



FACHHOCHSCHULE KIEL
University of Applied Sciences

EVALUIERUNG VON ELM ALS FRONTEND FÜR WEBAPPLIKATIONEN

Fachhochschule Kiel

Bachelor-Thesis

zur Erlangung des akademischen Grades
Bachelor of Science

vorgelegt von

Philipp Meißner

Matrikelnummer: 922432

Erstprüfer:	Prof. Dr. Robert Manzke
Zweitprüfer:	Prof. Dr. rer. nat. Carsten Meyer
Abgabetermin:	07.07.2016

Abstract

Motiviert durch die Versprechungen, die durch die funktionale Programmiersprache Elm gemacht werden [3, Vgl.], ist das Ziel der vorliegenden Bachelor-Thesis, die Programmiersprache, welche in natives JavaScript kompiliert wird, für die Verwendung als Frontend für Webapplikationen anhand einer empirischen Untersuchung zu evaluieren.

Im Zuge dessen wurde eine Single Page Application in nativen Elm-Code überführt. Dabei wurde versucht weitestgehend die Cascading Style Sheets und JavaScript-Dateien beizubehalten. Diese praktische Ausarbeitung wurde anhand unterschiedlichster Bewertungskriterien, wie beispielsweise die Zuverlässigkeit der Applikation, die Performanz einer Elm-Applikation und Portabilität zwischen unterschiedlichen Betriebssystemen und Browsern evaluiert. Ferner wurde die Dateigröße einer minimalen Elm-Applikation ausgewertet und anderen Frameworks gegenübergestellt. Darüber hinaus wurde der erzeugte Quellcode auf Wartbarkeit, Lesbarkeit, und Wiederverwendbarkeit untersucht.

Die praktische Ausarbeitung und Evaluation ergab, dass die Programmiersprache Elm die aufgestellten Kriterien in fast allen Punkten erfüllt. Lediglich die Interoperabilität von bestehenden JavaScript (JS)-Skripten in eine Elm-Applikation, sowie die Dateigröße einer minimalen Elm-Applikation erwiesen sich als nicht vollständig erfüllt. Die Interoperabilität war dahingehend nicht vollständig funktional, als dass es notwendig war Änderungen an den bestehenden JS-Skripten anzufertigen, sowie die nicht praktikable Möglichkeit Cascading Style Sheet (CSS)-Dateien nativ in Elm einzubinden. Der Elm-Compiler lieferte keinen Weg die Dateigröße der kompilierten Applikation zu minimieren. Stattdessen musste auf externe Werkzeuge zurückgegriffen werden. Abgesehen davon erwies sich eine Elm-Applikation als sehr performant, effizient und zuverlässig. Zusätzlich gibt es bereits eine Vielzahl an unterstützenden Werkzeugen zur Entwicklung mit Elm.

Die Ergebnisse erlauben die Schlussfolgerung, dass die Entwicklung mit Elm sehr empfehlenswert für Frontend-Entwickler ist. Das Typensystem in Elm erlaubt dem Elm-Compiler eine Vielzahl an Fehlern vorab zu finden. Ferner ist es für einen Entwickler nicht weiter notwendig die erzeugte Elm-Applikation auf allen Browsern separat testen zu müssen, da der Elm-Compiler die Applikation in natives JS kompiliert, welches kompatibel mit dem ECMAScript-Standard ist. Daraus entsteht eine enorme Zeitersparnis, die gepaart mit einer effizienten Applikation und einem zuverlässigen Compiler einen deutlich positiven Eindruck von Elm hinterlässt.

Inhaltsverzeichnis

Abstract	i
Abkürzungsverzeichnis	iv
Abbildungsverzeichnis	v
1 Einleitung	1
2 Theoretische Grundlagen	2
2.1 Funktionale Programmiersprachen	2
2.2 Grundlagen der Programmiersprache Elm	3
2.2.1 Geschichte	3
2.2.2 Konzept	4
2.2.3 Umsetzung	5
2.2.4 Einführung in die Elm-Architektur	8
3 Evaluierung der Programmiersprache Elm	23
3.1 Bewertungsmuster	23
3.2 Allgemeine Bewertungskriterien	24
3.2.1 Wartbarkeit und Lesbarkeit	24
3.2.2 Zuverlässigkeit	25
3.2.3 Portabilität	25
3.2.4 Effizienz	25
3.2.5 Wiederverwendbarkeit	26
3.3 Web-spezifische Bewertungskriterien	26
3.3.1 Browser Kompatibilität	26
3.3.2 Interoperabilität	27
3.3.3 Asynchrone Verarbeitung	27
3.3.4 Dateigröße	28
3.4 Empirische Analyse	29
3.4.1 Programmablauf	29
3.4.2 Entwicklungsumgebung	29
3.4.3 Grundaufbau	30
3.4.4 Überführung des Views	34
3.4.5 Beheben von JavaScript-Fehlern	35
3.4.6 Auslagern des Views	36
3.4.7 Hinzufügen von Klick-Events	37
3.4.8 Modularisierung der Applikation	40
3.4.9 Asynchrones Laden von Daten	42

3.4.10	Beobachtungen	45
3.4.11	Auswertung	47
4	Fazit	59
	Literaturverzeichnis	62
	Eidesstattliche Erklärung	65

Abkürzungsverzeichnis

API	Application Programming Interface
CSS	Cascading Style Sheet
DOM	Document Object Model
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
ICNDb	The Internet Chuck Norris Database
ID	Identifier
IDE	Integrated Development Environment
JS	JavaScript
JSON	JavaScript Object Notation
MVU	Model-View-Update
NPM	Node Package Manager
REPL	Read-Evaluate-Print-Loop
SPA	Single Page Application
URL	Uniform Resource Locator

Abbildungsverzeichnis

2.1	Eine iterative Funktion	2
2.2	Eine rekursive Funktion	3
2.3	Ein simpler Zähler auf einer Webseite	4
2.4	Das Model-View-Update (MVU)-Konzept von Elm [22]	5
2.5	Beispiel der dynamischen Typisierung	7
2.6	Das Tool elm-repl in der Kommandozeile	9
2.7	Ein Union-Type in Elm	10
2.8	Ein erweiterter Union-Type in Elm	11
2.9	Ein Record in Elm	12
2.10	Ein Tupel in Elm	13
2.11	Eine Variablendeklaration in Elm	13
2.12	Eine Konstrollstruktur in Elm	14
2.13	Konstrollstruktur in C++	14
2.14	Eine Funktion in Elm	15
2.15	Anwendung einer anonymen Funktion in Elm	15
2.16	Definition einer Funktion ohne Typen Alias	16
2.17	Definition einer Funktion mit Typen Alias	17
2.18	Mögliche Deklarationen eines Elm-Moduls	17
2.19	Mögliche Formen der Importierung eines Elm-Moduls	17
2.20	Die Online-Integrated Development Environment (IDE) von Elm	18
2.21	Installation des Node Package Manager (NPM)	19
2.22	Installation von Elm	19
2.23	Überprüfung der erfolgreichen Installation von NodeJs und dem NPM	20
2.24	Installation eines externen Paketes über den Elm-Paketmanager	21
2.25	Der gestartete Elm-Webserver	21
3.1	Die fünf meistgenutzten Browser im Mai 2016 [19]	27
3.2	Zeigt die Kommunikation zwischen Client und Webserver	29
3.3	Grundaufbau der <i>index.html</i> , um die Elm-Applikation zu inji- zieren	31
3.4	Grundgerüst der Elm-Applikation	32
3.5	Eine beispielhafte Initialisierung der Elm-Applikation mit ini- tialen, dynamischen Werten	33
3.6	Elm-Compilerfehler bei fehlenden Funktionen im Namensraum .	35
3.7	JavaScript-Fehler innerhalb der Developer-Tools	35
3.8	Notwendiger Code, um JavaScript-Fehler zu beheben.	36
3.9	Deklaration einer Sektion des Views in Elm	36
3.10	Ausgelagerter View in eine eigene Funktion	37

3.11	Aufruf der einzelnen Sektionen im View	37
3.12	Funktion zum scrollen bei einem Klick	38
3.13	Definition eines neuen Union Types, sowie eines Ports für die Kommunikation zwischen Elm und JS	38
3.14	Erweiterung des Views um einen <i>onWithOptions</i> Ereignisbe- handler in Elm	39
3.15	Entgegennahme der Daten aus der Elm-Applikation und an- schließendes scrollen	40
3.16	Einbindung und Aufruf der ausgelagerten View-Funktionen . . .	41
3.17	Einbindung und Aufruf der ausgelagerten View-Funktionen . . .	41
3.18	OnClick-Ereignisbehandlung in Elm	43
3.19	<i>Msg</i> -Deklaration mit asynchronem Request	43
3.20	Asynchroner Hypertext Transfer Protocol (HTTP)-Request in Elm	44
3.21	Dekodierung eines Json-Objektes in Elm	44
3.22	<i>Update</i> -Fälle für den asynchronen Request	44
3.23	Ausgehender HTTP-Request und eingehende Antwort	45
3.24	Ein <i>div</i> -Element in HyperText Markup Language (HTML) und Elm	46
3.25	Auswertung der Versuchskriterien	48
3.26	Automatisierter Signaturvorschlag durch den <i>elm – compiler</i> . .	48
3.27	Fehlermeldung eines nicht vorhandenen importierten Moduls . .	49
3.28	Fehlermeldung eines nicht vorhandenen Attributes	49
3.29	Fehlermeldung einer falschen Typzuweisung	50
3.30	Eine redundante Kontrollstruktur in Elm	50
3.31	Fehlermeldung einer fehlenden Klammer	51
3.32	Fehlermeldung einer mehrfach definierten Funktion	52
3.33	Fehlerfreier Kompiliervorgang durch den Elm-Compiler	52
3.34	Zeitliche Auswertung des Kompiliervorganges in Elm	53
3.35	Benchmark der TodoMVC in unterschiedlichen Programmier- sprachen	54
3.36	Dateigrößen der Frameworks Angular2, Elm, Ember und React in Kilobyte	58

1 Einleitung

Die Entwicklung von Webapplikationen kostet Zeit und dementsprechend Geld. Doch mit der Veröffentlichung der Webseite ist es nicht getan. Oftmals werden Fehler in der Webapplikation erst zur Laufzeit bemerkt und müssen daraufhin behoben werden. Wieder entstehen Kosten, nicht nur durch den Zeitaufwand der erneuten Entwicklung, sondern auch durch mögliche Ausfälle der Webapplikation. Die neue funktionale Programmiersprache Elm verspricht Besserung, indem Laufzeitfehler der Vergangenheit angehören sollen [3, Vgl. f.]. Dies soll nicht zuletzt an dem eigens entwickelten Compiler liegen, der den geschriebenen Quellcode überprüft und eine immense Anzahl von Fehlern finden soll. Zusätzlich sollen die Fehlermeldungen sehr präzise und aussagekräftig sein, wodurch dem Entwickler bei der Programmierung zusätzlich Zeit erspart wird. Darüber hinaus gilt die dargestellte Applikation als sehr performant. Im Rahmen dieser wissenschaftlichen Arbeit sollen diese Aussagen überprüft werden. Zu diesem Zweck werden in Kapitel 2 „Theoretische Grundlagen“ notwendige Grundlagen geklärt. Unter anderem behandelt das Kapitel das Konzept funktionaler Programmiersprachen, gefolgt von einer Einführung in die Grundlagen der Programmiersprache Elm. Hier werden Im Anschluss werden in Kapitel 3 „Evaluierung der Programmiersprache Elm“ Bewertungskriterien vorgestellt und erläutert, anhand derer die vorherigen Aussagen überprüft werden. Dazu gehören unter anderem die Evaluation der Performanz, die Klärung der Frage wie hilfreich der Elm-Compiler tatsächlich ist und ob mit der Verwendung von Elm Kompromisse bei der Kompatibilität mit unterschiedlichen Browsern gemacht werden müssen. Dem folgt die Dokumentation und Erläuterung des praktischen Teils dieser wissenschaftlichen Arbeit, in der zunächst einmal der geplante Programmablauf, sowie die genutzte Entwicklungsumgebung vorgestellt werden. und mündet in einer Auswertung anhand der aufgestellten Bewertungskriterien. Auf Grundlage der Auswertung werden im letzten Kapitel 4 „Fazit“ die Ergebnisse gegenübergestellt. Das Ziel dieser wissenschaftlichen Arbeit soll es sein, die Daseinsberechtigung der Programmiersprache Elm im Bereich der Frontend¹-Webentwicklung zu prüfen und eine Aussage darüber zu treffen, ob Elm die herkömmliche Entwicklung von Webapplikationen verbessert und tatsächlich eine valide Alternative darstellt.

¹Bezeichnet weitestgehend den Teil einer Webapplikation, der dem Nutzer dargestellt wird.

2 Theoretische Grundlagen

2.1 Funktionale Programmiersprachen

Jede Programmiersprache lässt sich in ein Programmierparadigma, einteilen. Prof. Dr. Uwe Kastens beschreibt Programmierparadigmen als „[...] unterschiedliche Prinzipien, mit denen die Ausführung von Programmen beschrieben wird“[17]. Ferner bezeichnet er die imperative, objektorientierte, funktionale und logische Programmierung als am Wichtigsten [17, Vgl.]. Jedes Programmierparadigma birgt unterschiedliche Merkmale der Programmierung wie beispielsweise rekursive Funktionen, Funktionen höherer Ordnung oder Polymorphie. Es ist nicht ausgeschlossen, dass eine Programmiersprache mehreren Prinzipien folgt und unterschiedliche Merkmale vereint. Funktionale Programmiersprachen sind stark an die Mathematik angelehnt. Programme werden hier eher als mathematische Funktionen formuliert, die eingehende Argumente immer gleichermaßen zu einem Ergebnis auswerten. Bei funktionalen Program-

```
1 iterative_function(n) {  
2   sum = 0;  
3   for(i = 0; i <= n; i++) {  
4     sum += i;  
5   }  
6 }  
7 console.log(iterative_function(10)); // => 55
```

Abbildung 2.1: Eine iterative Funktion

miersprachen ist es üblich, dass eine Variable nach ihrer Initialisierung ihren zugewiesenen Wert für die gesamte Laufzeit des Programmes beibehält. Man spricht hier von der Unveränderlichkeit einer Variable. Dadurch kann eine Variable als fester Ausdruck angesehen werden, der immer denselben Wert liefert. Typischerweise gibt es in funktionalen Programmiersprachen keine Schleifen, da dies bereits eine Verletzung der Unveränderlichkeit von Variablen bedeuten würde, wie klar erkennbar bei der Iteration in Abbildung 2.1 ist. Hier wird bei jedem Durchlauf der Schleife die Variable i inkrementiert und mit dem neuen Wert versehen. Es ist allerdings auch möglich, eine Schleife in einer funktionalen Programmiersprache zu verwirklichen. In Abbildung 2.2 ist die zuvor gezeigte iterative Funktion als rekursive Funktion implementiert worden.

„Eine Funktion heißt rekursiv, wenn sie sich selbst direkt aufruft oder wenn ein Aufruf einer anderen von ihr aufgerufenen Funktion dazu führt, dass sie ihrerseits wieder aufgerufen wird.“[14]

```

1 recursive_function(n) {
2   if (n < 1) return 0
3   else return n + recursive_function(n-1);
4 }
5 console.log(recursive_function(10)); // => 55

```

Abbildung 2.2: Eine rekursive Funktion

In beiden Fällen ist das Ergebnis gleich, jedoch besitzt die rekursive Implementierung keinerlei Seiteneffekte¹. Es wird an keiner Stelle der Wert einer Variable verändert, es werden lediglich Aufrufe mit neuen Werten durchgeführt. Die iterative Implementierung verursacht hingegen zwei Seiteneffekte. Es wird sowohl die Variable *sum*, sowie *i* bei jedem Schleifendurchlauf überschrieben. Bei der Rekursion hingegen wird die Funktion *recursive_function* mit einem neuen Wert aufgerufen, ohne jemals die ursprüngliche Variable verändert zu haben. Üblicherweise werden Funktionen in funktionalen Programmiersprachen als Funktionen höherer Ordnung angesehen. Das heißt, dass eine Funktion eine andere Funktion als Argument entgegennehmen kann, oder eine Funktion als Rückgabewert hat. Diese Art von Funktionen ist bekannt unter dem Begriff *Lambda Funktion* oder *anonyme Funktion*. Eine solche Funktion hat entsprechend keinen Namen, sondern kann nur einmalig aufgerufen werden. Im folgenden Beispiel sehen wir den Aufruf einer anonymen Funktion in Elm:

$$(\backslash x \rightarrow x * 2) [2, 3, 4] \Longrightarrow [4, 6, 8] \quad (2.1)$$

Grundsätzlich wird mit einer solchen Lambda-Funktion ein mathematisches Abbildungsgesetz formuliert. Das mathematische Gegenstück zu Funktion 2.1 ist $x \rightarrow x * 2$ und beschreibt eine Funktion, die einen Eingabeparameter x auf $x * 2$ abbildet. Hier wird klar erkenntlich, wie nahe die funktionale Schreibweise dem mathematischen Abbildungsgesetz ist. Auch Elm gehört dem funktionalen Programmierparadigma an und vereinheitlicht dessen Konzepte.

2.2 Grundlagen der Programmiersprache Elm

2.2.1 Geschichte

Elm wurde im Rahmen der Bachelorarbeit „Elm: Concurrent FRP for Functional GUIs“ von Evan Czaplicki entwickelt [2, Vgl.]. Auslöser für die Entwicklung von Elm war für Czaplicki die Schwierigkeit, ein Bild auf einer Webseite sowohl horizontal, als auch vertikal zu zentrieren.[8, Vgl.] Es gab keine für ihn annehmbare, leichte Lösung für dieses Unterfangen, ohne damit weitere Probleme zu schaffen. Unter der Leitfrage „Wie würde Web-Programmierung aussehen, wenn wir neu starten könnten?“², machte Czaplicki sich Gedanken,

¹Ein Seiteneffekt beschreibt die mehrmalige Zuweisung einer Variable mit einem Wert.

²[5, Vgl. Eigene Übersetzung]

welche Veränderungen an den aktuellen, etablierten Programmiersprachen für die Webentwicklung wünschenswert wären und entwickelte den ersten Prototypen von Elm.

2.2.2 Konzept

Elm verfolgt eine ganz eigene Implementierung des Model-View-Controller Paradigmas. Hier wird es Model-View-Update (MVU) genannt. Anhand des Beispiels in Abbildung 2.3 lässt sich das Muster in drei grundlegende Stationen unterteilen und erklären. Die Abbildung 2.3 zeigt einen simplen Zähler, der

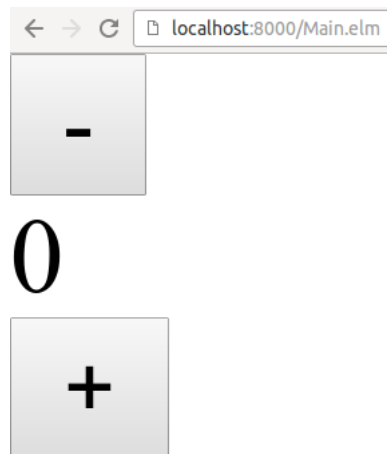


Abbildung 2.3: Ein simpler Zähler auf einer Webseite

über zwei Knöpfe inkrementiert und dekrementiert werden kann. Der aktuelle Stand des Zählers wird zwischen den Knöpfen angezeigt und kann sowohl negative, als auch positive Werte annehmen. Der angezeigte Zählerwert ist das sogenannte *Model* und zeigt den aktuellen Status der Applikation an. Interagiert ein Nutzer nun mit einem der beiden Knöpfe um den Zähler zu erhöhen oder zu reduzieren, wird diese Aktion an die sogenannte *Update*-Funktion weitergegeben. Zusätzlich zur auszuführenden *Aktion*, bekommt diese Funktion auch noch das aktuelle *Model* übergeben. Die *Update*-Funktion nimmt sämtliche Einwirkungen durch den Nutzer von außen entgegen und wendet diese Aktionen auf das aktuelle *Model* an. Dabei wird jedoch nicht das *Model* direkt verändert, sondern ein neues *Model* mit den geänderten Werten zurückgegeben, da sonst ein Seiteneffekt die Folge wäre. Damit dieser Vorgang zügig vonstattengeht, nutzt Elm persistente Datenstrukturen, womit nur die tatsächlich geänderten Attribute eines Models im neuen *Model* gesetzt werden, die unveränderten Attribute hingegen werden übernommen. Das Ergebnis der *Update*-Funktion wird weitergereicht an die *View*-Funktion. Sie beschreibt das Aussehen der Webseite und kümmert sich um die Darstellung des *Models*. Der zuvor beschriebene Datenfluss in Elm wird noch einmal in Abbildung 2.4 visualisiert.

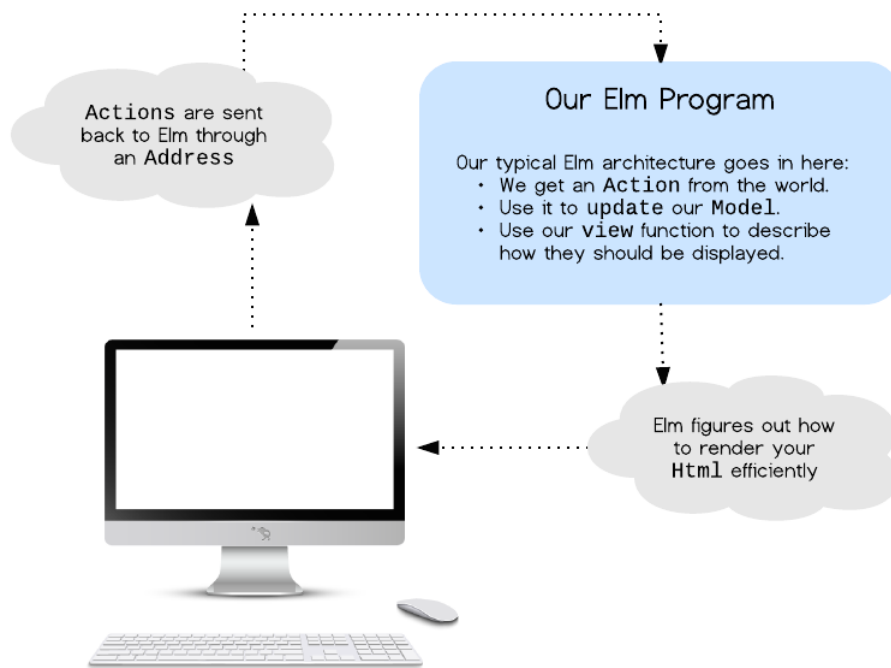


Abbildung 2.4: Das MVU-Konzept von Elm [22]

2.2.3 Umsetzung

Der vom Programmierer verfasste Programmcode wird vor der endgültigen Nutzung zu JS, HTML und CSS kompiliert und in die Webseiten integriert. Entsprechend fungiert in Elm verfasster Code im Endeffekt wie natives JavaScript (JS), nutzt allerdings noch einige weitere vertiefende Konzepte, um viele Problematiken von JS zu umgehen und auszumerzen. Unter anderem verspricht Elm, dass generierter Code keinerlei Laufzeitfehler erzeugt.³ Sämtliche Fehlerquellen werden vom Compiler zuvor erkannt, abgefangen und an den Programmierer weitergeleitet, um sie zu beheben. Damit dies funktioniert, implementiert Elm unter anderem Konzepte des funktionalen Programmierparadigmas. Diese und die weiteren Grundkonzepte werden im folgenden erläutert.

