



FACHHOCHSCHULE KIEL
University of Applied Sciences

EVALUIERUNG VON ELM ALS FRONTEND FÜR WEBAPPLIKATIONEN

Fachhochschule Kiel

DRAFT

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

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, At accusam aliquyam diam diam dolore dolores duo eirmod eos erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd magna no rebum. sanctus sea sed takimata ut vero voluptua. est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur

Inhaltsverzeichnis

Abstract	i
1 Einleitung	7
2 Theoretische Grundlagen	8
2.1 Funktionale Programmiersprachen	8
2.2 Grundlagen der Programmiersprache Elm	9
2.2.1 Geschichte	9
2.2.2 Konzept	10
2.2.3 Umsetzung	11
2.2.4 Einführung in die Elm-Architektur	16
3 Evaluierung der Programmiersprache Elm	30
3.1 Bewertungsmuster	30
3.2 Bewertungskriterien	31
3.2.1 Wartbarkeit und Lesbarkeit	31
3.2.2 Zuverlässigkeit	32
3.2.3 Portabilität	32
3.2.4 Effizienz	32
3.2.5 Wiederverwendbarkeit	33
3.3 Web-spezifische Bewertungskriterien	33
3.3.1 Browser Kompatibilität	34
3.3.2 Interoperabilität	34
3.3.3 Asynchrone Verarbeitung	34
3.3.4 Dateigröße	35
3.4 Empirische Analyse	36
3.4.1 Programmablauf	36
3.4.2 Entwicklungsumgebung	36
3.4.3 Grundaufbau	37
3.4.4 Überführung des Views	41
3.4.5 Beheben von JavaScript-Fehlern	42
3.4.6 Auslagern des Views	43
3.4.7 Hinzufügen von Klick-Events	43
3.4.8 Modularisierung der Applikation	47
3.4.9 Beobachtungen	51
3.4.10 Auswertung	54
3.4.11 Fazit	57
4 Fazit	58

5 Eidesstattliche Erklärung	59
------------------------------------	-----------

Abkürzungsverzeichnis

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

Abbildungsverzeichnis

2.1	Eine iterative Funktion	8
2.2	Eine rekursive Funktion	9
2.3	Ein simpler Zähler auf einer Webseite	10
2.4	Das Model-View-Update Konzept von Elm	11
2.5	Beispiel der dynamischen Typisierung	13
2.6	Performanz von Elm im Vergleich zu anderen Web-Frameworks. [1]	14
2.7	Das Tool elm-repl in der Kommandozeile	17
2.8	Ein Union-Type in Elm	18
2.9	Ein erweiterter Union-Type in Elm	19
2.10	Ein Record in Elm	19
2.11	Ein Tupel in Elm	20
2.12	Eine Variablendeklaration in Elm	21
2.13	Eine Kontrollstruktur in Elm	21
2.14	Kontrollstruktur in C++	22
2.15	Eine Funktion in Elm	22
2.16	Anwendung einer anonymen Funktion in Elm	23
2.17	Definition einer Funktion ohne Typen Alias	24
2.18	Definition einer Funktion mit Typen Alias	24
2.19	Mögliche Deklarationen eines Elm-Moduls	24
2.20	Mögliche Formen der Importierung eines Elm-Moduls	25
2.21	Die Online-Integrated Development Environment (IDE) von Elm	26
2.22	Installation des Node Package Manager (NPM)	26
2.23	Installation von Elm	27
2.24	Überprüfung der erfolgreichen Installation von NodeJs und dem NPM	27
2.25	Installation eines externen Paketes über den Elm-Paketmanager	28
2.26	Der gestartete Elm-Webserver	29
3.1	Die fünf meistgenutzten Browser im Mai 2016 [2]	34
3.2	Grundaufbau der <i>index.html</i> , um die Elm-Applikation zu inji- zieren	38
3.3	Grundgerüst der Elm-Applikation	39
3.4	Eine beispielhafte Initialisierung der Elm-Applikation mit ini- tialen, dynamischen Werten	40
3.5	Elm-Compilerfehler bei fehlenden Funktionen im Namensraum .	41
3.6	JavaScript-Fehler innerhalb der Developer-Tools	42
3.7	Notwendiger Code, um JavaScript-Fehler zu beheben.	42
3.8	Deklaration einer Sektion des Views in Elm	43

3.9	Ausgelagerter View in eine eigene Funktion	44
3.10	Aufruf der einzelnen Sektionen im View	44
3.11	Funktion zum scrollen bei einem Klick	45
3.12	Definition eines neuen Union Types, sowie eines Ports für die Kommunikation zwischen Elm und JavaScript (JS)	45
3.13	Erweiterung des Views um einen <i>onWithOptions</i> Ereignisbe- handler in Elm	46
3.14	Entgegennahme der Daten aus der Elm-Applikation und an- schließendes scrollen	47
3.15	Einbindung und Aufruf der ausgelagerten View-Funktionen . . .	48
3.16	Einbindung und Aufruf der ausgelagerten View-Funktionen . . .	48
3.17	OnClick-Ereignisbehandlung in Elm	50
3.18	<i>Msg</i> -Deklaration mit asynchronem Request	50
3.19	Asynchroner Hypertext Transfer Protocol (HTTP)-Request in Elm	50
3.20	Dekodierung eines Json-Objektes in Elm	51
3.21	<i>Update</i> -Fälle für den asynchronen Request	51
3.22	Ausgehender HTTP-Request und eingehende Antwort	52
3.23	Ein <i>div</i> -Element in HyperText Markup Language (HTML) und Elm	53

Tabellenverzeichnis

3.1	Auswertung der Versuchskriterien	55
-----	--	----

1 Einleitung

Die heutige Welt ist sehr schnelllebig und spielt sich immer mehr im Internet ab. Firmen präsentieren sich auf ihren Webseiten und akquirieren dadurch Neukunden. Ganze Geschäfte leben nur noch durch den Online-Handel und besitzen keinerlei Verkaufsläden, in denen ein Kunde das Produkt vorab in den Händen halten kann. Umso wichtiger ist es, das Produkt auf der Webseite außergewöhnlich gut zu präsentieren, um den Kunden zu überzeugen. Webseiten dieser und im Grunde jeglicher Art zielen immer darauf ab, einem Nutzer Informationen bereitzustellen und entsprechend angenehm zu präsentieren. Doch die Entwicklung solcher Systeme ist komplex und geht weit über das Design hinaus. Folglich ist ein großer Teil der Ausgaben von Firmen, die sich online präsentieren, die Bezahlung von Entwicklern für ihre Webapplikationen. Diese können simple Webseiten ohne großartige Funktionen sein, aber auch komplexe Systeme wie ein automatisierter Online-Handel, in dem die Nutzer ihren gesamten Einkauf abwickeln können, mitsamt Bezahlung. Demzufolge ist es für die Firmen von großem Interesse, dass die angestellten Entwickler zügig Ergebnisse in der Entwicklung der Webapplikationen machen. Damit der Entwickler effektiv arbeiten kann, braucht er ihn unterstützende Systeme, angefangen bei den Werkzeugen wie seiner Entwicklungsumgebung, bis hin zur tatsächlichen Programmiersprache. Diese wissenschaftliche Arbeit befasst sich mit der neuen Programmiersprache Elm, welche eine funktionale, den deklarativen Programmierparadigmen folgende, Programmiersprache ist. Sie wurde zu Beginn ihrer Entwicklung für die Erstellung von grafischen Benutzeroberflächen und der Verbildlichung mathematischer Funktionen genutzt, bewegt sich nun jedoch immer weiter in Richtung der Webentwicklung und kommt mit einigen Neuheiten, Veränderungen und einer aktiven Gemeinschaft an Open-Source Entwicklern.

