

{Website, Page, Header, Body, Footer, Navbar, Dropdown, Link, Text, Headline, Paragraph, List unordered, List ordered, Table, Tablerow, Form, Label, Input, Textarea, Placeholder, Id}

Jedem Schlüsselwort ist ein Doppelpunkt anzufügen. Sämtliche Elemente sind zudem mit runden Klammern umschlossen, um die Elemente syntaktisch voneinander zu trennen. Ausschließlich das Schlüsselwort Website wird als Startpunkt in den Code nicht in Klammern gepackt. Elemente welche in höherer Anzahl auftreten, sind Kommata getrennt.

Aus Gründen des Umfangs kann in diesem Bericht nicht auf die genaue syntaktische Verwendung jedes einzelnen Elementes eingegangen werden. Bei Interesse kann die Grammatik der Modellierungssprache allerdings im Parser der Sprache nachgeschaut werden. Die entsprechende Klasse trägt den Namen WebsiteParser.

## 5 Testen der Modellierungssprache

Für das Testen sämtlicher Programmteile wurde kein spezielles Testing Framework verwendet. Stattdessen habe ich händisch eigens eine Testumgebung entwickelt, die genau den nötigen Umfang abdeckt, um sowohl die Modellierungssprache als auch den Generator und den Discoverer ausreichend zu testen. Auf die Tests des Generators und des Discoverers wird in den folgenden Abschnitten eingegangen.

Die Klasse WebsiteParser definiert die Grammatik der Modellierungssprache und erbt von der Klasse RegexParser, die zu der genutzten Parserkombinator Bibliothek gehört. In der Klasse RegexParser sind einige Funktionen definiert, mit welchen Text eingelesen und geparkt werden können. Die in den Tests verwendete Funktion parseAll beispielsweise erwartet als Parameter einen Parser und anschließend den zu parsenden Text, wobei dieser dann komplett bis zum Ende gelesen wird. Die Funktion gibt dann einen Wert vom Typ ParseResult zurück. In diesem Wert befindet sich dann entweder das Ergebnis aus dem erfolgreich geparkten Text oder aber eine Fehlermeldung, sofern das Parsen nicht erfolgreich und der Text fehlerhaft war. Mithilfe des Scala Pattern Matching können diese beiden Alternativen im konkreten Fall dann abgefragt werden.

Diese Funktionalität eignet sich Unit Test artige Tests für sämtliche Elemente der Modellierungssprache zu entwickeln, da für jedes Element ein eigener Parser in der Klasse WebsiteParser definiert wurde. Als Tests wurden also für alle Elemente der Sprache ein Aufruf der Funktion parseAll mit einem korrekten, des Elements entsprechenden, Texts geschrieben, der den Text damit auf Korrektheit prüft. Praktisch wäre es möglich beispielsweise die daraus erstellten Instanzen abzufragen. Da sich die Struktur dieser Tests allerdings gut dazu eignet in einem



nächsten Schritt die korrekte Generierung abzuprüfen, wurde hier davon abgesehen, da die Überprüfung der korrekten Generierung semantisch die Überprüfung der Modellierungssprache selbst beinhaltet. Auf genauere Details der Generator Tests wird im nächsten Abschnitt eingegangen.

Auf diese Weise wurde das Testen aller Elemente einzeln und in Abhängigkeit zueinander abgedeckt und eine vollständige Testung ist damit gewährleistet. Des Weiteren wurde auch ein Test geschrieben, der Explizit einen Fehlerfall testet und durch einen beabsichtigten syntaktischen Fehler getriggert wird.

Sämtliche Tests der Modellierungssprache können in dem Scala Objekt `TestWebsiteParser` ausgeführt und angesehen werden.

Diese in den Tests definierten Modelle dienen, zusammen mit dem Modell in der Datei `input.txt`, als Beispielm Modelle, wobei sämtliche Modelle, welche nicht den Parser des Elements `Website` testen, selbstverständlich nur als Teilmodelle zu erachten sind.

## 6 Testen des Generators

Die Testumgebung in Bezug auf den Generator funktioniert mithilfe der sogenannten *Scala Assertions*. Dabei kann der Funktion `assert` ein boolescher Ausdruck übergeben werden, der eine *Assertion Exception* wirft, wenn der Ausdruck zum booleschen `false` ausgewertet wird.

Wie bereits im vorigen Abschnitt genannt, wurde das Testen des Generators in die Tests der Modellierungssprache integriert. Im Falle, dass ein Eingabetext syntaktisch gültig ist, wird die *Assertion Funktionalität* verwendet. Dies geschieht indem die `parseAll` Funktion im Erfolgsfall eine Instanz des geparsen Elements zurückgibt. Diese Instanzen besitzen eine Funktionsdefinition `toHtml`, welche je Instanz einen passenden HTML Code generiert. Innerhalb der Funktion `assert` kann dann die `toHtml` Funktion der Instanz aufgerufen und mit einem zu erwartenden Teil von HTML Code in Form eines Strings verglichen werden. Anschließend wird eine Meldung ausgegeben, dass Parsen und Generieren erfolgreich waren. Nochmals zur Erinnerung: Schlägt die Methode `assert` fehl, das heißt, entspricht der generierte HTML Code nicht dem zu erwartenden HTML Code, dann wird eine *Assertion Exception* ausgelöst. Damit würde das Programm abgebrochen werden und gibt direkte Rückmeldung darüber, ob ein Test erfolgreich war oder nicht.