Keine Seiteneffekte

In Elm sind Seiteneffekte ausgeschlossen. Sämtliche Variablen sind unveränderlich und können nur einmalig mit einem Wert initiiert werden. Danach bleibt diese Variable bis zum Ende der Laufzeit unverändert. Wie bereits beschrieben gibt es in Elm das *Model*, welches die Informationen über den Status der Applikation enthält. Beschreibt das *Model* beispielsweise den aktuellen Stand eines Zählers und wird dieser erhöht, muss auch das *Model*, um aktuell zu bleiben, verändert werden. Hier käme es zu einem Seiteneffekt. Realisiert wird diese Veränderung dadurch, dass ein neues *Model* mit den gleichbleibenden Daten, sowie dem zu ändernden, aktualisierten Wert erstellt wird. Da ein

³[3, Vgl.]

neues *Model* erstellt wurde, gibt es nun keinen Seiteneffekt mehr. Das vorherige *Model* wird schlichtweg verworfen und mit einem neuen *Model* ersetzt. Das Konzept der unveränderbaren Werte wurde aus der Mathematik übernommen. Um Funktionen und ihre Korrektheit garantieren zu können, wird dort dasselbe Prinzip der unveränderlichen Variablen angewandt. Betrachtet man beispielsweise den Ausdruck 2.2, so fällt auf, dass die Schreibweise lediglich in den meisten imperativen Programmiersprachen sinnvoll ist, allerdings einen Seiteneffekt darstellt.

$$x = x + 1 \tag{2.2}$$

In einer imperativen Programmiersprache wird durch den Ausdruck 2.2 der aktuelle Wert in x ausgelesen, um 1 inkrementiert und das Ergebnis in die Variable x geschrieben. Mathematisch betrachtet ist diese Aussage jedoch schlichtweg falsch, denn es existiert kein x , welches diese Aussage wahr werden lässt, wie durch Abbildung 2.3 deutlich wird.

$$x = x + 1 \leftrightarrow 0 = 1 \tag{2.3}$$

Die meisten imperativen Programmiersprachen nutzen das rechtsassoziative Gleichheitszeichen als Zuweisung, während es in der Mathematik als Vergleichsoperator angesehen wird. Die eigentliche Bedeutung des Ausdrucks 2.2 ist mathematisch ausgedrückt:

$$x_1 := x_0 + 1 \tag{2.4}$$

Es ist klar erkennbar, dass x_1 und x_0 unterschiedliche Variablen sind, wodurch die Aussage nun als wahr eingestuft werden kann. Das beschriebene Konzept wird laut Dr. Steffen Jost „referentielle Transparenz“ genannt und beschreibt die Kontinuität des Wertes einer Variable. Ferner gibt er an, dass Funktionen die stets gleiche Ergebnisse bei gleichbleibenden Eingabeparametern liefern, ebenfalls eine Eigenschaft der referentiellen Transparenz sind [9, Vgl.]. Beispiele für solche Funktionen sind $\sin(x)$ oder $\text{add}(x, y)$. Sie berechnen immer dieselben Werte, völlig unabhängig davon, wie oft oder zu welchem Zeitpunkt sie ausgeführt werden. Ein beliebtes Gegenspiel ist die *random()* Funktion, die einen zufälligen Wert zurückliefert und somit die Eigenschaft der referentiellen Transparenz verletzen würde. Doch auch diese Funktion kann als reine Funktion⁴ implementiert werden, wenn sie einen Wert abhängig von einem Übergabeparameter berechnet wie beispielsweise *random(seed)*.

Elm-Compiler

Laufzeitfehler sollen mit Elm in Vergessenheit geraten. Dafür soll der integrierte Compiler sorgen. Die Fehlermeldungen des Compilers sind sehr hilfreich und genau [3, Vgl.]. Bei der herkömmlichen Entwicklung eines Frontends mit JS trifft man häufig auf den Wert *undefined*. Dieser Wert beschreibt, dass die dazugehörige Variable noch nicht initialisiert wurde und somit keine nutzbaren

⁴Eine reine Funktion besitzt alle Eigenschaften der referentiellen Transparenz.

Daten enthält. Trifft man nun auf diesen Wert und versucht eine andere Funktion darauf auszuführen, so kann das geschriebene Programm entsprechend abstürzen. Der Elm Compiler überprüft den Programmcode nach exakt diesen Situation beziehungsweise analysiert, ob Variablen und Funktionen vorab initialisiert wurden, welche Parameter die einzelnen Funktionen erwarten und ob die Rückgabeparameter dem Typen entsprechen, der von anderen Funktionen erwartet wird. Der Elm-Compiler kompiliert den Elm-Code in ausführbaren JS-Code.

Statische Typisierung

Anders als bei nativem JS gibt es in Elm keine dynamische Typisierung. Das bedeutet, dass sowohl die Typen einer Variable, als auch die Rückgabewerte von Funktionen bereits bei der Kompilierung bekannt sein müssen. Natives JS erlaubt es, dass die Typen von Variablen erst zur Laufzeit überprüft werden und sich zusätzlich in dieser Zeit ändern können. So ist der Quellcode in Abbildung 2.5 konform mit JS. Der Typ der Variable *i* wurde in der Abbil-

```
1 var i = 1;  
2 i = "Test";
```

Abbildung 2.5: Beispiel der dynamischen Typisierung

dung 2.5 während der Laufzeit von *number* zu *string* geändert. Da Elm stark typisiert ist, gibt es keine Möglichkeit, dass eine Funktion verschiedene Datentypen zurück gibt oder eine Variable mehrere Typen während der Laufzeit annimmt.

Performanz

Obwohl Daten nicht verändert, sondern durchgehend ein neuer, aktualisierter Datensatz erstellt wird, soll die Performanz nicht darunter leiden. Möglich soll das durch die Verwendung eines virtuellen Document Object Model (DOM) werden. Laut Evan Czaplicki wird dabei das echte DOM bei jedem „Frame“⁵ in eine abstrakte Version kopiert. Auf diese abstrakte Version werden die Änderungen angewandt. Zunächst klingt diese Vorgehensweise sehr langsam und aufwändig, um jedoch die Geschwindigkeit zu gewährleisten wird das aktuelle abstrakte DOM mit dem neuen, veränderten DOM verglichen und nach Unterschieden gesucht. Jede Unterschiedlichkeit wird daraufhin zu einer Liste hinzugefügt, in der sämtliche Änderungen festgehalten werden. Anschließend wird diese Liste an den Browser zurückgegeben, so dass alle Änderungen für den Nutzer sichtbar gemacht werden können. Daraus resultiert, dass nur noch die tatsächlich neuen oder veränderten Elemente im DOM des Nutzers aktualisiert werden müssen. Dieser zu verändernde Teil stellt nur einen Bruchteil des

⁵Ein Frame stellt ein Bild dar, dass der Nutzer auf der Webseite sieht. Anhand der „Frame-rate“ (Bilder pro Sekunde) kann eine Aussage über die Geschwindigkeit der Darstellung einer Webseite gemacht werden.

kompletten DOM dar. Man kann dadurch von einer immensen Effizienzsteigerung ausgehen [5, Vgl.].

Interoperabilität

Es wäre sehr aufwendig die gängigsten JavaScript-Bibliotheken und -Frameworks wie jQuery oder AngularJS komplett zu verwerfen und mit Elm zu realisieren. Glücklicherweise bietet Elm eine ausgereifte Interoperabilität, wodurch alle Garantien der funktionalen Programmierung übernommen werden können, selbst wenn die externen Bibliotheken diese nicht gewährleisten. Die Elm-Applikation kommuniziert über sogenannte *Ports* mit den JS-Bibliotheken.

Praktische Anwendungsgebiete

Elm ist noch recht neu und befindet sich im ständigen Wandel. Für viele Entwickler ist Elm entsprechend noch keine wirkliche Alternative zu ihren gegenwärtig genutzten Frameworks, obgleich die Entwicklung mit den gängigen Werkzeugen oftmals steinig ist. Nur wenige Unternehmen nutzen derzeit Elm in ihrem Produktionsumfeld. Die derzeit größten Nutzer sind NoRedInk, Prezi und CircuitHub [3, Vgl.]. Alle Betriebe überführen Stück für Stück bereits bestehende Teile ihres Frontends zu Elm. Der NoRedInk-Mitarbeiter Richard Feldman gibt an, dass der überführte Elm-Code über etwa acht Monaten hinweg keinerlei Laufzeitfehler erzeugt hat, anders als die vorherige Implementierung [10, Vgl.]. Dennoch wird es wohl noch eine Weile dauern, bis sich mehr Firmen der Vorstellung hingeben ihr lauffähiges System in die vielversprechende Programmiersprache Elm zu portieren, nicht zuletzt, weil sie sich noch in einem sehr frühen Stadium befindet und somit noch nicht völlig ausgereift ist. Es existiert jedoch bereits eine Vielzahl an Projekten die mit Elm verwirklicht wurden. Dabei sind viele dieser Projekte kleinere Retro-Spiele, die über den Browser gespielt werden können [24, vgl.]. Dieser Einblick zeigt bereits, dass Elm in vielerlei Hinsicht Besserung für die Entwicklung von Webapplikationen verspricht. Doch wie praktikabel sind diese Versprechungen? Ist die Programmiersprache effizient und intuitiv, oder durch ihr noch frühes Entwicklungsstadium unausgereift? Auf der offiziellen Webseite von Elm wird angegeben, dass Elm die beste funktionale Programmiersprache des Browsers sei.⁶ Diese Aussage wird im Rahmen dieser Arbeit anhand der Entwicklung, respektive der Überführung einer Webseite in natives Elm überprüft. Die Evaluierung erfolgt schließlich anhand einiger Bewertungskriterien.

2.2.4 Einführung in die Elm-Architektur

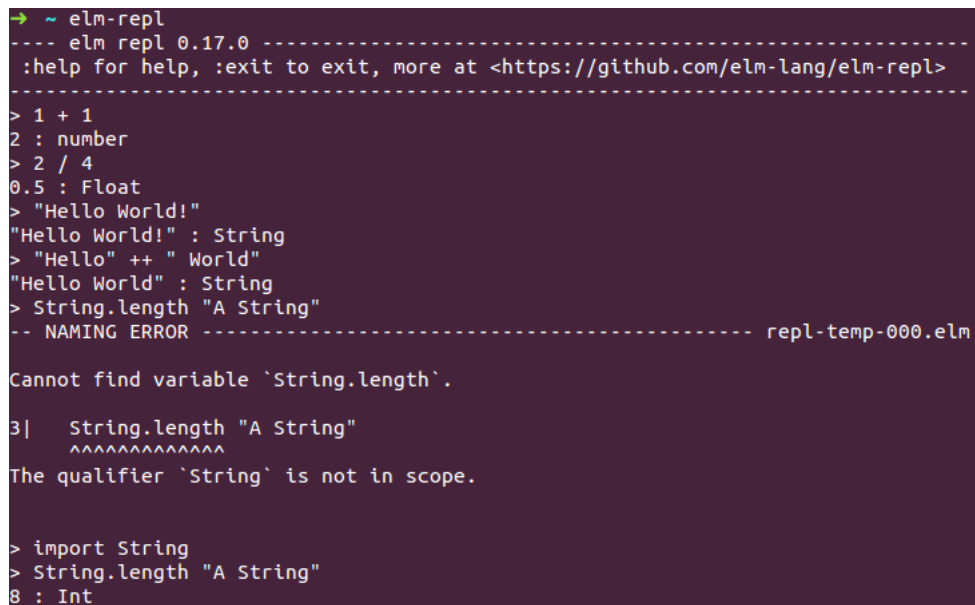
Vor der Evaluierung der Programmiersprache Elm ist es jedoch notwendig, die Elm-Architektur zu erläutern und einige Grundaspekte zu erklären. Dieses Kapitel soll einen Einstieg in die neue Programmiersprache Elm ermöglichen und stellt nicht nur die Grundkonzepte der Programmiersprache, sondern auch

⁶ „the best of functional programming in your browser“ [3, Vgl. Eigene Übersetzung]

einige mitgelieferte Werkzeuge vor. Im Rahmen dieses Kapitels wurden teilweise Aspekte der offiziellen Elm-Dokumentation⁷ entnommen, in Hinblick auf den praktischen Teil dieser Arbeit angeordnet und mit passenden Beispielen versehen.

Elm-REPL

Ein nützliches Tool um mit Werten in Elm zu interagieren und kleinere Algorithmen zu testen, ist die *elm – repl*. Read-Evaluate-Print-Loop (REPL) bezeichnet dabei die Iteration, welche ein Programmcode durchlebt. Zunächst wird der Quellcode gelesen (read), danach ausgewertet (evaluate) und das Ergebnis ausgegeben (print). Dieser Vorgang wiederholt sich (loop), bis die Entwicklung fertig ist. *Elm – repl* wird über die Kommandozeile aufgerufen und gestartet. In Abbildung 2.6 ist das Tool abgebildet und zeigt beispielhafte Eingaben und Evaluierungen. Wie aus der Abbildung 2.6 ersichtlich wird, werden



```
→ ~ elm-repl
---- elm repl 0.17.0 -----
:help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
-----
> 1 + 1
2 : number
> 2 / 4
0.5 : Float
> "Hello World!"
"Hello World!" : String
> "Hello" ++ " World"
"Hello World" : String
> String.length "A String"
-- NAMING ERROR ----- repl-temp-000.elm

Cannot find variable `String.length`.

3|   String.length "A String"
   ^^^^^^^^^^^^^^^^^
The qualifier `String` is not in scope.

> import String
> String.length "A String"
8 : Int
```

Abbildung 2.6: Das Tool elm-repl in der Kommandozeile

auch Fehler ausgegeben. In diesem Beispiel wurde versucht auf die Bibliothek *String*, genauer die Funktion *length* zuzugreifen. Diese Bibliothek wurde allerdings noch nicht eingebunden, wodurch es zu dem Fehler kam. Nachdem die Bibliothek über das Kommando *import String* importiert wurde, war der Fehler behoben. Die *elm – repl* erlaubt es, alle Bibliotheken die das Paket *core* mitliefert, zu importieren und Algorithmen zu evaluieren.

Basisdatentypen

Elm hat nur eine geringe Anzahl an Basisdatentypen, mithilfe derer sämtliche weiterführende Konstrukte abgeleitet werden können:

⁷[3, Vgl.]

- 42 : number
- True : Bool
- 'a' : Char
- [1, 2, 3] : List

Durch die Kombination dieser Basisdatentypen können weitere komplexere Datentypen wie *Strings*, *Integer* oder *Floats*, wie es in Abbildung 2.6 zu sehen ist, erzeugt werden.

Basisfunktionen

Elm kommt mit einer Vielzahl an grundlegenden Funktionen, um notwendige arithmetische Operationen zu ermöglichen. Dazu gehören Funktionen zur Addition (+), Subtraktion (−), Multiplikation (*) und Division (/). Um Vergleiche zu vollziehen gibt es die Funktionen zur Prüfung der Gleichheit (==) und Ungleichheit (/ =). Wie in Abbildung 2.6 zu sehen ist, können zwei *Strings* zu einem verbunden werden, indem die Funktion ++ angewendet wird.

Union Types

In vielen Applikationen können Datentypen unterschiedliche Zustände annehmen. So kann beispielsweise ein Tag die Zustände *Montag* bis *Sonntag* annehmen. Die in anderen Programmiersprachen als *algebraische Datentypen*[15, Vgl.] bekannte Aufzählung von Zuständen wird in Elm *UnionType* genannt und beschreibt auch hier eine Aufzählung von endlich vielen Zuständen eines Datentypen. Die Einführung eines solchen *UnionTypes* erlaubt es dem Compiler den Quellcode auf fehlende Zustände zu überprüfen. Ist es zum Beispiel das Ziel, eine Meldung abhängig vom aktuellen Wochentag auszugeben, müssen alle Zustände und ihre Folge dafür deklariert werden. Fehlt eine Deklaration, kann der *Elm – Compiler* dies anhand des *UnionTypes* erkennen. Betrachtet

```

1  type Day = Monday | Tuesday | .. | Sunday
2  storeStatus : Day -> String
3  storeStatus day =
4    case day of
5      Monday ->
6        "Opened "
7      Sunday ->
8        "Closed "
```

Abbildung 2.7: Ein Union-Type in Elm

man die Abbildung 2.7 ist klar erkennbar, dass die Funktion *storeStatus* nicht alle Fälle, die der Typ *Day* annehmen kann, behandelt. Aufgrund der vorherigen Deklaration eines *UnionType* kann der *Elm – Compiler* einsehen,

welche möglichen Zustände der Typ *Day* annehmen kann. Dadurch wird ein Fehler geworfen, insofern die fehlenden Typen nicht ergänzt werden. Dieses Konzept der algebraischen Datentypen erscheint zunächst sehr ähnlich den *Enumerationen* in beispielsweise *C++* oder anderen Programmiersprachen. Tatsächlich gibt es diese Form der Datenabstraktion in einer Vielzahl von funktionalen Programmiersprachen, mit dem wohl bekanntesten Vertreter der funktionalen Programmiersprache *Haskell*. Der Vorteil eines *UnionTypes* in Elm gegenüber *Enumerationen* in *C++* oder Java ist, dass jeder Zustand der Aufzählung optionale Parameter übergeben bekommen kann. Damit bietet sich die Möglichkeit viel detailliertere Konstrukte zu erzeugen, ohne die Typensicherheit die sich durch die *UnionTypes* ergibt zu verlieren. Das Beispiel der Abbildung 2.7 könnte entsprechend erweitert werden, so dass die Tage *Montag* bis *Freitag* zusätzlich noch eine Öffnungszeit besitzen. Wie in Ab-

```

1  type Day = Monday Int Int | .. | Sunday
2  storeStatus : Day -> String
3  storeStatus day =
4      case day of
5          Monday start end->
6              "Opened from:"
7                  ++ start "until"
8                  ++ end
9          Sunday ->
10             "Closed "
```

Abbildung 2.8: Ein erweiterter Union-Type in Elm

Abbildung 2.8 zu sehen ist, wurden die einzelnen Tage in der Deklaration um die Parameter *Int* erweitert. In diesem Fall sollen die *Integer* vereinfacht die Start- und Endzeit der Öffnungszeit widerspiegeln. Auffällig ist, dass die Signatur der Funktion *storeStatus* im Vergleich zur vorherigen Abbildung 2.7 nicht verändert wurde. Der übergebene Datentyp ist noch immer ein *Day*, mit dem Zusatz, dass die Wochentage zwei weitere Parameter *start* und *end* besitzen. Auf diese Weise können komplexe Konstrukte definiert werden, während die Typensicherheit gewahrt bleibt.

Records

Eine weitere Datenstruktur in Elm sind die sogenannten *Records*. Ein *Record* ist vergleichbar mit einem *Objekt* in JS und verfolgt eine sehr ähnliche Syntax [7, Vgl.]. Es ist möglich, eigene Felder in einem *Record* zu definieren, mit allen Datentypen die bisher erzeugt wurden. Dazu gehören nicht nur die Basis-Datentypen, sondern auch alle daraus konstruierten. Auch können verschiedene Datentypen in einem *Record* kombiniert werden. In Zeile 1 der Abbildung 2.9 ist erkennbar, wie ein *Record* mit initialen Werten erstellt wird. Lediglich das Gleichheitszeichen für die Zuweisung unterscheidet sich hierbei

```

1 university = { typ = "Fachhochschule "
2               , gruendungsdatum = 1969
3               , ort = "Kiel" }
4 university.typ —> "Fachhochschule "
5 .ort university —> "Kiel"
6
7 {university | ort = "Flensburg" }

```

Abbildung 2.9: Ein Record in Elm

von herkömmlichen erstellen eines *Records* in JS. Um ein Feld in einem *Record* direkt anzusprechen, kann die übliche Schreibweise aus Zeile 4 gewählt werden. Zeile 5 hingegen führt zum gleichen Ergebnis, nutzt intern jedoch eine *anonyme Funktion*. Der Unterschied eines *Records* im Gegensatz zu einem *Objekt* in JS liegt darin, dass Felder nicht dynamisch hinzugefügt oder gelöscht werden können. Sobald ein Feld definiert wurde, ist es für die gesamte Laufzeit vorhanden. Die Werte eines Feldes können nur bedingt verändert werden. Zeile 7 der Abbildung 2.9 zeigt, wie das Feld *Ort* des *Records* mit einem neuen Wert versehen wird. Dabei ist zu beachten, dass aufgrund der Unveränderlichkeit einer Datenstruktur in Elm nicht der *Record* selbst verändert wird. Viel mehr wird ein neuer *Record* erstellt, der die Änderung, sowie gleichbleibenden Felder des alten *Record* annimmt. Ein weiterer Unterschied gegenüber JS ist, dass ein Feld nie den Wert *undefined* oder *null* annehmen kann, da eingeführte Felder stets initialisiert werden müssen.

Tupel

Sollte es notwendig sein, mehr als einen Wert als Rückgabewert zu haben, so bietet sich die Nutzung von *Tupeln* an. Ein *Tupel* ist eine weitere Datenstruktur in Elm und kann eine beliebig feste Anzahl an Werten beinhalten. Jeder Wert kann dabei einen unabhängigen Datentyp einnehmen. Die Abbildung 2.10 zeigt, wie ein *Tupel* von einer Funktion zurückgegeben wird. Der Rückgabewert des beispielhaften Aufrufes in Zeile 10 ist ein *Tupel*, bestehend aus einem *Bool* und einem *String*, wie es in der Signatur in Zeile 3 definiert wurde. Soll das *Tupel* noch weiter verwendet werden, liefert die Bibliothek *Basics* aus dem Paket *elm – lang/core* die Funktionen *fst* und *snd*, die entsprechend den ersten oder zweiten Wert eines *Tupels* zurückliefern. Auf diese Weise kann ein einzelner Wert des *Tupels* extrahiert werden. Aufgrund der fehlenden Wege *Tupel* mit mehr als zwei Werten auszuwerten, sollte nur bedingt auf *Tupel* zurückgegriffen und stattdessen ein *Record* verwendet werden.

