



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 Was ist eine funktionale Programmiersprache?	6
2.2 Was ist Elm?	7
2.2.1 Geschichte	7
2.2.2 Konzept	8
2.2.3 Umsetzung	9
3 Evaluierung der Programmiersprache Elm	15
3.1 Bewertungsmuster	15
3.2 Bewertungskriterien	16
3.2.1 Entwicklungsgeschwindigkeit	16
3.2.2 Wartbarkeit	16
3.2.3 Zuverlässigkeit	17
3.2.4 Portabilität	17
3.2.5 Effizienz	17
3.2.6 Erlernbarkeit	17
3.2.7 Wiederverwendbarkeit	17
3.2.8 Lesbarkeit	18
3.2.9 Modularität	18
3.3 Ansprüche an eine Webapplikation	18
3.3.1 Browser Kompatibilität – Portabilität	19
3.3.2 Interoperabilität	19
3.3.3 Asynchrones Laden	20
3.3.4 Dateigröße	20
3.4 Empirische Analyse	20
3.4.1 Versuchsaufbau	20
3.4.2 Hypothesen und Vermutungen	21
3.4.3 Versuchsdurchführung	21
3.4.4 Beobachtungen	28
3.4.5 Auswertung	30
4 Fazit	33
5 Eidesstattliche Erklärung	34

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

Abbildungsverzeichnis

2.1	Eine iterative Funktion	7
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
3.1	Die fünf meistgenutzten Browser im Mai 2016	19
3.2	Grundgerüst eines HTML-Dokumentes, um die Elm Applika- tion zu laden	22
3.3	Der gestartete Elm-Webserver	23
3.4	Eine beispielhafte Initialisierung der Elm-Applikation mit über- gebenen Werten	24

Tabellenverzeichnis

3.1	Auswertung der Versuchskriterien	30
-----	--	----

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 dann, 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 Entwicklung 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 Was ist eine funktionale Programmiersprache?

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 Programmiersprachen dem deklarativen Programmierparadigma an. Es ist jedoch nicht ausgeschlossen, dass eine Programmiersprache mehreren Kategorien zugehörig ist und dadurch die Merkmale von mehr als einem Paradigma unterstützt. Zur Gruppe der deklarativen Programmiersprachen zählt man unter anderem Abfragesprachen wie SQL, sowie funktionale Programmiersprachen wie Lisp oder Scheme. Programmiersprachen der deklarativen Programmierung haben ihren Ursprung in der Mathematik. Programme werden hier als mathematische Funktionen formuliert, die nicht länger beschreiben, was getan werden soll, sondern lediglich vorgeben, welches Ergebnis am Ende erwartet wird. Bei funktionalen Programmiersprachen ist es üblich, dass eine Variable nach ihrer Initialisierung ihren zugewiesenen Wert für die gesamte Laufzeit des Programmes beibehält und unveränderlich bleibt. Es ist dementsprechend stets nachzuvollziehen, welchen Wert ein Ausdruck besitzt, wodurch insbesondere akademische Anforderungen an ein Programm, wie etwa die Beweisführung, erfüllt werden können. Zusätzlich können auch unendliche Datenstrukturen behandelt werden. Typischerweise gibt es in funktionalen Programmiersprachen keine Schleifen, da dies bereits eine Verletzung der Unveränderlichkeit von Variablen bedeuten würde, wie klar erkennbar bei der Iteration in Abbildung 2.1 ist. Hier wird bei jedem Durchlauf der Schleife die Variable i inkrementiert und mit dem neuen Wert versehen.

Es ist allerdings auch möglich eine Schleife in einer funktionalen Programmiersprache zu verwirklichen. In Abbildung 2.2 ist die zuvor gezeigte iterative Funktion als rekursive Funktion implementiert worden.