Elm verspricht eine blitzschnelle Darstellung von Inhalten selbst bei riesigen Datenmengen mit Hilfe einer Technik die *virtual – dom* genannt wird. Auch soll es keinerlei Laufzeitfehler mehr geben, da die gesamte Sprache mit Garantien ausgeschmückt ist, die im Zusammenspiel mit dem eigens entwickelten Compiler alle möglichen Fehler vorab findet und darauf hinweist. All diese Versprechungen werden anhand mehrerer Bewertungskriterien während einer empirischen Entwicklung einer Webapplikation geprüft.

2 Theoretische Grundlagen

2.1 Funktionale Programmiersprachen

In der Programmierung gibt es zwei Programmierparadigmen, die zur Kategorisierung von Programmiersprachen dienen und sich im Laufe der Zeit entwickelt haben. Dabei beschreiben die Paradigmen verschiedene Prinzipien der Programmierung. Die zugrunde liegenden Kategorien werden häufig als imperative und deklarative Programmierung bezeichnet, in welchen sich jede Programmiersprache einordnen lässt. Jede Kategorie birgt weitere Unterkategorien und dient der Verfeinerung der Prinzipien. Häufig gehören funktionale

```
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

Programmiersprachen dem deklarativen Programmierparadigma an. Es ist jedoch nicht ausgeschlossen, dass eine Programmiersprache mehreren Kategorien zugehörig ist und dadurch die Merkmale von mehr als einem Paradigma unterstützt. Zur Gruppe der deklarativen Programmiersprachen zählt man unter anderem Abfragesprachen wie SQL, sowie funktionale Programmiersprachen wie Lisp oder Scheme. Programmiersprachen der deklarativen Programmierung haben ihren Ursprung in der Mathematik. Programme werden hier als mathematische Funktionen formuliert, die nicht länger beschreiben, was getan werden soll, sondern lediglich vorgeben, welches Ergebnis am Ende erwartet wird. Bei funktionalen Programmiersprachen ist es üblich, dass eine Variable nach ihrer Initialisierung ihren zugewiesenen Wert für die gesamte Laufzeit des Programmes beibehält und unveränderlich bleibt. Es ist dementsprechend stets nachzuvollziehen, welchen Wert ein Ausdruck besitzt, wodurch insbesondere akademische Anforderungen an ein Programm, wie etwa die Beweisführung, erfüllt werden können. Zusätzlich können auch unendliche Datenstrukturen behandelt werden. 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. In beiden Fällen ist das Ergebnis gleich, jedoch besitzt die rekursi-

```

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

ve Implementierung keinerlei Seiteneffekte. 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 entgegen nehmen 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 durch einen Verweis 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. Auch Elm gehört dem deklarativen Programmierparadigma an und vereinheitlicht dessen Konzepte.

2.2 Grundlagen der Programmiersprache Elm

2.2.1 Geschichte

Elm ist eine funktionale Programmiersprache. Sie wurde im Rahmen der Bachelorarbeit „Elm: Concurrent FRP for Functional GUIs“ (<http://Elm-lang.org/papers/concurrent-frp.pdf>) von Evan Czaplicki entwickelt und im April 2012 offiziell veröffentlicht. Auslöser für die Entwicklung von Elm war für Czaplicki die Schwierigkeit, ein Bild auf einer Webseite sowohl horizontal, als auch vertikal zu zentrieren. Es gab keine für ihn annehmbare, leichte Lösung für dieses Problem, ohne damit weitere Probleme zu schaffen.

Unter der Leitfrage „What would web programming look like if we could restart?“ („Wie würde Web-Programmierung aussehen, wenn wir neu starten könnten?“) machte er sich Gedanken, 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

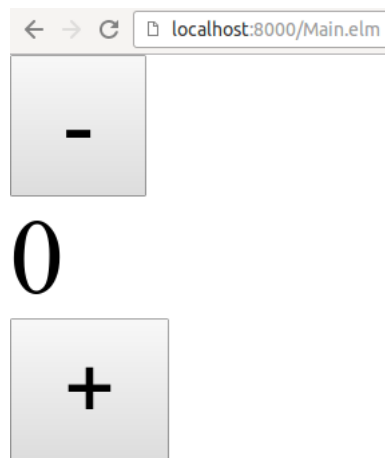


Abbildung 2.3: Ein simpler Zähler auf einer Webseite

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*, sprich den momentanen Zählerwert übergeben. Die *Update*-Funktion nimmt sämtliche Einwirkungen durch den Nutzer von außen entgegen und wendet diese Aktionen auf das aktuelle *Model* an. Das bedeutet in diesem konkreten Fall, dass das *Model* erhöht oder reduziert wird. Dabei wird jedoch nicht das *Model* direkt verändert, sondern ein neues *Model* mit den geänderten Werten wird 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 Website, soll heißen wie das *Model* dargestellt wird. Elm nutzt ein virtuelles Document

Object Model (DOM), wodurch nur tatsächliche Änderungen im Browser angezeigt werden, anstatt dauerhaft das komplette DOM stetig zu aktualisieren. Das DOM beschreibt die Schnittstelle zum Datenzugriff auf das Objektmodell eines HTML-Dokumentes. Der Datenfluss in Elm wird noch einmal in Abbildung 2.4 visualisiert.

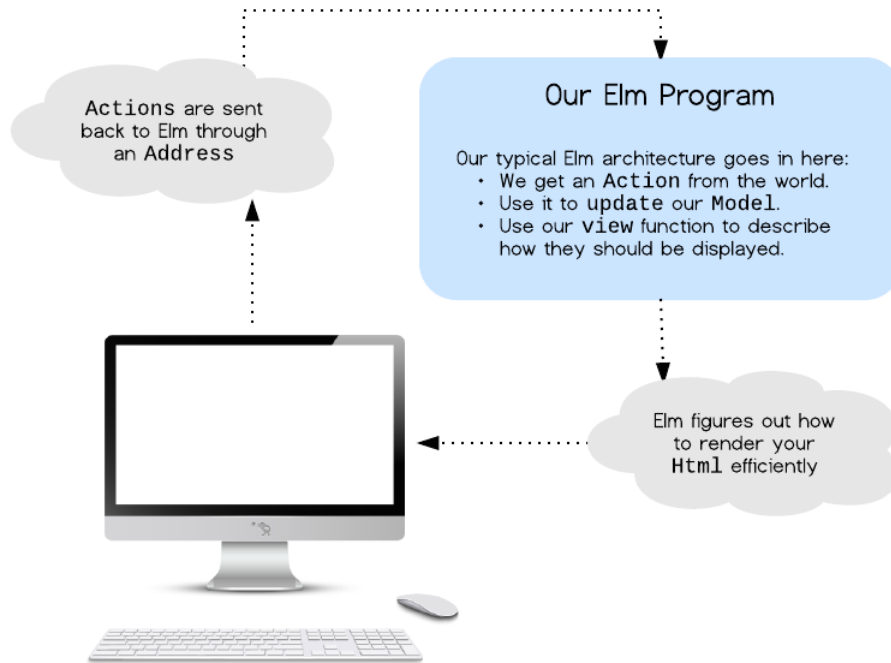


Abbildung 2.4: Das Model-View-Update Konzept von Elm

2.2.3 Umsetzung

Der vom Programmierer verfasste Programmcode wird vor der endgültigen Nutzung zu JS, HTML und Cascading Style Sheet (CSS) kompiliert und in die Webseiten integriert. Entsprechend fungiert in Elm verfasster Code im Endeffekt wie natives 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 (<http://elm-lang.org/>) erzeugt. Sämtliche Fehlerquellen werden vom Compiler zuvor erkannt, abgefangen und an den Programmierer weitergeleitet, um sie zu beheben. Damit dies funktioniert, implementiert Elm mehrere Konzepte des deklarativen Programmierparadigmas.