Dieses Testen wurde, ebenso wie die Tests der Modellierungssprache für alle möglichen Teilmodelle erstellt.

Im Testablauf stellt nur der Test der Website, also der Test eines gesamten Beispielm Modells, einen Unterschied zu den sonstigen Modellen dar. Die Instanz des Website Elements besitzt keine `toHtml` Funktionsdefinition, hat stattdessen allerdings die Funktionen `buildWebsite` und `analyzeSemantics`. Für das Testen dieser Instanz wird die korrekte Ausführung dieser beiden Funktionen getestet, indem die durch die Funktion generierten Dateien eingelesen und abermals mithilfe der `assert` Funktion auf den zu erwartenden HTML Code getestet wird.

Die Funktion `analyzeSemantics`, welche die festgelegten Modellrestriktionen prüfen soll, wird in der Funktion `buildWebsite` verwendet und Fehlerfälle, die aus der Verletzung dieser Modellrestriktionen resultieren, werden ebenfalls vollständig getestet. Diese Tests sind im unteren Abschnitt des Objekts `TestWebsiteParser` zu finden und sind je mit einem Kommentar versehen, welcher semantische Fehler getriggert wird.

Somit sind auch die Generation aller Instanzen und alle Modellrestriktionen durch Tests abgedeckt.

## 7 Architektur des Programms

Die grobe Architektur des Programms kann in der Abbildung 2 betrachtet werden.

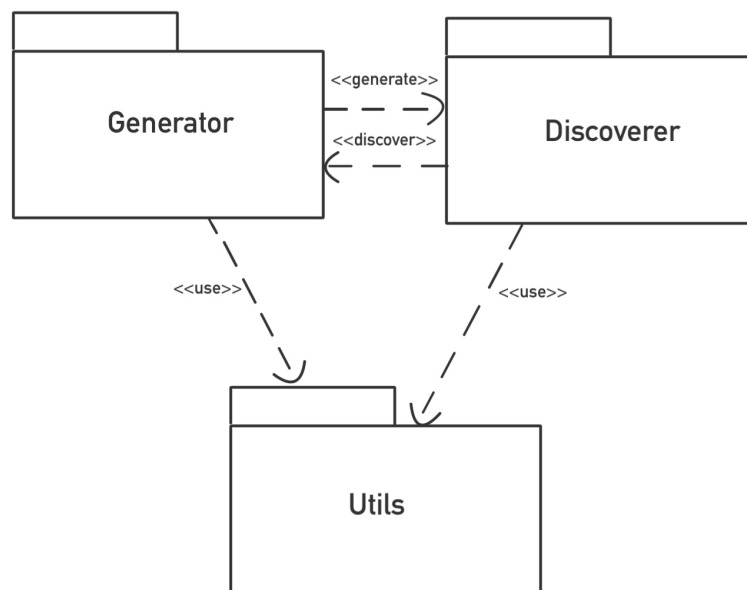


Abbildung 2: Architektur des Gesamtprogramms

Dabei enthält das Modul Generation sämtliche Klassen und Objekte zur Definition und Nutzung der Modellierungssprache, sowie den Generator selbst und das Objekt `TestWebsiteParser` zum Testen des Parsers und des Generators. Das Modul Discovering enthält die `Discoverer` Klasse und ein Test Objekt `TestDiscoverer` und das Modul Utils enthält zwei Klassen, `Reader` und `Writer`, welche entsprechende Lese- und Schreibfunktionalität implementieren. Des Weiteren gibt es eine `Main` Klasse, mit welchen sowohl der Generator als auch der Discoverer genutzt werden können.

Innerhalb des Moduls Generation sind die Definition der Modellierungssprache und der Generator durch ihre unterschiedlichen Klassen und Objekte klar getrennt. Das Objekt `Absyn`, welches sich in dem Modul befindet, bildet die Schnittstelle zwischen Modellierungssprache und Generator, indem sie die von der Sprache erstellten konkreten Instanzen aller Elemente definiert, welche ihrerseits alle eine Funktionsdefinition, `toHtml`, besitzen, die vom Generator für die Codegeneration genutzt wird. Außerdem besitzen diese Elementinstanzen eine weitere Funktionsdefinition, `toString`, welche das Element in Form der Modellierungssprache repräsentiert. Somit stellt das Objekt `Absyn` auch eine modulübergreifende Schnittstelle zwischen den geparsen Elementen und dem Discoverer dar.

