Paul Bryan Alexander Hinzen *M.Nr.11105958*
Jakob Hippe *M.Nr. 11105059*

Praxisarbeit SS 2017

Entwicklung einer Anwendung mithilfe von Elementen aus visuellen Programmiersprachen für die Erstellung interaktiver MockUps

Dokumentation zu MockITUp

Technology
Arts Sciences
TH Köln

Freitag, 18. August 2017

Inhalt

Abbildungsverzeichnis

1. Einleitung

In diesem Dokument wird die Entwicklung der Anwendung MockITUp dokumentiert, sowie dokumentiert wie und wo diese Anwendung findet und wo die Einstiegspunkte für die Weiterentwicklung sind.

Im Laufe dieser Praxisarbeit sollte eine Anwendung entwickelt werden, welche es einem mithilfe von Elementen aus visuellen Programmiersprachen verschiedener Paradigmen auf einfache und spielerischer Art und Weise ermöglicht interaktiv Mock-Ups von Mobilen Anwendungen zu erstellen. Mock Ups sind ein wichtiges Werkzeug im Softwareproduktentwicklungszyklus, die es ermöglichen die Anwendung so einfach und intuitiv wie möglich für den Endnutzer zu gestalten. Allerdings auch in den Fällen wo die Kunden mit den Mock-Ups konfrontiert sind, bleiben sie statische Bilder, siehe Abbildung 1, im Gegensatz zu einem dynamischen Prototyp, der nächste große Schritt in einem Softwareproduktentwicklungszyklus, was zu Problemen führen kann. Diese MockUps schmücken die ersten Handgefertigten Skizzen aus und fügen Details wie Schatten, Texturen, Bilder und Transparenz hinzu, jedoch ihr Hauptzweck ist das Entwickeln der Handlungsleitung der Endnutzer, im Englischen Affordance genannt. Allerdings können diese MockUps ebenfalls unerkennbare Probleme in der Benutzerführung und intuitiven Bedingung beherbergen, welche erst bei einem Usability Test mit einem Prototyp und Testpersonen ans Licht treten. Um diese Fehlerquelle zu eliminieren soll die Anwendung MockITUp nicht nur die Erstellung der MockUps für mobile Anwendungen vereinfachen, sondern im gleichen Zug eine protypische und interaktive UI ohne weiteren Aufwand generieren, welche es ermöglichen soll Affordance-Probleme in einem früheren Entwicklungsstadium aufzudecken.

