



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	5
2 Theoretische Grundlagen	6
2.1 Funktionale Programmiersprachen	6
2.2 Grundlagen der Programmiersprache Elm	7
2.2.1 Geschichte	7
2.2.2 Konzept	8
2.2.3 Umsetzung	9
2.2.4 Einführung in die Elm-Architektur	14
3 Evaluierung der Programmiersprache Elm	27
3.1 Bewertungsmuster	27
3.2 Bewertungskriterien	28
3.2.1 Entwicklungsgeschwindigkeit	28
3.2.2 Wartbarkeit und Lesbarkeit	28
3.2.3 Zuverlässigkeit	29
3.2.4 Portabilität	29
3.2.5 Effizienz	29
3.2.6 Wiederverwendbarkeit	30
3.3 Ansprüche an eine Webapplikation	30
3.3.1 Browser Kompatibilität – Portabilität	31
3.3.2 Interoperabilität	31
3.3.3 Asynchrones Laden	32
3.3.4 Dateigröße	32
3.4 Empirische Analyse	32
3.4.1 Programmablauf	32
3.4.2 Entwicklungsumgebung	33
3.4.3 Hypothesen und Vermutungen	33
3.4.4 Versuchsdurchführung	33
3.4.5 Beobachtungen	40
3.4.6 Auswertung	41
4 Fazit	45
5 Eidesstattliche Erklärung	46

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

Abbildungsverzeichnis

2.1	Eine iterative Funktion	6
2.2	Eine rekursive Funktion	7
2.3	Ein simpler Zähler auf einer Webseite	8
2.4	Das Model-View-Update Konzept von Elm	9
2.5	Beispiel der dynamischen Typisierung	11
2.6	Performanz von Elm im Vergleich zu anderen Web-Frameworks. [1]	12
2.7	Das Tool elm-repl in der Kommandozeile	15
2.8	Ein Union-Type in Elm	16
2.9	Ein erweiterter Union-Type in Elm	17
2.10	Ein Record in Elm	17
2.11	Ein Tupel in Elm	18
2.12	Eine Variablendeklaration in Elm	19
2.13	Eine Kontrollstruktur in Elm	19
2.14	Kontrollstruktur in C++	20
2.15	Eine Funktion in Elm	20
2.16	Anwendung einer anonymen Funktion in Elm	21
2.17	Definition einer Funktion ohne Typen Alias	22
2.18	Definition einer Funktion mit Typen Alias	22
2.19	Mögliche Deklarationen eines Elm-Moduls	22
2.20	Mögliche Formen der Importierung eines Elm-Moduls	23
2.21	Die Online-Integrated Development Environment (IDE) von Elm	24
2.22	Installation des Node Package Manager (NPM)	24
2.23	Installation von Elm	25
2.24	Überprüfung der erfolgreichen Installation von NodeJs und dem NPM	25
2.25	Installation eines externen Paketes über den Elm-Paketmanager	26
3.1	Die fünf meistgenutzten Browser im Mai 2016	31
3.2	Grundgerüst einesHyperText Markup Language (HTML)-Dokumentes, um die Elm Applikation zu laden	35
3.3	Der gestartete Elm-Webserver	35
3.4	Eine beispielhafte Initialisierung der Elm-Applikation mit über- gebenen Werten	37
3.5	Resultat des Elm-Codes in HTML	37
3.6	Deklaration einer Sektion des Views in Elm	38

Tabellenverzeichnis

3.1	Aufstellung der Hypothesen	34
3.2	Auswertung der Versuchskriterien	42

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

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

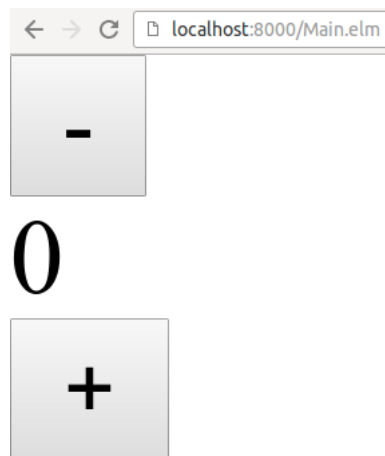


Abbildung 2.3: Ein simpler Zähler auf einer Webseite

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.