## 8 Eingabeartefakte des Discoverers

Der Discoverer ist ein rein textbasierter Parser. Er parst HTML Code und kann daraus wieder Code der domänenspezifischen Modellierungssprache erzeugen. Aufgrund dessen wurde entschieden, dass die einzig zulässigen Eingabeartefakte HTML Dateien sind. Dateien eines anderen Formats kann der Discoverer nicht sinnvoll verarbeiten.

## 9 Implementierung des Discoverers

Implementiert wurde der Discoverer als ein Parser, der HTML Code einliest und durch Identifikation aller variablen Bestandteile des HTML Codes Elementinstanzen erstellen kann. Dies funktioniert indem der Discoverer durch String Operationen Beginn und Ende aller variablen Bestandteile lokalisiert, diese in Variablen zwischenspeichert um entweder direkt das zu erstellende Element zu instanziiieren, wenn dieses ein Blattelement in der Repräsentation des abstrakten Syntaxbaums ist, oder alle discover Funktionen der Kindelemente aufzurufen, bis alle notwendigen Informationen vorliegen, um das zu erstellende Element zu instanziiieren.

Hierfür wurde entsprechend für sämtliche Elemente jeweils eine discover Funktion geschrieben, um den Code hinreichend zu modularisieren und unterschiedliche Abhängigkeiten der Elemente untereinander realisieren zu können.

Der Discoverer kann dabei nur HTML Code genau so, wie der Generator ihn generiert, sinnvoll interpretieren. Allerdings ist es dem Discoverer möglich alle nicht bekannten Elemente zu überlesen, sodass in HTML Code, in welchem nur Teile fremd sind, diejenigen Abschnitte, die dem Discoverer bekannt sind, dennoch korrekt interpretiert werden können. Der Abschnitt, der von dem Discoverer überlesen wird, wird dann in eine Textdatei namens failure.txt geschrieben, die im Anschluss vom Benutzer zum Auslesen von Fehlern genutzt werden kann.

Dadurch, dass der Discoverer selbst eigentlich nur Instanzen aus dem HTML Code liest und die Instanzen durch ihre toString Funktionen selbst wissen, wie ihre korrekte Repräsentation in Form der Modellierungssprache aussieht, ist es praktisch nicht möglich syntaktisch nicht validen Code der Modellierungssprache im Discovering Prozess zu generieren. Entweder eine Instanz wurde erkannt und kann stets wieder in korrekten HTML Code oder korrekten Code der Modellierungssprache umgewandelt werden, oder eine Instanz wird nicht korrekt erkannt. Dann wird sie aber beim Discovering Prozess allerdings auch gänzlich verworfen und kann daher keinen falschen Code produzieren. Um sich zu vergewissern, dass der vom Discoverer erzeugte Modellierungssprachen Code wirklich valide ist, kann stets die Testinfrastruktur des Parsers und Generators genutzt werden. Automatisch passiert diese Validierung allerdings nicht, da sie eben praktisch tatsächlich nicht möglich sein kann.

## 10 Testen des Discoverers

Die Tests für den Discoverer sind in eine ähnliche Testumgebung eingebettet, wie die Tests, welche die Funktionalität des Generators validieren. Für die Umgebung wurden wieder die Scala Assertions verwendet, wobei die Tests für den Discoverer stets einen Input String erwarten, welche ein Stück HTML Code repräsentieren, der dann discover wird. Der Discoverer parst

dann den Text und erstellt, wie im vorigen Abschnitt erklärt, bei korrektem Eingabetext die entsprechende Instanz, auf welche dann die `toString` Funktion der Instanz ausgeführt wird, die also die Modellierungssprachenrepräsentation des Elements erstellt. Daraufhin wird diese Repräsentation mit einem zu erwartenden String verglichen und als Argument in die `assert` Funktion übergeben, damit diese abermals prüft, ob die Ausgabe und das zu erwartende Ergebnis gleich sind. Bei korrektem Eingabeinput wird für alle korrekt gefundenen Instanzen jeweils das Ergebnis anschließend auf die Standardausgabe ausgegeben.

Dieses Prinzip wurde, wie beim Generator, für alle alleinstehenden Teilelemente der Sprache sowie für alle zusammengesetzten Teilelemente und die gesamte Website durchgeführt, sodass hier nun auch wieder eine vollständige Testabdeckung herrscht.

Des Weiteren findet sich bei den Tests zum Discoverer auch ein Test, der den Fall abdeckt, dass ein Teil des HTML Codes nicht richtig ist und somit überlesen werden muss. In diesem konkreten Test geht es um das Discovern eines Textes, dessen HTML Repräsentation eine h4-Überschrift, sowie einen Paragraphen enthält, wobei der Code für die Überschrift nicht korrekt ist. In der Eingabe wurde im öffnenden Tag die schließende Klammer weggelassen, sodass sich hieraus ein ungültiger Abschnitt HTML Code ergibt. Der Test prüft dann darauf, dass diese falsche Überschrift überlesen und nur der darauffolgende Paragraph korrekt discovered wird, was anschließend wieder über die `assert` Funktion validiert wird.