Keine Seiteneffekte

Ein Seiteneffekt beschreibt die mehrmalige Zuweisung einer Variable mit einem Wert. In Elm ist das allerdings nicht möglich. 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 darstellt. 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 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 der Ausdruck 2.2 den aktuellen Wert in x auslesen, um 1 inkrementieren und das Ergebnis der Operation in die Variable x schreiben. Mathematisch betrachtet ist diese Aussage jedoch schlichtweg falsch, denn es existiert kein x , welches diese Aussage wahr werden lässt:

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

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 nicht dieselbe Variable sind wodurch die Aussage nun als wahr eingestuft werden kann. Das beschriebene Konzept wird referentielle Transparenz genannt und beschreibt die Kontinuität des Wertes einer Variable. Des Weiteren basieren Funktionen in Elm auf dem Konzept von reinen Funktionen (*pure functions*). Das bedeutet, dass eine Funktion stets das gleiche Ergebnis liefert, insofern auch die Eingabeparameter gleich bleiben, unabhängig vom Zeitpunkt der Ausführung. Beispiele für eine reine Funktion 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 $\text{random}()$ Funktion, die einen (semi-)zufälligen Wert zurückliefert und somit als eine unreine Funktion gilt. Doch auch diese Funktion kann zu einer reinen Funktion gemacht werden, wenn man sie einen Wert abhängig von einem Übergabeparameter berechnen lässt wie beispielsweise $\text{random}(\text{seed})$.

Elm-Compiler

Laufzeitfehler sollen mit Elm in Vergessenheit geraten. Dafür soll der integrierte Compiler sorgen. Die Fehlermeldungen des Compilers sind sehr strikt und deuten exakt auf die Programmzeile die für den jeweiligen Fehler verantwortlich ist. 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 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 dieser Situation beziehungsweise analysiert, ob Variablen und Funktionen vorab initialisiert wurden, welche Parameter die einzelnen Funktionen erwarten und ob die Rückgabeparameter dem Typen entsprechen, den die anderen Funktionen erwarten. Befolgt man die Anweisungen des Compilers, soll das in einem stark strukturierten, lesbaren und funktionierenden Code münden. Dem Programmierer werden weniger Möglichkeiten gegeben, bestimmte Ziele zu erreichen, doch dadurch soll auf lange Sicht einheitlicher, lesbarer und besser zu wartender Code erzeugt werden. Passend dazu gibt es bereits viele Erweiterungen für gängige Editoren wie Sublime Text und Atom, welche beim Speichern des Projektes den Code entsprechend des Style Guides strukturieren und formatieren. So kann einerseits die Lesbarkeit des Codes vereinheitlicht, andererseits die Fehlerquellen in Form von Einrückungsfehlern oder vergessenen Kommas o.ä. verringert werden.

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 in JS. Der Typ der Variable *i* wurde in der Abbildung 2.5

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

Abbildung 2.5: Beispiel der dynamischen Typisierung

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.

Modularität

Um geschriebenen Code auch in Zukunft wartbarer zu machen, ist Elm modular aufgebaut und leicht erweiterbar. Es ist denkbar einfachen Code

zu importieren. Die importierten Module verstehen sich als gekapselt, wodurch sie in keiner Weise mit dem bereits verfügbaren Code kollidieren können.

Performanz

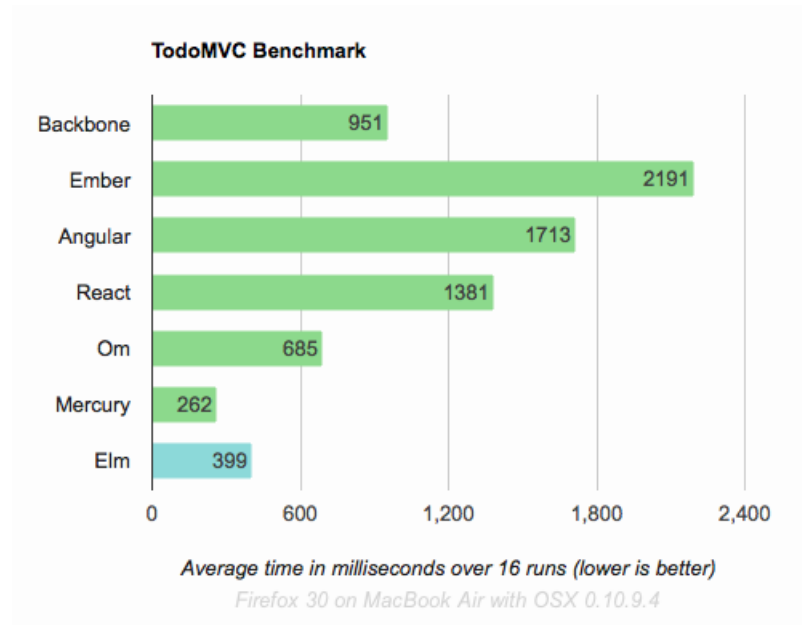


Abbildung 2.6: Performanz von Elm im Vergleich zu anderen Web-Frameworks. [1]

Obwohl Daten nicht verändert, sondern lediglich als neuer aktualisierter Datensatz betrachtet werden und einzelne Funktionen stark ausgelagert werden können, leidet die Performanz nicht darunter. Laut Abbildung 2.6 überzeugt Elm mit einer sehr guten Geschwindigkeit. Möglich wird das durch die Verwendung eines virtuellen DOM. Dabei wird das echte DOM bei jedem „Frame“ in eine abstrakte Version kopiert. Auf diese abstrakte Version werden die Änderungen angewandt. Zunächst klingt diese Vorgehensweise sehr langsam und aufwändig, doch um 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 Elemente im DOM des Nutzers aktualisiert werden müssen. Dieser zu verändernde Teil stellt nur einen Bruchteil des kompletten DOM dar. Man kann dadurch von einer immensen Effizienzsteigerung ausgehen.

Interoperabilität

Es wäre sehr aufwendig die gängigsten JavaScript-Bibliotheken und -Frameworks wie jQuery oder AngularJS komplett zu verwerfen und mit Elm zu realisieren

respektive neu zu programmieren. Glücklicherweise bietet Elm eine ausgereifte Interoperabilität, wodurch alle Garantien der deklarativen Programmierung übernommen werden können, selbst wenn die externen Bibliotheken diese nicht gewährleisten. Die genannten externen Bibliotheken sind zum Großteil nicht deklarativ programmiert, weswegen normalerweise keinerlei Garantien seitens Elm gemacht werden können. Jedoch ist Elm von den externen Bibliotheken isoliert und kommuniziert nur über sogenannte Ports mit JavaScript. Die Kommunikation durch die Ports funktioniert in beide Richtungen, womit sämtliche Daten bei Bedarf ausgetauscht werden können. Da in Elm wie bekannt vorab die Typen der Eingabeparameter und Rückgabewerte spezifiziert werden müssen, werden die Garantien weiterhin gewahrt.