Variablen

Die Lebenszeit eines *Records* ist die gesamte Laufzeit des Programmes. Eine *Variable* in Elm hingegen bleibt nur für die Dauer der Funktion in der sie definiert wurde erhalten und ist auch nur in diesem Namensraum gültig. Außerhalb

```

1  type Food = { name: String, sort: String }
2
3  isCookie : Food -> (Bool, String)
4  isCookie food =
5      if food.sort == "cookie" then
6          (True, food.name)
7      else
8          (False, "No cookie.")
9
10 isCookie { name = "Oreo", sort = "cookie" }
11 -> (True, "Oreo ")

```

Abbildung 2.10: Ein Tupel in Elm

der Funktion ist die Variable nicht mehr bekannt. In Elm wird eine Variable innerhalb eines *let..in*-Konstrukts erzeugt. *Let* bezeichnet dabei den Abschnitt, in welchem sämtliche Variablen erstellt werden und einen Wert zugewiesen bekommen. Die innerhalb dieses Blockes definierten Variablen können dabei aufeinander zugreifen, unabhängig von ihrer Reihenfolge. Mit Hilfe des *let..in*-Blockes kann die Logik einer Funktion noch weiter ausgelagert werden, ohne eine explizit neue Funktion zu definieren. Abbildung 2.11 zeigt beispielhaft die

```

1  meaningOfLife =
2      let
3          fortyTwo = oneHundred - 58
4          oneHundred = 100
5      in
6          fortyTwo
7
8  meaningOfLife -> 42

```

Abbildung 2.11: Eine Variablendeklaration in Elm

Deklaration einer Funktion *meaningOfLife*, die innerhalb des *let..in*-Blockes die Variablen *fortyTwo* und *oneHundred* deklariert und ihnen einen Wert zuweist. Die Inhalte eines *let..in*-Blockes müssen eingerückt werden, während hingegen eine übliche Funktion auch in einer Zeile verfasst werden kann. Des Weiteren ist erkennbar, dass die Variable *oneHundred* erst nach der Deklaration von *fortyTwo* deklariert wird, jedoch bereits vorher nutzbar ist. Der *elm - compiler* optimiert diese Programmstelle während des Kompilierungsvorganges entsprechend. Die beiden deklarierten Variablen sind lediglich in dem definierten *let..in*-Block verfügbar. Eine Verwendung außerhalb des Blockes hätte einen Compilerfehler zur Folge. Es besteht ferner die Möglichkeit ganze Funktionen innerhalb des Blockes zu definieren und zu nutzen.

Kontrollstrukturen

Elm bietet die Möglichkeit verschiedene Kontrollstrukturen anzuwenden. Dabei wird ein Record auf mögliche Werte hin überprüft und abhängig vom Wert ein anderer logischer Pfad gewählt. In Abbildung 2.12 ist eine herkömmliche

```
1 isItACookie food =  
2   if food == "cookie" then  
3     True  
4   else if food == "grapefruit" then  
5     False  
6   else  
7     False
```

Abbildung 2.12: Eine Konstrollstruktur in Elm

if-Abfrage zu sehen. In Elm sind die Schlüsselwörter *if*, *then* und *else* notwendig und unterteilen die Bedingung von der angewandten Folge, die nach dem Schlüsselwort *then* folgt. Insofern keine *if*-Bedingung zutrifft, wird der *else*-Fall angewandt. Elm verfolgt auch hier die statische Typisierung und fordert, dass jede mögliche Verzweigung denselben Typen zurückliefert. Eine Bedingung muss in Elm immer *True* oder *False*, sprich einen *Bool* liefern. In anderen Programmiersprachen wie beispielsweise *C++* evaluiert eine Bedingung zu *True*, wenn sie nicht 0 oder *false* ist. Das Beispiel der Abbildung 2.13 zeigt, dass 0 zu *false* auswertet. In Elm hingegen meldet der *elm – compiler* einen Fehler, sobald eine Bedingung zu einem anderen Typen als *Bool* auswertet. Abbildung 2.7 zeigt, wie *UnionTypes* überprüft werden können. Die Kontrollstruktur ähnelt dabei dem *switch-case*-Konstrukt von anderen Programmiersprachen wie JS oder C++.

```
1 string whatIsZero() {  
2   if (0) return "0 equals to TRUE";  
3   else return "0 equals to FALSE";  
4 }  
5 cout << whatIsZero() << endl;  
6 //=> 0 equals to FALSE
```

Abbildung 2.13: Konstrollstruktur in C++

Funktionen

Um in irgendeiner Art und Weise eine Interaktion mit den erstellten *Records* zu vollziehen, wird eine Funktion benötigt. Solch ein Programmkonstrukt wird mittels eines Funktionsnamen und einem Gleichheitszeichen realisiert. Die Abbildung 2.14 macht deutlich, wie eine simple Addition zweier Integer-Werte

```

1  add : Int -> Int -> Int
2  add a b =
3      a + b
4
5  add 3 4  —> 7

```

Abbildung 2.14: Eine Funktion in Elm

umgesetzt wird. Die Buchstaben *a* und *b* deuten an, dass die Funktion *add* zwei Parameter erwartet. Der gesamte Inhalt nach dem Gleichheitszeichen ist der sogenannte Rumpf einer Funktion und beschreibt den anzuwendenden Algorithmus. Der Funktionsaufruf ist in Zeile 5 sichtbar. Auffällig ist, dass eine Funktion nie explizit einen Wert zurück gibt, wie es in anderen Programmiersprachen wie JS oder C++ durch den Befehl *return* geschieht. Elm gibt implizit die letzte ausführbare Zeile als Rückgabewert zurück. Der beispielhafte Aufruf der Funktion *add* in Zeile 5 hätte dementsprechend 7 als Ergebnis. Des Weiteren ist erkennbar, dass Elm ohne die Nutzung von Kommata oder Klammern auskommt. Klammern werden erst notwendig, wenn mehrere Funktionen geschachtelt werden und eine Auswertung der Befehle von links nach rechts nicht ausreicht. Zusätzlich zu einer explizit benannten Funktion, gibt es die *anonyme* Funktion. Sie kommt ohne einen Funktionsnamen aus und wird an Funktionen höherer Ordnung weitergereicht. *List.filter* ist eine Funk-

```

1  List.filter : (a -> Bool) -> List a -> List a
2  List.filter (\str -> str == "a") ["a", "b", "c", "a"]
3  —> ["a", "a"]

```

Abbildung 2.15: Anwendung einer anonymen Funktion in Elm

tion höherer Ordnung und erwartet eine anonyme Funktion als Parameter, auf Basis derer eine Liste gefiltert wird. Aus der Abbildung 2.15 wird ersichtlich, dass die übergebene Liste, bestehend aus 4 Einträgen, auf die Präsenz des Buchstabens *a* hin überprüft wird. Dabei wird jeweils ein einzelner Wert an die anonyme Funktion weitergereicht, die in Form der Variable *str* repräsentiert und mit dem String *a* auf Gleichheit überprüft wird. Einträge die diesem Vergleich entsprechen, werden einer neuen Liste hinzugefügt. Wird das Ende der zu überprüfenden Liste erreicht, wird die Ergebnisliste als Rückgabewert ausgegeben.

Signaturen

Die jeweils erste Zeile der Abbildung 2.14 und 2.15 zeigt den Aufbau einer Signatur. Solch eine Signatur beschreibt die Funktion. Dabei sind Informationen über die Anzahl und der jeweilige Typ der Übergabeparameter, sowie der Typ des Rückgabewertes enthalten. Der letzte Typ einer Signatur ist dabei

immer der Rückgabewert, die vorherigen Typen stellen die Typen der Übergabeparameter in der übergebenen Reihenfolge dar. Die einzelnen Parameter sind durch einen Pfeil ($->$) voneinander getrennt. Wird eine Funktion anstelle eines einfachen Datentypen als Parameter übergeben, wird dies wie in Abbildung 2.15 durch das Einklammern der Typen angedeutet. Die Typen innerhalb der Klammer stellen wiederum die Typen der Übergabeparameter und Rückgabewerte der Funktion dar.

Anhand der Abbildung 2.14 lässt sich erkennen, dass die *add*-Funktion zwei Parameter vom Typ *Int* erwartet und letzten Endes ein Ergebnis des Typ *Int* liefert. Fehlt einer Funktion die Signatur, wird der *elm - compiler* eine Warnung ausgeben und zusätzlich eine passende, jedoch teilweise allgemeinere Signatur ausgeben. Würde die Funktion in Abbildung 2.14 keine Signatur enthalten, würde der *elm - compiler* die Signatur *add : number -> number -> number* vorschlagen. Da ein *Int* nur eine Spezifizierung des Basisdatentypen *number* darstellt, ist das nicht verwunderlich. Der *elm - compiler* nutzt implizit die eigens erarbeiteten Signaturen bei der Überprüfung des Quellcodes, sollte der Entwickler keine Signatur angegebenen haben.

Typen Alias

Je komplexer ein Record wird, desto länger und unübersichtlicher wird auch die dazugehörige *Signatur*. Dementsprechend bietet sich eine Abkürzung der Signatur an, die in Elm als *type alias* bezeichnet wird. Dabei handelt es sich um eine Repräsentation einer komplexen Datenstruktur in einer kurzen Schreibweise. In Abbildung 2.16 wird eine Funktion mit einem komplexen *Record* als

```

1 isOldEnough : { name : String
2               , profile : String
3               , age : Int
4               } -> Bool
5 isOldEnough user =
6     ...

```

Abbildung 2.16: Definition einer Funktion ohne Typen Alias

Übergabeparameter beschrieben. An dieser Stelle hat der *Record* lediglich drei Felder, führt allerdings schon zu einem recht unübersichtlichen Quellcode. Mit Hilfe des *type alias* kann die Signatur gekürzt werden, indem die Repräsentation des *Records* ausgelagert wird. In Abbildung 2.16 können die Auswirkung davon betrachtet werden. Nachdem der *type alias* erstellt wurde, kann das Konstrukt mit dem entsprechenden *alias* angesprochen werden. Zeile 5 verdeutlicht, wie der *alias* genutzt wird. Der Algorithmus der Funktion bleibt völlig unberührt, wodurch diese Änderung eher kosmetischer Natur ist.

```

1 type alias User = { name : String
2                   , profile : String
3                   , age : Int
4                   }
5 oldEnough : User -> Bool

```

Abbildung 2.17: Definition einer Funktion mit Typen Alias

```

1 module MyModule exposing (..)
2 module MyModule exposing (add, anotherMethod)

```

Abbildung 2.18: Mögliche Deklarationen eines Elm-Moduls

Module

Oftmals ist es sinnvoll Quellcode der sich nur auf eine Problemlösung bezieht zu gruppieren. Auch in Elm ist es möglich Code auszulagern in sogenannte *Module*. Sie beschreiben eine Art Container, in dem der Code isoliert von den anderen Projektteilen betrachtet werden kann. Um ein *Modul* in Elm zu erstellen, muss eine neue Datei vom Typ *elm* erzeugt werden. Abbildung 2.18 zeigt zwei Beispiele, wie eine Quelldatei als Modul deklariert werden kann. Zeile 1 gibt dabei sämtliche Funktionen die im Modul beinhaltet sind nach außen weiter, sobald das Modul importiert wird. Dies sollte nur gemacht werden, wenn das Modul sich um eine Art Bibliothek handelt, in der sämtliche Funktionen verfügbar gemacht werden sollen. In Zeile 2 hingegen wird ersichtlich, wie nur ausgewählte Funktionen nach außen sichtbar gemacht werden. Auf der anderen Seite hingegen würde der Entwickler das Modul *MyModule* importieren und daraufhin die Funktionen *add* und *anotherMethod* nutzen können.

Importierung

Jeder Entwickler hat ferner die Möglichkeit die zuvor erstellten Module an einer anderen Stelle zu importieren. Dabei wird das gesamte Modul in den aktuellen Namensraum geladen und nutzbar gemacht. In welcher Ausprägung die Funktionen des importierten Moduls eingebunden werden, hängt von der Spezifikation des Imports ab. Die Abbildung 2.19 zeigt vier mögliche Arten das

```

1 import MyModule
2 import MyModule exposing (..)
3 import MyModule exposing (add, anotherMethod)
4 import MyModule as MyVeryOwnModuleName

```

Abbildung 2.19: Mögliche Formen der Importierung eines Elm-Moduls

Modul *MyModule* zu importieren. Zunächst einmal werden alle Funktionen,

die das Modul selbst über das Stichwort *exposing* freigibt geladen. Das Einbinden wie in Zeile 1 hat zur Folge, dass sämtliche Funktionen mit dem expliziten Modulnamen voran angesprochen werden müssen. Die Funktion *add* wird beispielsweise mit *MyModule.add* aufgerufen. Zeile 2 wiederum hat zur Folge, dass alle Funktionen des Moduls *MyModule*, darunter auch die Funktion *add*, in den Namensraum des aktuellen Moduls geladen werden. Logischerweise ist an dieser Stelle für den Aufruf der *add*-Funktion nichts weiter notwendig, außer wenn das aktuelle Modul eine gleichnamige Funktion besitzt. In diesem Fall ist die Nutzung analog der Anwendung aus dem vorherigen Beispiel. Die Einbindung in Zeile 3 wirkt sich reduzierender auf den aktuellen Namensraum aus. Damit ist gemeint, dass nur die explizit genannten Funktionen nach dem Stichwort *exposing* in den Namensraum geladen werden, alle anderen Funktionen müssen mit dem entsprechenden Modulnamen vorweg angesprochen werden. Schlussendlich kann der Namensraum des eingebundenen Moduls durch den Zusatz des *as* Stichwortes, gefolgt vom gewünschten Namensraum verändert werden. Die Nutzung der Funktionen ist analog zu den vorherigen Beispielen und kann beliebig kombiniert werden.

Online-IDE

Damit begonnen werden kann mit Elm zu programmieren, ist es nicht notwendig sämtliche Tools auf dem lokalen Gerät zu installieren. Vielmehr haben die Entwickler von Elm eine Online IDE erstellt, mit der sofort online entwickelt werden kann. Die Abbildung 2.20 zeigt diese IDE mit einem typischen

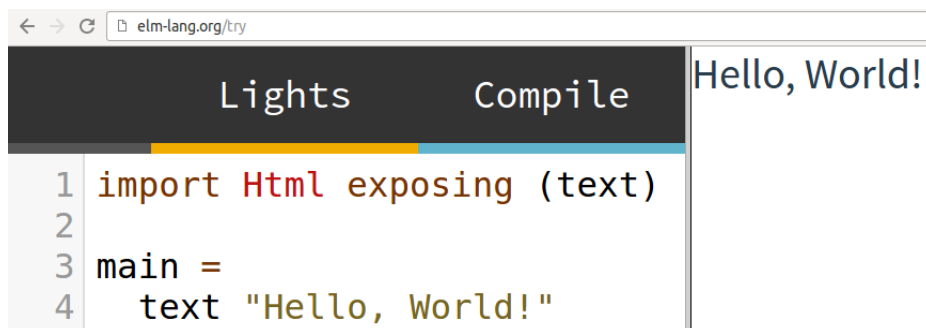


Abbildung 2.20: Die Online-IDE von Elm

Hello – World Beispiel. Auf der linken Seite ist eigentliche IDE erkennbar, während die rechte Seite das Ergebnis nach dem Kompiliervorgang ausgibt. Sollte es einen Fehler während des kompilierens geben, wird die entsprechende Fehlermeldung des *elm – compiler* dort ausgegeben. Mit Hilfe der Schaltfläche *Lights* können zum Einen die Farben der IDE invertiert werden, während der Knopf *Compile* den geschriebenen Quellcode kompiliert und ausführt. Kleinere Projekte können entsprechend bequem mit diesem Tool realisiert werden. Die IDE bietet einen schnellen Einstieg in die Programmiersprache.

Installation

Die Online-IDE bietet einen schnellen Einstieg in die Programmiersprache, ohne Programme lokal installieren zu müssen. Jedoch sind manche Konzepte nicht in der Online-IDE nutzbar, so dass Elm lokal installiert werden muss. Das ist beispielsweise der Fall, sobald spezifische Pakete aus dem Paketmanager von Elm installiert und benutzt werden möchten. Mehr Informationen über den Elm-eigenen Paketmanager folgen in der Sektion Start eines Projektes.

Die Elm-Webseite liefert Anleitungen um Elm auf Windows, Mac und allen NodeJs-kompatiblen Betriebssystemen zu installieren. Im folgenden werden die Kommandos zur Installation von Elm und dem NPM unter Ubuntu 14.04 64bit erläutert, da der praktische Teil dieser wissenschaftlichen Arbeit unter diesem Zielsystem vorgenommen wurde. Die Programmiersprache Elm wird über den NPM verbreitet und muss somit zuvor installiert werden. Da der NPM wiederum auf der Plattform NodeJs basiert, muss auch das dazugehörige Paket installiert werden. Für die Installation sind die Kommandos aus Abbildung 2.21 in der Kommandozeile auszuführen. Dadurch wird das NodeJs-Paket

```
1 $ sudo apt-get update
2 $ sudo apt-get install nodejs
3 $ sudo apt-get install npm
```

Abbildung 2.21: Installation des NPM

installiert und ausführbar gemacht. Anschließend sollte überprüft werden, ob die Installation erfolgreich war und sowohl NodeJS, als auch der NPM verfügbar sind. Das Kommando in Zeile 1 der Abbildung 2.22 liefert ein Ergebnis,

```
1 $ node -v && npm -v
2 $ npm install -g elm
```

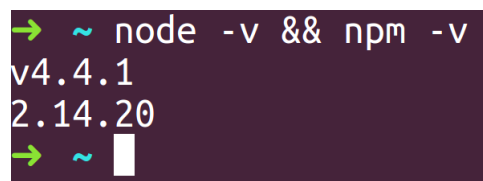
Abbildung 2.22: Installation von Elm

wie es in Abbildung 2.23 zu sehen ist. Elm kann nun über den NPM mit dem Kommando aus Zeile 2 installiert werden. Das Kürzel `-g` installiert die neueste Elm-Version global für alle Projekte auf dem System. Entfernt man es, wird die Zielversion nur für den aktuellen Ordner zugänglich gemacht. Um die in dieser wissenschaftlichen Arbeit behandelte Version 0.17 zu installieren, ist der Zusatz `@0.17` direkt hinter dem globalen Flag notwendig. Nachdem alle Kommandos fehlerfrei ausgeführt wurden sollte Elm über die Konsole nutzbar sein.

Start eines Projektes

Nachdem die Installation von Elm erfolgreich war, kann nun ein neues Projekt lokal erstellt werden. Dazu gehört zunächst einmal die Initialisierung des

Projektordners, wodurch notwendige Ordner erstellt und Pakete installiert werden. Die Entwickler von Elm stellen dafür den Paketmanager *elm - package* bereit. Dieses Tool wird zur Installation von externen Modulen, sogenannte Bibliotheken oder Pakete, benutzt. Zusätzlich hilft der Paketmanager dabei, das eigene Projekt lauffähig zu halten und nicht durch auftretende Aktualisierungen einzelner Pakete die Lauffähigkeit des eigenen Projektes zu verletzen. Dabei verfolgen die Entwickler des Paketmanagers bestimmte Versionsregeln, die auf jedwede Änderung an einem Paket angewendet wird. Alle Versionsnummern haben dabei genau drei Felder *MAJOR.MINOR.PATCH*, wobei der Beginn der Versionsnummern stets 1.0.0 ist. Die Felder der Versionsnummern ändern sich abhängig der Änderungen an der API eines Paketes. Die harmloseste Änderung ist dabei der *PATCH*. Das bedeutet, dass die Application Programming Interface (API) unverändert ist und keinerlei Gefahr für die Lauffähigkeit besteht. Es kann sich hierbei um beispielsweise Verbesserungen eines Algorithmus oder andere interne Änderungen handeln, die jedoch für den außenstehenden Entwickler keine Bedeutung haben. Der nächstgrößere Schritt stellt die Aktualisierung der *MINOR* Versionsnummer dar. Wie bei der Aktualisierung auf die nächste *PATCH*-Version, ist auch hier eine Aktualisierung unbedenklich. Das *MINOR*-Feld sagt aus, dass neue Methoden hinzugefügt wurden, alte Methoden jedoch unberührt blieben. Wichtige Änderungen an einem Paket bei der alte Methoden maßgeblich verändert, oder sogar gelöscht wurden, werden mit dem Feld *MAJOR* ausgedrückt. Hat sich dieses Feld verändert, wird eine Aktualisierung aller Wahrscheinlichkeit nach dazu führen, dass ein Programm nicht mehr lauffähig ist und Änderungen am eigenen Quellcode vorgenommen werden müssen. Nutzt ein Entwickler beispielsweise das Paket *elm - lang/html* mit der Version 1.2.3, wird der importierende Programmcode voraussichtlich kompatibel mit allen zukünftigen Versionen des Pakets bis Version 2.0.0 sein. Ab diesem Zeitpunkt sind die vorgenommenen Änderungen so massiv, dass eine Aktualisierung nicht ohne weiteres möglich ist. Der Paketmanager *elm - package* erzwingt die beschriebene Versionierung bei jedem Paket, bevor es veröffentlicht werden kann.[4, Vgl.]. Die Abbildung 2.24 zeigt, wie ein Paket installiert und der Projektordner eingerichtet wird. Der Paketmanager installiert nach der Bestätigung das angeforderte Paket und alle darin enthaltenen, externen Abhängigkeiten. Des Weiteren wird die Datei *elm - package.json* erstellt, mit grundlegenden Informationen über das eigene Projekt, wie beispielsweise die Versionsnummer, eine Zusammenfassung oder die zu verwendende Lizenz, sowie eine Liste von



```
→ ~ node -v && npm -v
v4.4.1
2.14.20
→ ~
```

Abbildung 2.23: Überprüfung der erfolgreichen Installation von NodeJs und dem NPM

```

→ elm-application elm-package install elm-lang/html
Some new packages are needed. Here is the upgrade plan.

Install:
  elm-lang/core 4.0.1
  elm-lang/html 1.0.0
  elm-lang/virtual-dom 1.0.2

Do you approve of this plan? (y/n) y
Downloading elm-lang/core
Downloading elm-lang/html
Downloading elm-lang/virtual-dom
Packages configured successfully!
→ elm-application ls
elm-package.json  elm-stuff

```

Abbildung 2.24: Installation eines externen Paketes über den Elm-Paketmanager

installierten externen Paketen. Der Ordner *elm-stuff* enthält die zuvor installierten Pakete. Nun muss noch manuell eine *.elm*-Datei erstellt werden, üblicherweise mit dem Namen des Projektes. In dieser Datei können die zuvor gezeigten Konzepte genutzt werden, um das gewünschte Programm zu verwirklichen.

Fertigstellung

Zum Kompilieren eines Elm-Programmes wird das Tool *elm-make* genutzt. Es kompiliert den gesamten Quellcode eines Ordners, oder die Dateien, welche dem Befehl hinzugefügt wurden. Ferner kann der Entwickler entscheiden, ob eine *.html*-Datei erstellt werden soll, in der bereits das kompilierte Elm-Programm eingebaut wird, oder eine *.js*-Datei, so dass das Einbinden oder die Auslieferung selbstständig vorgenommen werden kann.