Die Tests zum Discoverer befindet sich im Scala Object `TestDiscoverer` und können dort ausgeführt und nachvollzogen werden.

## 11 Modellannotationen

Modellannotationen sind in der Modellierungssprache nicht vorgesehen. Die Sprache soll als domänenspezifische Sprache insbesondere Personen dienen, welche keine oder kaum Programmierkenntnisse haben und sich insbesondere mit den Eigenschaften von HTML nicht auskennen. Modellannotationen fordern aber genau solche Kenntnisse, um überhaupt sinnvoll genutzt werden zu können. Entsprechend wurden diese bei der Entwicklung der Modellierungssprache nicht berücksichtigt.

## 12 Modellvorverarbeitung

Modellvorverarbeitungen finden ebenfalls nicht statt, da es hierfür keine Notwendigkeit gibt. Das Modell besteht stets nur aus einer textuellen Angabe in Form der Modellierungssprache, die direkt geparkt und anschließend vom Generator verarbeitet werden kann.

## 13 Modellrestriktionen

Modellrestriktionen definiert die Modellierungssprache in verhältnismäßig hoher Anzahl. Hierbei werden alle Fehler abgedeckt, die ein Modell erzeugen kann und zuvor nicht durch Einschränkungen in der Syntax abgefangen wurden.

Eine Reihe gleichartiger Modellrestriktionen gilt dabei für die Sprachelemente `Tablerow`, `Text`, `Unordered` und `Ordered List`, `Dropdown`, `Body` und `Footer`. Die Grammatik für diese Elemente ist vom Parser durch eine Funktion namens `repsep` definiert, die eine Liste von Elementen, mit einem beliebigen Separator separiert, parst. Hierdurch ist es möglich, dass Kommata getrennte Listen von Elementen gelesen werden können. Diese Funktion parst allerdings beliebig viele solcher Elemente, auch keins. Es ist daher möglich zum Beispiel ein Textelement zu definieren, das leer ist. Der Parser würde eine solche Definition als korrekt und valide interpretieren. Entsprechend muss nach dem Parsen eine weitere Funktionalität implementiert sein, die es aber dennoch verbietet, dass Elemente leer sind. Der Generator prüft also ob die oben genannten Elemente leere Listen enthalten oder nicht. Ist dies der Fall, wird eine Fehlermeldung ausgegeben und das Programm mit einem Fehlercode abgebrochen.

Dazu ergibt sich eine weitere Restriktion aus der Verwendung von Forms. In der Modellierungssprache bestehen die Inhalte einer Form stets aus Labels in Kombination mit entweder einzeiligen Input Feldern oder mehrzeiligen Textareas. Die entsprechenden HTML Tags der Labels, Input Feldern und Textareas besitzen ein Attribut, womit eine Zuordnung von Label und Input Feld oder Textarea hergestellt wird. Bereits HTML zwingt den Programmierer dazu das Label mit dem darauffolgenden Input Feld oder der darauffolgenden Textarea mit jeweils dem selben Zuordnungsattribut zu versehen. Andernfalls wird ein Fehler angezeigt und die Website kann nicht korrekt dargestellt werden. Da die Modellierungssprache darauf ausgelegt ist, insbesondere von Personen ohne HTML Kenntnissen genutzt zu werden, ist es wesentlich, dass ein solcher Fehler bereits von der Modellierungssprache selbst abgefangen wird, um der Person die Auseinandersetzung mit dem HTML Fehler dahinter zu ersparen. Hierbei prüft der Generator also ob die ID des Labels zur ID des Input Felds oder der Textarea passen. Sind diese ungleich, wird auch hier eine Fehlermeldung ausgegeben und das Programm endet mit einem Fehlercode.

Des Weiteren wird überprüft, ob die Anzahl von Zeilen und Spalten in Tabellen übereinstimmt und der Generator erlaubt nur bis zu einschließlich zehn Elemente in der Navbar, um zu verhindern, dass die Navbar, welche sich horizontal ausdehnt, in der Breite unübersichtlich lang wird. Auch diese beiden Fehler erzeugen bei ihrem Aufkommen eine Ausgabe mit einer Fehlermeldung und den Programmabbruch mit Fehlercode.

Alle Restriktionen werden gesondert im Objekt `TestWebsiteParser` getestet, indem ihre Fehler absichtlich getriggert und auf die korrekte Fehlermeldung mithilfe der `assertion` Funktionalität geprüft wird. Somit ist die Testabdeckung auch für alle Fehlerfälle, die der Generator bearbeiten muss, vollständig.