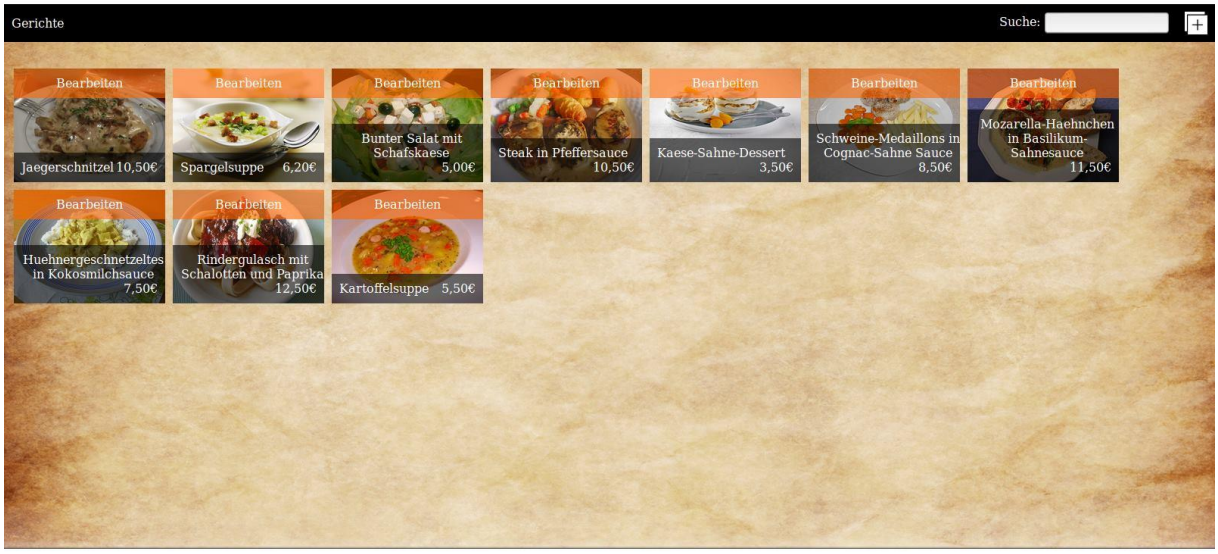


ABBILDUNG 1: EIN KLASSISCHES MOCK-UP FÜR EINE REZEPTSAMMLUNGSANWENDUNG

Die Anwendung MockITUp soll mehrere dieser inhärenten Probleme von Mock-Ups lösen, indem sie zum einen von ungeschulten Personal in kürzester Zeit erstellt werden können und auf der anderen Seite im Beisein von Kunden in Echtzeit entwickelt werden können. Am Ende dieser Designsession kann der Kunde dann sein Smart-Device der Wahl oder die lokale Anwendungsinstanz benutzen um den erstellten Mock-Up wie eine richtige Anwendung benutzen zu können. Durch Persistierung soll es dem Kunden möglich sein, das fertige Mock-Up mitzunehmen und im Detail weiter zu testen.

Ein ähnliches aber nicht gleiches Projekt gab es bereits bei Massachusetts Institute of Technology, kurz MIT, allerdings ging es dort um eine Anwendung in welcher der Benutzer eine lauffähige mobile Anwendung per „Baukastensystem“ erstellen kann, siehe Abbildung 2. Jedoch ist diese Applikation bereits mit einer eigenen visuellen Programmiersprache für die Logik hinter den Elementen ausgestattet, welche es den meisten nicht programmiererfahrenen Nutzern verhindert diese Applikation zur Gänze zu nutzen. Außerdem ist es unser Ziel den Entwicklungsprozess von mobilen Anwendungen zu verbessern und beschleunigen anstatt vollständige Applikationen zu erstellen. Ähnlichkeiten mit der Applikation vom MIT stammen daher von der Analyse derer nach ihren Stärken und Schwächen. [1]

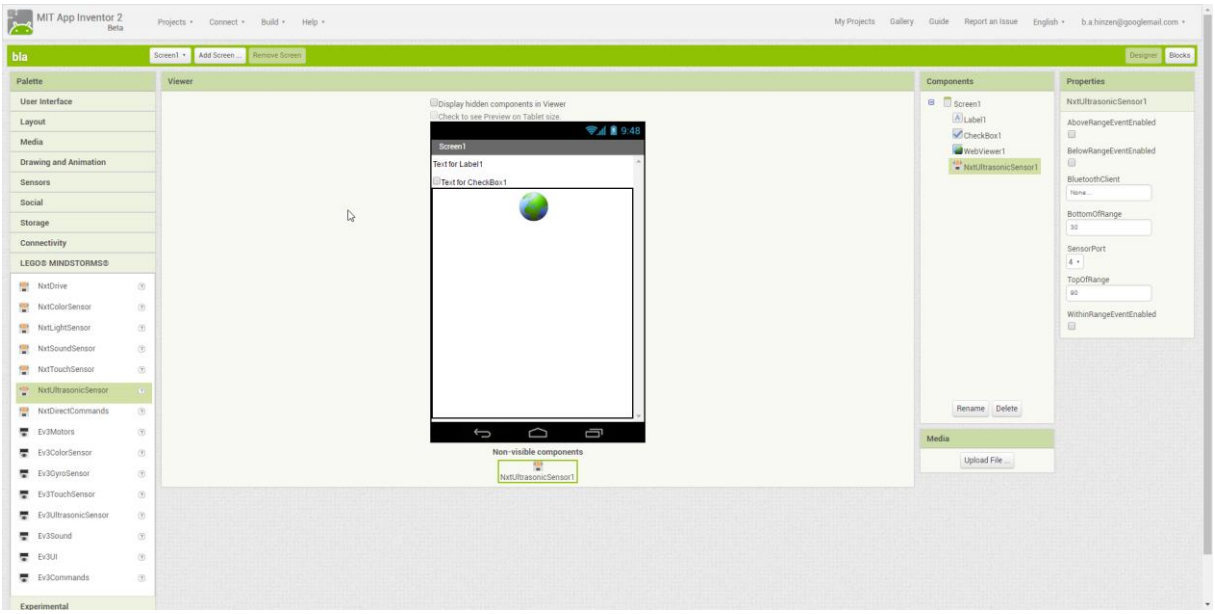


ABBILDUNG 2: MIT APP INVENTOR

Um festzustellen was für eine prototypische Implementierung erforderlich ist sollte außerdem stichprobenartig festgestellt werden, welche die am häufigsten vorkommenden Elemente und Aktionen in einer mobilen Anwendung sind. Dadurch kann priorisiert werden welche Elemente und Aktionen in das „Proof of Concept“ implementiert werden, welches im Verlauf dieser Arbeit erstellt

[1] http://ai2.appinventor.mit.edu

werden soll. Dafür werden im ersten Teil dieses Dokumentes einige unterschiedliche Applikationen auf zwei verschiedenen mobilen Betriebssystemen untersucht.