Debugger

Nicht nur die detaillierten Fehlermeldungen des Compilers sind ein großer Pluspunkt von Elm, sondern auch der mitgelieferte und einfach zu bedienende Debugger. Anders als bei anderen Programmiersprachen zeigt der Debugger nicht nur einen einfachen Stacktrace mit den letzten Rücksprungadressen an. Vielmehr ermöglicht es der Debugger in Elm die Zeit zurückzudrehen. In gängigen Debuggern ist es nicht einfach, ein bestimmtes Verhalten, das möglicherweise zum Absturz des Programms geführt hat, zu reproduzieren. Um den Fehler erneut zu erhalten und dadurch ein Verständnis des Fehlers aufzubauen ist es oft notwendig den genauen Hergang und Ablauf manuell nachzuahmen. Der Elm Debugger hingegen speichert den Status einer jeden Variable zu jedem Zeitpunkt und zeichnet außerdem die Wechsel auf. Aufgrund dessen ist es sehr einfach möglich mit Hilfe eines Sliders auf der Website die Zeit buchstäblich zurückzudrehen und den vorherigen Status wieder aufzurufen. Weiterhin besteht die Möglichkeit sämtliche Daten zu diesem Zeitpunkt zu betrachten und live zu verändern. Dadurch wird nicht nur der aktuell begutachtete Status verändert und der Effekt zeitgleich auf dem Bildschirm angezeigt, sondern auch alle folgenden Status mitsamt den Daten werden entsprechend aktualisiert. Evan Czaplicki hat dahingehend ein Video angefertigt, das den Vorgang sehr detailliert beschreibt ([LINK](#)). Alternativ kann die Abbildung XY betrachtet werden, die einen Zeitstrahl mit Werten, sowie die Möglichkeiten die sich durch den Debugger ergeben, aufzeigt.

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 wohl derzeit größten Nutzer sind NoRedInk, Prezi und CircuitHub. Alle Betriebe überführen Stück für Stück bereits bestehende Teile ihres Frontends zu Elm. NoRedInk gibt an, dass der überführte Elm-Code in den letzten acht Monaten keinerlei Laufzeitfehler erzeugt hat, anders als die vorherige Implementierung

(<https://www.youtube.com/watch?v=zBHB9i8e3Kc>). 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 können¹. Dazu gehört unter anderem Tetris, Pong und Space Invaders. Weiterhin bietet Elm sehr einfache Möglichkeiten Formen wie Kreise, Vierecke, Hexagone und vieles mehr zu erzeugen, ohne großartig in die Mathematik einzusteigen. 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? „The best functional programming in your browser“ - Diese Aussage wird anhand verschiedener Bewertungskriterien überprüft. Im Zuge dessen wird eine Webseite mit kleineren Modulen in Elm erzeugt. Die fertige Webseite respektive der erzeugte Quellcode wird anhand der zuvor erstellten Bewertungskriterien ausgewertet. Auch die Beobachtungen während der Entwicklung, wie etwa unvorhergesehene Probleme, gehen mit in die Wertung ein.

2.2.4 Einführung in die Elm-Architektur

TODO: Evtl. Absatz Praktische Anwendungsgebiete hier rein mergen und abändern? ... Um eine Programmiersprache jedoch anzuwenden, ist es notwendig zuvor die Architektur kennenzulernen. Dieses Kapitel soll eine Einführung in die Programmiersprache Elm darstellen und die grundlegenden Funktionen näher bringen.

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 Algorithmus 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 2.7 ist das Tool abgebildet und zeigt beispielhafte Eingaben und Evaluierungen. Wie aus der Abbildung 2.7 ersichtlich wird, werden 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 *importString* importiert wurde, war der Fehler behoben. Die *elm – repl* erlaubt es, alle Bibliotheken die das Paket *core* mitliefert, zu importieren.

¹[3, vgl.]

```

→ ~ 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 2.7: Das Tool elm-repl in der Kommandozeile

Basisdatentypen

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

- `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 2.7 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 2.7 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 Datentyp* 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 man die

```

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 2.8: Ein Union-Type in Elm

Abbildung 2.8 ist klar erkennbar, dass die Funktion *storeStatus* nicht alle Fälle, die der Typ *Day* annehmen kann, behandelt werden. 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 2.8 könnte entsprechend erweitert werden, so dass die Tage *Montag* bis *Freitag* zusätzlich noch eine Öffnungszeit besitzen. Wie in Abbildung 2.9 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 2.8 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 gewährt bleibt.

```

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 2.9: Ein erweiterter Union-Type in Elm

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. 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 2.10 ist erkenn-

```

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

```

Abbildung 2.10: Ein Record in Elm

bar, wie ein *Record* mit initialen Werten erstellt wird. Lediglich das Gleichheitszeichen für die Zuweisung unterscheidet sich hierbei 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 einzige Möglichkeit einen *Record* zu verändern liegt darin, den Inhalt der Felder zu manipulieren. Zeile 7 der Abbildung 2.10 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. Des Weiteren ist es nicht möglich rekursive Felder zu erzeugen.

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 ist unabhängig von den anderen und kann einen geeigneten Datentyp annehmen. Die folgende Abbildung 2.11 zeigt, wie ein solches Tupel von einer Funktion zurückgegeben werden kann. Der Rückgabewert des bei-

```
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 2.11: Ein Tupel in Elm

spielhaften 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 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 2.12 zeigt beispielhaft die

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

Abbildung 2.12: 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 Kompiliervorganges 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 2.13 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 2.13: Eine Kontrollstruktur 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* liefern und somit einen *Bool*

liefern. In anderen Programmiersprachen wie beispielsweise *C++* evaluiert eine Bedingung zu *True*, wenn sie nicht 0 oder *false* ist. Am Beispiel in Abbildung 2.14 sieht man, 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 2.8 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 2.14: Kontrollstruktur in C++

Funktionen

Um in irgendeiner Art und Weise eine Interaktion mit den erstellten *Records* zu vollziehen, benötigt der Entwickler auch in Elm eine Funktionen. Solch ein Programmkonstrukt wird mittels eines Funktionsnamen und dem Gleichheitszeichen realisiert. Abbildung 2.15 zeigt, wie eine simple Addition zweier

```

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

```

Abbildung 2.15: Eine Funktion in Elm

Integer-Werte 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++ meist 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 ablaufen sollen 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

Funktionstitel aus und wird an Funktionen höherer Ordnung weitergereicht. *List.filter* ist eine solche Funktion und erwartet eine anonyme Funktion als Parameter, auf Basis derer eine Liste gefiltert wird. Aus der Abbildung 2.16 wird ersichtlich, dass die übergebene Liste, bestehend aus 4 Einträgen, auf die Präsenz des Buchstabens *a* hin überprüft wird. Dabei wird jeweils ein 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 2.15 und 2.16 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 (\rightarrow) voneinander getrennt. Wird eine Funktion anstelle eines einfachen Datentypen als Parameter übergeben, wird dies wie in Abbildung 2.16 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 2.15 lässt sich erken-

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

Abbildung 2.16: Anwendung einer anonymen Funktion in Elm

nen, 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 2.15 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 2.17 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 2.17: 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 2.17 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 2.18: Definition einer Funktion mit Typen Alias

Module

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

Abbildung 2.19: Mögliche Deklarationen eines Elm-Moduls

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 2.19 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 2.20 zeigt vier mögliche Arten das

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

Abbildung 2.20: 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 2.21 zeigt diese IDE mit einem typischen *Hello – World* Beispiel. Auf der linken Seite ist eigentliche IDE erkennbar,