## 14 Modell-Refactorings

Einige der Sprachkonstrukte, welche die Modellierungssprache definiert, sind anfällig für Code-Smells. Dadurch, dass die Modellierungssprache eine deklarative Sprache ist, ist die Varietät der Code-Smells aber eher gering. Denn die Syntax und damit die Art, wie die Sprache zu verwenden ist, lässt wenig Spielraum dafür semantisch gleiche Dinge auf syntaktisch unterschiedlicher Art darzustellen, wie es in imperativen Sprachen oft sehr gehäuft möglich ist. Die Sprachkonstrukte, bei welchen Code-Smells aber auftreten können, sind im Wesentlichen solche, bei welchen eine beliebige Anzahl an Elementen in Form einer Liste eingebettet werden kann. Diese Einbettung von Elementen in ein übergeordnetes Element stellt auch die Weise dar, wie die Sprache ver-

wendet werden sollte, um eine hinreichende Modellqualität zu gewährleisten. Ein Smell würde dann auftreten, wenn stattdessen das übergeordnete Element mehrfach hintereinander mit nur einem oder wenigen Unterelementen genutzt würden, obwohl alle Unterelemente in ein einziges Überelement einsortiert werden könnten.

Ein konkretes Beispiel für solche Smells lässt sich anhand des Elements Text gut darstellen. In Listing 2 ist ein Beispiel eines Textelements zu sehen, wie es tatsächlich verwendet werden sollte.

Listing 2: Beispiel eines Textelements in guter Modellqualität

```
(Text :
2   (Headline 4: (Kurzbeschreibung:)),
   (Paragraph: (In der Veranstaltung werden Grundlagen vertieft.))
4 )
```

In Listing 3 ist ein Beispiel eines Textelements, das einen Code-Smell aufweist. Hier wurde hintereinander das übergeordnete Element Text zweimal mit je einem einzelnen Unterelement verwendet, obwohl die Unterelemente problemlos zusammen in ein Textelement gefasst werden können, ohne die Semantik des Modells zu verändern.

Listing 3: Beispiel eines Textelements mit Code-Smell

```
(Text :
2   (Headline 4: (Kurzbeschreibung:))
   ),
4 (Text :
   (Paragraph: (In der Veranstaltung werden Grundlagen vertieft.))
6 )
```

Elemente, bei welchen diese Art von Code-Smell auftreten können sind neben Textelementen auch Unordered Lists und Forms. Alle weiteren Elemente, welche Listen von Unterelementen besitzen, weisen semantische Unterschiede auf, je nachdem ob Unterelemente in einem Element zusammengefasst, oder mehrere Elemente mit voneinander getrennten Unterelementen erstellt werden. So hat beispielsweise die Aufteilung von Listenelementen auf zwei Ordered Lists den Effekt, dass beim Beginn der zweiten Liste die Zählung der Listenelemente selbstverständlich wieder bei 1 beginnt, wohingegen die Zusammenfassung aller Listenelemente in einer Ordered List bewirkt, dass alle Elemente von 1 an fortlaufend durchnummeriert werden.

Die Analyse, um automatisiert solche Code-Smells zu finden, ist praktisch relativ einfach. Ein Refactoring Modul, welches dem Programm hinzugefügt werden könnte, würde dabei das Modell einfach nach dem Vorkommen von Textelementen, Unordered Lists und Forms absuchen und dann einen Code-Smell vermelden, sobald diese Elemente jeweils direkt hintereinander zwei mal oder häufiger verwendet würden.

Ein automatisches Refactoring könnte erfolgen, indem die gefundenen hintereinander definierten Elemente auf ein einzelnes Element zusammengeschmolzen und all ihre Unterelemente in der richtigen Reihenfolge diesem einen Element hinzugefügt werden. Dies ist ebenfalls programmatisch leicht umsetzbar. Alternativ kann ein manuelles Refactoring natürlich auch stets geschehen. Dafür muss der Nutzer der Sprache händisch die obsoleten Elemente auf ein einzelnes zusammenführen. Dabei kann ihm die automatische Analyse helfen, indem sämtliche Code-Smells ermittelt und angezeigt werden, was insbesondere in großen Modellen sehr hilfreich sein kann.

## 15 Modell-Splitting

Die Modellierungssprache bietet ein relatives hohes Maß an Model-Splitting. Viele der Elemente lassen sich unabhängig und ohne Kenntnis von anderen Elementen definieren. Um den Grad zu bestimmen, inwiefern die Sprache aufteilbar ist, lässt sich die Modellierungssprache prinzipiell in zwei verschiedene Arten Sprachkonstrukte unterscheiden. Zum einen sind dies Elemente, die in ihrem Überelement gehäuft vorkommen und zum anderen sind das Elemente, die in ihrem Überelement genau einmal auftreten.

Alle Elemente, die genau einmal in ihrem Elternelement auftreten, sind praktisch nicht aufteilbar. Dazu zählen der Header, die Navbar, der Body und der Footer. Beim modellieren einer Website, müssen diese Elemente je Seite genau einmal definiert werden und können ohne Kenntnis über ihre Kindelemente nicht sinnvoll umgesetzt werden. Eine von anderen Elementen unabhängige Bearbeitung ist also kaum möglich.