Weiterführend wird in dieser Arbeit des Entwicklungsprozesses der Anwendung MockTUp vom Anfang bis zum momentanen Stand dokumentiert die auf den Ergebnissen der vorrausgegangenen Untersuchung der mobilen Anwendungen basiert.

Der letzte Teil dieses Dokumentes wendet sich der Technischen Dokumentierung der Anwendung und des Source Codes zu, um die Weiterentwicklung und Erweiterbarkeit von MockITUp sicherzustellen und Einstiegspunkte für die Implementierung neuer Features zu dokumentieren.

2. Untersuchung der Häufigkeit von Elementen und Aktionen in mobilen Anwendungen

Im Laufe der Untersuchung wurden auf zwei verschiedenen mobilen Betriebssystemen, Ubuntu Touch und Android 7.0, jeweils dreizehn mobile Anwendungen analysiert. Dabei wurde von jedem Screen der mobilen Anwendungen ein Screenshot erstellt, siehe Abbildung 4, und die Elemente durchnummeriert. Ihr Typ und ihre möglichen Aktionen wurden bestimmt und festgehalten um am Ende eine stichprobenartige Verteilung aufzustellen, welche zeigen kann, welche die essentiellen Elemente und Aktionen für dieses Projekt sind und dementsprechende Priorität bei der Implementierung von MockITUp haben. Diese Untersuchung soll keine Studie zu der Häufigkeit der Elemente und Aktionen darstellen, sondern nur stichprobenartige Daten liefern um die Implementierungsreihenfolge der Funktionen des Projekts festlegen zu könne. Trotzdem wurden einige Muster für die Zählung der Elemente und Aktionen festgelegt welche in den jeweiligen Abschnitten erklärt werden.

In den folgenden Abschnitten werden die Verteilungen der Elemente und Aktionen zunächst in Bezug auf die Betriebssysteme betrachtet um dann anschließend ein Fazit über die Gesamtverteilung zu ziehen. Weitere Tabellen und Diagramme zur Anschauung befinden sich im Anhang. Hierbei zu beachten ist, das die Prioritäten die in den folgenden Abschnitten zu Stande kommen den Aufwand der Implementierung dieser Features nicht mit in Betracht sind und daher nur eine Maßangabe ihrer Wichtigkeit in einer prototypischen Implementierung der Anwendung sind.

2.1. Häufigkeit von Elementen in mobilen Anwendungen

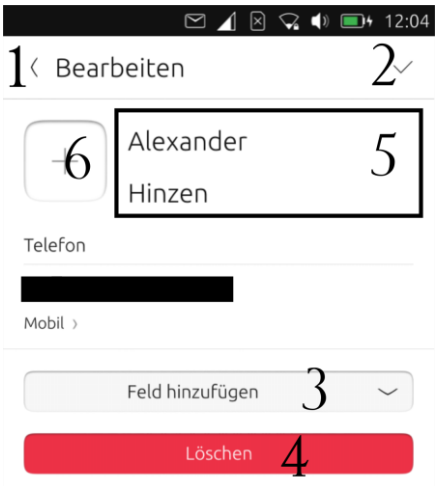


ABBILDUNG 3: SCREENSHOT AUS DER KONTAKTE-APPLIKATION AUF UBUNTU TOUCH

Elemente die in einer *List View* vorkommen werden nur einmal pro Screen gezählt, da sich quasi unendlich wiederholen können sofern man weiter scrollt. Außerdem wurden nur dynamischen Labels und Bilder als diese gezählt, das heißt, Labels und Bilder die sich durch Aktionen oder über Zeit ändern können und nicht statische Texte sind.

2.1.1 Elemente auf Ubuntu Touch

Ubuntu Touch ist eine Variante der Ubuntu Linux-Distribution, die auf Debian basiert und von Canonical entwickelt wird. Ziel von Ubuntu Touch ist es eine einheitliche, jedoch dynamische Benutzeroberfläche anzubieten, die es ermöglicht touchbasiert sowie auch klassische Desktopanwendungen darzustellen. Zu Beginn des Jahres 2016 wurde das erste Gerät werksmäßig mit Ubuntu Touch auf den Markt gebracht. [2]

Aufgrund der neuen Ansätze in der Benutzeroberflächengestaltung im Gegensatz zu den eher klassischen mobilen Betriebssystemen Android und iOS wurde es im Folgenden untersucht.