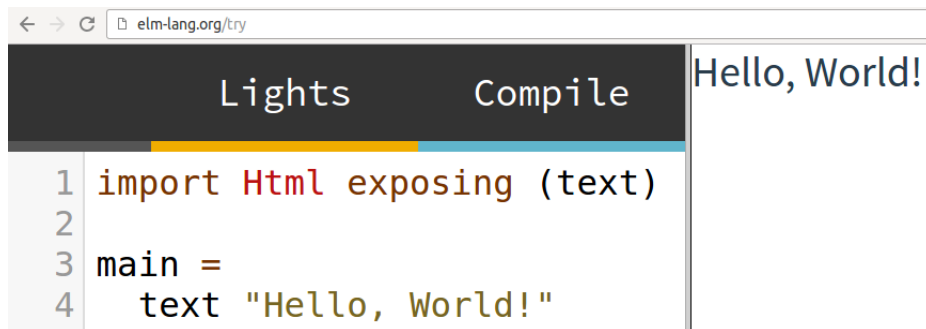


Abbildung 2.21: Die Online-IDE von Elm

während die rechte Seite das Ergebnis nach dem Kompilervorgang 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 des 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 2.22 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
```

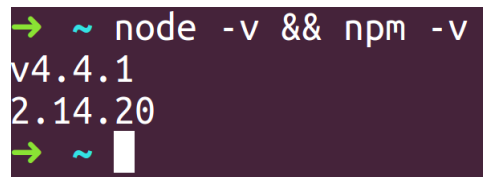
Abbildung 2.22: Installation des NPM

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 2.23 liefert ein Ergebnis,

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

Abbildung 2.23: Installation von Elm

wie es in Abbildung 2.24 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.



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

Abbildung 2.24: Überprüfung der erfolgreichen Installation von NodeJs und dem NPM

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* 1.2.3, wird es voraussichtlich kompatibel mit allen zukünftigen Versionen 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. TODO: [?]. Die Abbildung 2.25 zeigt, wie ein Paket installiert und der Projektord-

```
→ 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
```

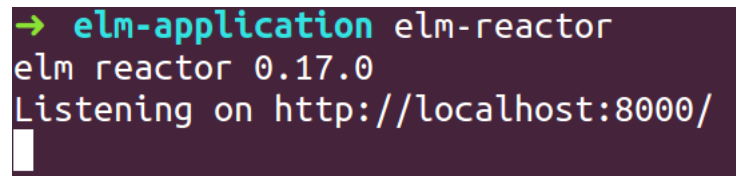
Abbildung 2.25: Installation eines externen Paketes über den Elm-Paketmanager

ner 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 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 2.26: Der gestartete Elm-Webserver

Erklärung des Elm-Reactors. Siehe Evaluierung, der auskommentierte Teil.

NULL/Undefined/Maybe in Elm?

Konzept des Maybe erklären, da es auch in der Applikation mit den initialen Übergabeparametern genutzt wird.

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), welches in HTML, CSS und JS programmiert wurde, in nativen Elm-Code überführt, insofern möglich. 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 erörterten Kriterien ausgewertet. Das Kapitel mündet schließlich in einem Fazit, in dem sämtliche Beobachtungen, die Auswertung anhand der Bewertungskriterien abschließend wertend zusammengefasst werden.

3.1 Bewertungsmuster

Das fertige Template ist eine SPA mit Elementen, wie sie typischerweise auf einer solchen Webseite vertreten sind. Eine SPA ist, wie der Name bereits suggeriert, eine Webseite mit effektiv nur einer aktiven Seite und ohne Unterseiten (ausgenommen Impressum, AGB und Datenschutz). 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
 - ScrollSpy zur Darstellung der aktuellen Position
- Titelbild mit einem vertikal und horizontal zentriertem Text
- Service-Sektion
- Twitter-Bootstrap-CSS mitsamt allen Funktionen

- Vorgefertigtes, responsive Design
- Aufklappbares Menü
- Portfolio mit Bildern, wobei ein Klick einen asynchronen Request ausführt und Daten nach lädt
- Formular zur Kontaktaufnahme mit Validierung der eingegebenen Daten

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 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 XY der Washington University zusammen. 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.

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 führen würden. Erkennt der *elm* – *compiler* diese Fehler oder umgeht den Absturz, gilt das Kriterium als erfüllt.

3.2.3 Portabilität

Nicht alle Programmiersprachen und die damit erzeugten Programme funktionieren auf allen Endgeräten. 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

„Zeit ist Geld.“ ist auch heute noch immer eine wahre Aussage. Entsprechend ist es von Vorteil, wenn Quellcode schnell erzeugt, getestet und als fertiges Produkt (Software) an den Kunden ausgeliefert werden kann. Dabei ist es wichtig, dass der Compiler den Quellcode schnell in ein lauffähiges Programm verwandelt und auch das daraus resultierende Programm 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 Caching durch den Compiler zu verhindern, wird vor jedem Kompiliervorgang die zuvor erzeugten Programmdateien gelöscht. Im Anschluss wird das arithmetische Mittel der verbleibenden acht Werte ermittelt. TODO. Die Performance der Programmiersprache wird nicht anhand des hier entwickelten Projektes, sondern der externen Applikation [?] *TodoMVC Performance Comparison* ermittelt. Dieses Tool ermöglicht es, anhand der Beispiel-Applikation "TodoMVC", die in mehreren Programmierspra-

chen realisiert wurde, die Performance des Programmes 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 Umgebungen der Applikation sind ebenso in nativem JS oder einer zu JS kompilierenden Programmiersprache geschrieben. Dadurch kann eine Aussage über die tatsächliche Effizienz der Applikation im Vergleich gemacht 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 als solches anhand der eingebauten Möglichkeiten die diese bietet objektiv beurteilen zu können. Da sich diese wissenschaftliche Arbeit allerdings ausdrücklich mit der Evaluierung von Elm für Webapplikationen auseinandersetzt, müssen auch diese Kriterien untersucht werden und einen höheren Stellenwert bei der Bewertung einnehmen. Webapplikationen können grundsätzlich in zwei Typen unterteilt werden. Zunächst gibt es die SPA, bestehend aus nur einer einzigen Seite, bei der typischerweise nur wenig Programmlogik vorhanden ist und der Client lediglich die Informationen anzeigt. Des Weiteren gibt es die Rich-Internet-Applikationen

(RIA: <http://www.itwissen.info/definition/lexikon/rich-Internet-application-RIA.html>).

Diese Art der Webapplikation besitzt im wesentlichen mehr Programmlogik, die der Client ausführen kann. Ein weiteres Merkmal sind oftmals auch die Anzahl der vorhandenen Unterseiten, wodurch wiederum mehr Logik erforderlich wird. Bei der Erstellung einer SPA muss generell weniger Aufwand betrieben werden, um eine fertige Präsentation zu erstellen. Jede Webapplikation lebt von einem Frontend, sowie einem Backend. Typischerweise bezeichnet das Frontend dabei alle Komponenten, die an den Benutzeroberfläche des Nutzers gesendet werden. Dazu gehören unter anderem die Komponenten HTML, CSS und JS. Der Nutzer interagiert mit dieser Darstellung der Oberfläche. Das Ge-

genstück zum Frontend stellt das sogenannte Backend dar. Dabei handelt es sich um Komponenten, die dem Webserver zugehörig sind. Unter anderem ist das zum Beispiel eine Datenbank, die eigentliche Webapplikation und natürlich der Webserver selbst. Der Nutzer agiert mit dem Backend nur über zuvor im Frontend eingerichtete Schnittstellen.

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 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 die fünf meistgenutzten Browser Google Chrome, Safari, Internet Explorer, Firefox und Opera (vgl. Abbildung 3.1).

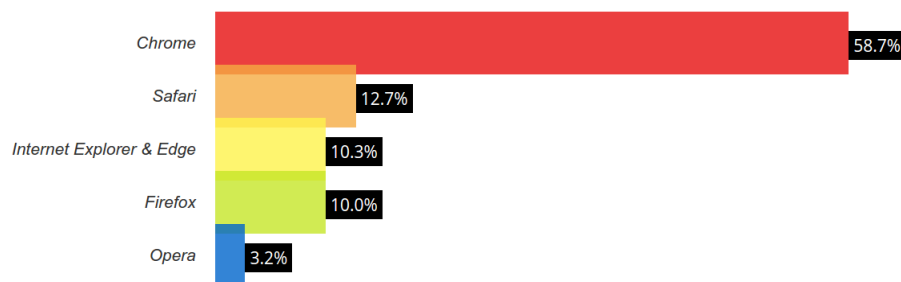


Abbildung 3.1: Die fünf meistgenutzten Browser im Mai 2016 [2]

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.

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, sollten sie tatsächlich angefordert werden. 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. Da initial gewisse Daten noch nicht vorhanden sind, verringert sich die Datenmenge der zu übertragenden Informationen. Dadurch kann der Webserver entlastet und die initiale Ladezeit der Webseite verringert werden. Um das Kriterium der asynchronen Verarbeitung zu erfüllen, sollte Elm die Möglichkeit liefern, asynchrone Requests an einen Server zu schicken, Daten anzufordern und die Antwort darzustellen. Die Applikation darf dabei nicht in einen undefinierten Zustand kommen, sprich durch beispielsweise eine langsame Übertragungsrate Fehler bei der Darstellung erzeugen oder die Ausführung von nachfolgenden Operationen verzögern.

3.3.4 Dateigröße

Die Größe einer Datei wirkt sich auf die Dauer der Übertragungszeit aus. Ist die Datenmenge groß, steigt entsprechend auch die Dateigröße an. Eine schlechte Übertragungsrate kann somit zu erheblichen Verzögerungen der fertigen 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* 150 Kilobyte nicht übersteigen.

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 Entwicklungsumgebung unter der die Applikation ausgearbeitet wurde vorgestellt. Die Entwicklung wird in einzelne Abschnitte unterteilt und einzeln erläutert.

3.4.1 Programmablauf

Abbildung XY: Zeigt die Kommunikation zwischen Client und Webserver Die Abbildung XY 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.

3.4.2 Entwicklungsumgebung

Die zugrunde liegende Elm-Version ist 0.17 und wird unter Ubuntu 14.04 64bit installiert. Als Editor wird Atom mit den Elm-spezifischen Plugins *language – elm*, *linter – elm – make* und *elm – format* 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, 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 XY), Mozilla Firefox (Version 47) und Opera (Version 38). In Bezug auf die Abbildung 3.1 fällt auf, dass Safari als einziger der fünf meistgenutzten Browser nicht getestet wird. Zum Zeitpunkt dieser wissenschaftlichen Arbeit stand kein entsprechendes Testgerät zur Verfügung. Die in der Abbildung genannten Browser decken 94,9% aller genutzten Browser ab. Abzüglich des Browsers Safari, werden entsprechend die verbleibenden 82,2% aller Browser überprüft.

TODO: (<https://github.com/triforkse/atom-elm-format>)
(<https://github.com/triforkse/atom-elm-format>)
(<https://atom.io/packages/language-elm>)

3.4.3 Grundaufbau

Damit der Nutzer letzten Endes eine Webseite besuchen kann, muss zunächst ein auslieferbares Dokument erzeugt werden. Hierfür wird zunächst die *index.html*-Datei aus dem Template *Agency* 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 Code der für die Darstellung des Templates innerhalb des *body*-Elementes definiert wurde entfernt. Lediglich die *script*-Tags bleiben innerhalb des *body*s bestehen, so dass sämtliche vordefinierte JS-Dateien weiterhin geladen werden.

Damit in das Grundgerüst die Elm-Applikation geladen werden kann, ist ein weiterer *script*-Tag notwendig, der die kompilierte Elm-Applikationsdatei *elm.js* einbindet. Des Weiteren muss die Elm-Applikation explizit aufgerufen und gestartet werden. Die Abbildung 3.2 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
3. *worker*: Initialisiert die Applikation ohne grafische Benutzeroberfläche

Die Abbildung 3.2 zeigt, dass für dieses praktische Beispiel Version 2 genutzt wird. Diese Version bietet den Vorteil, dass eine Elm-Applikation gezielt in einen vordefinierten Bereich einer gesamten Webseite platziert werden kann. Auf diese Weise kann eine Elm-Applikation problemlos in eine bestehende Webseite eingebaut und erweitert werden. Ferner erlaubt diese Art der Injizierung zusätzliche externe JS- und CSS-Dateien hinzuzufügen. 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