Die Elemente, die jedoch öfter als einmal in ihrem Elternelement vorkommen, sind allesamt sehr gut aufteilbar, zumindest in dem Maße in dem die Elemente als ein in sich geschlossenes Konstrukt erachtet werden. Für alle Elemente, die zum Beispiel im Body vorkommen, gilt nämlich, dass diese in ihrer Gesamtheit völlig unabhängig von allen anderen Elementen definiert werden können. Der Modellierer muss keine Kenntnisse über den Text, der auch im Body vorkommt, haben, wenn er gerade eine Tabelle modelliert. Bei einer feingranularen Betrachtung der einzelnen Elemente im Body, kann jedoch festgestellt werden, dass das Splitting hier stellenweise nicht mehr ganz so einfach vollziehbar ist. Im Beispiel der Tabelle etwa kann es durchaus relevant sein, Kenntnis über die gesamte Tabelle zu haben, um beispielsweise der Modellrestriktion gerecht zu werden, dass die Anzahl von Zeilen und Spalten in der Tabelle gleich sein muss. Beim Modellieren reicht es also nicht sich mit den Tabledatas auseinanderzusetzen ohne sich nicht auch Gedanken über die anderen Teilelemente der Tabelle zu machen. Außerdem gilt für alle Elemente, dass die Reihenfolge, in welcher sie definiert werden, relevant ist, da diese natürlich Auswirkung auf die Platzierung der Elemente in der fertigen Website hat. Abgesehen davon können aber sonstige Body Elemente stets unabhängig betrachtet werden.

Was sich ebenfalls sehr gut und vermutlich auch am sinnvollsten für das Modell-Splitting eignet, ist das Splitting nach Pages. Es ist möglich jede Page unabhängig von den anderen Pages zu erstellen und zu bearbeiten. Einzig in der gesamten Website muss beim Zusammenfügen darauf geachtet werden die einzelnen Pages wiederum in die richtige Reihenfolge zu bringen, da die Erstellung und Benennung der generierten HTML Dateien maßgeblich von der Reihenfolge der Pages abhängt.

## 16 Spezifikationsloser Test

Ein möglicher spezifikationsloser Test des Generators, den man durchführen könnte, bestünde aus irgendeinem zufälligen Modell, welches dem Generator zum Kompilieren übergeben wird. Dabei würde die Überprüfung, ob die Ausgabe valide ist oder nicht, bestenfalls von einer Person durchgeführt werden, die intuitiv und ohne zuvor eine Erwartung entwickelt zu haben, wie die Ausgabe aussehen sollte, entscheidet, ob das Ergebnis nun gut ist oder nicht.



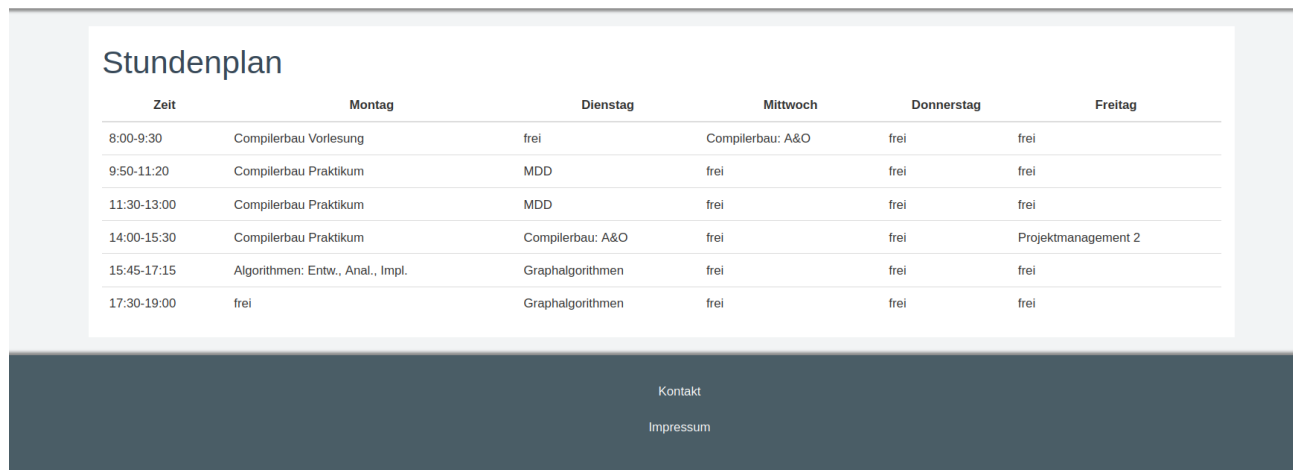
## 17 Erzeugung der Referenzseite

Die Referenzwebsite kann von dem entwickelten Generator in einer zufriedenstellenden Genauigkeit generiert werden. Die meisten generierten Seiten sind dabei optisch identisch zu der Referenz. Bei einigen gibt es kleine Unterschiede in der genauen Positionierung der Elemente oder bei Abständen zwischen den Elementen. Dies sind letztlich allerdings Details, worunter die Ästhetik der generierten Seiten nicht leidet.

Ein Beispiel hierfür ist die Seite, welche den Stundenplan darstellt. Hier ist die größte Abweichung zwischen der originalen Referenzseite und der generierten Seite zu erkennen. In Abbildung 3 ist die originale Referenzseite sehen. Darunter zeigt Abbildung 4 die generierte Seite.