[2] https://wiki.ubuntuusers.de/Ubuntu_Touch/
https://de.wikipedia.org/wiki/Ubuntu

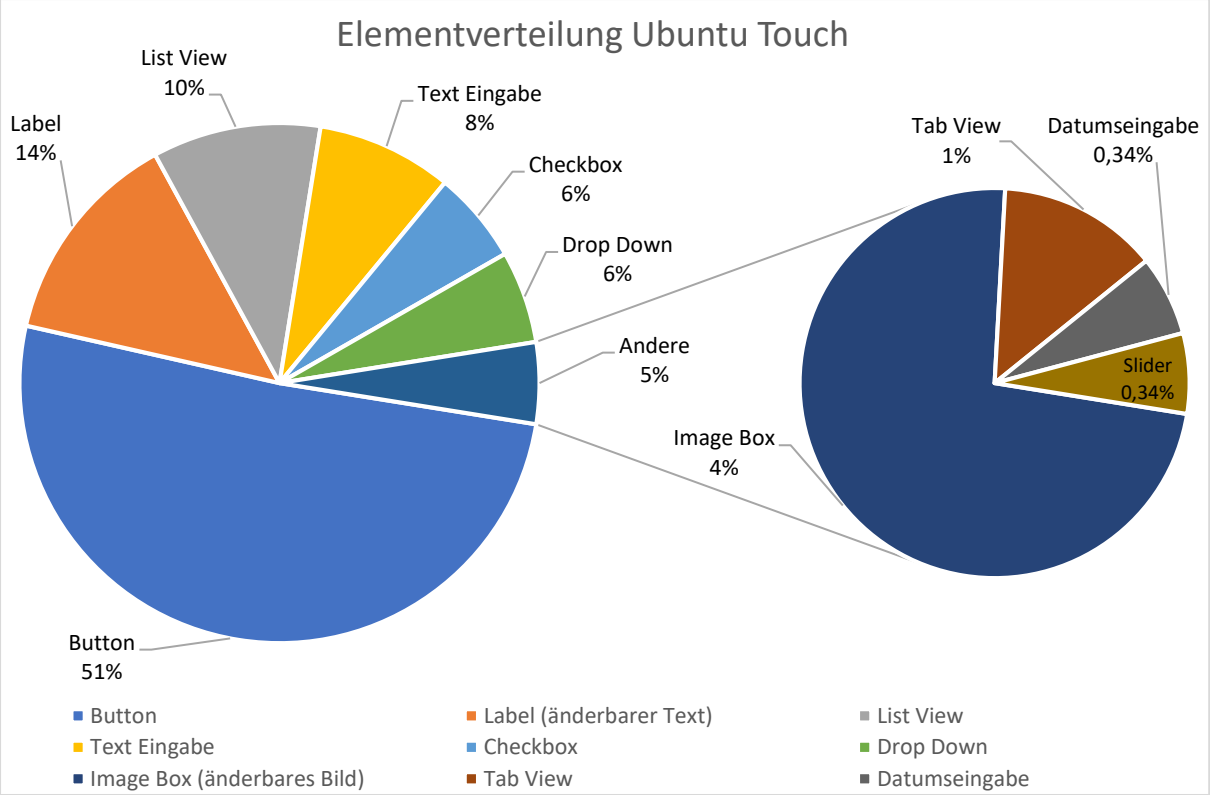


ABBILDUNG 4: ELEMENTVERTEILUNG AUF UBUNTU TOUCH

In Abbildung 5 ist die Verteilung der verschiedenen Elemente von Ubuntu Touch prozentual dargestellt. Auf Ubuntu Touch enthält eine Applikation im Durschnitt 22 Elemente, wovon bereits 51% Buttons sind. Darauf folgen eine relativ gleichverteilte Mischung an Labels, List Views, Text Eingaben, Check Boxes und Drop Down Menüs, welche 44% aller Elemente auf Ubuntu Touch bilden. Die verbleibenden 5% teilen sich eher speziellere Elemente wie dynamische Bilder oder Tab Views.