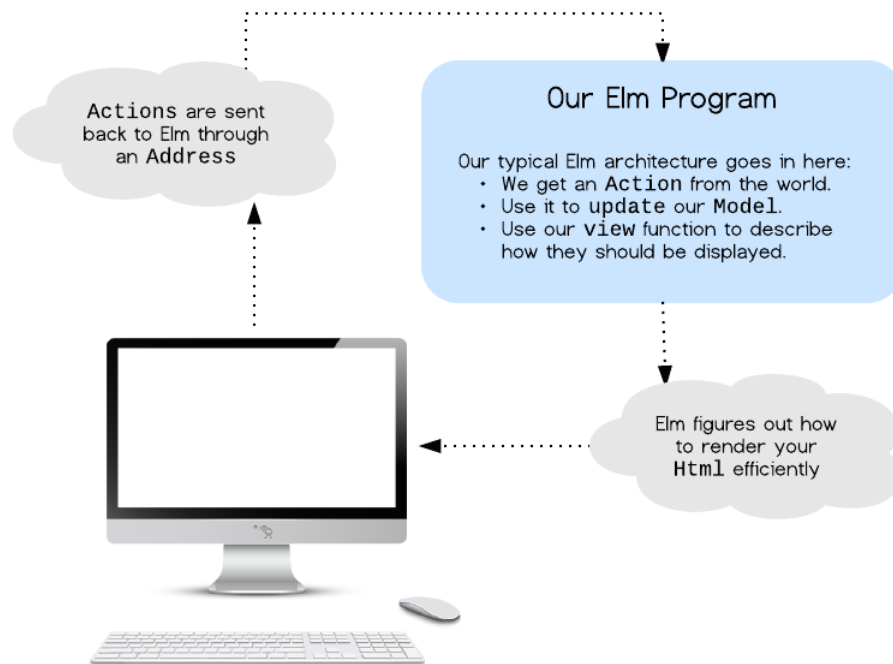


Abbildung 2.4: Das Model-View-Update Konzept von Elm

2.2.3 Umsetzung

Der vom Programmierer verfasste Programmcode wird vor der endgültigen Nutzung zu JavaScript (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

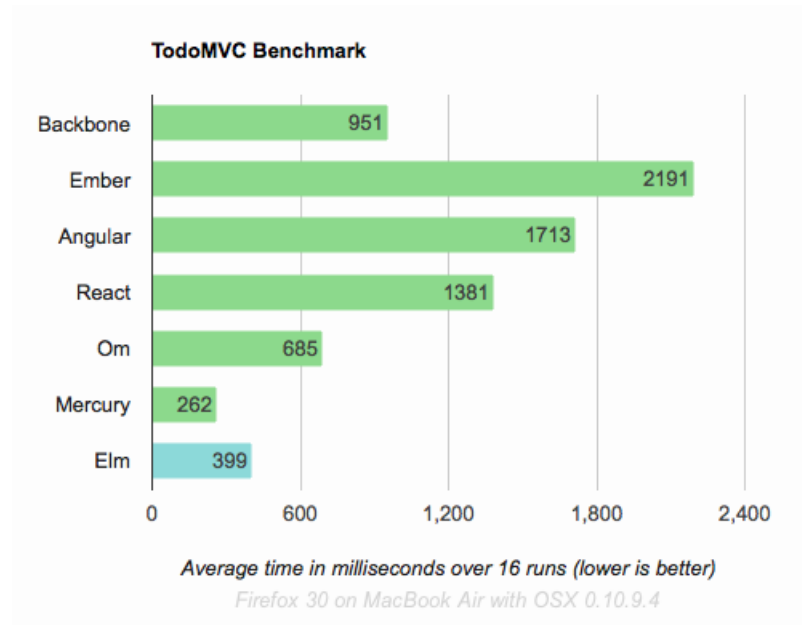


Abbildung 2.6: Performanz von Elm im Vergleich zu anderen Web-Frameworks. [1]

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[2, 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,