```

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 3.2: 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 3.3: Grundgerüst der Elm-Applikation

fullscreen-Applikation gestartet und ist für die Zwecke dieser Arbeit nicht geeignet, weswegen auf die manuell Kompilierung 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 einerseits die Funktionen um HTML-Elemente zu erzeugen, andererseits drei 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. Dabei variieren stets die Übergabeparameter, wodurch die Applikation leicht erweitert und komplexer werden kann. So verlangt die Funktion *beginnerProgram* nur 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 3.4 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 3.4: 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 3.3 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 3.3 wird ersichtlich, dass zunächst das gesamte Paket *elm – lang/html* importiert wird. Die vorherige Installation des Pakets ist analog zu der in der Abbildung 2.25 beschriebene Vorgehensweise. Des Weiteren wird der Grundaufbau der Elm-Applikation ersichtlich. Inner-

halb der *main*-Funktion werden dabei 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 Ausweich-Model (*initialModel*) erstellt, falls keine Daten übergeben wurden. Die Funktion *view* erzeugt in der Abbildung 3.3 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* auf der Webseite <http://mbylstra.github.io/html-to-elm/> zur Hilfe genommen. Der gesamte zuvor im *body*-Tag befindliche HTML-Code wird kopiert und zur Konvertierung in das entsprechende Textfeld des Tools eingefügt. Der konvertierte Elm-Code sollte daraufhin auf der gegenüberliegenden Seite zu sehen sein. Durch einen Klick auf *copy* wird der erstellte Elm-Code in die Zwischenablage kopiert, so dass der Inhalt in die *view*-Funktion der *Main.elm*-Datei eingefügt werden kann. Der kopierte Text fungiert dabei als kompletter Rückgabewert der *view*-Funktion. Das Tool *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 3.5: Elm-Compilerfehler bei fehlenden Funktionen im Namensraum

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 3.5 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 3.3 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.

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* (<https://learn.jquery.com/using-jquery-core/document-ready/>) erst nach dem kompletten Seitenaufbau zu laden. Innerhalb der Datei *agency.js* muss nicht nur die Funktion *\$('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 3.6 zu sehen ist. Um diesen zu

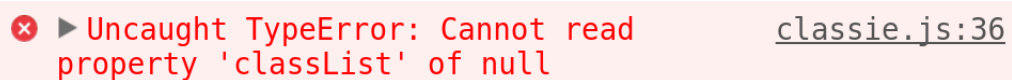


Abbildung 3.6: JavaScript-Fehler innerhalb der Developer-Tools

beheben, muss der gesamte Dateiinhalt in eine *\$(document).ready()*-Funktion geschachtelt werden. Zusätzlich dazu ist es notwendig, dass die Codezeile wie sie in Abbildung 3.7 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 3.7: 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 3.6 zustande kommt. Nachdem diese beiden Fehler behoben wurden, kann nun die *Scroll – Spy*-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 3.8 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 3.8: Deklaration einer Sektion des Views in Elm

Sektion des zu überführenden Templates sehen die daraus resultierende Funktion teilweise aus wie in Abbildung 3.9. Die einzelnen Klassen und ID's 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 3.10 betrachtet werden. Das Resultat der *view*-Funktion unverändert aus, ist jedoch nun im Code übersichtlicher.