2.1.2 Elemente auf Android

Android ist wie Ubuntu Touch ein kostenfreies, opensource mobiles Betriebssystem, welches auf über 80% der mobilen Geräte installiert ist. (Stand 2016). Android wird durch ein von Google gegründetes Konsortium, der Open Handset Alliance, entwickelt und ist seit 2008 auf dem Markt. Die Basis des Betriebssystems ist der Linux-Kernel, allerdings weicht Android von klassischen Linux-Distributionen stark ab, denn es sich orientiert sich weniger an Unix, sondern implementiert stattdessen viele Konzepte in Java.[3]

Um neben Ubuntu Touch ein klassisches mobiles Betriebssystem zu untersuchen wurde Android 7.0 analysiert, da durch den hohen Marktanteil von Android sich wiederspiegeln sollte, welche die am häufigsten vorkommenden Elemente in klassischen mobilen Applikationen sind.

[3] https://de.wikipedia.org/wiki/Android_(Betriebssystem)

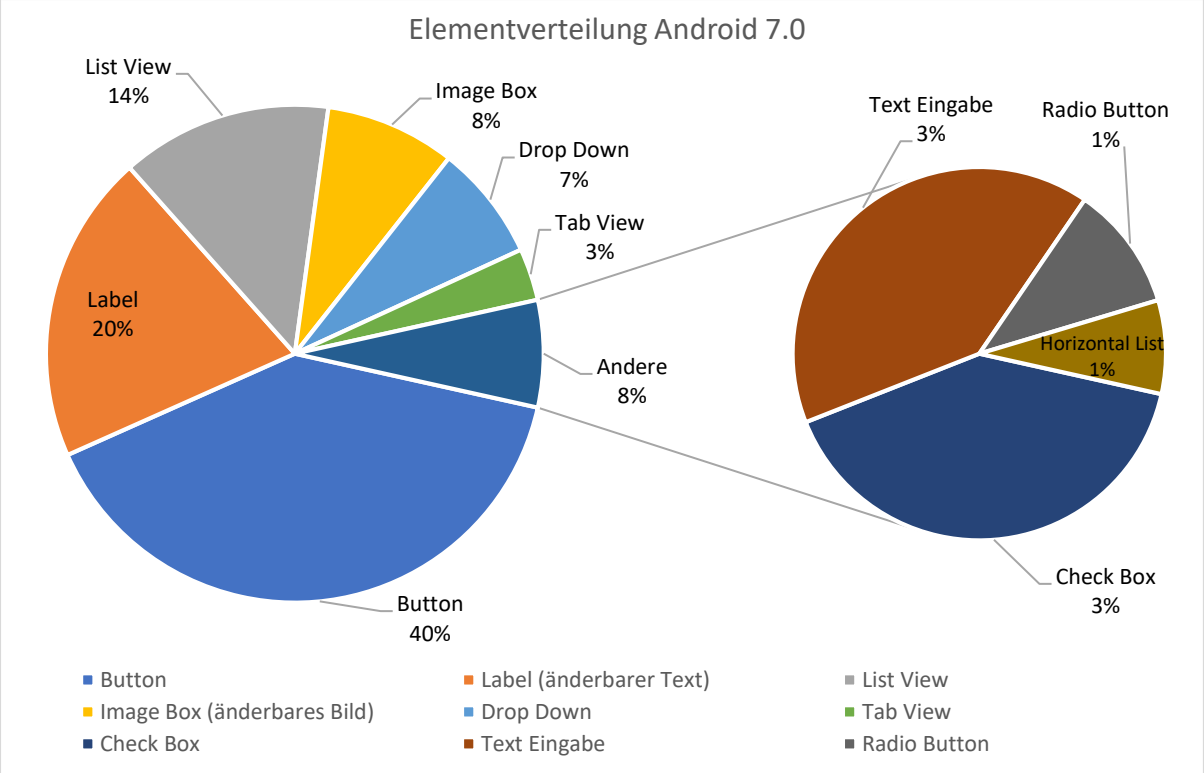


Abbildung 5: Elementverteilung auf Android 7.0

Abbildung 6 stellt die prozentuale Aufteilung aller Elemente der analysierten Applikationen auf Android dar. Android Anwendungen enthalten im Durchschnitt 40 Elemente, somit beinahe doppelt so viele wie eine durchschnittliche Ubuntu Touch Applikation. Die meist dynamischeren Inhalte von Android Anwendungen benötigen meist speziellere Elemente als die solchen auf Ubuntu Touch, die eher an Rich-Client Anwendung erinnern. Dadurch lässt sich die erhöhte Anzahl an Elementen und ihrer Diversität erklären welche sich im Diagramm wiederspiegelt. Trotz der unterschiedlichen Designparadigmen der Betriebssysteme, sind 40% der enthaltenen Elemente in Android Anwendungen ebenfalls Buttons, gegenüber 51% auf Ubuntu Touch. Dynamische Inhalte, Labels, List Views und Image Boxes, bilden mit aufsummierten 42% einen deutlich größeren Block als auf Ubuntu Touch. Die restlichen 18% teilen sich speziellere Kontroll- und Interaktionselemente, wie Tab Views, Drop Down Menüs und Text Eingaben.