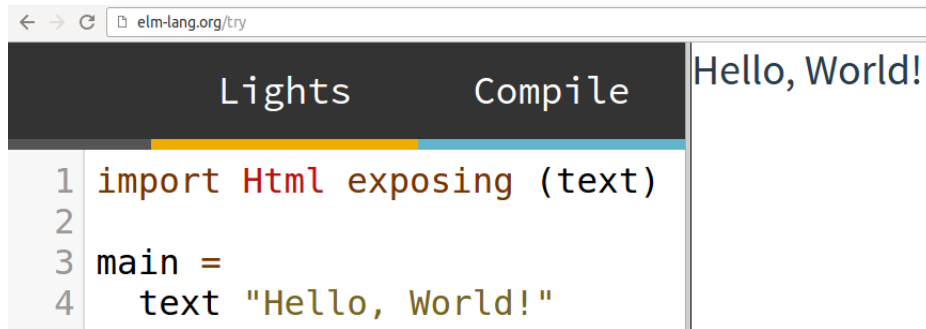


Abbildung 2.21: Die Online-IDE von Elm

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
```

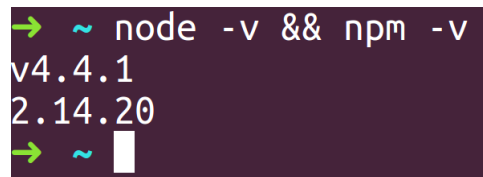
Abbildung 2.22: Installation des NPM

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.

3 Evaluierung der Programmiersprache Elm

3.1 Bewertungsmuster

Der folgende Teil dieser wissenschaftlichen Arbeit befasst sich mit der Entwicklung eines Modells, um die Programmiersprache Elm und ihre Nutzung als Frontend für Webapplikationen bewerten zu können. Dabei soll ein bereits fertiges Template von den ursprünglichen Programmiersprachen HTML, CSS und JS in die Programmiersprache Elm portiert werden. Während der Entwicklung/Portierung wird die Programmiersprache Elm gegenüber mehreren Gesichtspunkten verglichen und ausgewertet. Das fertige Template ist eine Single Page Application (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 + Funktionen
 - Vorgefertigtes, responsive Design
 - Hamburger Menü
- Portfolio mit Bildern, wobei ein Klick einen asynchronen Request ausführt und Daten nach lädt
- Formular zur Kontaktaufnahme mit Validierung der Korrektheit 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 für einen Newsletter anmelden. Die Eingaben des Nutzers sollen auf Richtigkeit überprüft werden.

3.2 Bewertungskriterien

Während der Bearbeitung, respektive 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 Entwicklungsgeschwindigkeit

Gerade für Startups ist es wichtig, ein erstes Produkt so schnell wie möglich bereitzustellen. Die Entwickler müssen entsprechend mit wenigen Mitteln ein fertiges (Software-)Produkt ausliefern können, selbst wenn sie noch kein oder wenig Vorwissen zu einer Programmiersprache haben. Dementsprechend sollte eine Programmiersprache nur eine geringe Anzahl an primitiven Konzepten aufweisen, die sich leicht erweitern lassen. Beispielsweise hat die Programmiersprache C standardmäßig keine Anreihung von Buchstaben, auch bekannt als *String*. Jedoch gibt es den Datentyp *char*, der einen einzelnen Buchstaben repräsentiert. Durch die Verknüpfung mehrerer *char*'s, kann letztendlich der Datentyp *String* umgesetzt werden. TODO: Weg, oder re-write. Bibliotheken erlauben üblicherweise schnelleres/leichteres programmieren, aufgrund der Sammlung von Algorithmen.

3.2.2 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.

3.2.3 Zuverlässigkeit

Für einen Entwickler von Software, egal wofür und mit welcher Programmiersprache gearbeitet wird, 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 unterstreichen 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.

3.2.4 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 *node.js* unterstützen. Gibt es keinerlei Differenz in Form von Fehlern oder Warnungen während des Kompilierens, gilt die Portabilität als erfüllt.

3.2.5 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. 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 Programmiersprachen realisiert wurde, zu testen. Dabei werden automatisch gewisse Aktionen in der jeweiligen Sprache durchgeführt, die benötigte Zeit gemessen und 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.

3.2.6 Wiederverwendbarkeit

Es ist sehr ineffizient vorhandene Codestücke neu zu verfassen, oder händisch kopieren zu müssen. Besser ist es, wenn Funktionen mehrfach genutzt werden können. Auf diese Weise müssen Änderungen auch 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 Ansprüche an eine Webapplikation

Diese 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 Gegenstück 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 – Portabilität

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. Natürlich muss die Applikation letzten Endes auch auf den gängigen Browsern fehlerfrei funktionieren, insbesondere die fünf meistgenutzten Browser Google Chrome, Safari, Internet Explorer, Firefox und Opera (vgl. Abbildung 3.1).

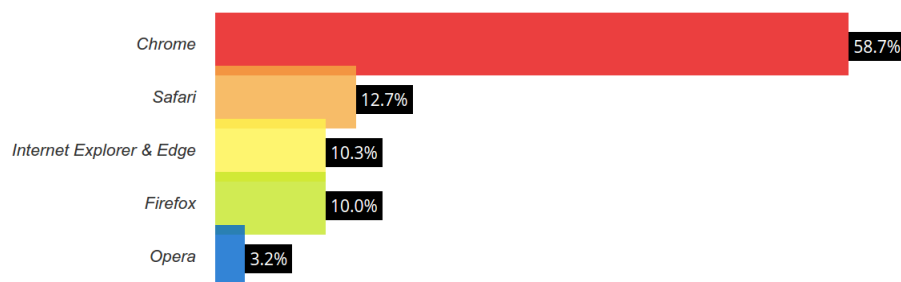


Abbildung 3.1: Die fünf meistgenutzten Browser im Mai 2016

3.3.2 Interoperabilität

Auch bei den Webapplikationen ist es wichtig, bereits existente Frameworks und Problemlösungen nutzen zu können. Dementsprechend müssen externe JS und CSS Bibliotheken ohne große Probleme eingebettet werden können. Dies kann bequem über die vorhandenen HTML-Tags und ihre Attribute gemacht werden.

3.3.3 Asynchrones Laden

Nicht immer ist es sinnvoll, sämtliche Daten die auf einer Webseite angezeigt werden auch direkt an den Nutzer zu schicken. Sind beispielsweise große Datenmengen erst nach mehrmaligem Scrollen durch den Nutzer sichtbar, kann es ausreichen die Daten asynchron nachzuladen, sobald der Viewport, sprich der für den Nutzer sichtbare Bereich, sich diesem Inhalt nähert. Auf diese Weise kann der Server entlastet und die Ladezeit für den Nutzer verringert werden. Dieses Verhalten wird „Lazy Load“ genannt (XY). Besonders wenn davon auszugehen ist, dass der Nutzer nur einen sehr eingeschränkten Viewport oder eine schlechte Internetverbindung hat, bietet sich das asynchrone nachladen von Daten an.

3.3.4 Dateigröße

Die Größe einer Datei wirkt sich auf die Dauer der Übertragungszeit aus. Ist die Datei groß, müssen entsprechend viele Informationen vom Server auf den Client des Nutzers übertragen werden, um die Webseite ansprechend darzustellen. Die Dateigröße kann mit einfachen Mitteln verringert werden. Beispielsweise können alle nicht unbedingt notwendigen Zeichen aus der Datei entfernt werden. Dazu gehören Zeichen wie „Leerzeichen“ (Whitespaces) oder Zeilenumbrüche. Doch auch die Verkürzung von Variablen ist denkbar. Anstelle von „EinZiemlichLangesWort“ reicht beispielsweise „a“ komplett aus. Offensichtlich muss die Variable im gesamten Code entsprechend umbenannt werden. Hierzu bedarf es Algorithmen, welche die Nutzung von Variablen genau untersuchen und Fehler durch Ersetzungen vermeiden.