In beiden Fällen ist das Ergebnis gleich, jedoch besitzt die rekursive 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.


```

iterative_function(n) {
  sum = 0;
  for(i = 0; i <= n; i++) {
    sum += i;
  }
}
console.log(iterative_function(10)); // => 55

```

Abbildung 2.1: Eine iterative Funktion

```

recursive_function(n) {
  if (n < 1) then 0
  else recursive_function(n-1) + n;
}
console.log(recursive_function(10)); // => 55

```

Abbildung 2.2: Eine rekursive Funktion

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 dieser Funktion 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 Was ist 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

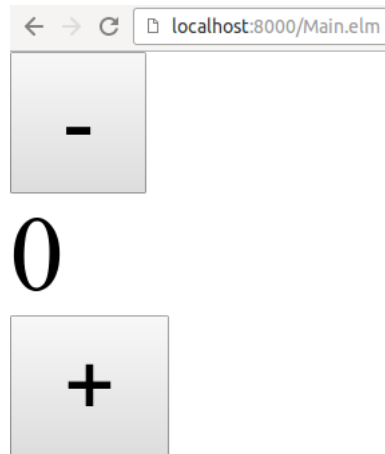


Abbildung 2.3: Ein simpler Zähler auf einer Webseite

Elm verfolgt eine ganz eigene Implementierung des Model-View-Controller Paradigma. Hier wird es Model-View-Update (MVU) genannt. Anhand der Abbildung 2.3 lässt sich das Muster in drei grundlegende Stationen unterteilen und erklären. Die Abbildung 2.3 zeigt einen simplen Zähler der über zwei Knöpfe inkrementiert und dekrementiert werden kann. Der aktuelle Stand des Zählers wird dazwischen angezeigt und kann sowohl negative, wie auch positive Werte annehmen. Der angezeigte Zählerwert ist das sogenannte *Model* und zeigt den aktuellen Status der Applikation an. Betätigt ein Nutzer nun einen 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 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. Damit dieser Vorgang zügig vonstatten geht, nutzt Elm persistente Datenstrukturen, womit nur die tatsächlich geänderten Attribute eines Models im neuen *Model* gesetzt werden, die unveränderten Attribute hingegen werden kopiert. Das Ergebnis der Update-Funktion wird weitergereicht an die View-Funktion. Diese Funktion 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 an-

gezeigt werden, anstatt dauerhaft das komplette DOM stetig zu aktualisieren. Das DOM beschreibt die Schnittstelle zum Datenzugriff auf das Objektmodell eines HyperText Markup Language (HTML)-Dokumentes. Der Datenfluss in Elm wird noch einmal in Abbildung 2.4 visualisiert.

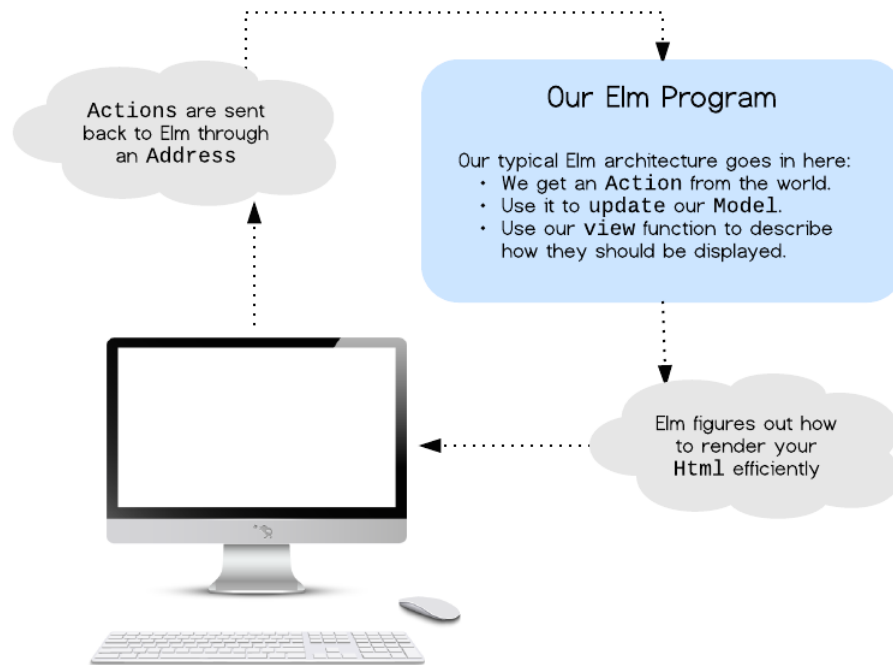


Abbildung 2.4: Das Model-View-Update Konzept von Elm

2.2.3 Umsetzung

Der vom Programmierer verfasste Programmcode wird vor der endgültigen Nutzung zu JavaScript, HTML und Cascading Style Sheet (CSS) kompiliert und in die Webseiten integriert. Entsprechend fungiert in Elm verfasster Code im Endeffekt wie natives JavaScript, nutzt allerdings noch einige weitere vertiefende Konzepte, um viele Problematiken von JavaScript 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 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 folgenden Code, 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 in der Mathematik das Gleichheitszeichen 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 JavaScript 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 JavaScript, 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 JavaScript erlaubt es, dass die Typen von Variablen erst zur Laufzeit überprüft werden und sich zusätzlich während der Laufzeit ändern können. So ist der Quellcode in Abbildung 2.5 konform in JavaScript. Der Typ der