2.1.3 Betrachtung der Gesamtverteilung der Elemente

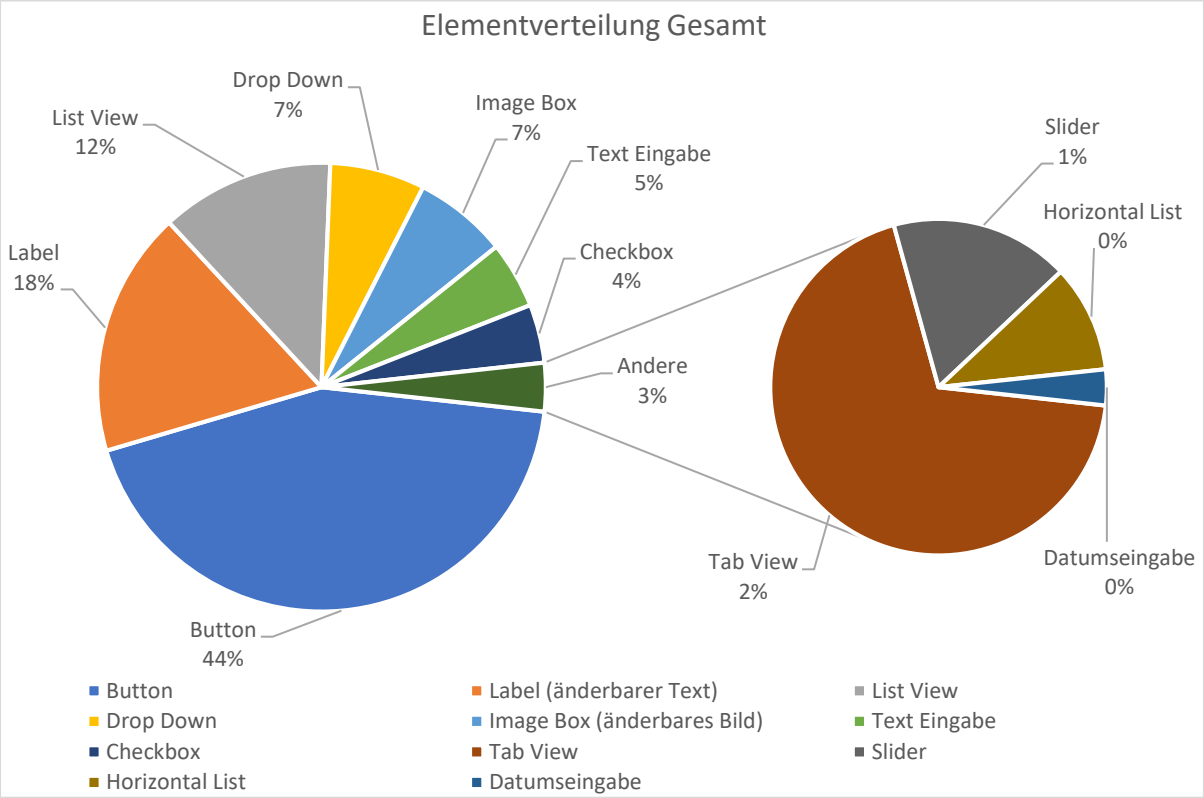


ABBILDUNG 6: GESAMTVERTEILUNG DER ELEMENTE

In Abbildung 7 wird die Gesamtverteilung der Elemente über beide Betriebssysteme hinweg dargestellt. Aus diesem Diagramm wurde abgeleitet welche Elemente mit Vorrang bei der Entwicklung von MockITUp implementiert wurden. Je häufiger ein Element im Durchschnitt in einer mobilen Anwendung vorkommt, umso wahrscheinlicher ist es, dass ein Kunde oder Entwickler es im MockUp-Prozess verwenden möchte.

Daraus lässt sich ableiten, dass zuerst der Button als das Element mit der meisten Funktionalität implementiert werden sollte. Anschließend kann die MockITUp-Anwendung um die dynamischen Inhaltsträger Label, List View und Image Box erweitert werden. Neben dem Button sollte mit Priorität die Text Eingabe implementiert werden um die Inhaltsträger auch mit Daten füllen zu können. Mit einer Implementation mit diesen fünf Elementen, die durchschnittlich 86 % aller Elemente ausmachen, sollte es möglich sein die meisten Anwendungen zu prototypisieren, da einige fehlende Elemente auch durch Kombination von Aktionen und implementierten Elemente abgebildet werden können.