Elm Reactor

```

→ elm-application elm-reactor
elm reactor 0.17.0
Listening on http://localhost:8000/

```

Abbildung 2.25: Der gestartete Elm-Webserver

Wahlweise kann während der Entwicklung das mitgelieferte Tool *elm-reactor* zum Testen der Elm-Applikation genutzt werden. Der *elm-reactor* stellt eine Art Webserver dar, durch den die Applikation automatisch bei jedem Aufruf kompiliert und gestartet wird, um sie dem Nutzer zugänglich zu machen. Der Nutzer kann die Applikation im Browser über das lokale Netzwerk öffnen und mit ihr interagieren. Die Abbildung 2.25 zeigt den gestarteten

Webserver in der Kommandozeile. Er wird standardmäßig auf dem Port 8000 gestartet.

3 Evaluierung der Programmiersprache Elm

Die Programmiersprache Elm wird mit Hilfe einer empirischen Analyse in Form der Entwicklung eines praktischen Beispiels evaluiert. Dabei wird ein fertiges Template einer Single Page Application (SPA)¹, welches in HTML, CSS und JS programmiert wurde, in nativen Elm-Code überführt. Ausschließlich die vorhandenen JS und CSS-Dateien sollen bestehen bleiben. Zur aussagekräftigen Evaluierung der Programmiersprache Elm hinsichtlich einer Nutzung im Bereich der Webentwicklung bedarf es zunächst einiger Bewertungskriterien, anhand derer eine Aussage möglich ist. Nachdem ein Bewertungsmuster erstellt und erläutert wurde, befasst sich der darauf folgende Teil der wissenschaftlichen Arbeit mit der Überführung des Templates in nativen Elm-Code. Zuletzt werden Beobachtungen, die während der Entwicklung auftraten, erläutert und die zuvor erwähnten Kriterien ausgewertet.

3.1 Bewertungsmuster

Das fertige Template ist eine SPA mit Elementen, wie sie typischerweise auf einer solchen Webseite vertreten sind. Die SPA wird genutzt um ein Produkt oder Konzept schnell und einfach zu präsentieren, ohne den Nutzer mit Informationen zu überfluten und Unübersichtlichkeit in Form von tief verlinkten Unterseiten zu erzeugen. Oftmals wird eine SPA auch als Startseite benutzt und bietet nur eine geringe Anzahl an Funktionen. Die fertige SPA soll unter anderem die folgenden, typischen Elemente enthalten:

- Navigation mit Anchor-Elementen
 - Verkleinern der Navigation nach x Pixeln
 - ScrollSpy² zur Darstellung der aktuellen Position
- Titelbild mit einem vertikal und horizontal zentriertem Text
- Service-Sektion
- Twitter-Bootstrap-CSS³ mitsamt allen Funktionen

¹Eine SPA bezeichnet eine Webseite mit effektiv nur einer aktiven Seite und ohne Unterseiten (ausgenommen Impressum, AGB und Datenschutz).

²Scrollspy gibt die aktuelle Position in einem HTML-Dokument an.

³Twitter-Bootstrap ist ein bekanntes CSS-Framework.

- Vorgefertigtes, responsive Design⁴
- Aufklappbares Menü
- Portfolio mit Bildern, wobei ein Klick einen asynchronen Request ausführt und Daten nachlädt
- Formular zur Kontaktaufnahme

Mit diesen Elementen kann eine typische SPA verwirklicht werden. Die Navigation bietet dabei die Möglichkeit für den Nutzer schnell zwischen einzelnen Sektionen der Seite zu wechseln. Das initiale Titelbild mit einem zentrierten Text gibt den Kontext der Präsentation an und soll das Interesse des Nutzers anregen. Die folgende Service-Sektion wird dazu genutzt, allgemeine Informationen über das beworbene Produkt anzuzeigen. In der darauf folgenden Portfolio-Sektion werden dem Nutzer mehrere Bilder des Produktes angezeigt, wobei ein Klick auf eines der Bilder dazu führt, dass ein Popup erzeugt wird, in welches mit Hilfe eines AJAX-Requests Informationen asynchron vom Server angefordert und im Nachhinein geladen werden. Zuletzt kann sich der Nutzer ein Kontaktformular ausfüllen, das auf Korrektheit hin überprüft wird.

3.2 Allgemeine Bewertungskriterien

Während der Überführung des Templates soll der erzeugte Code, sowie der Weg dahin analysiert werden. Hierbei sind Aspekte wie Wiederverwendbarkeit und Effizienz von großer Bedeutung. Aber auch die Produktivität während der Arbeit mit dem Code soll betrachtet werden. Die Bewertungskriterien setzen sich zum Großteil aus den zugrunde liegenden Kriterien aus dem Dokument „Evaluating Programming Languages“ der University of Washington zusammen [1, Vgl.]. Im Folgenden soll die Notwendigkeit der Kriterien und ihre eigentliche Bedeutung verständlich gemacht werden.

3.2.1 Wartbarkeit und Lesbarkeit

Es ist unabdingbar, dass verfasster Quellcode wartbar ist. Dazu gehört einerseits, dass der Code lesbar ist, unabhängig von der Zeit die sich ein Entwickler bereits mit der Codebasis auseinandergesetzt hat. Damit Quellcode lesbarer wird, reicht es schon aus Kommentare im Code zuzulassen, die beispielsweise eine Funktion und ihr Ziel beschreiben, oder wichtige Informationen über einen Algorithmus enthalten. Des Weiteren sollten Funktionen und Variablen kurze und prägnante Namen haben, wodurch die Verständlichkeit des Quellcodes unterstützt wird. Die Programmiersprache Elm kann als wartbar bezeichnet werden, insofern es Möglichkeiten zum Kommentieren des Codes gibt, oder bestenfalls automatisch Informationen in Form von Kommentaren über Funktionen und Verhaltensweisen von Algorithmen generiert werden.

⁴Bezeichnet ein Design, dass abhängig von der Größe des Gerätes HTML-Elemente unterschiedlich positioniert.

3.2.2 Zuverlässigkeit

Für einen Entwickler von Software ist es wichtig, dass das ausgelieferte Programm letzten Endes fehlerfrei funktioniert. Dazu gehört, dass das Programm nicht unvorhergesehen abstürzt, andere Systeme beeinträchtigt, oder dem späteren Nutzer der Software anderweitig Probleme beschert. In den meisten Fällen helfen die Editoren mit denen der Quellcode geschrieben wird bereits, indem syntaktische Fehler durch Markierungen sichtbar, oder Vorschläge zur Vervollständigung des angefangenen Codes gemacht werden. Wichtig ist entsprechend, dass die Programmiersprache auf offensichtliche Fehler, entweder durch Plugins für den Editor oder durch den Compiler selbst, aufmerksam macht und sie somit verhindert und nicht erst im Produktionssystem den Fehler zulässt. Als Testfall wird Quellcode bewusst mit Fehlern versehen, die zu einem Absturz des Programmes während der Laufzeit, oder zu anderen Problemen führen würden. Erkennt der *elm – compiler* diese Fehler oder umgeht den Absturz, gilt das Kriterium als erfüllt.

3.2.3 Portabilität

Häufig sind Programmiersprachen nicht auf allen Systemen lauffähig, um so Applikationen zu erstellen. Es kommt dabei sehr stark auf die Hardwarekomponenten und das Betriebssystem des Zielsystems an. Es ist wünschenswert, dass Quellcode nur einmal geschrieben und auf das Zielsystem übertragen werden kann, ohne großartige Änderungen am Quellcode vornehmen zu müssen. Letzten Endes wird versucht den Quellcode auf mehreren Zielsystemen zu kompilieren. Elm liefert Installationsanweisungen für die Betriebssysteme *Mac*, *Windows* und allen Betriebssystemen, die *nodejs* unterstützen. Gibt es keinerlei Differenz in Form von Fehlern oder Warnungen während des Kompilierens, gilt die Portabilität als erfüllt.

3.2.4 Effizienz

Die Entwicklung von Applikationen ist zeitaufwändig und verursacht Kosten. Entsprechend ist es von Vorteil, wenn Quellcode schnell erzeugt, getestet und als fertiges Produkt in Form von Software an den Kunden ausgeliefert werden kann. Dabei ist es wichtig, dass der Compiler den Quellcode schnell in ein lauffähiges Programm umwandelt und das Resultat effizient arbeitet, sprich schnell ist. Damit eine Aussage über die notwendige Zeit für das kompilieren des Quellcodes getroffen werden kann, wird der Code zehn mal kompiliert und die benötigte Zeit gemessen. Anschließend wird der höchste und niedrigste Wert verworfen. Um ein Caching⁵ durch den Compiler zu verhindern, werden vor jedem Kompiliervorgang die zuvor erzeugten Programmdateien im Ordner *elm – stuff* gelöscht. Im Anschluss wird das arithmetische Mittel der verbleibenden acht Werte ermittelt. Die Performance der Programmiersprache wird

⁵Das lokale Zwischenspeichern der Daten wird als Caching bezeichnet und verhindert in diesem Fall, dass die Applikation in vollem Umfang neu kompiliert wird.

nicht anhand des hier entwickelten Projektes, sondern der externen Applikation *TodoMVC Performance Comparison*⁶ ermittelt. Dieses Tool ermöglicht es, anhand der Beispiel-Applikation *TodoMVC*, die in mehreren Programmiersprachen realisiert wurde, die Performance der Applikation in den einzelnen Programmiersprachen zu testen. Dabei werden automatisch gewisse Aktionen in der jeweiligen Sprache durchgeführt, die benötigte Zeit gemessen und die Ergebnisse am Ende gegenübergestellt. Mit Hilfe des Tools werden 20 Messungen durchgeführt und das Endergebnis betrachtet. Ist Elm signifikant schneller als der Großteil der anderen getesteten Programmiersprachen, gilt das Kriterium als erfüllt. Die anderen Applikationsumgebungen sind ebenso in nativem JS oder einer zu JS kompilierenden Programmiersprache geschrieben. Dadurch kann eine Aussage über die tatsächliche Effizienz der Applikation im Vergleich getroffen werden.

3.2.5 Wiederverwendbarkeit

Vorhandene Codeteile neu zu verfassen oder händisch kopieren zu müssen ist sehr ineffizient. Besser ist es, wenn Funktionen mehrfach genutzt werden können. Auf diese Weise müssen Änderungen nicht mehrmals vorgenommen werden und die Fehleranfälligkeit sinkt. Ferner sollten Funktionen nicht nur wiederverwendbar, sondern isoliert in einem eigenen Modul definiert werden können. Besteht die Möglichkeit Module zu erstellen und diese an unterschiedlichen Stellen beliebig anzuwenden, gilt das Kriterium als erfüllt. Erweitert wird das Kriterium durch einen zusätzlichen Zugriffsschutz von außen, das bedeutet, dass innerhalb eines Moduls definiert werden kann, welche Funktionen nach außen hin sichtbar sind.

3.3 Web-spezifische Bewertungskriterien

Die bisherigen Kriterien ermöglichen es, die Programmiersprache objektiv beurteilen zu können. Da sich diese wissenschaftliche Arbeit allerdings ausdrücklich mit der Evaluierung von Elm für Webapplikationen auseinandersetzt, müssen auch diese Kriterien bewertet werden. Zu diesem Zweck wird eine SPA mit nur wenig Programmlogik erstellt, die jedoch die Grundkonzepte einer Webseite nutzen. Diesbezüglich werden im folgenden einige Aspekte vorgestellt.

3.3.1 Browser Kompatibilität

Ein weiterer wichtiger Aspekt ist die Kompatibilität der Browser mit den genutzten Programmiersprachen. Da der Browser die Darstellung des HTML und CSS Quellcodes übernimmt, sowie die Manipulationen des DOM durch JS, ist es wichtig, dass der Browser den vorhandenen Quellcode lesen und ausführen kann. Sämtliche Sprachen wie HTML, CSS und JS befinden sich im konstanten Wandel und werden stets weiter entwickelt. Dabei werden nicht

⁶Abrufbar unter <https://github.com/evancz/todomvc-perf-comparison/>.

nur vorhandene Fehler behoben, sondern auch neue Eigenschaften hinzugefügt, sowie teilweise ersetzt oder verworfen. Diese Änderungen können darin münden, dass Nutzer unterschiedlicher Browser, auch unterschiedliche Ergebnisse angezeigt bekommen, manche Features gar nicht erst funktionieren oder die Applikation im schlimmsten Fall abstürzt. Dementsprechend ist es wichtig, dass die Applikation auf den gängigen Browsern fehlerfrei funktioniert, insbesondere den fünf meistgenutzten Browsern Google Chrome, Safari, Internet Explorer, Firefox und Opera (vgl. Abbildung 3.1). Sollte die Applikation fehlerfrei in allen Browsern starten und die Funktionalität vollends gegeben sein, kann dieses Kriterium als erfüllt angesehen werden.

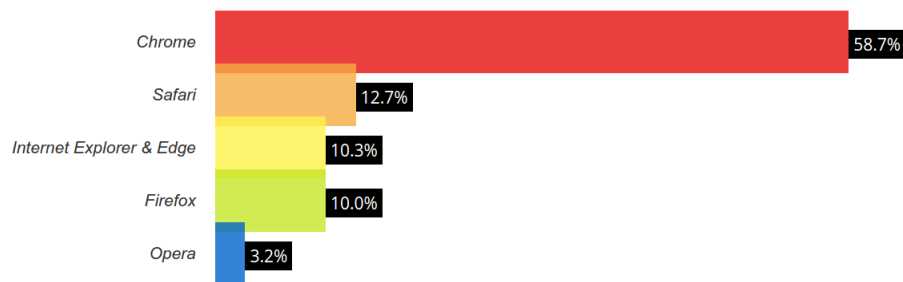


Abbildung 3.1: Die fünf meistgenutzten Browser im Mai 2016 [19]

3.3.2 Interoperabilität

Auch bei den Webapplikationen ist es wichtig, bereits existente Frameworks und Problemlösungen nutzen zu können. Folglich müssen externe JS und CSS Bibliotheken ohne große Probleme eingebettet werden können, ohne den Quellcode zu verändern. Sollte es die Möglichkeit geben bestehende, externe Bibliotheken einzubinden oder anderweitig nutzbar zu machen, gilt das Kriterium als erfüllt.

3.3.3 Asynchrone Verarbeitung

Große Datenmengen, deren Übertragung einige Sekunden in Anspruch nimmt, oder Aufgaben die langwierig sind, sollten nicht synchron ausgeführt werden. Stattdessen bietet es sich an, asynchrone Aufgabenblöcke zu definieren, wie beispielsweise das Nachladen von Daten. Am Beispiel der hier geplanten SPA bedeutet dies, dass Informationen von einem Webserver erst angefordert und sichtbar gemacht werden, sobald der Nutzer diese durch einen Klick auf ein Element anfordert. Um das Kriterium der asynchronen Verarbeitung zu erfüllen, sollte Elm die Möglichkeit liefern, asynchrone Daten anzufordern und die Antwort darzustellen. Die Applikation darf dabei nicht in einen undefinierten Zustand kommen.

3.3.4 Dateigröße

Die Größe einer Datei wirkt sich negativ auf die Dauer der Übertragungszeit aus. Eine schlechte Übertragungsrate kann zu erheblichen Verzögerungen der Darstellung führen. Um die Dateigröße von üblichen JS-Dateien zu verringern, werden oftmals externe Tools zur Entfernung von Leerzeichen oder der Verkürzung von Funktions- und Variablennamen genutzt. Für die anschließende Bewertung dieses Kriteriums sollte der Compiler die Möglichkeit zur automatischen Verkleinerung der Dateigröße mit einer Effizienz von mindestens 50% bieten. Zusätzlich sollte die Dateigröße eines Elm-Programmes mit den eingebundenen Standard-Bibliotheken *elm – lang/core* und *elm – lang/html* die Dateigröße der Frameworks *AngularJs*, *Ember* und *React* nicht übersteigen. Die Größen der einzelnen Frameworks werden dabei der Statistik von Anton Vynogradenko [23, Vgl.] entnommen.

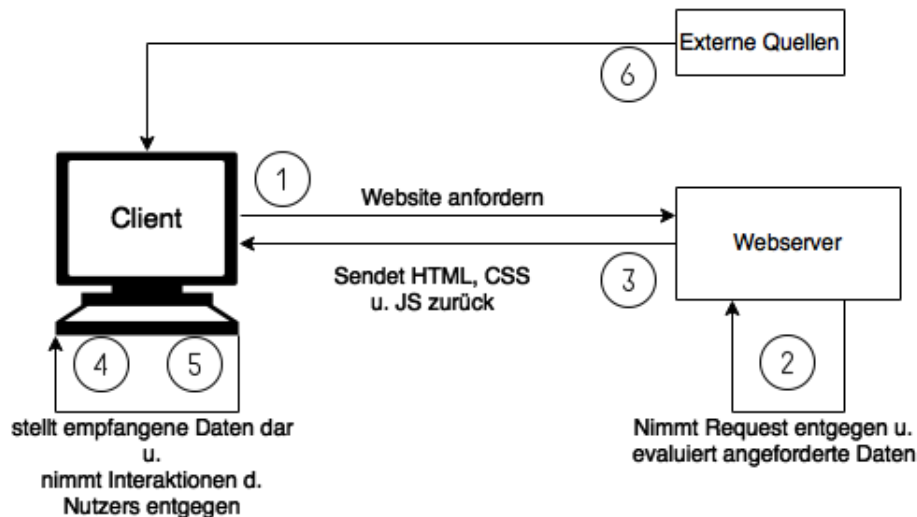


Abbildung 3.2: Zeigt die Kommunikation zwischen Client und Webserver

3.4 Empirische Analyse

Die folgende Sektion unterstützt und erläutert die praktische Ausarbeitung der SPA. Dabei wird zunächst der typische Programmablauf der Applikation erörtert. Ferner wird die genutzte Entwicklungsumgebung vorgestellt. Die Entwicklung wird in einzelne Abschnitte unterteilt und erläutert.

3.4.1 Programmablauf

Die Abbildung 3.2 zeigt die geplante Kommunikation zwischen dem Client und dem Webserver. In Schritt 1 fordert der Nutzer die Webseite an und erzeugt dadurch einen Request. Dieser geht in Schritt 2 bei dem Webserver ein und wird verarbeitet. Abhängig von der angeforderten URL erzeugt der Webserver eine Antwort, mit allen zu sendenden Informationen und dem dazugehörigen HTML, CSS und JS Code. Im nächsten Schritt 3 werden diese Daten zurück an den Client gesendet. Der Client nimmt die Daten entgegen, wie in Schritt 4 beschrieben. Des Weiteren macht der Client die Daten entsprechend sichtbar, so dass der Nutzer eine vollständige Präsentation der Webseite sieht. Schritt 5 beschreibt die mögliche Interaktion des Nutzers mit dem ihm präsentierten Dokument. Jede Interaktion wird durch JS erkannt und erzeugt eine Reaktion. Unter Umständen ist dies ein Request an den Server, womit bei Schritt 1 begonnen wird. Externe Daten wie CSS oder bestehende JS-Skripte werden asynchron nachgeladen, wie es Schritt 6 andeutet. Dabei kann dieser Schritt zeitlich gesehen zu jedem Zeitpunkt nach Schritt 3 geschehen.

3.4.2 Entwicklungsumgebung

Die zugrunde liegende Elm-Version 0.17 wird auf einem Testgerät mit dem Betriebssystem Ubuntu 14.04 64bit installiert. Die Hardware enthält 16GB RAM und einen Intel Core i7-4702HQ Prozessor mit einer Taktung von 8x2.2GHz.

Als Editor wird Atom mit den Elm-spezifischen Plugins *language – elm*⁷, *linter – elm – make*⁸ und *elm – format*⁹ verwendet. Diese Plugins unterstützen die Entwicklung von Elm-Programmen, indem der Quellcode beim speichern automatisch dem Style-Guide entsprechend formatiert, eventuelle Fehler bei der Kompilierung direkt im Editor sichtbar gemacht und der Code syntaktisch gefärbt wird, sowie Vorschläge zur Vervollständigung des geschriebenen Quellcodes gemacht werden. Da die Pakete ständig aktualisiert und verändert werden, wird an dieser Stelle von einer detaillierten Beschreibung zur Installation und Verwendung abgesehen und auf die Dokumentationen der einzelnen Pakete verwiesen. Der angefertigte Quellcode wird durch den Compiler *elm – make* kompiliert. Anschließend wird die fertige SPA mit den Browsern Google Chrome (Version 51), Internet Explorer (Version 11), Mozilla Firefox (Version 47) und Opera (Version 38) getestet. Die getesteten Browser entsprechen dabei 82,2% der fünf meistgenutzten Browser im Mai 2016, wie Abbildung 3.1 zeigt. Zum Zeitpunkt dieser wissenschaftlichen Arbeit stand kein Testgerät mit dem Browser Safari zur Verfügung.

3.4.3 Grundaufbau

Damit der Nutzer letzten Endes eine Webseite besuchen kann, muss ein auslieferbares Dokument erzeugt werden. Hierfür wird zunächst die *index.html*-Datei aus dem Vorlage *Agency*¹⁰ übernommen und entsprechend angepasst. Die Datei dient dabei als Grundgerüst für die Elm-Applikation und lädt externe CSS und JS-Dateien. Zunächst wird sämtlicher Code der Datei innerhalb des *body*-Elementes entfernt. Lediglich die *script*-Tags bleiben bestehen, so dass sämtliche vordefinierte CSS und JS-Dateien weiterhin geladen werden.

Damit die Elm-Applikation in das Grundgerüst geladen werden kann, ist ein weiterer *script*-Tag notwendig, der die kompilierte Applikation *elm.js* einbindet. Des Weiteren muss die Elm-Applikation explizit aufgerufen und gestartet werden. Die Abbildung 3.3 zeigt die *index.html* mit den notwendigen Änderungen in Zeile 22, in welcher die kompilierte Elm-Applikation eingebunden wird, sowie den Zeilen 43 bis 46, in denen die Applikation gestartet wird und ein Ziel-HTML-Element übergeben bekommt. Damit die Datei *elm.js* eingebunden werden kann, muss sie zunächst erstellt werden. Dafür wird der Elm-Compiler *elm – make* mit dem Zusatz *Main.elm – –output elm.js* genutzt. Hierbei wird die Datei *Main.elm* kompiliert und das Ergebnis in die Datei *elm.js* gespeichert. Es bestehen insgesamt drei Möglichkeiten die Elm-Applikation innerhalb der *index.html* aufzurufen:

1. *fullscreen*: Der erzeugte Code der Applikation wird in den *body*-Tag einer HTML-Datei geladen und überschreibt den sonstigen HTML-Code
2. *embed*: Der erzeugte Code der Applikation wird in den übergebenen DOM-Knoten geladen

⁷Abrufbar unter <https://atom.io/packages/language-elm>.