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 Modals 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

```

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 3.9: 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 3.10: Aufruf der einzelnen Sektionen im View

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 3.11 zu sehen ist. Dabei wird zunächst ein *Ereignisbehandler* an die Link-Elemente mit der Klasse *page – scroll* gebunden. Klickt der Nutzer 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, da sich dieser nicht Teil der Elm-Applikation ist. 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

```

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 3.11: Funktion zum scrollen bei einem Klick

JS-Skripten herrscht. Die Elm-Bibliothek *Html.Events* enthält die Methode *onWithOptions*, mit Hilfe derer ein Klick auf ein Element des *views* in Elm eine *Msg* erzeugt. Diese *Msg* wird an die *update*-Funktion weitergeleitet und kann dort entsprechend ausgewertet werden. Bislang gibt es in der Elm-

```

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

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 3.12. 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 3.12 zu sehen ist zu kompilieren, wird der Compiler mit der Fehlermeldung "You are declaring port *sendClickedElemToJs* in a normal moduleantworten. Die Fehler-

meldung 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 den Zusatz der *onWithOptions*-Funktion, sowie den Übergabeparametern *ClickedElem* und dem *href*-Wert erweitert werden. In Abbildung 3.13 kann beispielhaft die Erweiterung an einigen Stellen begutachtet werden. Die Funktion *onWithOptions* bekommt dabei zunächst einmal das *Event*, auf das sie einen Ereignishandler 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 Ergebnisbehandler in denen der Link geschachtelt ist ausführen. Dies würde zu einem undefinierten Verhalten führen. Der letzte Parameter ist die 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

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

Abbildung 3.13: Erweiterung des Views um einen *onWithOptions* Ereignisbehandler in Elm

notwendige Erweiterung 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 Codestelle die für die Initialisierung der Elm-Applikation zuständig ist die Funktion aus Abbildung 3.14 hinzugefügt. Durch das Kommando *app.ports.sendClickedElemToJs.subscribe* 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.


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

Abbildung 3.14: Entgegennahme der Daten aus der Elm-Applikation und anschließendes scrollen

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 verschieben. Eine solche Strukturierung hilft dabei, die womöglich fehlerbehafteten Teile der Applikation schneller zu finden und die einzelnen Bereiche der Applikation deutlicher voneinander zu trennen. 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 an den entsprechenden Stellen 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 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 3.16 und 3.15 zeigen die *view*-Funktion des Hauptmoduls, sowie den Aufruf um die angelegten Module zu importieren. Letzten 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.

```

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 3.15: Einbindung und Aufruf der ausgelagerten View-Funktionen

```

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 3.16: Einbindung und Aufruf der ausgelagerten View-Funktionen

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.

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) (<https://api.icndb.com>) genutzt. Sie liefert bei jeder Anfrage einen zufälligen Witz 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 ein zufälliger Witz präsentiert werden. Dafür wird über die externe API ein zufälliger String angefordert, vom Server generiert und letztlich an die Elm-Applikation zurückgegeben. 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 und das Ergebnis in den View einzuarbeiten. Dafür 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 3.17 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 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 3.18 zeigt die hinzugefügten *Msg*-Typen. Nachdem die neuen Interaktionsmöglichkeiten in der Elm-Applikation implementiert wurden, muss nun der eigentliche Request ausführbar gemacht

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

Abbildung 3.17: OnClick-Ereignisbehandler in Elm

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

Abbildung 3.18: *Msg*-Deklaration mit asynchronem Request

werden. Die Funktion *GetRandomString* soll dabei ein unverändertes Modell, 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 3.19 erkennbar ist. Dabei wird die Funktion

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

Abbildung 3.19: Asynchroner HTTP-Request in Elm

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 3.20 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 Modell mit dem