2.2 Häufigkeit von Aktionen in mobilen Anwendungen

Neben der Festlegung der Implementierungsreihenfolge der Elemente für die MockUps, mussten die wichtigsten Aktionen die mit diesen Elementen durchgeführt werden ermittelt werden um festzustellen mit welcher Priorität sie zu MockITUp hinzugefügt werden sollten.

Hierbei wurde parallel zu der Zählung der Elemente bei der Beschriftung dieser in den Screenshots die möglichen Aktionen der Elemente festgehalten und zum Ende hin einmal für die jeweilige Plattform und im Anschluss über die Betriebssysteme hinweg aufsummiert und veranschaulicht.

Diese stichprobenartige Zählung und Gegenüberstellung der möglichen Aktionen auf den Systemen, ermöglicht einen Einblick darauf welche Aktionen am häufigsten genutzt werden und das mit geringerem Aufwand als eine Untersuchung auf die tatsächliche Häufigkeit der Nutzung der Aktionen durch Anwender.

2.2.1. Aktionen auf Ubuntu Touch

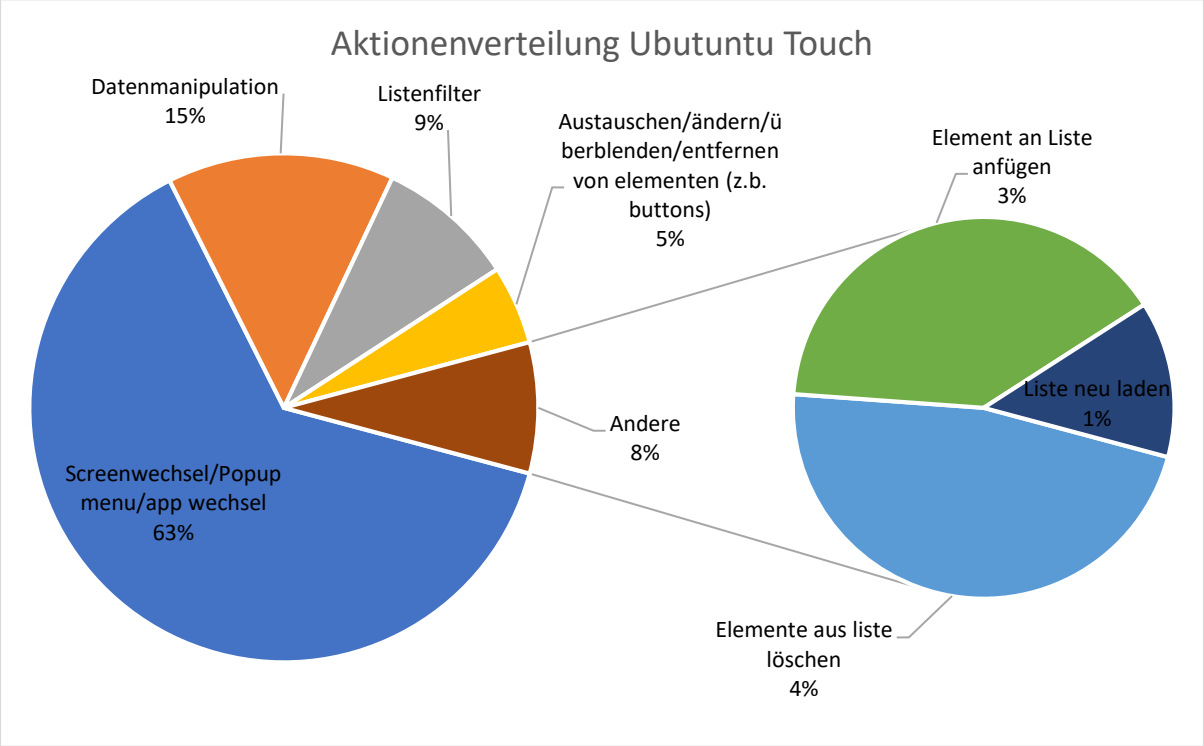


ABBILDUNG 7: AKTIONENVERTEILUNG AUF UBUNTU TOUCH

In Abbildung 7 ist dargestellt welche Aktionen auf Ubuntu Touch durchgeführt werden können und wie häufig sie vorkommen. Mit 63% ist der Wechsel von einem Screen der Applikation zu einem anderen oder der Wechsel vom gesamten Anwendungskontext zu einem anderem, die am Häufigsten vorkommende Aktion. Das Manipulieren von Daten, also die Änderung von Stammdaten oder Bewegungsdaten durch den Nutzer belegt 15% der möglichen Aktionen und ist gefolgt mit dem Filtern von Listen nach bestimmten Kategorien oder das Durchsuchen dieser nach Wörtern mit 9%. Die restlichen 13% der möglichen Aktionen bestehen aus dynamischen Verändern der Benutzeroberfläche und listenspezifischen Aktionen.