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

Abbildung 2.5: Beispiel der dynamischen Typisierung

Variable *i* wurde in der Abbildung 2.5 während der Laufzeit von *number* zu *string* geändert. Da Elm stark typisiert ist, gibt es keine Möglichkeit, dass eine Funktion verschiedene Datentypen zurück gibt. Die Limitierung ist auch einer der Gründe, weshalb der Elm-Compiler immens viele Fehler bereits beim Kompiliervorgang aufdeckt. Dies ist das Ziel und der Vorteil des statischen Typensystems.

Modularität

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

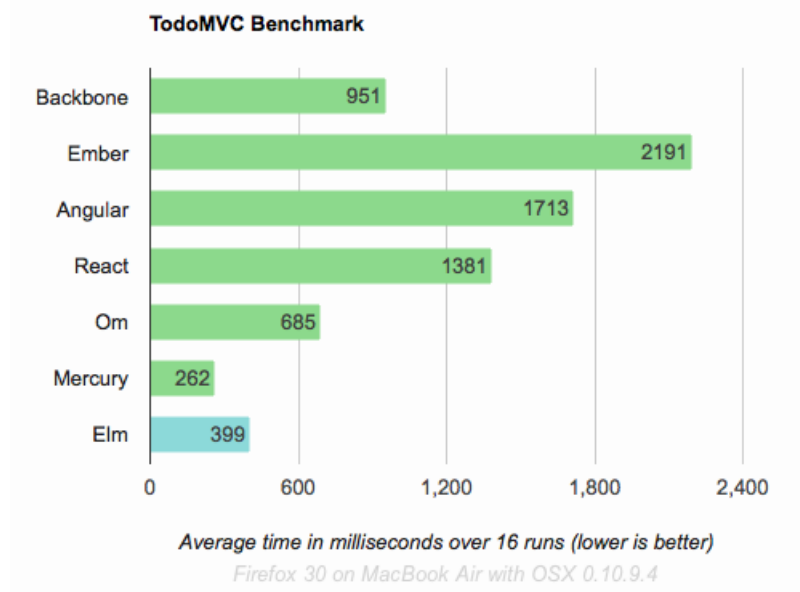


Abbildung 2.6: Performanz von Elm im Vergleich zu anderen Web-Frameworks. [1]

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

Performanz

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 erneut 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 dann anhand der zuvor erstellten Bewertungskriterien ausgewertet. Auch die Beobachtungen während der Entwicklung, wie etwa unvorhergesehene Probleme, gehen mit in die Wertung ein.

¹[2, vgl.]

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 JavaScript in die Programmiersprache Elm portiert werden. Während der Entwicklung/Portierung wird dann die Programmiersprache Elm gegenüber mehreren Gesichtspunkten verglichen und ausgewertet. Das fertige Template ist eine Single-Page-Applikation mit Elementen, wie sie typischerweise auf einer solchen Webseite vertreten sind. Eine Single-Page-Applikation ist, wie der Name bereits suggeriert, eine Webseite 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 also 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 Buchstabe repräsentiert. Durch die Verknüpfung mehrerer *char*'s, kann letztendlich der Datentyp *String* umgesetzt werden.

3.2.2 Wartbarkeit

Geschriebener Quellcode muss wartbar sein. Das bedeutet, dass auch Entwickler auch nach einiger Zeit an der nicht daran gearbeitet wurde den Programmcode verstehen und entsprechend verändern kann. Dazu gehört, dass der Quellcode gut dokumentiert ist, respektive erläuternde Kommentare hinzugefügt wurden, oder der Code selbsterklärend ist. Auch sollten Funktionen sinnvoll und markant benannt werden, so dass dies bereits Aufschlüsse darüber gibt, was das Ziel der Methode ist. Ein weiterer Schritt um die Wartbarkeit

von Quellcode zu gewährleisten sind Tests, durch die eine Methode und ihre eigentliche Funktion getestet und auf Fehler überprüft werden kann.