⁸Abrufbar unter <https://atom.io/packages/linter-elm-make>.

⁹Abrufbar unter <https://github.com/triforkse/atom-elm-format>.

¹⁰Abrufbar unter <https://github.com/BlackrockDigital/startbootstrap-agency> [20].

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1">
7      <meta name="description" content="">
8      <meta name="author" content="">
9      <title>Agency - Start Bootstrap Theme</title>
10     <!-- Bootstrap Core CSS -->
11     <link href="css/bootstrap.min.css" rel="stylesheet">
12     <!-- Custom CSS -->
13     <link href="css/agency.css" rel="stylesheet">
14     <!-- Custom Fonts -->
15     <link href="font-awesome/css/font-awesome.min.css" rel="stylesheet" type="text/css">
16     <link href="https://fonts.googleapis.com/css?family=Montserrat:400,700" rel="stylesheet">
17     <link href='https://fonts.googleapis.com/css?family=Kaushan+Script' rel="stylesheet">
18     <link href='https://fonts.googleapis.com/css?family=Droid+Serif:400,700,400italic' rel="stylesheet">
19     <link href='https://fonts.googleapis.com/css?family=Roboto+Slab:400,100,300' rel="stylesheet">
20
21     <!-- Compiled Elm-Application -->
22     <script type="text/javascript" src="elm.js"></script>
23   </head>
24   <body id="body" class="index">
25     <!-- jQuery -->
26     <script src="js/jquery.js"></script>
27
28     <!-- Bootstrap Core JavaScript -->
29     <script src="js/bootstrap.min.js"></script>
30
31     <!-- Plugin JavaScript -->
32     <script src="http://cdnjs.cloudflare.com/ajax/libs/jquery-easing/1.3/jquery.easing.min.js"></script>
33     <script src="js/classie.js"></script>
34     <script src="js/cbpAnimatedHeader.js"></script>
35
36     <!-- Contact Form JavaScript -->
37     <script src="js/jqBootstrapValidation.js"></script>
38     <script src="js/contact_me.js"></script>
39
40     <!-- Custom Theme JavaScript -->
41     <script src="js/agency.js"></script>
42     <!-- Injects the elm-application -->
43     <script type="text/javascript">
44       var bodyElement = document.getElementById('body');
45       app = Elm.Main.embed(elmApp);
46     </script>
47   </body>
48 </html>

```

Abbildung 3.3: Grundaufbau der *index.html*, um die Elm-Applikation zu injizieren

```

1  module Main exposing (..)
2  import Html exposing (..)
3  import Html.Attributes exposing (..)
4  •import Html.Events exposing (..)
5  import Html.App as Html
6  •main =
7      Html.programWithFlags { init = init
8                              , view = view
9                              , update = update
10                             , subscriptions = subscriptions
11                             }
12
13  •subscriptions model =
14      Sub.batch []
15
16  •init passedModel =
17      Maybe.withDefault initialModel passedModel ! []
18
19  type alias Model =
20      { async_content : String }
21
22  •initialModel =
23      { async_content = "" }
24
25  •view model =
26      div [ id "elm-view" ] []
27
28  •update msg model =
29      case msg of
30          _ ->
31              ( model, Cmd.none )

```

Abbildung 3.4: Grundgerüst der Elm-Applikation

3. *worker*: Initialisiert die Applikation ohne grafische Benutzeroberfläche

Die Abbildung 3.3 zeigt, dass für dieses praktische Beispiel Version 2 genutzt wird. Daraus ergibt sich der Vorteil, dass die Applikation gezielt in einen vordefinierten Bereich der Webseite platziert werden kann. Ferner erlaubt diese Art der Initialisierung das Hinzufügen externer JS- und CSS-Dateien. Nutzt man beispielsweise Version 1, so ist die Elm-Applikation im absoluten Vordergrund und lässt die Interaktion mit anderen Elementen die zuvor auf der Webseite definiert wurden nicht mehr zu. Version 3 wiederum erzeugt keine grafische Benutzeroberfläche, wodurch ebenso keine Interaktion möglich ist. Wahlweise besteht die Möglichkeit über den mitgelieferten Elm-Webserver *elm – reactor* die Applikation manuell zu starten. Jedoch wird die Elm-Applikation dabei als *fullscreen*-Applikation gestartet und ist für die Zwecke dieser Arbeit nicht geeignet, weswegen auf die manuelle Kompilierung und Initialisierung zurückgegriffen werden muss.

Zusätzlich zum Grundgerüst der *index.html* muss nun noch das Grundgerüst der eigentlichen Elm-Applikation erstellt werden. Wie im Kapitel Konzept beschrieben, ist Elm nach einem Model-View-Update-Konzept aufgebaut. Entsprechend sind das die drei notwendigen Funktionen, die es zu realisieren gilt, damit die Elm-Applikation lauffähig ist. Um HTML-Code zu erzeugen gibt es das Elm-Paket *elm – lang/html*. Es liefert unter anderem die Funktionen *beginnerProgram*, *program* und *programWithFlags*. Diese kümmern sich um die Bereitstellung und Auslieferung der Applikation, so dass sich Entwickler ganz auf die eigentliche Programmierung konzentrieren können. Die Funktion *beginnerProgram* verlangt die bekannten Funktionen *model*, *view* und *update* als Übergabeparameter. Hierbei können jedoch keine asynchronen Funktionen wie HTTP-Requests genutzt werden. Dafür gibt es wiederum die erweiterte Funktion *program*, die als vierten Übergabeparameter sogenannte *subscriptions* erwartet. Sie werden für die Kommunikation zwischen Elm und JS, sowie Verbindungen zu Websockets genutzt. Die dritte und letzte Möglichkeit der Initialisierung ist die Funktion *programWithFlags*. Hierbei wird die Übergabe eines initialen *Models* an die Elm-Applikation ermöglicht, um den Zustand der Applikation dynamisch setzen zu können. Die Abbildung 3.5 zeigt beispielhaft die Implementierung

```
<script type="text/javascript">
  var elmApp = document.getElementById('elm-application');
  app = Elm.Main.embed(elmApp,
    { location: "test"
      , width: window.innerWidth - 15
      , height: window.innerHeight
    });
```

Abbildung 3.5: Eine beispielhafte Initialisierung der Elm-Applikation mit initialen, dynamischen Werten

der *Html.App.programWithFlags*-Funktion. Der zweite Übergabeparameter

an die *Elm.Main.embed*-Funktion ist dabei ein *Record* mit den gewünschten initialen Werten. Für die Umsetzung dieses praktischen Beispiels ist der Grundaufbau wie in Abbildung 3.4 notwendig. Dies sind die minimal notwendigen Funktionen, um in den weiteren Schritten die einzelnen Sektionen des Views von HTML nach Elm zu portieren, in einzelne Funktionen auszulagern und letzten Endes der Versuch, die gesamte Applikation zu modularisieren. Aus der Abbildung 3.4 wird ersichtlich, dass zunächst das gesamte Paket *elm – lang/html* importiert wird. Die vorherige Installation des Pakets ist analog zu der in der Abbildung 2.24 beschriebene Vorgehensweise. Innerhalb der *main*-Funktion werden alle notwendigen Funktionen an die *HTML*-Bibliothek weitergereicht. Dazu gehören die *init*, *view*, *update* und *subscriptions*-Funktionen. Wie der Name bereits suggeriert initiiert die *init*-Funktion ein *Model*. Dabei wird dieses entweder anhand der dynamisch übergebenen Werte, oder durch ein vordefiniertes Model (*initialModel*) erstellt, falls keine Daten übergeben wurden. Die Funktion *view* erzeugt in der Abbildung 3.4 zunächst ein *div*-Element, während die *update*-Funktion auf jede eingehende Interaktion ein Tupel mit einem unveränderten *model*, sowie keinem *Effekt* zurück gibt. Um die Applikation aufzurufen muss die Datei *index.html* im Browser geöffnet werden.

3.4.4 Überführung des Views

Nachdem der Grundaufbau der Elm-Applikation beidseitig ausgeführt wurde, können nun die einzelnen Sektionen der originalen *index.html* aus dem *Agency*-Template nativ in Elm überführt werden. Dafür wird das Online-Tool *html – to – elm*¹¹ zur Hilfe genommen. Der gesamte HTML-Code des *body*-Tags wird kopiert und zur Konvertierung in das entsprechende Textfeld des Tools eingefügt. Der konvertierte Code wird in die *view*-Funktion der *Main.elm*-Datei als Rückgabewert eingefügt. Das Tool *html – to – elm* sollte den Code bereits nach den Regeln des Style-Guides formatiert haben, so dass hiermit der erste Schritt der Portierung abgeschlossen ist. Um Namenskonflikte im aktuellen Namensraum zwischen Funktionen zu verhindern, werden innerhalb der importierten *HTML*-Bibliothek die genutzten Funktionen zur Erstellung von HTML-Code explizit genannt. Der Compiler sollte nach dem Entfernen des Kommandos *exposing(..)* eine Warnung ausgeben und auf die fehlenden Funktionen hinweisen, wie es in Abbildung 3.6 gezeigt wird. Schlägt der Compiler dabei vor, die Funktion aus dem Namensraum *Html.Attributes* aufzurufen, muss sie entsprechend in die Liste der zu ladenden Funktionen der *Attribute* hinzugefügt werden, wie in Abbildung 3.4 in Zeile 3. Die Funktionen zur Generierung von HTML-Tags hingegen sollten in die Liste der *import Html*s stehen. Nachdem alle Funktionen korrekt in den aktuellen Namensraum aufgenommen wurden, sollten keinerlei Compilerfehler auftreten.

¹¹Abrufbar unter <http://mbylstra.github.io/html-to-elm/>.

```

-- NAMING ERROR -----
Cannot find variable `class`

70|                                     , div [ class
                                     ^^^^^^
Maybe you want one of the following?

    clamp
    Html.Attributes.class
    Basics.clamp
    Debug.crash

-- NAMING ERROR -----
Cannot find variable `h4`

71|                                     [ h4 []
                                     ^^
Maybe you want one of the following?

    Html.h4

```

Abbildung 3.6: Elm-Compilerfehler bei fehlenden Funktionen im Namensraum

3.4.5 Beheben von JavaScript-Fehlern

Das Öffnen der Seite mitsamt der Developer-Tools von Google-Chrome verrät, dass die bisher eingebundenen JS-Dateien nicht fehlerfrei funktionieren. Nutzt man das Template wie vorgesehen, ist sämtliche Funktionalität vorhanden. Bei der Überführung des Templates in Elm gilt es allerdings zu beachten, dass die Elm-Applikation den *view* ausliefert und diesen erst verzögert darstellt. Dem menschlichen Betrachter fällt dies zwar nicht auf, jedoch greifen die JS-Skripte auf ein noch nicht geladenes Element zu. Es ist dementsprechend notwendig, die einzelnen Skripte anzupassen und mit Hilfe der $\$(document).ready()$ -Funktion des Frameworks *jQuery*¹² erst nach dem kompletten Seitenaufbau zu laden. Innerhalb der Datei *agency.js* muss nicht nur die Funktion $\$(\text{'a.page - scroll'}).bind(...)$, sondern auch die nachfolgenden Funktionen innerhalb des $\$(function(..))$ -Blocks liegen. Die JS-Datei *cbpAnimatedHeader.js* hingegen erzeugt einen Fehler, wie in Abbildung 3.7 zu sehen ist. Um diesen zu be-

✖ ▶ Uncaught TypeError: Cannot read property 'classList' of null classie.js:36

Abbildung 3.7: JavaScript-Fehler innerhalb der Developer-Tools

heben, muss der gesamte Dateiinhalt in eine $\$(document).ready()$ -Funktion geschachtelt werden. Zusätzlich ist es notwendig, dass die Codezeile wie sie in

¹²Abrufbar unter <https://learn.jquery.com/using-jquery-core/document-ready/>.

Abbildung 3.8 zu sehen ist, als erstes in der Funktion *scrollPage()* hinzugefügt wird. Der Code hat zur Folge, dass das Navigations-Element ausgewählt

```
header = document.querySelector( 'navbar-fixed-top' );
```

Abbildung 3.8: Notwendiger Code, um JavaScript-Fehler zu beheben.

und an die *classie*-Funktion weitergegeben wird. Fehlt diese Zeile, erfolgt ein Aufruf der *classie*-Funktion mit einem undefinierten Wert, wodurch der Fehler in Abbildung 3.7 zustande kommt. Nachdem diese beiden Fehler behoben wurden, kann nun die *ScrollSpy*-Funktionalität auf der Webseite betrachtet werden. Scrollt der Nutzer nun auf oder ab und befindet sich dabei in einer innerhalb der Navigation definierten Sektion, so wird das entsprechende Gegenstück in der Navigationsleiste farblich hinterlegt. Des Weiteren verkleinert sich die Navigationsleiste, wenn der Nutzer eine bestimmte Entfernung nach unten gescrollt hat.

3.4.6 Auslagern des Views

Um die einzelnen Sektionen der Webseite im Quellcode klar voneinander zu trennen, werden die einzelnen Sektionen in eine jeweils eigene Funktion ausgelagert. Für jede Sektion wird hierfür eine gleichnamige Funktion angelegt, die dem Muster in Abbildung 3.9 folgt. Die *id* eines HTML-Elementes könnte entsprechend für den Funktionsnamen *nameDerSektion* genutzt werden, da dieses Attribut ohnehin einzigartig in einem gesamten HTML-Dokument vorkommt und eine klare Namensgebung liefert. Am Beispiel der *portfolio*-

```
9  -- nameDerSektion - bspw.: "navigation"
10  nameDerSektion : Html Update.Msg
11  nameDerSektion =
12      section [ class "bg-light-gray", id "navigation" ]
13          [ div [ class "container" ]
14              [ div [ class "row" ]
```

Abbildung 3.9: Deklaration einer Sektion des Views in Elm

Sektion des zu überführenden Templates sieht die daraus resultierende Funktion teilweise aus wie in Abbildung 3.10. Die einzelnen Klassen und Identifier (ID) wurden beibehalten und übernommen. Nachdem die einzelnen Sektionen in Funktionen ausgelagert wurden fällt auf, dass die Sektionen nicht mehr angezeigt werden. Dafür ist es notwendig die ursprüngliche *view*-Funktion mit den Funktionsaufrufen der Sektionen zu versehen. Dieses Vorgehen kann in der Abbildung 3.11 betrachtet werden. Das Resultat der *view*-Funktion bleibt unverändert, ist jedoch nun im Code übersichtlicher.

```

10 portfolio : Html
11 portfolio =
12     section [ class "bg-light-gray", id "portfolio" ]
13         [ div [ class "container" ]
14             [ div [ class "row" ]
15                 [ div [ class "col-lg-12 text-center" ]
16                     [ h2 [ class "section-heading" ]
17                         [ text "Portfolio" ]

```

Abbildung 3.10: Ausgelagerter View in eine eigene Funktion

```

62 view model =
63     div [ id "elm-view" ]
64         [ navigation
65           , header
66           , services
67           , portfolio
68           , about
69           , team
70           , clients
71           , contacts
72           , footer
73           , popups
74         ]

```

Abbildung 3.11: Aufruf der einzelnen Sektionen im View

3.4.7 Hinzufügen von Klick-Events

Der derzeitige Stand der überführten Applikation stellt bereits sämtliche Sektionen auf der Webseite dar und erlaubt das Öffnen eines Modals¹³ durch einen Klick auf eines der Portfolio-Elemente. Bei einem Klick auf ein Element in der Navigationsleiste springt der Viewport des Browsers jedoch ohne jegliche Verzögerung direkt auf die ausgewählte Sektion. Das ursprüngliche Template hingegen nutzt die Funktionalität *smooth – page – scrolling*, wodurch der Nutzer einen sichtbaren Scroll-Effekt erhält, während automatisch zur ausgewählten Sektion gescrollt wird. Innerhalb der Datei *agency.js*, in der vorab bereits die Funktion des *Scroll – Spys* wiederhergestellt wurde, wird in den Zeilen 9 bis 15 die Funktionalität des automatischen scrollens bei einem Klick umgesetzt, wie in Abbildung 3.12 zu sehen ist. Dabei wird zunächst ein *Ereignisbehandler* an die Link-Elemente mit der Klasse *page – scroll* gebunden. Klickt der Nutzer

¹³Ein Modal ist Fenster im Browser, welches sich über dem gesamten Hauptfenster öffnet. Es verhindert eine Interaktion mit dem Hauptfenster.

```

9      $('a.page-scroll').bind('click', function(event) {
10          var $anchor = $(this);
11          $('html, body').stop().animate({
12              scrollTop: $($anchor.attr('href')).offset().top
13          }, 1500, 'easeInOutExpo');
14          event.preventDefault();
15      });

```

Abbildung 3.12: Funktion zum scrollen bei einem Klick

eines dieser Elemente, so wird die darauf folgende Funktion ausgeführt. Diese Funktion wiederum liest Informationen des angeklickten Elementes aus, insbesondere das *href*-Attribut, welches die ID zur angeklickten Sektion enthält. Mittels der *animate*-Funktion wird letztlich zur gewünschten Sektion gescrollt. Diese Funktionalität ist nicht verfügbar, da das externe JS versucht auf HTML-Elemente innerhalb der Elm-Applikation zuzugreifen. Um dieses Problem zu lösen, kann einerseits die Funktion innerhalb der Datei *agency.js* umgeschrieben werden, so dass sämtliche Klicks innerhalb des *body*-Elementes abgefangen werden. Andererseits kann der Klick innerhalb der Elm-Applikation registriert und über sogenannte Ports nach außen getragen werden. In diesem Beispiel ist das die bevorzugte Methode, da so eine klare Trennung zwischen der Elm-Applikation und den externen JS-Skripten herrscht. Die Elm-Bibliothek *Html.Events* enthält die Methode *onWithOptions*, mit Hilfe derer ein Klick auf ein Element eine *Msg* erzeugt. Diese *Msg* wird an die *update*-Funktion weitergeleitet und kann dort entsprechend ausgewertet werden. Bislang gibt es in

```

13      type Msg
14          = ClickedElem String
15
16      update msg model =
17          case msg of
18              ClickedElem elem ->
19                  ( model, sendClickedElemToJs elem )
20
21      port sendClickedElemToJs : String -> Cmd msg

```

Abbildung 3.13: Definition eines neuen Union Types, sowie eines Ports für die Kommunikation zwischen Elm und JS

der Elm-Applikation noch keine *Msg*-Typen, welche die *update*-Funktion verarbeiten könnte. Dementsprechend wird zunächst ein neuer Union-Type *Msg* mit dem Eintrag *ClickedElem String* angelegt. Der Union-Type muss nun noch in der *update*-Funktion behandelt werden. Hierbei wird ein neuer *case* angelegt, mit dem zuvor erstellten Eintrag *ClickedElem* und dem Übergabeparameter *elem* vom Typ *String*. Der vorherige Eintrag "*_* — >", welcher für alle

Fälle gilt, kann entfernt werden. Der Rückgabewert des *ClickedElem*-Falles ist dabei ein Tupel, bestehend aus dem unveränderten *model*, sowie einem Effekt. In diesem Fall ist der Effekt das Senden der ausgewählten Elemente nach außen zu einem JS-Skript, das die Daten verarbeitet. Der Effekt wird dabei über den Port *sendClickedElemToJs* nach außen geschickt. Der Port muss definiert werden mit dem Kommando in Zeile 21 aus der Abbildung 3.13. Das Kommando ist lediglich die Signatur der gewünschten Kommunikationsweise und wird durch den Elm-Compiler während des Kompiliervorgangs entsprechend verarbeitet. Beim Versuch den Code wie er in Abbildung 3.13 zu sehen ist zu kompilieren, wird der Compiler mit der Fehlermeldung „Sie deklarieren den Port *sendClickedElemToJs* in einem normalen Modul“¹⁴ antworten. Die Fehlermeldung bedeutet, dass das Modul in dem ein Port definiert wird explizit als ein Port-Modul deklariert werden muss. Die erste Zeile des Hauptmoduls muss folglich noch mit dem Schlüsselwort *port* versehen werden. Damit die Elm-Applikation die Funktion *ClickElem* aufruft, bedarf es noch des Aufrufes der zuvor erwähnten *onWithOptions*-Methode. Diese wird in den *view* an den entsprechenden Stellen eingesetzt. Die Links der Navigationsleiste befinden sich in der Funktion *navigation* und müssen um die *onWithOptions*-Funktion, sowie den Übergabeparametern *ClickedElem* und dem *href*-Wert erweitert werden. In Abbildung 3.14 kann beispielhaft die Erweiterung an einigen Stellen begutachtet werden. Die Funktion *onWithOptions* bekommt dabei zunächst einmal das *Event*, auf das sie einen Ereignisbehandler setzen soll. In diesem Fall ist dies ein *click*. Darauf folgt ein optionaler *Record* von Typ *Options*. Hier müssen die Werte *stopPropagation* und *preventDefault* auf *True* gesetzt werden. Fehlt diese Einstellung, so wird das scrollen sofort ausgeführt und kann möglicherweise andere Ereignisbehandler in denen der Link geschachtelt ist ausführen. Dies würde zu einem undefinierten Verhalten führen. Der letzte Parameter ist die Decoder-Funktion *Json.succeed*. Sie führt dazu, dass die übergebene Funktion *ClickedElem* direkt und ohne weitere Überprüfungen ausgeführt wird. Seitens der Elm-Applikation ist die notwendige Erweite-

```
a
[ class "page-scroll"
, href "#team"
, onWithOptions "click"
  { stopPropagation = True
  , preventDefault = True
  }
  (Json.succeed (ClickedElem "#team"))
]
```

Abbildung 3.14: Erweiterung des Views um einen *onWithOptions* Ereignisbehandler in Elm

¹⁴ „You are declaring port *sendClickedElemToJs* in a normal module“- Eigene Übersetzung.

rung nun abgeschlossen. Jedoch muss noch die *index.html* um eine sogenannte *subscription* erweitert werden. Das bedeutet, dass innerhalb der *index*-Datei die Nachricht der Elm-Applikation des angeklickten Elements entgegengenommen und verarbeitet werden muss. Diesbezüglich wird in der *index.html* nach der Initialisierung der Elm-Applikation die Funktion aus Abbildung 3.15 hinzugefügt. Durch das Kommando *app.ports.sendClickedElemToJs.subscribe*