```

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

```

Abbildung 3.20: 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 3.21: *Update*-Fälle für den asynchronen Request

Fällen ist in Abbildung 3.21 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 3.22 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 des *Popup*-Moduls mittels des Kommandos *text model.async_content* angezeigt wird.

3.4.9 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.



Abbildung 3.22: Ausgehender HTTP-Request und eingehende Antwort

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 3.23 zeigt ein *div*-Element,

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

Abbildung 3.23: 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 in 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 2.701 angemeldeten Nutzern [4, vgl.], gibt es noch die offizielle Google-Gruppe. Offiziell gibt es derzeit 205 Elm-Bibliotheken [5, 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 [6, vgl.] die in, oder mit Elm realisiert wurden. Ferner wurden 6.365 Probleme in diesen Projekten gemeldet, von denen 4.834 bereits gelöst und 1.531 noch geöffnet sind. 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 einem der genannten Medium meist in wenigen Minuten aushelfen. War dies nicht der Fall, wurden stets weitere Nutzer kontaktiert, bis das Problem endgültig gelöst wurde.

3.4.10 Auswertung

- Das bedeutet, dass nativer Code in Elm kürzer und weniger anfällig für Flüchtigkeitsfehler wie beispielsweise das Schließen eines Tags ist. Der Entwickler wird weniger syntaktische Fehler machen.

1.1. Elm hat leicht erlernbare Grundkonzepte, die adaptiert und erweiterbar sind, um Produktivität für die Entwickler zu gewährleisten.

2.1. Elm Code kann mindestens kommentiert werden, wenn nicht sogar eine Funktion zur automatischen Generierung von Signaturen einer Funktion existiert -> Kommentare können mit ‘-’ (single-line) oder ‘- a comment -’ erstellt werden. Eine vollautomatische, explizite Erstellung der Signaturen besteht derzeit nicht. Der Compiler wird jedoch nach dem Kompilervorgang einen Signaturvorschlag geben. Dieser Vorschlag ist auf einer niedrigeren Ebene, als es eine Deklaration wäre, ist jedoch ausreichend und wesentlich sinnvoller als eine Signatur komplett unberücksichtigt zu lassen. Beispiel: Abbildung XY (*compiler_{sig}suggestion* + *compiler_{own}sig*) ““ type alias Model = counter: Int increaseModel: Model -> Model increaseModel model = model | counter = model.counter + 1

vs. Kompilervorschlag type alias Model = counter: Int increaseModel: a | counter : number -> a | counter : number increaseModel model = model | counter = model.counter + 1 ““

3.1. Die Erweiterungen des Editors oder der Compiler warnt spätestens bei der Kompilierung, wenn nicht bereits während der Entwicklung vor syntak-

Kriterium	Erfüllt
1. Entwicklungsgeschwindigkeit 1.1	✓
2. Wartbarkeit 2.1	✓
3. Zuverlässigkeit 3.1 3.2 3.3	✓ ✓ ✓
4. Portabilität 4.1	ausstehend
5. Effizienz 5.1 5.2	✓ ausstehend
6. Wiederverwendbarkeit 6.1	✓
7. Modularität 7.1 7.2	✓ ✓
8. Lesbarkeit 8.1	✓
9. Dateigröße 9.1	ausstehend
10. Interoperabilität 10.1 10.2	✓ Nein/Nur bedingt
11. Asynchrones Laden 11.1	✓

Tabelle 3.1: Auswertung der Versuchskriterien

tischen Fehlern → Syntaktische Fehler werden vom Compiler erkannt und ein Fehler angezeigt. Der Fehler wird unterstützt durch einen Hinweis auf die mögliche Fehlerquelle (Abbildung XY – *Compiler_{syntax}error*). Die Fehlermeldung wird erst nach der Kompilierung eingeblendet. ““ div [id elm-view"] [Navigation.view model Header.view , Services.view] ““

3.2. Gibt es keine Fehlermeldungen, wird die Applikation ohne Laufzeitfehler funktionieren → Aufgrund der Garantien, dass es keine Seiteneffekte gibt und alle Variablen unveränderlich sind, ist die Korrektheit syntaktische und semantische Korrektheit des Codes gewährleistet. Lediglich die logische Implementierung eines Algorithmus kann noch Fehler aufweisen, dies kann jedoch nicht von einem Compiler überprüft werden. 3.3. Fehlermeldungen werden sehr spezifisch auf das eigentliche Problem hinweisen → Wie in Abbildung iii.i zu sehen ist, gibt der Compiler nicht nur an, in welcher Zeile ein Fehler gefunden wurde, sondern liefert zusätzlich noch einen Hinweis, worin eine wahrscheinliche Fehlerquelle liegt. In diesem Beispiel weist der Compiler auf das fehlende

Komma als Fehlerquelle hin.

4.1 1. Die SPA wird auf allen gängigen Browsern fehlerfrei dargestellt -> AUSSTEHEND

5.1. Der Kompilervorgang wird nur wenige Sekunden dauern, jedoch mehr Zeit in Anspruch nehmen, als die herkömmliche Entwicklung mit reinemHTML,CSS undJS -> Bei der herkömmlichen Entwicklung ist kein expliziter Kompilervorgang notwendig, stattdessen übernimmt der Browser die Darstellung des übergebenenHTML,CSS undJS Quellcodes. Der Kompilervorgang der Elm-Applikation dauert wie in Abbildung XY zu sehen ist im Durchschnitt XY Sekunden. Es wurde 10x gemessen, der höchste und niedrigste Wert gestrichen und dann das arithmetische Mittel der verbleibenden 8 Messungen ermittelt.

5.2 2. Die erzeugte Elm-Applikation ist deutlich schneller während der Laufzeit -> AUSSTEHEND (Abbildung - Performance - Evtl. neu?)

6. 1. Codeteilen lassen sich problemlos auslagern und wiederverwenden -> Der komplette View wurde von einer großen Darstellung in mehrere Funktionen unterteilt und letztlich in ein eigenes Modul überführt. Analog dazu wurden auch das Model, sowie die Update-Funktion ausgelagert. Das einmalig definierte 'Model' kann in sämtlichen Modulen wiederverwendet werden, auch die einzelnen Funktionen eines jeden Moduls finden erneut Verwendung, insofern gewünscht.

7.1. Ausgelagerte Codeteile sind isoliert voneinander und als einzelnes Modul nutzbar -> Diese Eigenschaft ist gegeben. 7.2. Module können Funktionen nach außen verbergen -> Jedes Modul kann mit dem Stichwort 'expose' einzelne Funktionen nach außen hin zugänglich machen. Das importierende Modul wiederum kann einzelne Funktionen in den derzeitigen globalen 'Scope' laden.

8.1. Die automatische Formatierung des Elm-Codes macht diesen lesbarer und spart Zeit -> Fehler werden Dank der Formatierung schnell sichtbar.

9.1. Die Dateigröße der Elm-Applikation wird kleiner als bei vergleichbaren Frameworks ausfallen -> Die kompilierte Elm-Applikation hat eine Größe von etwa 300kb. Allerdings kann diese Größe um etwa 60Prozent verringert werden, wenn der Code 'minified' wird.

10.1. BestehendeJS-Skripte können mit der Elm-Applikation interagieren oder funktionieren bereits einwandfrei -> Die Integration vorhandenerJS-Skripte verlief problemlos. Benötigt ein externes Skript Daten von der Elm-Applikation, so kann mit Hilfe von 'Ports' eine gesicherte Kommunikation in beide Richtungen stattfinden.

10.2. BestehenderCSS-Code kann nativ in Elm eingebunden werden -> Das Einbinden vonCSS Quellen ist derzeit nicht ohne Probleme nativ in Elm möglich. Es verursacht ein „flackern“ und ist nicht für den Gebrauch in einem fertigen System geeignet.CSS-Klassen und 'inline-styling' sind jedoch nativ in Elm implementiert und erlauben ein Styling der Elemente. ExternesCSS muss jedoch über das Grundgerüst derHTML-Datei eingebunden werden.

11.1. Elm erlaubt asynchrone Requests ohne Seiteneffekte zu erzeugen -> Ein asynchroner Request stellt implizit einen Seiteneffekt dar, wird in Elm allerdings als gemanagter Seiteneffekt bezeichnet. In jedem Fall ist der Nutzer gezwungen eine Funktion bei einer fehlerhaften Übertragung zu übergeben, die

dann die erwartenden Typen/Werte liefert.

3.4.11 Fazit

1. Installation sehr einfach 1. externe Abhängigkeiten werden automatisch installiert 2. Installation war schnell 3. Erklärung zur Installation war deutlich 2. Modularität möglich 1. Jedes View-Element als eigenes Modul 1. Paralleles arbeiten möglich 2. Entwickler bekommen nur die notwendigen Informationen für das Interface 3. Tools 1.HTML2ELM sehr hilfreich 1. Kleinere Fehler 1. ' verursacht Fehler (Weiterfolgende Dirs wurden vernachlässigt) 2. required -> required " (wants Bool, got String); novalidate same 2. Elm-Compiler 1. Fehler vonHTML2ELM (2. Punkt) wurden sofort erkannt 2. Fehlende Funktionen fürHTML undCSS Attribute wurden in den Fehlermeldungen vorgeschlagen („Did you mean Html.Attributes.required?“ für „required true“) 4. Erklärungen der Elm-Struktur in den Guides sehr gut

4 Fazit

TODO:

1. Gegenüberstellung
 - Was war das Ziel und der Wissensstand zu Beginn
 - Was waren die Erwartungen
 - Gab es unvorhergesehene Probleme? Wenn ja, welche?
 - Entsprach das Endergebnis (und der Weg dorthin) dem vorherigen Ziel?
2. Aufzählung der maßgeblich wichtigen Punkte
3. Résumé
 - Erfüllt Elm die Anforderungen der Webentwicklung?
 - Einstufung, für welche Anwender Elm geeignet ist
 - Aktuell (noch) bestehende Probleme, die es zu lösen gilt

5 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)

Literaturverzeichnis

- [1] CZAPLICKI, Evan: TodoMVC Benchmark. In: *Blazing Fast HTML Virtual DOM in Elm* (2014), Juli
- [2] LLC., AWIO WEB S.: *Web Browser Market Share*. Webseite, Juni 2016. – Online einsehbar unter <https://www.w3counter.com/globalstats.php?year=2016&month=5> - abgerufen am 06. Juni 2016
- [3] WESTBY, Luke: *builtwithelm*. Webseite, Dezember 2015. – Online einsehbar unter <http://builtwithelm.co/> - abgerufen am 07. Juni 2016
- [4] *Elm-Slack Nutzer*. <http://elmlang.herokuapp.com/>
- [5] *Elm Packages*. <http://package.elm-lang.org/>
- [6] *Github Elm-Repositories*. <https://github.com/search?utf8=%E2%9C%93&q=language%3Aelm&type=Repositories&ref=searchresults>

TODO: Literatur hinzufügen + Referenzen + importieren