Die Tabelle der generierten Seite wurde, wie zu sehen ist, gegenüber der originalen Seite in der Breite zusammengestaucht. Außerdem sind die Inhalte in den Zellen der Tabelle zentriert, was im Original nicht der Fall ist. Wie bereits genannt, leidet die Ästhetik allerdings nicht unter der leichten Änderung der Optik. Persönlich empfinde ich die generierte Version sogar ansprechender als die Referenz, da sie übersichtlicher erscheint und weniger breite Lücken zwischen den Spalten enthält.





**THM** | CAMPUS GIESSEN | MNI  
TECHNISCHE HOCHSCHULE MITTELHESSEN  
Mathematik, Naturwissenschaften und Informatik

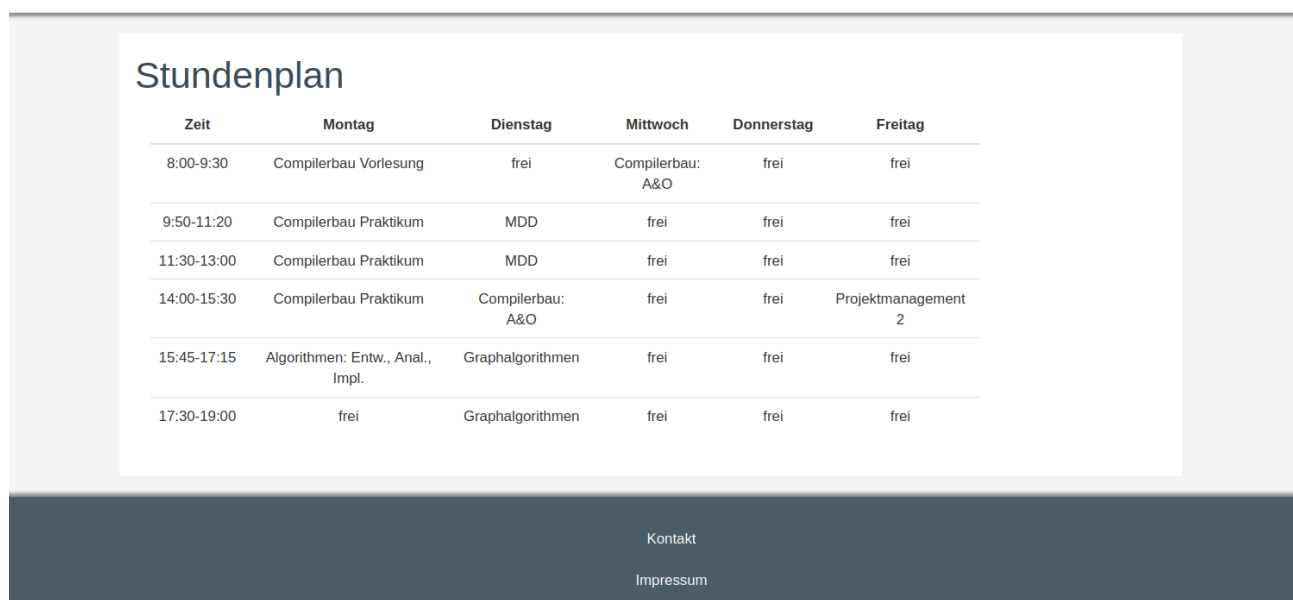
[Startseite](#) [Veranstaltungen -](#) [Projekte -](#) [Literaturempfehlungen](#) [Stundenplan](#)

### Stundenplan

Zeit	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
8:00-9:30	Compilerbau Vorlesung	frei	Compilerbau: A&O	frei	frei
9:50-11:20	Compilerbau Praktikum	MDD	frei	frei	frei
11:30-13:00	Compilerbau Praktikum	MDD	frei	frei	frei
14:00-15:30	Compilerbau Praktikum	Compilerbau: A&O	frei	frei	Projektmanagement 2
15:45-17:15	Algorithmen: Entw., Anal., Impl.	Graphalgorithmen	frei	frei	frei
17:30-19:00	frei	Graphalgorithmen	frei	frei	frei

[Kontakt](#)  
[Impressum](#)

Abbildung 3: Stundenplan der Referenzseite



**THM** | CAMPUS GIESSEN | MNI  
TECHNISCHE HOCHSCHULE MITTELHESSEN  
Mathematik, Naturwissenschaften und Informatik

[Startseite](#) [Veranstaltungen -](#) [Projekte -](#) [Literaturempfehlungen](#) [Stundenplan](#)

### Stundenplan

Zeit	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
8:00-9:30	Compilerbau Vorlesung	frei	Compilerbau: A&O	frei	frei
9:50-11:20	Compilerbau Praktikum	MDD	frei	frei	frei
11:30-13:00	Compilerbau Praktikum	MDD	frei	frei	frei
14:00-15:30	Compilerbau Praktikum	Compilerbau: A&O	frei	frei	Projektmanagement 2
15:45-17:15	Algorithmen: Entw., Anal., Impl.	Graphalgorithmen	frei	frei	frei
17:30-19:00	frei	Graphalgorithmen	frei	frei	frei

[Kontakt](#)  
[Impressum](#)

Abbildung 4: Stundenplan der generierten Seite

Ein Beispiel für eine Seite, die der Generator identisch zur Referenz erstellt hat, ist die Seite zur Veranstaltung Effiziente Algorithmen in der Computergrafik. Diese Seite kann in der Abbildung 5 betrachtet werden.