```
app.ports.sendClickedElemToJs.subscribe(function(elem){
    $('html, body').stop().animate({
        scrollTop: $(elem).offset().top
    }, 2000);
});
```

Abbildung 3.15: Entgegennahme der Daten aus der Elm-Applikation und anschließendes scrollen

werden sämtliche Daten die Elm über den Port *sendClickedElemToJs* sendet entgegengenommen und die nachfolgende Funktion ausgeführt. Diese Funktion wurde in diesem Beispiel lediglich aus der Datei *agency.js* kopiert. Sie ermittelt das HTML-Element im Baum anhand der übergebenen ID und scrollt innerhalb von zwei Sekunden zur obersten Stelle des übergebenen Elements.

3.4.8 Modularisierung der Applikation

Durch das Auslagern aller einzelnen Sektionen der darzustellenden HTML-Elemente ist die daraus resultierende *Main.elm*-Datei recht unübersichtlich. Abhängigkeiten zwischen einzelnen Funktionen sind kaum mehr erkennbar und die Programmlogik ist nicht unterscheidbar von Funktionen, die für die Darstellung des *Views* zuständig sind. Folglich ist es sinnvoll einzelne Teile der Applikation in mehrere Dateien und Ordner zu unterteilen. Dabei werden die einzelnen Programmteile des *Views* ausgelagert in eigene Module. Dasselbe Prinzip wird für das *Model* und die *update*-Funktion angewandt. Im Gegenzug sollen die ausgelagerten Programmteile global im Hauptmodul importiert und aufgerufen werden.

View

Jede Funktion die bisher eine Sektion des *Views* erzeugt hat, wird in eine neu angelegte *.elm*-Datei verschoben und die Funktion umbenannt zu *view*. Die Datei bekommt dabei den Namen, den die Funktion vorher trug. Ferner wird ein Ordner *View* erzeugt, in den all diese Dateien hin verschoben werden. Damit die neuen Dateien als Module eingebunden werden können, müssen sie als solches definiert werden. Das bedeutet, dass jede Datei mit dem Kommando *module View.NameDerDatei exposing (view)* initialisiert werden muss. Der Zusatz *view* nach dem Kommando *exposing* gibt an, dass die Funktion *view* nach außen hin sichtbar ist und vom importierenden Modul genutzt werden

```

15  import View.Team as Team
16  import View.Navigation as Navigation
17  import View.Header as Header
18  import View.Services as Services
19  import View.Portfolio as Portfolio
20  import View.About as About
21  import View.Clients as Clients
22  import View.Contacts as Contacts
23  import View.Footer as Footer
24  import View.Popups as Popups
25  import Model.Model as Model
26  import Update.Update as Update

```

Abbildung 3.16: Einbindung und Aufruf der ausgelagerten View-Funktionen

kann. Nachdem alle Funktionen in ein eigenes Modul ausgelagert wurden, müssen alle Module im Hauptmodul *Main.elm* importiert werden. Des Weiteren müssen die *view*-Funktionen der einzelnen Module im Hauptmodul aufgerufen werden. Die Abbildungen 3.17 und 3.16 zeigen die *view*-Funktion des Hauptmoduls, sowie den Aufruf um die angelegten Module zu importieren. Letzten

```

47  view : Model.Model -> Html Update.Msg
48  view model =
49      div [ id "elm-view" ]
50          [ Navigation.view model
51            , Header.view
52            , Services.view
53            , Portfolio.view
54            , About.view
55            , Team.view
56            , Clients.view
57            , Contacts.view
58            , Footer.view
59            , Popups.view model
60          ]

```

Abbildung 3.17: Einbindung und Aufruf der ausgelagerten View-Funktionen

Endes müssen die einzelnen Module noch mit den notwendigen *imports* für die verwendeten Funktionen erweitert werden. Das Hauptmodul wiederum kann von den nicht benutzten *Html*- beziehungsweise *Html.Attributes*-Funktionen gesäubert werden.

Update

Nicht nur die für die Darstellung verantwortlichen Funktionen sollten modularisiert werden, sondern auch die Programmlogik. Hierfür wird ein weiterer Ordner *Update* erzeugt. Die bisherige *update*-Funktion wird demzufolge analog

zu den *View*-Funktionen ausgelagert in ein eigenes Modul im neu erzeugten Ordner *Update*. Das *Update*-Modul folgt derselben Namensgebung wie ein einzelner *View* und wird definiert als *Update.Update*. Zusätzlich müssen auch an dieser Stelle die *imports* der benutzten Pakete übernommen und definiert werden, wobei das Hauptmodul diese entfernen kann.

Model

Letztlich wird noch das *model*, das sämtliche Daten die den Status der Applikation beschreiben enthält, in ein eigenes Modul im Unterordner *Model* überführt. Die Einbindung dieses Moduls funktioniert analog zur Modularisierung von *Update* und *View*. Mit Hilfe dieser Modularisierung wird das angestrebte MVU-Konzept von Elm besonders deutlich.

3.4.9 Asynchrones Laden von Daten

Klickt man innerhalb der Portfolio-Sektion der Webseite auf ein Element, öffnet sich ein Modal in dem weitere Informationen dargestellt werden. Die bestehende Elm-Applikation wird nun um das Feature des asynchronen Ladens von Informationen erweitert, die anschließend in dem geöffneten Modal angezeigt werden. Beispielhaft wird hier die API von The Internet Chuck Norris Database (ICNDb)¹⁵ genutzt. Sie liefert bei jeder Anfrage einen zufälligen String zurück. Für diese Funktionalität muss zunächst das *model* erweitert und angepasst werden, da dies die einzige Möglichkeit in einer Elm-Applikation ist, Daten beziehungsweise den Status der Applikation zu speichern. Das *model* bekommt entsprechend ein weiteres Feld *async_content* vom Typ *String*. Bei einem Klick auf einen der Portfoliobeiträge soll ein Modal geöffnet und der zufällige String dargestellt werden. Dafür wird ein HTTP-Request an die API erzeugt und die Antwort entgegengenommen. Ebenso wäre es möglich einen Server für das Backend zu erstellen, auf dem eine Datenbank läuft, so dass Daten asynchron angefordert werden können. Das asynchrone Anfordern von Daten basiert sowohl bei einem lokalen, wie auch externen Server auf dem gleichen Konzept, weswegen dieser Schritt entfällt und der vorhandene externe Service genutzt wird. Ein solcher asynchroner Request stellt im Grunde eine Verletzung des Konzeptes von Elm dar, dass es keinerlei Seiteneffekte gibt. Da nicht bekannt ist, wann der Request endet und welchen Status die Antwort besitzt, ist zunächst nicht vorhersehbar, wie der Status der Applikation nach dem Request aussehen wird. Um dieses Problem zu vermeiden, ist es notwendig alle möglichen Fälle, sprich den Fall einer erfolgreichen, sowie fehlerhaften Übertragung, zu behandeln. Auf diese Weise ist gewährleistet, dass die Applikation sich nicht plötzlich in einem undefinierten Zustand befindet. Es bedarf zusätzlich der Bibliotheken *Http*, *Json.Decode* und *Task*, damit ein asynchroner Request ausgeführt werden kann. Die Bibliotheken müssen installiert und in die jeweiligen Module importiert werden. Des Weiteren gilt es den Klick auf den Portfoliobeitrag abzufangen, um den Request zu starten

¹⁵Abrufbar unter <https://api.icndb.com>.

und das Ergebnis im View anzuzeigen. Hierfür gibt es die *onClick* Funktion aus der *Html.Events*-Bibliothek. Sie bekommt eine auszuführende Funktion aus dem *Update*-Modul als Parameter, in diesem Fall *GetRandomString*, wie es in Abbildung 3.18 zu sehen ist. Bei einem Klick auf das Element wird eine *Msg* von diesem Typ erstellt und muss in der *update*-Funktion behandelt werden. Wie zuvor bei der Definition einer *Msg* um Daten durch Ports

```
a
[ class "portfolio-link"
, attribute "data-toggle" "modal"
, href "#portfolioModal1"
, onClick Update.GetRandomString
]
```

Abbildung 3.18: OnClick-Ereignisbehandler in Elm

nach außen zu kommunizieren, muss auch die Nachricht *GetRandomString* als *Msg*-Typ deklariert werden, um so in der *update*-Funktion behandelt werden zu können. Außerdem müssen die möglichen Statusfälle des Requests behandelt und als *Msg* hinzugefügt werden. Abbildung 3.19 zeigt die hinzugefügten *Msg*-Typen. Nachdem die neuen Interaktionsmöglichkeiten in der Elm-

```
type Msg
    = ClickedElem String
    | GetRandomString
    | FetchFail Http.Error
    | FetchSucceed String
```

Abbildung 3.19: *Msg*-Deklaration mit asynchronem Request

Applikation implementiert wurden, muss nun der eigentliche Request ausführbar gemacht werden. Die Funktion *GetRandomString* soll dabei ein unverändertes Model, sowie eine asynchron auszuführende Funktion in Form eines Effektes als Tupel zurückliefern. Dabei ist die asynchrone Funktion der eigentliche HTTP-Request, wie er in Abbildung 3.20 erkennbar ist. Dabei wird die Funktion *Http.get* genutzt, um einen Request an die *url* auszuführen. Die Antwort des Requests wird an die Funktion *decodeAnswer* gereicht, welche das eingehende JavaScript Object Notation (JSON)-Objekt dekodiert und die gewünschten Informationen herausfiltert. Die Dekodierfunktion kann in Abbildung 3.21 betrachtet werden. Sie filtert das geschachtelte JSON-Objekt, so dass nur der gewünschte String übrig bleibt. Abhängig vom Status des Requests, wird der jeweilige Fall in der *update*-Funktion aufgerufen. Bei einer erfolgreichen Übertragung wird die Funktion *FetchSucceed* ein neues Model mit dem

```

fetchAsync =
  let
    url =
      "http://api.icndb.com"
      ++ "/jokes/random"
      ++ "?firstName=John"
      ++ "&lastName=Doe"
  in
    Task.perform
      FetchFail
      FetchSucceed
      (Http.get decodeAnswer url)

```

Abbildung 3.20: Asynchroner HTTP-Request in Elm

```

decodeAnswer =
  Json.at
    [ "value"
      , "joke" ]
  Json.string

```

Abbildung 3.21: Dekodierung eines Json-Objektes in Elm

asynchron abgerufenen String zurückliefern. Der String wird dabei in das Feld *async_content* des *model* geschrieben. Bei einer fehlgeschlagenen Übertragung wird die Funktion *FetchFail* das unveränderte *model* zurückgeben. Alternativ kann auch eine mögliche Fehlermeldung in das *model* eingearbeitet und im *view* dargestellt werden. Die fertige *update*-Funktion mit allen abgehandelten

```

GetRandomString ->
  ( model, fetchAsync )
FetchFail err ->
  ( model, Cmd.none )
FetchSucceed newStr ->
  ( { model | async_content = newStr }, Cmd.none )

```

Abbildung 3.22: *Update*-Fälle für den asynchronen Request

Fällen ist in Abbildung 3.22 dargestellt. Der asynchrone Request wird dabei durch *GetRandomString* initiiert. Die Manipulation des *model* findet lediglich in der *FetchSucceed*-Funktion statt. Nachdem die Elm-Applikation an dieser Stelle fehlerfrei kompiliert wurde, kann der asynchrone Request auf der Webseite getestet werden. Mit Hilfe der Developer-Tools kann die ausgehende Anfrage an den ICNDb-Server beobachtet und ausgewertet werden. Auch die Antwort des Servers kann unbehandelt in Abbildung 3.23 begutachtet werden. Die Abbildung zeigt ferner wie der zurückgelieferte Text des Requests fehlerfrei dekodiert und in das *model* übertragen, sowie letztlich in der *view*-Funktion



Abbildung 3.23: Ausgehender HTTP-Request und eingehende Antwort

des *Popup*-Moduls mittels des Kommandos `text model.async_content` angezeigt wird.

3.4.10 Beobachtungen

In diesem Abschnitt sollen Auffälligkeiten, die während der Überführung des *agency*-Templates in eine native Elm-Applikation auftraten, aufgezeigt und erläutert werden. Es handelt sich hierbei nicht nur um Probleme, sondern auch um interessante Erkenntnisse im Hinblick auf die Unterschiede zwischen herkömmlicher Webentwicklung mit HTML, CSS und JS gegenüber Elm, die es zu erwähnen gilt.

Natives Einbinden von CSS in Elm nur beschränkt möglich

Herkömmlicherweise können Elemente einer HTML-Datei auf drei Arten gestylt werden.

1. Inline CSS:

Das Inline CSS erlaubt Design-Anpassungen für genau ein bestimmtes Element. Der verfasste CSS-Code kann nicht von anderen Elementen genutzt werden und wird über das HTML-Attribut *style* eingeführt.

2. Internes CSS:

HTML enthält den *style*-Tag. In diesem kann natives CSS verfasst, bestimmte Klassen oder Id's erstellt und sämtliche gängigen Design-Anpassungen durchgeführt werden.

3. Externes CSS:

Die Erweiterung des internen CSS ist das externe CSS, bei dem der gesamte CSS-Code ausgelagert wird in eine eigene *.css*-Datei. Diese wird wiederum mit Hilfe des *link*-Tag der HTML-Datei eingebunden.

In Elm ist es nicht möglich nativen CSS-Code zu verfassen. Jedoch kann mittels der Bibliothek *elm – lang/html* inline CSS erzeugt werden. Das Design wird dabei nativ in Elm über die *Html.style*-Funktion programmiert. Ferner ist das erstellen von *style*-Tags in Elm nicht möglich, wodurch auch die Möglichkeit für internes CSS entfällt. Es ist zwar praktisch möglich externe CSS-Dateien in die Elm-Applikation nativ in Elm einzubinden, jedoch stört das immens die Nutzbarkeit der Applikation. Die Elm-Applikation wird zunächst ohne das CSS geladen, während die externe Datei nachgeladen wird. Das bedeutet für den Nutzer der Webseite, dass zunächst nur Texte und Bilder ohne Styling angezeigt werden. Nachdem die externe Datei vollständig übertragen wurde, wird das darin enthaltene Design auf die HTML-Elemente angewandt. Der zeitliche Abstand zwischen initialem Laden und der Anwendung des Stylesheets hat ein sichtbares *flackern* zur Folge, wodurch eine Nutzung in einem fertigen System entfällt. Dies ist der Grund, weswegen bereits bestehende CSS-Dateien des Templates nicht nativ in Elm überführt, sondern über das HTML-Grundgerüst eingebunden wurden. Zusätzlich zu diesem Umstand wird inline- und internes CSS nicht vom Browser zwischengespeichert. Das hat eine längere Ladezeit zur Folge und sollte vermieden werden. Des Weiteren ist es nur bedingt möglich CSS-Code in Elm an mehreren Stellen zu verwenden, wie es üblicherweise mit in CSS definierten Klassen der Fall ist.

HTML-Code ist in Elm wesentlich kürzer

HTML-Tags mit den gleichnamigen Funktionen nativ in Elm zu erstellen ist einerseits signifikant kürzer, andererseits viel übersichtlicher als das Verfassen von herkömmlichen HTML-Code. Die Abbildung 3.24 zeigt ein *div*-Element,

HTML: <div> </div>
 Elm: div [[]]

Abbildung 3.24: Ein *div*-Element in HTML und Elm

wie es einerseits in klassischem HTML, andererseits in Elm realisiert wird. In Elm handelt es sich dabei um einen Funktionsaufruf mit zwei Parametern, in diesem Fall zwei leeren Listen. Ein HTML-Element hat dabei eine Anzahl von $5 + 2 * n_0$ Zeichen, wobei n_0 gleich der Zeichenanzahl des Tag-Namens und 5 die Zeichen für das öffnende, sowie schließende Tag sind. Am Beispiel des *div*-Elementes wäre $n_0 := 3$, wodurch die Implementierung in HTML $5 + 2 * 3$, sprich 11 Zeichen benötigt. In Elm hingegen benötigt die Erstellung eines HTML-Elementes $4 + n_1$ Zeichen, wobei n_1 gleich der Zeichenanzahl des Tags, sprich hier der Funktion *div*, und 4 die Zeichenanzahl für die Parameterliste ist. Am selben Beispiel wären das $4 + 3$, sprich 7 Zeichen. Die Konstanten 5 (HTML) und 4 (Elm) können vernachlässigt werden, wodurch sich die dynamischen Werte $2 * n_0$ und n_1 gegenüberstellen. Da alle HTML-Tags in Elm mit einer gleichnamigen Funktion erstellt werden, kann n_0 gleich n_1 gestellt werden. Daraus folgt, dass Elm im Vergleich zu HTML 50% weniger Zeichen für die Erstellung eines HTML-Elementes benötigt. Die Differenz der Zeichen

lässt sich auf das schließen der Tags in HTML zurückführen. Nativer Elm-Code arbeitet mit Einrückungen, wohingegen klassischer HTML-Code über die öffnenden und schließenden Tags geschachtelt wird. Dadurch entsteht bei HTML-Code der Vorteil, dass ein gesamtes Dokument theoretisch in nur einer Zeile ohne Absätze definiert werden kann, während in Elm die Einrückungen und die damit verbundenen Zeilenumbrüche notwendig sind. Daraus folgt, dass Elm-Code zwangsweise übersichtlicher bleibt, da die Einrückungen nicht umgangen werden können und dem Entwickler eine visuelle Rückmeldung liefern. Der Elm-Compiler wird den Elm-Code nicht in JS-Code umwandeln, sollten die Abstände inkorrekt sein. HTML-Code bietet die Möglichkeit gleichermaßen übersichtlich zu programmieren, jedoch gibt es hier keinen Compiler oder sonstige Warnhinweise für den Fall, dass geschriebener Code unübersichtlich wird. Durch die Einrückung und das damit verbundene Fehlen von schließenden Tags kam es zu weniger Flüchtigkeitsfehlern in Form von falschen Verschachtelungen oder dem simplen Fehlen der schließenden Tags bei einer tiefen oder über mehrere Absätze stattfindende Schachtelung.

Elm hat eine stark wachsende Open-Source Gemeinschaft

Trotzdem Elm erst im März 2012 entwickelt wurde, gibt es bereits eine Vielzahl an Gemeinschaften, um über die Sprache zu diskutieren, Hilfe zu erbitten oder gemeinsam an Projekten zu arbeiten. Neben einem eigenen Slack-Chatroom mit derzeit 2.835 angemeldeten Nutzern [16, Vgl.], gibt es noch die offizielle Google-Gruppe¹⁶. Offiziell gibt es derzeit 205 Elm-Bibliotheken [6, Vgl.], von denen jede einzelne ein Open-Source-Projekt darstellt und für alle Entwickler freigegeben ist. Auf der Webseite <https://github.com/> gibt es zur Zeit 2.845 öffentliche Projekte [12, Vgl.] die komplett oder zumindest teilweise in Elm realisiert wurden. Ferner wurden 6.365 Probleme in diesen Projekten gemeldet, von denen 4.834 bereits gelöst und 1.531 noch geöffnet sind [11, Vgl.]. Alles in Allem wird deutlich, dass Elm eine wachsende Anhängerschaft besitzt und die Probleme der Webentwicklung auf eine andere Weise angeht. Wann immer Fragen auftraten konnten die Nutzer in einer der genannten Medien meist in wenigen Minuten aushelfen. Des Weiteren ist Elm anwendbar auf allen Betriebssystemen, die den NPM unterstützen. Zusätzlich gibt es die öffentlichen Tools *elm – repl*, *elm – make* und *elm – package*, mit denen eine Elm-Applikation umgesetzt werden kann. Darüber hinaus gibt es Quellen wie das offizielle Handbuch für Elm¹⁷, oder das praktische Beispiel „Elm Tutorial“ [21]¹⁸ von Sebastian Porto, die mit Hilfe von Schritt-für-Schritt Anleitungen an die Programmiersprache heranführen.

3.4.11 Auswertung

Nachdem die SPA vollständig überführt und einige Beobachtungen erläutert wurden, steht die Auswertung der einzelnen Bewertungskriterien aus. Dieser

¹⁶Abrufbar unter <https://goo.gl/wzJup3>.

¹⁷Abrufbar unter <http://guide.elm-lang.org/>.

¹⁸Abrufbar unter <http://www.elm-tutorial.org/en>.

Abschnitt erläutert dabei in welchem Maße die Kriterien erfüllt wurden und inwiefern die Erkenntnisse verallgemeinerbar sind und auf andere Webapplikationen zutreffen.

Kriterium	Erfüllt
3.2.1 Wartbarkeit und Lesbarkeit	✓
3.2.2 Zuverlässigkeit	✓
3.2.3 Portabilität	✓
3.2.4 Effizienz	✓
3.2.5 Wiederverwendbarkeit	✓
Webspezifische Kriterien:	
3.3.1 Browser Kompatibilität	✓
3.3.2 Interoperabilität	teilweise erfüllt
3.3.3 Asynchrone Verarbeitung	✓
3.3.4 Dateigröße	teilweise erfüllt

Abbildung 3.25: Auswertung der Versuchskriterien

Auswertung: 3.2.1 Wartbarkeit und Lesbarkeit

Die Programmiersprache Elm lässt die Erstellung von manuellen Kommentaren zu. Die Kommentare können sich dabei über mehrere Zeilen erstrecken, oder lediglich eine einzige Zeile einnehmen. Für die jeweilige Distanz eines Kommentar gibt es gesonderte Kommandos. Das Atom-Plugin *elm – format* fügt außerdem Zeilenabstände vor, sowie hinter dem Kommentar ein, so dass es übersichtlich platziert wird. Die Grundlagen dieses Kriteriums wurden somit an dieser Stelle erfüllt. Das fortgeschrittene Kriterium der automatischen Erstellung von Kommentaren mit Informationen über beispielsweise die erwarteten Typen der Übergabeparameter einer Funktion sind nur bedingt erfüllt. Die Möglichkeit *Signatures* innerhalb der Elm-Applikation hinzuzufügen besteht und wird durch den *elm – compiler* unterstützt, allerdings nicht komplett automatisiert. Der Compiler gibt einen Signaturvorschlag in Form einer Warnung ab, sobald die Applikation kompiliert wird, wie es in Abbildung 3.26 demonstriert wird. Dem Signaturvorschlag kann unter der Prämisse

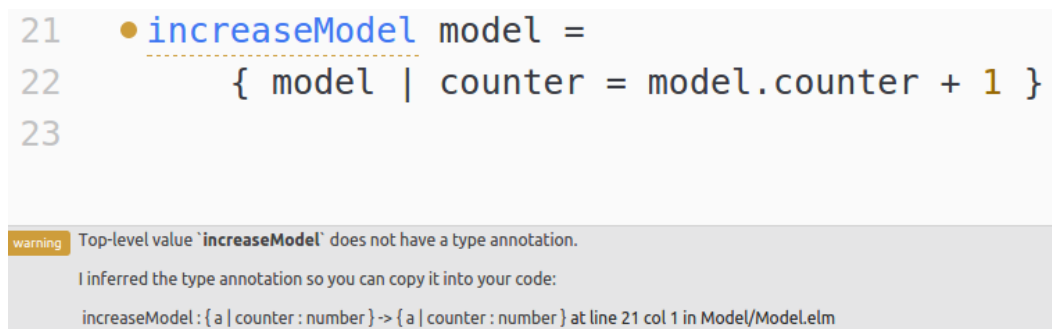


Abbildung 3.26: Automatisierter Signaturvorschlag durch den *elm – compiler*

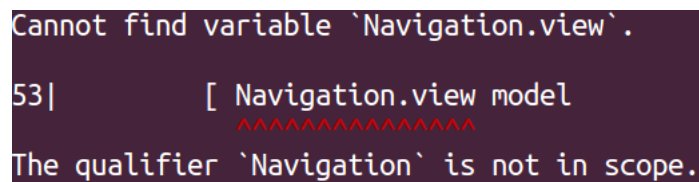
übernommen werden, dass sämtliche zuvor definierten Typen korrekt sind, da der *elm – compiler* die Vorschläge auf der vorherigen Analyse des gesamten Dokumentes stützt.

Auswertung: 3.2.2 Zuverlässigkeit

Um die Zuverlässigkeit des *Elm – Compiler* und die Fehlermeldungen bewerten zu können, werden syntaktische und semantische Fehler innerhalb der Applikation eingebaut. Daraufhin wird eine erwartete Fehlermeldung der tatsächlichen Fehlermeldung gegenübergestellt und die Qualität bewertet.

1. Entfernen eines importierten Moduls

Die Fehlermeldung des 1. Versuches sagt aus, dass die Variable *Navigation.view*



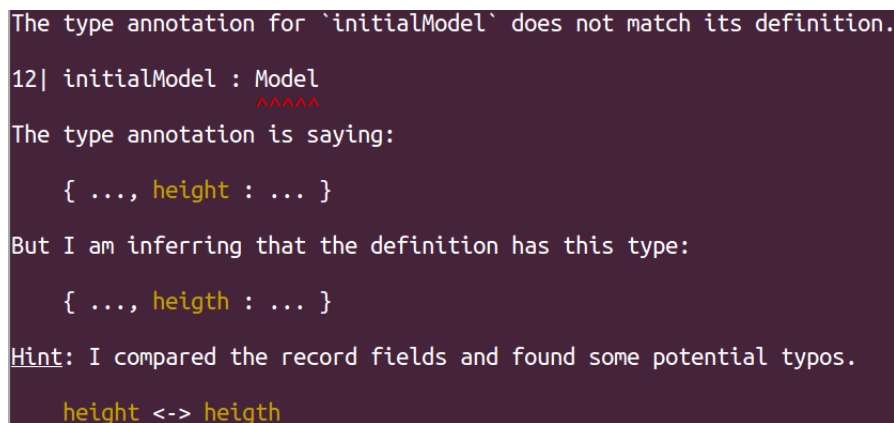
```
Cannot find variable `Navigation.view`.
53| [ Navigation.view model
    ~~~~~
The qualifier `Navigation` is not in scope.
```

Abbildung 3.27: Fehlermeldung eines nicht vorhandenen importierten Moduls

nicht gefunden wurde und das Modul *Navigation* nicht im aktuellen Namensraum vertreten ist. Ein Entwickler kann daraus schließen, dass das Modul nicht ordnungsgemäß eingebunden wurde, oder im Namensraum anders benannt wurde. In beiden Fällen wird der Entwickler die importierende Programmstelle aufsuchen, die sich am Beginn der Datei befindet. Die Fehlermeldung gibt einen hilfreichen Aufschluss über das eigentliche Problem.

2. Falsche Benennung eines Feldes

Im 2. Versuch wird durch die Fehlermeldung vermittelt, dass das *Model*