3.2.3 Zuverlässigkeit

Für einen Entwickler von Software, egal wofür und mit welcher Programmiersprache gearbeitet wird, ist es wichtig, dass das erzeugte 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.

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.

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.

3.2.6 Erlernbarkeit

Wie im vorherigen Punkt, ist es wirtschaftlich wichtig schnell zu arbeiten und Ergebnisse zu erbringen. Demzufolge sollte auch der Lernprozess schnell vonstatten gehen und die Programmiersprache eine gewisse Einfachheit in ihren Grundkonzepten verwirklichen, um dem Entwickler ein einfacheres Erlernen zu ermöglichen.

3.2.7 Wiederverwendbarkeit

Erzeugter Quellcode sollte nicht für jedes Projekt oder gar Modul innerhalb eines Projektes neu geschrieben und somit vervielfacht werden. Besser ist es

wenn der Quellcode lediglich einmal geschrieben wird und von anderen Modulen benutzt werden kann. Dadurch vereinfacht sich auch die Wartbarkeit des Quellcodes, da Änderungen nur an einer Stelle vorgenommen werden müssen und sich so weniger Fehler einschleichen können.

3.2.8 Lesbarkeit

Quellcode muss lesbar sein. Damit ist gemeint, dass Befehle stets einen prägnanten, einfachen und möglichst kurzen Namen haben sollten. Des Weiteren ist es notwendig, dass der geschriebene Quellcode gelesen werden kann, ergo der Sinn dahinter schnell verständlich wird. Jede Programmiersprache verfolgt dabei einen anderen Ansatz. Ist die Lesbarkeit gegeben, so kann Quellcode auch von Neulingen gelesen werden. Es ist für diese Neulinge nicht unbedingt notwendig die Syntax oder gar genaueres über einzelnen Befehle zu wissen, es reicht vollkommen aus, wenn das angestrebte Ziel des Quellcodes klar wird und adaptiert werden kann.

3.2.9 Modularität

Mit der Zeit wächst ein Projekt. Zumeist steigt simultan auch die Zahl der Menschen, die an einem Projekt mitarbeiten, wodurch Quellcode sehr schnell unübersichtlich wird. Um Übersichtlichkeit zu gewährleisten wird der geschriebene Programmcode oftmals ausgelagert und in einzelne Module unterteilt, die wiederum logische Teilblöcke des gesamten Programmes darstellen. Die einzelnen Module sollten voneinander isoliert funktionieren, so dass Änderungen in Modulen nur eine Anpassung in der Kommunikation erfordern. Weiterhin kann durch Module der Zugriff auf bestimmte Funktionen gewährt oder verweigert werden.

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 Single Page Application (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 JavaScript. 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 mit dem Backend.

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 JavaScript, ist es wichtig, dass der Browser den vorhandenen Quellcode lesen und ausführen kann. Sämtliche Sprachen wie HTML, CSS und Javascript 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).

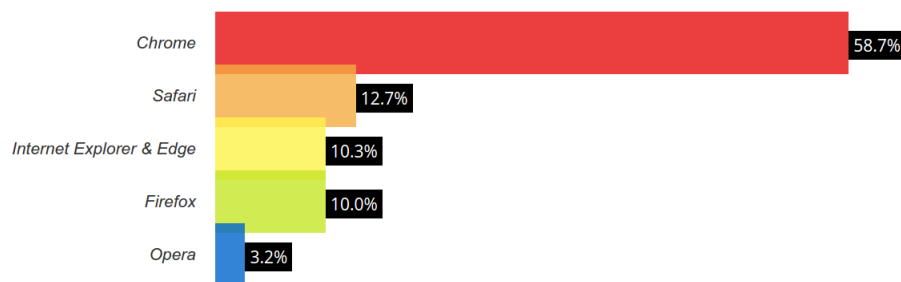


Abbildung 3.1: Die fünf meistgenutzten Browser im Mai 2016

3.3.2 Interoperabilität

Auch bei den Webapplikationen ist es wichtig, bereits existente Frameworks und Problemlösungen nutzen zu können. Dementsprechend müssen externe JavaScript 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, also 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, muss dann natürlich an jeder anderen Stelle auch ersetzt werden. Hierzu bedarf es Algorithmen, welche die Nutzung von Variablen genau untersuchen und Fehler durch Ersetzungen vermeiden.