3.4 Empirische Analyse

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 nun eine vollständige Präsentation der Webseite sieht. Schritt 5 beschreibt die mögliche Interaktion des Nutzers mit dem ihm präsentierten Dokument. Jede Interaktion führt zu einer Reaktion durch JS, oder erzeugt einen neuen Request an den Server, womit erneut bei Schritt 1 begonnen wird. Das Einbinden von CSS und JS aus externen Quellen wird durch Schritt 6 verbildlicht. Die gesamte Applikation wird von einem fertigen Tem-

plate, programmiert in HTML, CSS und JS in die Programmiersprache Elm überführt, insofern möglich. Der daraus entstehende Elm-Quellcode wird daraufhin mit dem Elm-Compiler kompiliert zu JS. Um ein generelles Verständnis der Programmiersprache Elm zu bekommen, dient die Dokumentation auf <http://elm-lang.org/docs> als Hilfestellung. Dort besteht die Möglichkeit sich mit den grundlegenden Konzepten vertiefend auseinanderzusetzen und diese zu erlernen.

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 mehreren Elm-Plugins verwendet, so dass der geschriebene Code automatisch beim Speichern formatiert und durch den Compiler auf Fehler überprüft wird. Lauffähig wird die geschriebene Applikation mit dem Elm-Compiler *elm – make* gemacht und mit Google Chrome Version 50 getestet.

3.4.3 Hypothesen und Vermutungen

Sämtliche Hypothesen wurden in der Tabelle 3.1 zusammengefasst. Dabei beziehen sich die einzelnen Vermutungen jeweils auf eines der zuvor erläuterten Kriterien in den Kapiteln Bewertungskriterien (3.2) und Ansprüche an eine Webapplikation (3.3). Die Nummerierungen der Hypothesentabelle 3.1 sind gleich der Auswertungstabelle 3.2, so dass ein Vergleich möglich ist.

3.4.4 Versuchsdurchführung

Zur Entwicklung der SPA auf die in dieser wissenschaftlichen Arbeit Bezug genommen wird, wurden eine Vielzahl von externen Werkzeugen zur Hilfe genommen. Zur Bearbeitung und Erstellung von Quellcode wurde der kostenfreie Editor *Atom*[?] genutzt und mit mehreren Plugins versehen. Dazu gehören die Pakete *elm – format* (<https://github.com/triforkse/atom-elm-format>), *language – elm* (<https://atom.io/packages/language-elm>) und *linter – elm – make* (<https://atom.io/packages/linter-elm-make>). Mit Hilfe dieser Plugins wird der Code automatisch beim Speichern 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. Da die Pakete ständig aktualisiert und verändert werden, wird an dieser Stelle von einer detaillierten Beschreibung zur Installation abgesehen und auf die Dokumentationen der einzelnen Pakete verwiesen. Damit die geschriebene Applikation letztendlich im Browser aufrufbar ist, muss die Elm-Applikation zunächst in der HTML-Datei aufgerufen und initialisiert werden. Dabei gibt es mehrere Möglichkeiten. Es ist sinnvoll hierbei zwischen vollautomatischem, sowie manuellem Grundaufbau zu unterscheiden.

Kriterium	Hypothese
1. Entwicklungsgeschwindigkeit	1.1 Elm hat leicht erlernbare Grundkonzepte, die adaptiert und erweiterbar sind, um Produktivität für die Entwickler zu gewährleisten
2. Wartbarkeit	2.1 Elm Code kann mindestens kommentiert werden, wenn nicht sogar eine Funktion zur automatischen Generierung von wichtigen Informationen existiert
3. Zuverlässigkeit	3.1 Die Erweiterungen des Editors, oder der Compiler selbst, warnen spätestens bei der Kompilierung, wenn nicht bereits während der Entwicklung vor syntaktischen Fehlern 3.2 Gibt es keine Fehlermeldungen, wird die Applikation ohne Laufzeitfehler funktionieren 3.3 Fehlermeldungen werden sehr spezifisch auf das eigentliche Problem hinweisen
4. Portabilität	4.1 Die SPA wird auf allen gängigen Browsern fehlerfrei dargestellt
5. Effizienz	5.1 Der Kompilervorgang wird nur wenige Sekunden dauern, jedoch mehr Zeit in Anspruch nehmen 5.2 Die erzeugte Elm-Applikation ist deutlich schneller während der Laufzeit
6. Wiederverwendbarkeit	6.1 Codeteilen lassen sich problemlos auslagern und wiederverwenden
7. Modularität	7.1 Ausgelagerte Codeteile sind isoliert voneinander und als einzelnes Modul nutzbar 7.2 Module können Funktionen nach außen verbergen
8. Lesbarkeit	8.1 Die geforderte Formatierung des Elm-Codes macht diesen lesbarer und spart Zeit
9. Dateigröße	9.1 Die Dateigröße der Elm-Applikation wird kleiner als bei vergleichbaren Frameworks ausfallen
10. Interoperabilität	10.1 Bestehende JS-Skripte können mit der Elm-Applikation interagieren oder funktionieren bereits einwandfrei 10.2 Bestehender CSS-Code kann nativ in Elm eingebunden werden
11. Asynchrones Laden	11.1 Elm erlaubt asynchrone Requests ohne Seiteneffekte zu erzeugen

Tabelle 3.1: Aufstellung der Hypothesen

Grundgerüst der Applikation

Vollautomatisch – Elm-reactor

Mit Hilfe des *elm – reactor* kann der gesamte Prozess der Kompilierung und Konfiguration eines Webservers automatisiert werden. Dafür muss lediglich der Ordner in dem die Elm-Applikation abgelegt wurde geöffnet und der *elm – reactor* gestartet werden. Die geschriebene Applikation kann daraufhin lokal mit jedem Browser über die URL *http : //localhost : 8000/* geöffnet werden, ohne die Notwendigkeit eines zusätzlichen Webservers. Die Abbildung 3.3) zeigt den laufenden Webserver. Eine weitere Möglichkeit ist, den Code automatisch zu kompilieren und zusätzlich eine *elm.html* Datei zu erzeugen. Diese Datei wird mit dem gesamten kompilierten Elm-Code bestückt und ausgeführt, sobald die Datei im Browser geöffnet wird. Für den Entwickler bedeutet dies, dass nur noch die erzeugten Dateien weitergegeben werden müssen.