```
The type annotation for `initialModel` does not match its definition.
12| initialModel : Model
    ~~~~~
The type annotation is saying:
  { ..., height : ... }
But I am inferring that the definition has this type:
  { ..., heigth : ... }
Hint: I compared the record fields and found some potential typos.
    height <-> heigth
```

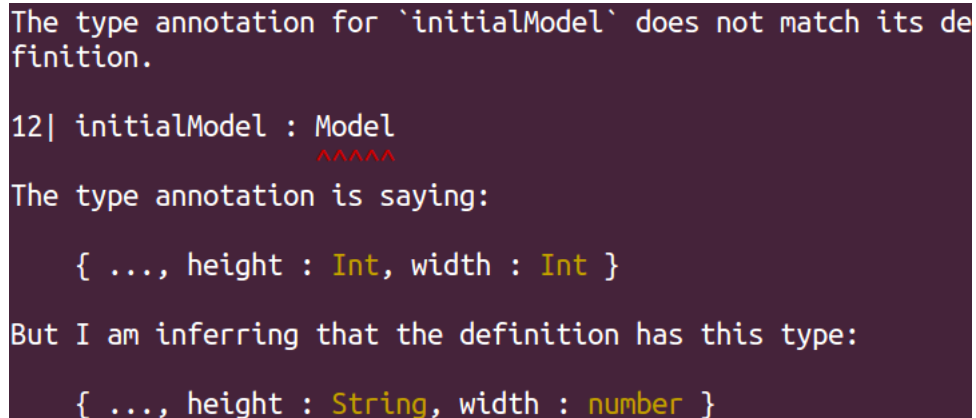
Abbildung 3.28: Fehlermeldung eines nicht vorhandenen Attributes

per Definition ein Feld mit dem Namen *height* hat. Im weiteren Verlauf der Applikation hingegen wird ebenfalls auf ein *Model* zugegriffen, diesmal jedoch auf das Feld mit dem Namen *heigth*. Es handelt sich offensichtlich um

einen Schreibfehler. Bei der herkömmlichen Entwicklung mit JS ist es möglich, dynamisch Felder in einem Objekt hinzuzufügen, wodurch selbst bei einer expliziten Überprüfung des Quellcodes der Fehler erst während der Laufzeit auftreten würde. Die Fehlermeldung weist hier explizit auf das Problem hin und veranschaulicht es.

3. Falsche Typzuweisung eines Feldes

In der 3. Fehlermeldung wird durch den Compiler erkannt, dass ein Typenfeh-



```
The type annotation for `initialModel` does not match its definition.

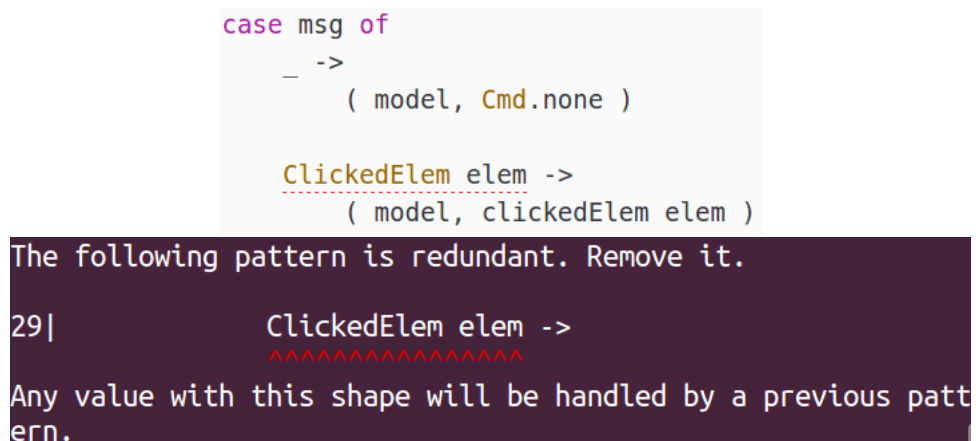
12| initialModel : Model
    ^^^^^
The type annotation is saying:
    { ..., height : Int, width : Int }
But I am inferring that the definition has this type:
    { ..., height : String, width : number }
```

Abbildung 3.29: Fehlermeldung einer falschen Typzuweisung

ler aufgetreten ist. Es ist versucht worden das Feld *height* mit dem Wert "0" zu initialisieren. Laut Definition ist das Feld allerdings von Typ *Int*, wohingegen versucht wird einen *String* zuzuordnen. Da Elm statisch typisiert ist, können Variablen nur einen Datentyp annehmen. JS liefert die Möglichkeit eine Variable dynamisch zu typisieren. Der Entwickler wird durch die Fehlermeldung explizit auf den entstandenen Fehler hingewiesen.

4. Einbau redundanter Kontrollen

Für die Demonstration der 4. Fehlermeldung wird einer Kontrollstruktur ein



```
case msg of
  - ->
    ( model, Cmd.none )

  ClickedElem elem ->
    ( model, clickedElem elem )

The following pattern is redundant. Remove it.

29| ClickedElem elem ->
    ^^^^^^^^^^^^^^^^^
Any value with this shape will be handled by a previous pattern.
```

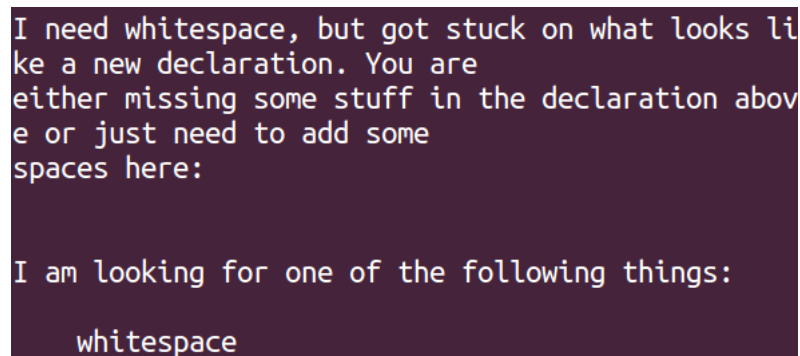
Abbildung 3.30: Eine redundante Kontrollstruktur in Elm

allumfassender Fall hinzugefügt. Der erste Fall der Kontrollstruktur in der

Abbildung 3.30 ist für jeden zu überprüfenden Fall gültig. Dementsprechend sind alle nachfolgenden Fälle überflüssig, da sie nie erreicht werden können. Der allumfassende Fall kann mit einem *if(true)*-Statement verglichen werden. Jeglicher *else*-Fall in einem solchen Konstrukt wird unmöglich eintreffen. Der *elm – compiler* erkennt diese Redundanz und gibt an, dass die Fälle bereits behandelt werden und entfernt werden sollten. Dadurch wird zusätzlich die Lesbarkeit des Quellcodes gewahrt, da der Entwickler aufgefordert wird unnötige Codeteile zu entfernen. Erneut gibt die Fehlermeldung ganz klar an, welche Schritte notwendig sind um den Fehler zu beheben.

5. Entfernen einer schließenden Klammer

Der 5. Versuch zeigt die Fehlermeldung nachdem eine schließende Klammer



```
I need whitespace, but got stuck on what looks like a new declaration. You are
either missing some stuff in the declaration above or just need to add some
spaces here:

I am looking for one of the following things:

whitespace
```

Abbildung 3.31: Fehlermeldung einer fehlenden Klammer

eines Statements entfernt wurde. Die daraus resultierende Fehlermeldung gibt an, dass entweder Teile innerhalb der Deklaration fehlen, oder Leerzeichen hinzugefügt werden müssen, um die passende Einrückung zu erreichen. Offensichtlich *fehlt* etwas in der Deklaration, die Fehlermeldung sollte jedoch wesentlich genauer ausfallen und auf die fehlende Klammer hindeuten. Der Entwickler muss nun die dazugehörige Stelle des Quellcodes absuchen, ohne genau zu wissen, welche möglichen Zeichen fehlen. Diese Fehlermeldung ist nicht ausreichend explizit, um den Fehler ohne Probleme ausfindig zu machen.

6. Mehrfache Definition einer Funktion

Die 6. Fehlermeldung bezieht sich auf die mehrmalige Deklaration einer Funktion. Der *elm – compiler* erkennt, dass mehrere Funktionen mit demselben Namen existieren und sich im gleichen Namensraum befinden. Der Entwickler wird durch die Fehlermeldung dazu aufgefordert, die Funktionen einzigartig zu benennen. Durch die Fehlermeldung erhält der Entwickler eine klare Problemlösung präsentiert.

Die erzeugten Fehler decken offensichtlich nicht alle möglichen Fehlerfälle ab, geben jedoch recht gut Aufschluss über das Verhalten des *elm – compiler* hinsichtlich gängiger Fehler. Zusammenfassend lässt sich sagen, dass die Fehlermeldungen des *elm – compiler* zuverlässig Fehler finden, selbst wenn sie über die syntaktische Ebene hinausgehen und den Entwickler über unnötige Kontrollstrukturen informieren. Ferner wird die Qualität der Fehlermeldungen

```

Naming multiple top-level values `view` makes things ambiguous. When you say
`view` which one do you want?

51| view model =
   view
Find all the top-level values named `view` and do some renaming. Make sure the
names are distinct!

```

Abbildung 3.32: Fehlermeldung einer mehrfach definierten Funktion

durch das Markieren der tatsächlichen Fehlerposition und dem Unterstreichen der Fehlerquelle verbessert. Da alle eingebauten Fehler gefunden und die dazugehörigen Fehlermeldungen im Durchschnitt sehr akkurat auf den Fehler hingewiesen haben, kann das Kriterium der Zuverlässigkeit als erfüllt angesehen werden.

Auswertung: 3.2.3 Portabilität

Die offizielle Webseite von Elm bietet Installationsdateien und -anleitungen für die gängigen Betriebssysteme *Mac* und *Windows*, sowie für alle Plattformen die den NPM unterstützen. Testweise wurde die Elm-Version der SPA unter Ubuntu 14.04 64bit, Windows 7 64bit und Windows 10 64bit kompiliert und auf Warnungen oder Fehler seitens des *elm – compiler* überprüft. In keinem der Fälle kam es zu Fehlern, geschweige denn Warnungen. Das Verhalten des Compilers war unter allen Betriebssystemen gleich und erzeugte keinerlei Anomalien. Das Betriebssystem *Mac* konnte nicht getestet werden, da zum Zeitpunkt der Auswertung kein passendes Endgerät zur Verfügung stand. Da sowohl *Mac*, als auch *Linux* auf dem Betriebssystem *Unix* basieren [?], wird an dieser Stelle davon ausgegangen, dass der Kompiliervorgang auch unter *Mac* ohne signifikante Probleme ausführbar ist. Die Abbildung 3.33 zeigt den fehlerfreien Kompiliervorgang der SPA unter Ubuntu 14.04 64bit mit Elm 0.17. Das Kriterium der Zuverlässigkeit gilt somit als vollständig erfüllt.