3.4 Empirische Analyse

3.4.1 Versuchsaufbau

Abbildung XY: Zeigt die Kommunikation zwischen Client und Webserver Die Abbildung XY zeigt die geplante Kommunikation zwischen dem Client, also dem Nutzer und dem Webserver. In Schritt 1 fordert der Nutzer die Webseite an, erzeugt also einen Request. Dieser geht in Schritt 2 bei dem Webserver ein und wird verarbeitet. Abhängig von der angeforderten URL erzeugt der Webserver dann eine Antwort, mit allen zu sendenden Informationen und dem dazugehörigen HTML, CSS und JavaScript 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 Javascript, oder erzeugt einen neuen Request an den Server, womit wieder 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 Template, programmiert in HTML, CSS und JavaScript in die Programmiersprache Elm überführt, insofern möglich. Der daraus entstehende Elm-Quellcode wird dann mit dem Elm-Compiler kompiliert zu JavaScript. 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. 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 dann mit Google Chrome Version 50 getestet.

3.4.2 Hypothesen und Vermutungen

TODO: Aufzählung importieren

3.4.3 Versuchsdurchführung

Installation

Zunächst einmal ist es notwendig, Elm und die dazugehörigen Komponenten zu installieren. Im folgenden wird die Installation unter Ubuntu 14.04 64bit erklärt. Elm wird über den NPM verbreitet. Um diesen zu installieren, sind folgende Kommandos auf der Kommandozeile auszuführen:

```
$ sudo apt-get update
$ sudo apt-get install nodejs
$ sudo apt-get install npm
```

Dadurch wird das NodeJs-Paket installiert und ausführbar gemacht. Aufgrund der Tatsache, dass viele Pakete nicht *nodejs*, sondern *node* als ausführbare Datei erwarten, wird noch sicherheitshalber ein Symlink erstellt.

```
$ sudo ln -s /usr/bin/nodejs /usr/bin/node
```

Anschließend sollte überprüft werden, ob die Installation erfolgreich war und sowohl NodeJS, wie auch NPM verfügbar sind. Das Kommando

```
$ node -v && npm -v
```

liefert den Output, wie er in Abbildung XY zu sehen ist. Elm kann nun über die Paketverwaltung NPM mit dem Kommando *npm install -g elm* installiert werden. Das *-g* -Flag 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 hier behandelte Version 0.17 zu installieren, ist der Zusatz *@0.17* direkt hinter dem globalen Flag notwendig. Elm ist nun vollständig installiert und kann genutzt werden. Die Entwicklung in Atom wird unterstützt durch 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 Pakete 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 (elm-lang.org). Damit die geschriebene Applikation letztendlich im Browser aufrufen zu können, 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.

Grundaufbau

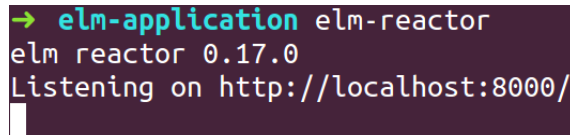
1) Vollautomatisch – Elm-reactor

Mit Hilfe des *elm-reactor*, einem mitgelieferten Tools um den erzeugten Elm-Code lauffähig zu machen und die Applikation debuggen zu können, 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. Der Code kann automatisch kompiliert und zusätzlich eine *elm.html* Datei erzeugt werden. Diese Datei wird dann mit dem gesamten kompilierten Elm-Code bestückt. Außerdem wird automatisch Code erzeugt, der den kompilierten JavaScript-Code ausführt. Für den Entwickler bedeutet dies, dass nur noch die erzeugten Dateien weitergegeben werden müssen. Die gesamte geschriebene Applikation ist darin enthalten und kann 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.

```
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 eines HTML-Dokumentes, um die Elm Applikation zu laden

Überlässt man dem Compiler das Einbinden der erzeugten JavaScript-Datei,



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

Abbildung 3.3: Der gestartete Elm-Webserver

so ist die gesamte Elm-Applikation im Vordergrund. Das ist nicht immer wünschenswert oder gar praktikabel. Einerseits, da externe Quellen für CSS und JavaScript 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 JavaScript-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 JavaScript 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 Javascript, sowie die Verbindung zu 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 Imple-