```
1  <html>
2    <head>
3      <title>Elm-Examples</title>
4      <script type="text/javascript" src="elm.js"></script>
5    </head>
6
7    <body>
8      <div id="elm-application">
9      </div>
10   </body>
11
12   <script type="text/javascript">
13     var stamperDiv = document.getElementById('elm-application');
14     Elm.Main.embed(stamperDiv);
15   </script>
16 </html>
17
```

Abbildung 3.2: Grundgerüst einesHTML-Dokumentes, um die Elm Applikation zu laden

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

Abbildung 3.3: Der gestartete Elm-Webserver

Manueller Grundaufbau

Überlässt man dem Compiler das Einbinden der erzeugten JS-Datei, so ist die gesamte Elm-Applikation im Vordergrund. Das ist nicht immer wünschenswert oder gar praktikabel. Einerseits, da externe Quellen für CSS und JS über natives Elm nicht reibungslos geladen werden können, andererseits weil nicht immer der gesamte darzustellende HTML-Code nur in Elm geschrieben wurde. Dementsprechend gibt es auch die Möglichkeit, eine HTML-Datei als Gerüst zu erzeugen, in die gezielt der JS-Code der Elm-Applikation injiziert wird. Das Gerüst ist vollständig wie eine klassische HTML-Datei aufgebaut. Abbildung 3.2 zeigt das Grundgerüst des HTML-Dokumentes. Codezeile 4

bindet die kompilierte Elm-Datei. Der DOM-Knoten in welchen die Applikation injiziert wird, ist in Zeile 8 definiert. Das injizieren der Applikation geschieht in den Zeilen 13 bis 14 und erhält als Parameter den zuvor erwähnten DOM-Knoten. Um die Elm-Applikation einfügen zu können, muss die Elm-Datei vorher in der Kommandozeile kompiliert werden mit dem Kommando `elm - make Datei.elm - -output = elm.js`. Bei dieser Art der Initialisierung kann nun noch zwischen drei weiteren Darstellungen unterschieden werden:

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.

Der Vorteil bei Version 2, so wie es auch in der Abbildung 3.2 realisiert wurde, ist, dass auch nur kleine Teile der gesamten Applikation in Elm implementiert werden können. Überlegt man ein bestehendes, möglicherweise komplexes Projekt zu portieren, genügt es kleinere Teile Stück für Stück zu portieren. Es muss nicht befürchtet werden, große Teile der bestehenden Applikation während der Portierung nutzlos zu machen. Ein weiterer Vorteil ist, dass externes CSS und JS in dem HTML-Dokument über die HTML-eigenen `<script>`-Tags geladen werden kann. Zusätzlich zum Grundgerüst der `elm.html` muss nun noch das Grundgerüst der eigentlichen Elm-App erstellt werden. Wie im Kapitel Theoretische Grundlagen beschrieben, ist Elm nach einem MVU-Konzept aufgebaut. Entsprechend sind das die unbedingt notwendigen Funktionen, die es zu realisieren gilt. Das standardmäßig mitgelieferte Paket `elm - lang/html` liefert die sogenannten `Html.App`-Funktionen. 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, ohne Neulinge direkt abzuschrecken. So verlangt die Funktion `Html.App.beginnerProgram` nur die bekannten Funktionen `model`, `view` und `update`. Hierbei können jedoch keine asynchronen Funktionen wie Hypertext Transfer Protocol (HTTP)-Requests genutzt werden. Dafür gibt es die erweiterte Funktion `Html.App.program`, die als vierten Übergabeparameter sogenannte `subscriptions` erwartet. `Subscriptions` werden für die Kommunikation zwischen Elm und JS, sowie Verbindungen über Websockets genutzt. Die dritte und letzte Möglichkeit ist die Funktion `Html.App.programWithFlags`. Hierbei wird die Übergabe eines initialen `Models` an die Elm-Applikation ermöglicht, um den Zustand der Applikation dynamisch setzen zu können. Abbildung 3.4 zeigt das Grundgerüst einer Elm-Applikation mit der Implementierung der Funktion `Html.App.programWithFlags`. Die `main` Funktion erstellt dabei die eigentliche Applikation, während die Funktionen `model`, `view` und `update` jeweils den Zustand und die gewünschte Darstellung der Applikation beschreiben, sowie vorgeben, wie mit der Interaktion durch den