```

→ Elm-App elm-make Main.elm --output elm.js
Success! Compiled 1 module.
Successfully generated elm.js

```

Abbildung 3.33: Fehlerfreier Kompiliervorgang durch den Elm-Compiler

Auswertung: 3.2.4 Effizienz

Innerhalb der Abbildung 3.34 können die Zeiten für die Kompilierung der Elm-Applikation betrachtet werden. Dabei wurde die Dauer des Kompiliervorganges mit und ohne Caching gemessen. Es wurden jeweils zehn Messwerte

ermittelt, der jeweils höchste und niedrigste Wert gestrichen und das arithmetische Mittel der verbleibenden acht Werte berechnet. Das Streichen des höchsten und niedrigsten Wert soll etwaigen Messfehlern entgegenwirken. Im Durchschnitt benötigt es 8.03 Sekunden, um die Applikation von Grund auf zu kompilieren. Erlaubt man das Caching der vorherigen Kompiliervorgänge, so ist der erste Durchgang mit 7.88 Sekunden vergleichsweise hoch. Die folgenden Durchgänge hingegen benötigen im Durchschnitt 0.3 Sekunden, unter der Berücksichtigung, dass auch hier der höchste und niedrigste Wert vernachlässigt wird. Dadurch verfällt der langwierigste Kompiliervorgang. Unter realen Umständen wird ein Entwickler die Applikation unter der Verwendung von Caching kompilieren, womit nur sehr wenig Zeit verloren geht. Ferner kann der Entwickler auf den mitgelieferten Webserver *elm – reactor* zurückgreifen, der ebenfalls die Applikation bei jedem Seitenaufruf neu kompiliert. In beiden Fällen sind die Werte von 8.03s beziehungsweise 0.3s für den Kompilervorgang mehr als annehmbar, insbesondere da ein Entwickler standardmäßig das Caching nutzt. Die Effizienz wird als erfüllt angesehen. Die Performanz

Durchlauf	Dauer ohne Caching	Dauer mit Caching
1	7,56s	7,88s
2	8,31s	0,34s
3	7,78s	0,34s
4	7,77s	0,31s
5	7,4s	0,22s
6	8,48s	0,28s
7	9,79s	0,19s
8	9,19s	0,30s
9	6,98s	0,33s
10	8,14s	0,28s
Durchschnitt:	8,03s	0,3s

Abbildung 3.34: Zeitliche Auswertung des Kompiliervorganges in Elm

von Elm wird mittels der Applikation *TodoMVC Performance Comparison* ermittelt. Die Applikation liefert Ergebnisse für die Benchmark-Tests der Programmiersprachen Backbone 1.1.2, Ember 1.4 mit Handlebars 1.30, Angular 1.2.14, React 0.10.0, Om 0.5 mit React 0.9, Mercury 3.1.7 mit virtual-dom 0.8, sowie Elm 0.12.3 mit virtual-dom 0.8. Dabei nutzen die Programmiersprachen *Elm*, *Om* und *Mercury* das Konzept der *virtual – dom*. Die zu testenden Programmiersprachen sind weit verbreitet in der Webentwicklung [18, Vgl.]. Die Abbildung 3.35 zeigt eine graphische Auswertung nach 20 ausgeführten Tests. Bei jedem Testdurchlauf wird dabei die Zeit gemessen, die für die Erstellung von 100 Elementen, dem Anklicken all dieser erstellten Elemente sowie dem Löschen aller Elemente benötigt wird. Der Test wurde auf dem in der Sektion 3.4.2 beschriebenen System ausgeführt. Abbildung 3.35 zeigt, dass Elm durchschnittlich 259ms für einen Test benötigt hat. Den niedrigsten Wert mit 191ms pro Durchlauf erzielte Mercury und ist damit mehr als ein viertel mal schneller als Elm. Den schlechtesten Wert erzielte das Framework AngularJs

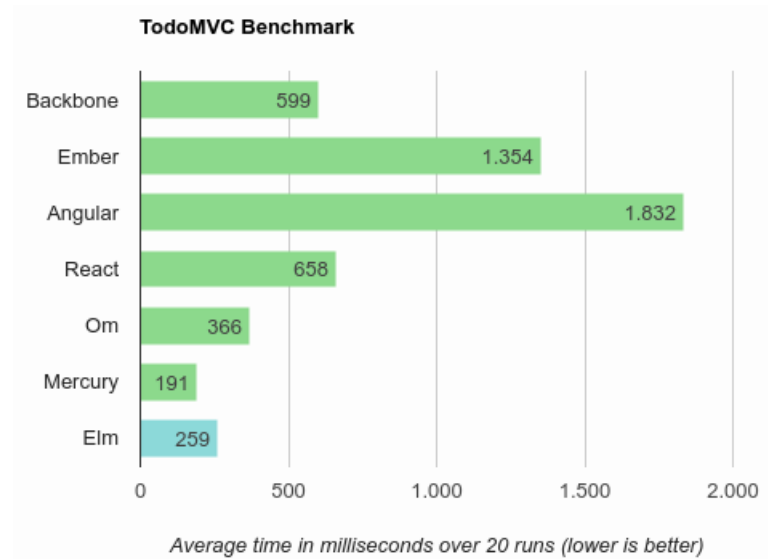


Abbildung 3.35: Benchmark der TodoMVC in unterschiedlichen Programmiersprachen

mit 1832ms pro Durchlauf. Dieser Wert schneidet mehr als 6x schlechter ab, als es bei Elm der Fall war. Obwohl die gegenübergestellten Programmiersprachen in der Testumgebung nicht den neuesten Versionen entsprechen, ist davon auszugehen, dass die grobe Platzierung selbst mit den neuesten Versionen ähnlich ausfällt. Die einzelnen Frameworks haben an der grundlegenden Umsetzung zur Darstellung von Daten im Browser nichts geändert, das bedeutet, dass Elm, Mercury und Om weiterhin mit neueren Versionen der *virtual – dom* arbeiten. Die Frameworks Angular, Backbone und Ember nutzen noch immer nicht standardmäßig das Konzept der *virtual – dom*. Lediglich das Framework React könnte bei einem Test der neuesten Version signifikante Veränderung aufweisen, da hier ein *virtual – dom* Konzept implementiert wurde. Das Kriterium der Performanz einer Elm-Applikation gilt als erfüllt. Im Vergleich zu anderen beliebten Frameworks schlägt sich Elm mit Bravour und wird nur von *Mercury*, einem extrem leichtgewichtigen Framework übertrumpft.

Auswertung: 3.2.5 Wiederverwendbarkeit

Das Konzept der Wiederverwendbarkeit, insbesondere der Modularität wird in Elm komplett durchgesetzt. Jede Datei in Elm ist zwangsläufig ein Modul, insofern man die darin enthaltenen Funktionen nutzen möchte. Zusätzlich ist auch ein Zugriffsschutz gegeben. Das bedeutet, dass Funktionen gesondert nach außen hin sichtbar und somit nutzbar gemacht werden können. Gängige imperative Programmiersprachen wie beispielsweise *C++* nutzen Klassen anstelle von Modulen. Dabei wird der Zugriff von außen auf eine Funktion innerhalb der Klasse durch die Schlüsselwörter *public*, *private* und *protected* definiert. Jedes Schlüsselwort bietet einen sichereren Zugriffsschutz. Ähnlich fungiert in Elm das Schlüsselwort *exposing*, wodurch einzelne Funktionen an die importierenden Module zugreifbar gemacht werden. Der Entwickler hat

dadurch die Sicherheit, dass nur die von ihm erwarteten Funktionen, sprich die API, genutzt werden. Dementsprechend ist das Bewertungskriterium der Wiederverwendbarkeit in allen Aspekten erfüllt.

Auswertung: 3.3.1 Browser Kompatibilität

Wie in Sektion 3.4.2 angesprochen, werden lediglich die Browser Google Chrome (Version 51), Internet Explorer (Version 11), Mozilla Firefox (Version 47) und Opera (Version 38) getestet. Dabei wird in jedem Browser die *index.html* der SPA aufgerufen und gewartet, bis die Seite komplett geladen ist. Anschließend werden die *onClick*-Elemente der Applikation getestet, indem auf die einzelnen Reiter in der Navigation geklickt wird. Die Webseite soll daraufhin zu den entsprechenden Stellen im Dokument scrollen und die Navigationsleiste kleiner werden. Außerdem soll das *ScrollSpy*-Feature dazu führen, dass die aktuelle Position im Dokument innerhalb der Navigationsleiste sichtbar gemacht wird. Sobald die *Portfolio*-Sektion erreicht wurde, wird das erste Element angeklickt und überprüft, ob ein asynchroner Request erzeugt, sowie die Ergebnisse in das geöffnete *Modal* eingepflanzt wurden.

Die Darstellung in allen Browsern ist komplett gleich. Es gibt nur marginale Unterschiede in der Darstellung der SPA selbst, die jedoch auf die Unterschiede der Browser selbst zurückzuführen sind. Zusätzlich erzeugt der Aufruf und die Interaktion mit der Webseite in keinem getesteten Browser einen Fehler. Ferner funktioniert die asynchrone Verarbeitung nach einem Klick auf eines der Portfolio-Elemente problemlos und zügig. Auch die Darstellung des Modals unterscheidet sich nicht. Alles in Allem scheint sich die kompilierte Elm-Applikation in allen getesteten Browsern gleich zu verhalten. Das Kriterium der Browser-Kompatibilität gilt als komplett erfüllt.

Auswertung: 3.3.2 Interoperabilität

Das Einbinden von externen JS-Frameworks oder bestehendem CSS-Code ist nicht vollständig in Elm gegeben. Die Programmiersprache Elm liefert mehrere Möglichkeiten eine Elm-Applikation im Browser darzustellen. Die Methoden dafür können bei Bedarf in der Sektion Grundaufbau nachgelesen werden. Abhängig von der Art der Initialisierung der Elm-Applikation entsteht eine Abhängigkeit zu einer *.html*-Datei, in welcher der Grundaufbau der Webseite erstellt und die Elm-Applikation geladen wird. Nutzt man eine *.html*-Datei und lädt dort die Elm-Applikation hinein, können bestehende CSS- oder JS-Frameworks innerhalb des *head*-Tag in einem *link*- oder *script*-Tag geladen werden. Hier entstehen keinerlei Probleme. Wird allerdings die Abhängigkeit der *.html*-Datei umgangen, wird die Elm-Applikation entweder im Vollbildmodus, oder im Hintergrund gestartet und muss somit zwangsweise das Einbinden externer Quellen übernehmen. Diese Funktionalität ist gegeben, jedoch sehr umständlich und nicht nutzbar für ein fertiges Produkt. Beim Einbinden von externem CSS durch nativen Elm-Code wird ein *link*-Tag innerhalb des *body*-Tags erstellt und der Inhalt asynchron nachgeladen. Üblicherweise wird externer CSS-Code innerhalb des *head*-Tags geladen und auf das gesamte Dokument

angewandt. Insofern wäre es nicht problematisch, jedoch bewirkt das Einbinden des Stylings innerhalb des *body*-Tags ein flickern, wodurch zunächst der gesamte Inhalt des Dokumentes komplett ohne und kurze Zeit später mit dem definierten CSS-Code angezeigt wird. Des Weiteren bietet die *elm-lang/html*-Bibliothek keine native Funktion für einen *script* oder *link*-Tag. Stattdessen muss die *Html.node*-Funktion genutzt werden, um ein eigenes Html-Element zu erzeugen. Eine Problemlösung steht an dieser Stelle aus.

Abgesehen vom initialen Einbinden externer Quellen funktioniert die Kommunikation zwischen der Elm-Applikation und externen JS-Skripten problemlos. Über spezifische Ports die jeweils für nur eine Aufgabe genutzt werden können ist die Kommunikation möglich. Die Änderungen um einen Port zu definieren und ein Modul für diese Aufgabe zu befähigen sind minimal. Der Zustand der Elm-Applikation ist dabei immer gewahrt und unmöglich in einen undefinierten Zustand zu bringen, da vorab definiert wird, welche Typen gesendet oder entgegengenommen werden. Wird beispielsweise ein *String* erwartet, jedoch ein *Objekt* seitens einem JS-Skript an die Elm-Applikation gesendet, nimmt die Applikation die Daten nicht entgegen und verwirft sie. Handelt es sich hingegen tatsächlich um den erwarteten Typ, sprich einem *String*, werden die Daten an die entsprechende Stelle zur *update*-Funktion weitergeleitet und die Daten ausgewertet.

Das Kriterium der Interoperabilität kann teilweise als erfüllt angesehen werden. Da die grundlegenden Funktionen des Kriteriums gegeben sind, jedoch nicht unter allen Umständen perfekt funktionieren, ist hier nicht von einer vollständigen Erfüllung auszugehen. Es ist nicht zu erwarten, dass die genannten Probleme bei Weiterführung des aktuell angewandten Konzepts der Programmiersprache behebbar sind. Die Elm-Applikation wird asynchron als eigenständiges Element innerhalb der DOM initialisiert. Dadurch ist es theoretisch notwendig zunächst die Applikation vollständig zu laden und im Nachhinein Änderungen am *head*-Tag, sprich außerhalb der Elm-Applikation, zu erlauben, um so weitere CSS oder JS-Dateien laden zu können. Erst wenn diese Vorgänge abgeschlossen sind, sollte dem Nutzer das Ergebnis der Webseite präsentiert werden. Folgt man dieser Herangehensweise würde dies zu weiteren offensichtlichen Problemen führen, allen voran die initiale Ladezeit, bis der Nutzer die Webseite fertig geladen betrachten kann. Alles in Allem bedarf es für eine Problemlösung weiterer Recherchen und voraussichtlich die Umgestaltung der in Elm genutzten Konzepte.

Auswertung: 3.3.3 Asynchrone Verarbeitung

Asynchrone Verarbeitung innerhalb einer nativen Elm-Applikation ist problemlos möglich. Dabei macht es keinen Unterschied, ob die asynchrone Aufgabe eine langwierige Berechnung, oder das Anfordern von externen Daten mittels eines HTTP-Requests ist. Die Bibliothek *elm - lang/core* liefert zur asynchronen Verarbeitung den Typ *Tasks*. Mit Hilfe dieses Typs können asynchrone Aufgaben definiert werden, die durch interne Algorithmen asynchron ausgeführt werden. Besteht bei einer Aufgabe die Gefahr, dass sie fehlschlagen könnte, können die Funktionen *fail* und *succeed* zur Hilfe genommen wer-

den. So kann der Zustand der Elm-Applikation in jedem Falle gesichert werden. Innerhalb der Applikation wurde ein asynchroner Request ausgeführt, bei dem von einem externen Webserver Daten angefordert wurden. Dadurch entstand die eine Abhängigkeit gegenüber der Antwortzeit des Webserver. Diese war unbekannt und konnte nicht vorhergesehen werden. Trotzdem war die Applikation auch während der Anfrage in vollem Umfang nutzbar. Sobald die Antwort des Webserver von der Elm-Applikation entgegengenommen werden konnte, wurden die übertragenen Daten dekodiert und in das *model* eingearbeitet. Daraufhin wurden die übermittelten Daten für den *view* verfügbar gemacht und eine aktualisierte Version der gesamten Applikation auf dem Bildschirm ausgegeben. Versuchsweise wurde die Übertragungszeit künstlich erhöht, so dass mehrere Sekunden auf die Antwort des Servers gewartet wurde. Dank der Umsetzung der asynchronen Verarbeitung kam es jedoch weder zu einem Laufzeitfehler, noch einer unvorhergesehenen Darstellung von Information. Ein weiterer Grund für die erwartete Darstellung ist, dass das Feld *async_content* in das die asynchronen Daten gespeichert werden, bereits bei der Initialisierung des *model* mit einem initialen Wert, genauer einem leeren String, versehen wurde. Dementsprechend wurde an der besagten Stelle in der das Feld des *model* dargestellt werden sollte ein leerer String angezeigt. Bei der Simulation einer fehlerhaften Übertragung wurde ebenfalls der Initialwert des Feldes angezeigt. Sämtliche Fälle die eintreten konnten wurden getestet und hatte in keinem Fall einen undefinierten Zustand der Elm-Applikation zur Folge. Das Kriterium kann somit als vollständig erfüllt angesehen werden.

Auswertung: 3.3.4 Dateigröße

Ermittelt wurden die Dateigrößen der Elm-Applikation, die in Abbildung 3.36 zusammengefasst wurden, indem eine neue Elm-Applikation initialisiert wurde. Dafür wurden die Pakete *elm-lang/html* (Version 1.0.0), *elm-lang/core* (Version 4.0.1) und *elm-lang/virtual-dom* (Version 1.0.2) installiert und in einem Testmodul *Main.elm* importiert. Das Testmodul beinhaltete zusätzlich die Funktionen *model*, *view*, *update* und *main*, um dem MVU-Konzept von Elm gerecht zu werden und einen minimal notwendigen Grundaufbau für eine funktionierende Applikation zu simulieren. Dabei liefert *model* einen Integer 0 zurück. Die Funktion *view* gibt den Wert von *model* in einem HTML-Element aus und die *update*-Funktion nimmt sämtliche Interaktionen des Nutzers entgegen und gibt das unveränderte *model* zurück. Die *main*-Funktion macht dabei die Elm-Applikation lauffähig über den Webserver *elm-reactor* oder einer expliziten Einbindung der Applikation in eine *.html*-Datei. Das Resultat der Kompilierung der Elm-Applikation misst 127 Kilobyte. Der *elm-compiler* liefert keinerlei Möglichkeit die Dateigröße zu minimieren. Aufgrunddessen wird das externe Tool *jscompress*¹⁹ genutzt, um die Dateigröße der kompilierten Elm-Applikation zu minimieren. Das Resultat ist eine Minimierung um 66%. Die Dateigröße misst nun 59 Kilobyte. Wird anschließend noch *gzip*²⁰ auf die

¹⁹Abrufbar unter <http://jscompress.com/>.

²⁰gzip ist ein Programm zur Datenkompression [13, Vgl.].

Name	Minimiert	Minimiert u. Komprimiert
Angular2	566KB	111KB
Ember 2.2.0	435KB	111KB
React 0.14.5 + React DOM	133KB	40KB
Elm 0.17	127KB	34KB

Abbildung 3.36: Dateigrößen der Frameworks Angular2, Elm, Ember und React in Kilobyte

minimierte Datei ausgeführt, misst die daraus resultierende Datei noch 34 Kilobyte. Im Gegensatz zu den Dateigrößen der Frameworks *AngularJs*, *Ember* und *React* ist Elm um bis zu 30.6% leichtgewichtiger (ausgehend von den minimierten und mit gzip komprimierten Dateien) und erfüllt somit das Kriterium einer geringen Dateigröße. Jedoch hat bietet der *elm – compiler* keinerlei Möglichkeit der Minimierung. Aufgrund dieser Tatsache gilt das Kriterium als nur teilweise erfüllt.

4 Fazit

Ziel dieser wissenschaftlichen Arbeit war es, die Programmiersprache Elm auf unterschiedlichste Gesichtspunkte hin zu evaluieren und speziell für die Verwendung im Bereich der Webentwicklung zu prüfen. Infolgedessen wurde eine SPA überführt in eine native Elm-Applikation. In der herkömmlichen Webentwicklung ist es üblich die darstellbare Oberfläche einer Webseite mit HTML, das Styling mit CSS und Interaktionen mit JS zu programmieren. Ein Entwickler hat dadurch zwangsweise drei Programmiersprachen zu bedienen. Einerseits mündet dieses Konzept in einer klaren Trennung der Applikation, andererseits muss jede Programmiersprache dem Entwickler bekannt sein. Ferner werden JS-Skripte genutzt, um die dargestellte Webseite weiter zu verändern, Interaktionen des Nutzers abzufangen und darauf zu reagieren, wodurch die Webseite unter Umständen undefinierte Zustände erreicht und ausgeführte Skripte im Konflikt miteinander stehen. Elm hingegen bringt die funktionale Programmierung mit einem expliziten Typensystem, reinen Funktionen und klar strukturierten Style-Guides in die Webentwicklung. Das hinter Elm stehende Konzept beruht darauf, zuverlässige Applikationen zu entwickeln. Der initiale Aufwand bei der Entwicklung einer Elm-Applikation ist dabei höher als bei der üblichen Entwicklung von Webseiten, der Aufwand die Applikation zu erweitern, Codeteile auszulagern und ungenutzte Teile zu entfernen ist hingegen wesentlich unkomplizierter. Zu Beginn der praktischen Ausarbeitung war unklar, inwiefern die native Implementierung der SPA in Elm mit den bestehenden JS-Skripten und CSS-Definitionen funktionieren wird und ob etwaige Änderungen notwendig sind. Bestenfalls wären sämtliche Garantien die Elm liefert, mit der Überführung des des HTML-Codes in der Applikation eingebunden, ohne die JS- und CSS-Dateien verändern zu müssen. Die Überführung der Elm-Applikation ergab, dass lediglich zwei der insgesamt neun geprüften Kriterien nur teilweise erfüllt wurden. Die verbleibenden Kriterien wurden vollständig erfüllt. Zu den unerfüllten Kriterien gehörte unter anderem die Interoperabilität, mit der geklärt werden sollte, inwiefern die Funktionalität vorhandener JS-Skripte gegeben sei, wenn der Rest der Applikation nativ in Elm verwirklicht wird. Ergebnis der Auswertung war, dass nicht alle JS-Skripte einwandfrei mit dem nativ in Elm programmierten Teil der Applikation, genauer der DOM, kommunizierten. Die Ereignisbehandler, die durch die bestehenden JS-Skripte erzeugt wurden, bezogen sich dabei auf Elemente die sich zur Zeit ihrer Initialisierung noch nicht im Zugriffsbereich des Skriptes befanden, da die Elm-Applikation erst nachträglich in die DOM injiziert wurde. Zusätzlich nutzt Elm das Konzept der *virtual-dom*, wodurch die tatsächliche Repräsentation des DOM nicht direkt von außen zugreifbar ist und sich unter Umständen verändert, wodurch die vorherigen Ereignisbehandler nicht

mehr auf die ursprünglichen Elemente in der DOM zugreifen. Die Dateigröße einer Elm-Applikation mit den Grundfunktionen zur Darstellung eines HTML-Elementes im Browser war zufriedenstellend gering. Jedoch wird die erzeugte Datei nicht minimiert, obwohl dies mit einfachsten Mitteln bewerkstelligt werden kann. Der Test ergab, dass die Datei um mehr als 60% kleiner sein könnte, wenn eine standardmäßige Verkleinerung der Datei durch den *elm – compiler* stattfinden würde.

Der selbst erzeugte Quellcode wies eine sehr hohe Wartbarkeit und Lesbarkeit auf, zum einen wegen der Vielzahl an unterstützenden Plugins die es für die Entwicklungsumgebungen gibt, zum anderen aufgrund der Hilfe durch den *elm – compiler*. Hier wird dem Entwickler viel Arbeit erspart und gleichzeitig Sicherheit über die Zuverlässigkeit der Applikation vermittelt. Die Fehlermeldungen waren sehr detailliert und überwiegend akkurat, wodurch davon ausgegangen werden kann, dass eine kompilierte Elm-Applikation keine Laufzeitfehler erzeugt. Zuverlässig ist der erzeugte Quellcode auch hinsichtlich der Nutzung auf unterschiedlichen Plattformen. Es gab keinerlei Fehler bei der Kompilierung des Codes unter den gängigsten Betriebssystemen. Trotz der Plattformunabhängigkeit überzeugte die Elm-Applikation mit hoher Effizienz. Die getestete Applikation erwies sich als deutlich schneller als andere beliebte Frameworks wie AngularJS oder React. Obwohl sich die Benchmarks auf ältere Versionen der getesteten Frameworks stützen, ist davon auszugehen, dass die Ergebnisse der aktuellsten Versionen ähnlich ausfallen. Um exakte Ergebnisse zu ermitteln, müsste die Applikation *ToDoMVC Performance Comparison* in allen Programmiersprachen auf die neueste Version aktualisiert werden. Im Rahmen dieser Arbeit hat sich ebenfalls ergeben, dass Code mit einfachsten Mitteln modularisiert werden konnte. Dadurch entstand eine hohe Übersichtlichkeit innerhalb des Quellcodes. Module konnten ferner beliebig oft an anderen Stellen eingebunden und genutzt werden. Die Elm-Applikation kompilierte nicht nur auf allen getesteten Plattformen fehlerfrei, sondern bot auch eine erstaunliche Browser-Kompatibilität. Die Darstellung in den getesteten Browsern war stets fehlerfrei und erzeugte fast gänzlich gleiche Ergebnisse. Marginale Unterschiede sind auf die verschiedenen Browser selbst zurückzuführen, die einige HTML-Elemente unterschiedlich darstellen. Ein Entwickler kann darauf nur mit entsprechendem CSS-Styling Einfluss nehmen. Trotz der unveränderbaren Datenstrukturen in Elm war es möglich Daten asynchron zu verarbeiten. Mit Hilfe eines *Tasks* konnte ein asynchroner Request an einen externen Server gesendet, die Antwort ausgewertet und auf dem Bildschirm dargestellt werden, ohne den Zustand der Applikation zu gefährden und einen Absturz zu erzeugen.

Zusammenfassend lässt sich sagen, dass Elm die Anforderungen der Webentwicklung fast vollständig erfüllt. Lediglich die Interoperabilität bedarf der Umsetzung eines anderen Konzeptes. Sämtliche Tests dieser wissenschaftlichen Arbeit beziehen sich bewusst auf grundlegende Aspekte der Programmiersprache, um so allgemeine Erkenntnisse daraus ziehen zu können. Die SPA ist denkbar einfach aufgebaut, zeigt jedoch alle Konzepte die Elm nutzt und für wichtig erklärt. Während der Entwicklung wurde deutlich, dass das *Model* den

Status der Applikation innehält. Dadurch lag der Fokus während der Entwicklung eher auf den zu verarbeitenden Daten, als auf der letztlichen Darstellung. Die Programmiersprache Elm bietet sich folglich eher für Web-Applikationen an, die deutlich mehr Logik als eine SPA beinhalten, oder mehr Interaktionen mit dem Nutzer anstreben. Gerade aufgrund der ausgeprägten Unterstützung durch den *elm – compiler* und die damit verbundene Zeiteinsparung ist Elm auch für Entwickler ohne, oder mit nur geringen Vorkenntnissen im Bereich der funktionalen Programmierung sinnvoll.

Literaturverzeichnis

- [1] BORNING, Alan ; CHEN, Harr ; YASUHARA, Ken: *Evaluating Programming Languages*. <http://courses.cs.washington.edu/courses/cse341/02sp/concepts/evaluating-languages.html>. Version: 2002. – abgerufen am 4. Juni 2016
- [2] CZAPLICKI, Evan: *Elm: Concurrent FRP for Functional GUIs*. <https://www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf>. Version: April 2012. – abgerufen am 22. April 2016
- [3] CZAPLICKI, Evan: *Elm Packages*. <http://elm-lang.org/>. Version: Mai 2012. – abgerufen am 1. Mai 2016
- [4] CZAPLICKI, Evan: *elm-package - Command line tool to share Elm libraries*. <https://github.com/elm-lang/elm-package>. Version: September 2013. – abgerufen am 20. Juni 2016
- [5] CZAPLICKI, Evan: *Blazing Fast HTML*. <http://elm-lang.org/blog/blazing-fast-html>. Version: Juli 2014. – abgerufen am 3. Mai 2016
- [6] CZAPLICKI, Evan: *Elm Packages*. <http://package.elm-lang.org/>. Version: Oktober 2014. – abgerufen am 21. Juni 2016
- [7] CZAPLICKI, Evan: *From JavaScript?* <http://elm-lang.org/docs/from-javascript>. Version: Juni 2015. – abgerufen am 5. Mai 2016
- [8] CZAPLICKI, Evan: *Erlang Factory SF 2016 Keynote Phoenix and Elm – Making the Web Functional*. <https://www.youtube.com/watch?v=XJ9ckqCMiKk&t=1740>. Version: März 2016. – abgerufen am 20. Juni 2016
- [9] DR. JOST, Steffen: *Einführung in die Funktionale Programmierung mit Haskell für Studierende mit Nebenfach Informatik*. <https://www.tcs.ifi.lmu.de/lehre/ss-2013/efpn/efp-material/folien-01-motivation-und-planung>. Version: April 2013. – abgerufen am 30. Juni 2016
- [10] FELDMAN, Richard: *Richard Feldman - Introduction to Elm*. <https://www.youtube.com/watch?v=zBHB9i8e3Kc&t=2478>. – abgerufen am 2. Mai 2016
- [11] GITHUB, Inc.: *Github Elm-Issues*. <https://github.com/search?q=language%3Aelm&ref=opensearch&type=Issues>. Version: Juni 2016. – abgerufen am 10. Juni 2016

- [12] GITHUB, Inc.: *Github Elm-Repositories*. <https://github.com/search?utf8=%E2%9C%93&q=language%3Aelm&type=Repositories&ref=searchresults>. Version: Juni 2016. – abgerufen am 21. Juni, 2016
- [13] GRIGORIK, Ilya: *Optimizing content efficiency*. <https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimize-encoding-and-transfer?hl=en>. Version: k.A.. – abgerufen am 1. Juli 2016
- [14] HEROLD, Helmut ; LURZ, Bruno ; WOHLRAB, Jürgen: *Grundlagen der Informatik*. Pearson Studium, 2007. – 195 S. – ISBN 978-3-8273-7305-2
- [15] HOFFMANN, Berthold: *Algebraische Datentypen*. <http://www.informatik.uni-bremen.de/agbkb/lehre/pi3/folien/Algebraisch.pdf>. Version: November 2009. – abgerufen am 4. Mai 2016
- [16] K.A.: *Elm-Slack Nutzer*. <http://elmlang.herokuapp.com/>. Version: k.A.. – abgerufen am 1. Juli 2016
- [17] KASTENS, Prof. Dr. U.: Programmierparadigma. (2013), September. <http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/technologien-methoden/Sprache/Programmiersprache/Paradigma>. – abgerufen am 29. Mai 2016
- [18] KUHN, Zach: Choosing a Front End Framework: Angular vs. Ember vs. React. (2015), Oktober. <http://smashingboxes.com/blog/choosing-a-front-end-framework-angular-ember-react>. – abgerufen am 1. Juli 2016
- [19] LLC., AWIO WEB S.: *Web Browser Market Share*. <https://www.w3counter.com/globalstats.php?year=2016&month=5>. Version: Juni 2016. – abgerufen am 06. Juni 2016
- [20] MILLER, David: *Start Bootstrap - Agency*. <https://github.com/BlackrockDigital/startbootstrap-agency>. Version: April 2016. – abgerufen am 18. April 2016
- [21] PORTO, Sebastian: *Elm Tutorial - Building web applications with Elm*. <http://www.elm-tutorial.org/en/>. Version: Dezember 2015. – abgerufen am 15. Mai 2016
- [22] SMITH, Scott: What About Elm? (2016), April. <http://www.slideshare.net/scottnelsonsmith/what-about-elm>. – abgerufen am 05. Mai 2016
- [23] VYNOGRADENKO, Anton: Sizes of JS frameworks, just minified + minified and gzipped. (2016), Januar. <https://gist.github.com/Restuta/cda69e50a853aa64912d>. – abgerufen am 23. Juni 2016

- [24] WESTBY, Luke: *builtwithelm*. <http://builtwithelm.co/>.
Version: Dezember 2015. – abgerufen am 07. Juni 2016

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt, nur die angegebenen Quellen benutzt und die in den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Kiel, den 07. Juli 2016

(Unterschrift)