```
<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

mentierung 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 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 Model (*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 des Views

TODO: Rewrite:

1. Konstruktion des Views
 - Import der HTML-Bibliothek
 - Aufruf der HTML-Funktion
 - Signatur der HTML-Funktion
 - Verschachtelung verschiedener Aufrufe
 - Unterschied HTML vs. Elm

- HTML: Öffnende + schließende Tags (Fehleranfällig, unübersichtlich, schachtelbar in einer Zeile)
- Elm: Einrückung (indentation sensitive)
- Mathematisch ausdrücken, dass Elm-Code im Gegensatz zu HTML 'kürzer' ist
- Fehleranfälligkeit von HTML klarstellen

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 sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat.

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.

Überführung des Views

Für die Darstellung einer SPA wird ein fertiges und kostenloses Theme von 'startbootstrap.com' verwendet. Im Zuge dessen werden alle verfügbaren Dateien heruntergeladen und die zuvor erstellten Dateien 'elm.html' und 'Main.elm' in denselben Ordner verschoben. Das Grundgerüst der 'elm.html' muss nun in die 'index.html' überführt werden, so dass die Elm-Applikation weiterhin in den vorhandenen HTML-Code injiziert wird. Zunächst einmal muss der HTML-Code des Themes in ausführbaren Elm-Code umgeschrieben werden, damit Elm Zugriff auf den kompletten 'view' bekommt. Nur so kann Elm die Interaktion des Benutzers mit den HTML-Elementen abfangen und entsprechend darauf reagieren. Um nicht den kompletten HTML-Code der 'index.html' per Hand in Elm-Code überführen zu müssen, wird das Tool 'html-to-elm' genutzt (<http://mbylstra.github.io/html-to-elm/>). Dieses Online-Tool erlaubt es HTML5-konformen Code in lauffähigen Elm-Code zu überführen. Der erzeugte Elm-Code wird dann in der 'Main.elm'-Datei von der 'view'-Funktion zurückgegeben., muss also entsprechend dort eingefügt werden. Um Modularität zu gewährleisten, wird jede Sektion der SPA, wie zum Beispiel die Navigation, die Team-Sektion oder der Footer im ersten Schritt in eine eigene Funktion ausgelagert. Jede dieser Funktionen wird dann in der 'view'-Funktion aufgerufen und ergibt am Ende die gesamte Webseite. Auf diese Weise können die einzelnen Teile der SPA unabhängig voneinander modifiziert und auf Korrektheit überprüft werden. Im nächsten Schritt wird der gesamte View-Code modularisiert.

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 fehlerbe-

hafteten 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’s 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 : String’. Bei einem Klick auf eines der Portfoliobeiträge soll entsprechend das Modal geöffnet und ein Titel prä-

sentiert 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.

3.4.4 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. HTML-Code ist in Elm kürzer

- 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

- 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 Javascript. Zuletzt bleibt noch die altbekannte Möglichkeit mittels CSS den Text in einem Element beidseitig zu zentrieren. Hierfür kann das externe CSS-Framework Flexbox zur Hilfe 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 (vs HTML: `< 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

3.4.5 Auswertung

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

Tabelle 3.1: Auswertung der Versuchskriterien

TODO: Importieren der schriftlichen Auswertung der Tabelle 3.1

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

ne, als es eine Deklaration wäre, ist jedoch ausreichend und wesentlich sinnvoller als eine Signatur komplett unberücksichtigt zu lassen. Beispiel: Abbildung XY (*compiler_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 reinem HTML, CSS und JS -> Bei der herkömmlichen Entwicklung ist kein expliziter Kompiliervorgang notwendig, stattdessen übernimmt der Browser die Darstellung des übergebenen HTML, CSS und JS 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 Mo-

dul 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. Bestehende JS-Skripte können mit der Elm-Applikation interagieren oder funktionieren bereits einwandfrei -> Die Integration vorhandener JS-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. Bestehender CSS-Code kann nativ in Elm eingebunden werden -> Das Einbinden von CSS Quellen ist derzeit nicht ohne Probleme nativ in Elm möglich. Es verursacht ein „flackern“ und ist nicht für den Gebrauch in einem fertigen System geeignet. CSS-Klassen und 'inline-styling' sind jedoch nativ in Elm implementiert und erlauben ein Styling der Elemente. 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