Abbildung 5: Seite zur Veranstaltung Effiziente Algorithmen in der Computergrafik

Alle weiteren generierten Seiten können in der Abgabe betrachtet werden. Die entsprechenden Dateien tragen die Namen file1.html-file11.html und wurden aus der Textdatei input.txt heraus generiert. Die originalen Seiten der Referenzwebsite ist in dem zusätzlichen Ordner namens Referenzwebsite zu finden.

Achtung: Bei Ausführung des Generators werden alte Dateien automatisch überschrieben. Das heißt, dass sobald die Datei input.txt zu einem neuen validen Modell umgeschrieben wurde, werden alle alten generierten HTML Dateien gelöscht und es liegen neue vor. Genauere Hinweise zur Nutzung der Programmmodule befinden sich in der Readme des Projekts.

## 18 Ausblick

Der Generator kann insbesondere im Hinblick auf Styling und Präzision verbessert und erweitert werden. Wie in den Unterschieden aus der generierten Referenzwebsite und der originalen Referenzwebsite zu erkennen ist, ist der Generator nicht dazu in der Lage Elemente variabel zu positionieren. Des Weiteren kann der Generator nicht auf die genutzten CSS Dateien zugreifen und entsprechende Änderungen im Styling vornehmen. Auch sämtliche Styling Optionen in Form von Attributen in den HTML Tags beeinflusst der Generator grundsätzlich nicht. All diese Werte sind statisch und bisweilen daher vorgegeben. Eine interessante Erweiterung des Generators bestünde also darin den Generator und die Modellierungssprache dahingehend anzupassen, dass auch solche Werte in den HTML und CSS Dateien automatisch generiert würden.

Die Anpassung der Modellierungssprache dahingehend ist relativ einfach und mit wenig Aufwand verbunden. Hierzu müssten die entsprechenden Objektdefinitionen der Sprachinstanzen jeweils zusätzliche Parameter erhalten, die das jeweilige Styling Attribut repräsentieren. Dieses Styling Attribut könnte seinerseits selbst als Knoten im AST realisiert werden, indem auch hierfür eine eigene Objektdefinition und damit ein Typ definiert würde. Letztlich würde das Styling Attribut bei dieser Vorgehensweise so interpretiert und verarbeitet werden wie Compiler von imperativen, stark typisierten Programmiersprachen die Typen der Sprachen verarbeitet. Rein semantisch passt diese Analogie auch, da sowohl Typen in Programmiersprachen, als auch die Styling Attribute in HTML Code die wesentlichen Eigenschaften ihrer Träger beschreiben und definieren.

Die reine Grammatik der Modellierungssprache könnte dann so angepasst werden, dass jedes Sprachelement wie im bisherigen Stil einen zusätzlichen Klammerausdruck mit dem Styling Attribut und gegebenenfalls neuen Schlüsselwörtern erhält.

Das Listing 4 ist ein Beispiel für die mögliche syntaktische Repräsentation für das Styling.

Listing 4: Beispiel eines Teilmodells mit gestyltem Paragraph

```
(Paragraph: (Styling: (Colour: (Red))), (Content.))
```

Zusätzlich müsste dann selbstverständlich auch der Discoverer angepasst werden, da dieser Instanzen bislang nur ohne Styling Attribute kennt. In diesem Schritt steckt vermutlich die meiste Arbeit, würde das Programm um Styling erweitert werden, da im Discoverer auch der Großteil des bereits bestehenden Codes überarbeitet werden müsste.

Des Weiteren könnte es außerdem spannend sein das Programm um einen Modell-Refactorer zu erweitern. Wie bereits im entsprechenden Abschnitt beschrieben, ist sowohl die Analyse, als auch das automatische Refactoring relativ leicht umzusetzen, bietet gleichzeitig aber einen großen Mehrwert, um große Modelle qualitativ hochwertig zu gestalten oder auch um das Modell einfach nur nach den Vorkommnissen der einzelnen Sprachelemente analysieren zu können.

## 19 Programmausführung

Sämtliche notwendige Informationen bezüglich der Ausführung des Generators, des Discoverers oder der Tests befinden sich in der Readme des Projekts. Bei weiteren Fragen und Unklarheiten, technischer wie inhaltlicher Art, stehe ich Ihnen gerne unter meiner E-Mail Adresse [madelaine.hestermann@mni.thm.de](mailto:madelaine.hestermann@mni.thm.de) zur Verfügung.