```
<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 übergebenen Werten

Benutzer umgegangen wird. Die Funktion *subscription* gibt in diesem Stand noch keinerlei Daten weiter und stellt eine Dummy-Funktion dar. *initialModel* erzeugt beim Aufruf ein Model mit initialen Werten, die dem Modeltypen (*type alias Model*) entsprechen müssen. Durch das Grundgerüst der *elm.html* und *Main.elm*-Datei kann die Applikation nun schrittweise erweitert und Funktionen hinzugefügt werden. Zuletzt wird der erzeugte Code modularisiert.

Konstruktion und Überführung des Views

Das gesamte Template *Agency* soll weitestgehend nativ in Elm programmiert werden. Ausschließlich die vorhandenen JS-Dateien sollen, wenn möglich, bestehen bleiben. Den Großteil des zu überführenden Codes macht dabei der HTML-Code aus. Sämtliche Sektionen des Templates werden zunächst in einzelne Funktionen überführt und anschließend gesammelt von einem zentralen Punkt im Code aus aufgerufen. Damit HTML-Code erzeugt werden kann, werden die entsprechenden Funktionen aus der Bibliothek *elm-lang/html* genutzt werden. Im Zuge dessen wird die Bibliothek installiert und in die **.elm*-Datei importiert. Mit Hilfe des Keywords *exposing* wird eine Liste der Funktionen erzeugt, die in den aktuellen Namensraum der Applikation geladen werden sollen. Diese Liste wird mit den verwendeten Funktionsnamen zur Erstellung der HTML-Elemente gefüllt, so dass die HTML-Elemente ohne die Definition des ursprünglichen Namensraum *Html* genutzt werden können.

```
-- div: (List Html.Attribute msg) (List Html.Html msg)
div
  [ class "someClass" id "someID" ]
  [ text "This is some text inside a div." ]
-- <div class="someClass" id="someID">This is some text inside a div.</div>
```

Abbildung 3.5: Resultat des Elm-Codes in HTML

Wie bereits erwähnt, hat erzeugt jede Funktion das gleichnamige HTML-Element als äquivalent. Ein *div*-Element wird dementsprechend mit dem Aufruf der Funktion *div [] []* erzeugt. Es ist erkennbar, dass die Funktion zwei Listen als Übergabeparameter erwartet. Die erste Liste enthält dabei optional sämtliche Attribute, die das HTML-Element besitzen soll. Dazu gehören

beispielsweise *class* oder *href*. Die zweite Liste kann optional weitere HTML-Elemente enthalten, um geschachtelte Konstrukte zu erzeugen. Die Signatur für die *div*-Funktion kann in Abbildung 3.5 betrachtet werden. Die Abbildung zeigt ferner, die Repräsentation von Elm-Code gegenüber dem daraus resultierenden HTML-Code. Auch ist beispielhaft die Zuweisung einer Klasse (*class*) zu sehen. Auffällig sind die fehlenden *Tags* des HTML-Codes. Diese werden offenbar erst während der Kompilierung erzeugt. In Elm ist lediglich das Einrücken und schachteln der Funktionen von Bedeutung, um das entsprechende HTML-Konstrukt zu erzeugen.

```
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 3.6: Deklaration einer Sektion des Views in Elm

Mit Hilfe des Tools *html – to – elm* kann fertiger HTML-Code automatisch in Elm-Code überführt werden. Das Tool wird für jede Sektion angewandt, um Zeit zu sparen und gleichzeitig die Funktionalität des Tools zu überprüfen. Für jede Sektion wird eine gleichnamige Funktion angelegt, die dem Muster in Abbildung 3.6 folgt. Die *id* eines HTML-Elementes könnte entsprechend stets für den Funktionsnamen *nameDerSektion* genutzt werden, da dieses Attribut ohnehin einzigartig in einem gesamten HTML-Dokument vorkommt und eine klare Namensgebung liefert.

Modularisierung

Damit ein Entwickler den Überblick über den vorhandenen Quellcode behält 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 zu finden und beispielsweise die Programmlogik noch deutlicher von der Applikationsdarstellung zu trennen. Dabei werden die einzelnen Funktionen des Views, also die für die Darstellung verantwortlichen Programmteile, ausgelagert in eigene Module. Dasselbe wird für die ‘Update’ und ‘Model’ relevanten Funktionen durchgeführt. Die notwendigen Funktionen eines jeden Moduls werden dann im Gegenzug vom Hauptmodul importiert und an den entsprechenden Stellen aufgerufen.

View

Jede bisherige Funktion aus dem View wird in den Ordner ‘View’ verschoben und als gleichnamiges Modul benannt. Jedes Modul bekommt dabei den Namen des Ordners in dem es zu finden ist, gefolgt vom Namen des Views, dass es darstellt. Das für die Navigation verantwortliche Modul wird entsprechend mit ‘module View.Navigation exposing (view)’ initialisiert und gibt die Funktion ‘view’ an jedes importierende Modul frei. Dieser Schritt dient lediglich der besseren Strukturierung des Quellcodes und der Vereinfachung für den Entwickler. Abbildung XY zeigt die Haupt-‘view’-Funktion nachdem sämtliche Teile des

‘View‘ modularisiert und entsprechend importiert wurden.

Update

Auch die Programmlogik kann modularisiert werden und wird dafür in den Unterordner *Update* verschoben. Hierfür werden sämtliche Typdeklarationen (*Msg*), sowie die dazugehörige Funktion *update* in das neue Modul ‘Update.Update‘ verschoben und auch die notwendigen Pakete hinzugefügt. Von außen kann auf das *Update*-Modul, nachdem es importiert wurde, mit dem Namespace *Update* zugegriffen werden.

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

Das fertige Template bietet die Möglichkeit auf einen Portfoliobeitrag zu klicken. Durch diesen Klick wird ein Modal geöffnet, in das weitere Informationen dargestellt werden. Üblicherweise werden diese zusätzlichen Informationen in einer SPA nachgeladen, um das initiale Laden der Applikation zu verkürzen und nur wirklich notwendige Daten anzuzeigen. Die bestehende Elm-Applikation wird nun um das Feature des asynchronen Ladens von Informationen erweitert. Zunächst muss 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 eines der Portfoliobeiträge soll entsprechend das Modal geöffnet und ein Titel präsentiert werden. In diesem Beispiel wird über eine externe API ein zufälliger String angefordert, vom Server generiert und dann 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. In diesem Fall ist es jedoch nicht notwendig ein zusätzliches Backend zu konfigurieren. 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 (Failed, Success, ..?), 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, also den Fall einer erfolgreichen, sowie fehlerhaften Übertragung, zu behandeln. Auf diese Weise ist gewährleistet, dass die Applikation sich nicht plötzlich in einem nicht definierten Zustand befindet. Einen asynchronen Request in Elm auszuführen bedarf mindestens zweier zusätzlicher Funktionen und der Importierung der Bibliotheken *Http*, *Json.Decode* und *Task*. Des Weiteren muss der Klick auf das Portfolio-Element abgefangen werden. Dafür gibt es die *onClick* Funktion

aus der *Html.Events*-Bibliothek. Sie bekommt die auszuführende Funktion als Parameter, sieht also wie folgt aus: `onClick Update.GetRandomString`. Die möglichen ‘Types‘ von eingehenden Nachrichten (‘Msg‘/Klicks) wird erweitert um `GetRandomString`, sowie auch die `update`-Funktion um diesen Typ erweitert werden muss. Der entsprechende `update`-Fall `GetRandomString` gibt dann das `model`, sowie einen Effekt `fetchAsync` zurück. Die Definition dieses Effektes ist der Grund, weshalb hier von einem ‘managed Effect‘ die Rede ist und der Seiteneffekt kontrolliert verläuft. `fetchAsync` ist hierbei erneut eine Funktion, die eine Nachricht (‘Msg‘) an die `update`-Funktion mit dem Ergebnis des Requests zurückgibt. Elm führt den Request in Form eines ‘Task‘ aus und erwartet eine Funktion für den Fall einer erfolgreichen Übertragung, sowie eine Funktion für jeglichen Fehlerfall. In beiden Fällen wird die entsprechende Funktion ausgeführt und an die `update`-Funktion zurückgegeben. Hier wird, insofern notwendig, ein neues `model` mit veränderten Werten erzeugt und letztlich das Ergebnis auf dem Bildschirm des Nutzers sichtbar gemacht. TODO: Allgemeiner beschreiben. Bei Bezug auf den Anwendungsfall, Abbildung einfügen.

3.4.5 Beobachtungen

Während der Entwicklung der Elm-Applikation kam es zu unvorhergesehenen Problemen, sowie teilweisen Erkenntnissen. Diese Beobachtungen werden im folgenden stichpunktartig zusammengefasst und anschließend erläutert.

1. Externes CSS kann nicht nativ über Elm geladen werden
 - 1.1. Inline-CSS muss manuell mehrfach abgeändert werden
 - 1.2. Keine Schachtelung möglich
 - 1.3. Browser kann inline-CSS nicht zwischenspeichern
2. Ein HTML-Element benötigt in Elm weniger Zeichen
 - 2.1. HTML-Code benötigt in Elm keine schließenden Klammern. Anders als HTML arbeitet Elm mit Einrückungen. Die gleichnamigen Funktionen um ein HTML-Element zu erzeugen benötigen dementsprechend lediglich den Funktionsaufruf, gefolgt von den zwei Argumenten. 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.
3. Vertikale und horizontale Zentrierung eines Elementes ist umfangreicher als vorerst erwartet
 - 3.1. Obwohl Elm ein Paket für genau dieses Anwendungsgebiet besitzt, ist die Anwendung dennoch weder simpel, noch voll funktionsfähig. Es bedarf einiger Transformationen der Elemente, um sie in nutzbare, validen HTML-Code zu formatieren. Zuletzt trifft der Entwickler auf die Problematik, zusätzlich noch die explizite Größe des

Fensters mit einbeziehen zu wollen, so dass das Element den gesamten Bildschirm ausfüllt und einen zentrierten Text beinhaltet. Die Fenstergröße zu ermitteln stellt ein größeres Hindernis dar, als zunächst erwartet. Die Möglichkeiten sind hier, halbwegs dynamisch die Fenstergröße bei der Initiierung der Elm-Applikation an diese weiter zureichen, oder über Ports abzufangen, wenn sich die Fenstergröße geändert hat. Die erste Lösung hilft nur, wenn die Größe des Fensters nicht weiter verändert wird. Dies kann jedoch nicht gewährleistet werden, wodurch die Lösung entfällt. Die zweite Lösung hingegen erzeugt eine zusätzliche Abhängigkeit zwischen Elm und externem JS. Zuletzt bleibt noch die altbekannte Möglichkeit mittels CSS den Text in einem Element beidseitig zu zentrieren. Hierfür kann das externe CSS-Framework Flexbox zuhulfe gezogen werden. Flexbox kümmert sich um die horizontale und vertikale Platzierung von Elementen, ohne auf die CSS-Eigenschaft „float“ zurückgreifen zu müssen, die gerade bei tief verschachtelten Elementen Probleme verursacht.

- 3.2. Das Paket ‘elm-lang/window‘ stellt eine Funktion ‘resize‘ zur Verfügung, die bei jeder Veränderung der Fensterdimensionen über eine sogenannte ‘subscription‘ in Elm einen Aufruf der ‘update‘ Funktion auslöst, an jene die neuen Fensterdimensionen (X, Y) weitergereicht werden.

4. Elm-Compiler

- 4.1. Kompiliert abhängig von der Tiefe der Elemente

- 4.1.1. Entwickler muss lesbaren Code erzeugen, da Elm sensitiv auf Tabs reagiert (vsHTML: `< div >< span >< /span >< /div >` wo unendliches verschachteln auch in einer Zeile möglich ist)

5. TODO:

- 5.1. Fehlende Beispiele importieren
- 5.2. Umschreiben der Beispiele

3.4.6 Auswertung

TODO: Importieren der schriftlichen Auswertung der Tabelle 3.2

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

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 3.2: Auswertung der Versuchskriterien

XY (*compiler_sig_suggestion* + *compiler_own_sig*) “ type alias Model = counter: Int increaseModel: Model -> Model increaseModel model = model | counter = model.counter + 1

vs. Kompiliervorschlag 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 syntaktischen 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 Kompiliervorgang 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 Kompiliervorgang 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 mög-

lich. 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. Externes CSS muss jedoch über das Grundgerüst der HTML-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.

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 Divs wurden vernachlässigt) 2. required -> required " (wants Bool, got String); novalidate same 2. Elm-Compiler 1. Fehler von HTML2ELM (2. Punkt) wurden sofort erkannt 2. Fehlende Funktionen für HTML und CSS 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] WESTBY, Luke: *builtwithelm*. Webseite, Dezember 2015. – Online einsehbar unter <http://builtwithelm.co/> - abgerufen am 07. Juni 2016

TODO: Literatur hinzufügen + Referenzen + importieren