2.2.2. Aktionen auf Android

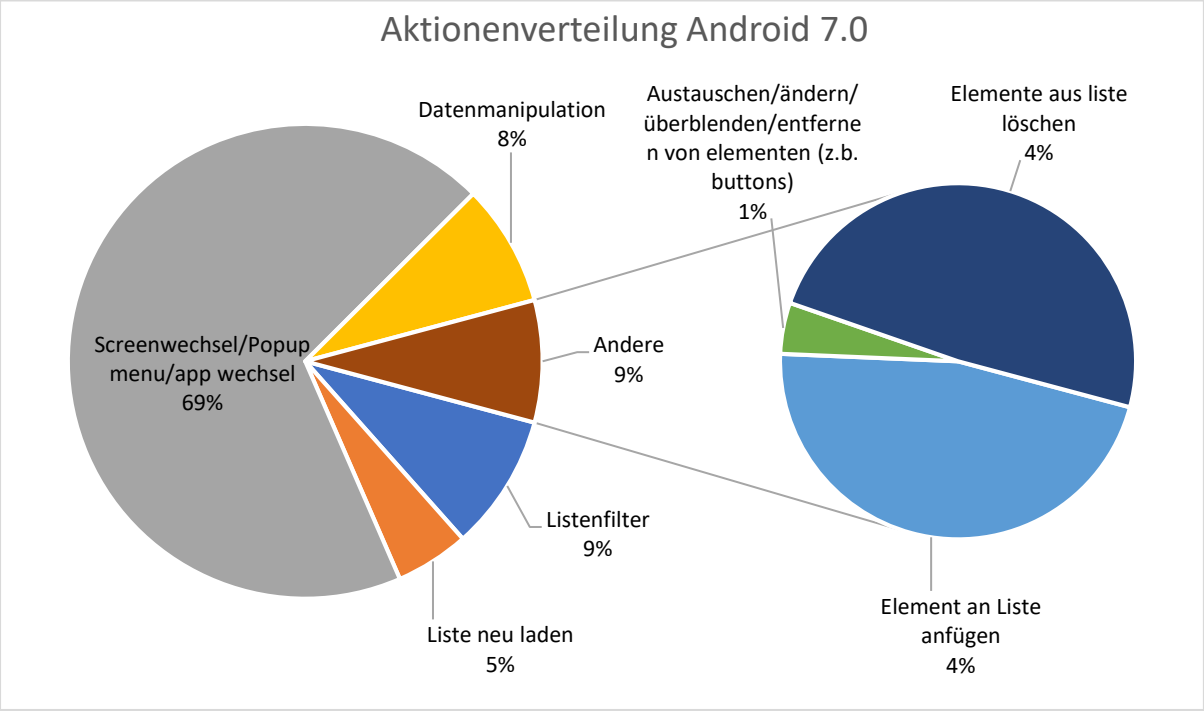


ABBILDUNG 8: AKTIONENVERTEILUNG AUF ANDROID 7.0

Im Gegensatz zu dem Vergleich der Elementverteilung der beiden Betriebssysteme in Abschnitt 2.1, sind die Aktionenverteilungen der beiden Systeme beinahe Deckungsgleich. Die einzige Aktion welche in der Rangfolge der Häufigkeit deutlich nach gesunken ist, ist das dynamische Verändern der Benutzeroberfläche, in Abbildung 8 zu erkennen. Alle anderen Aktionen haben ihren Stellenwert kaum verändert, lediglich die Listenaktionen weisen eine etwas höhere Häufigkeit auf Android auf als auf Ubuntu Touch. Die Manipulation von Daten ist mit 8% der gesamten Aktionen auf Android gegenüber den 15% auf Ubuntu Touch etwas geringfügiger vertreten. Der Wechsel oder die Veränderung von Kontext ist mit 69% allerdings auch auf Android die meist mögliche Aktion.

2.2.3. Gesamtverteilung der Aktionen

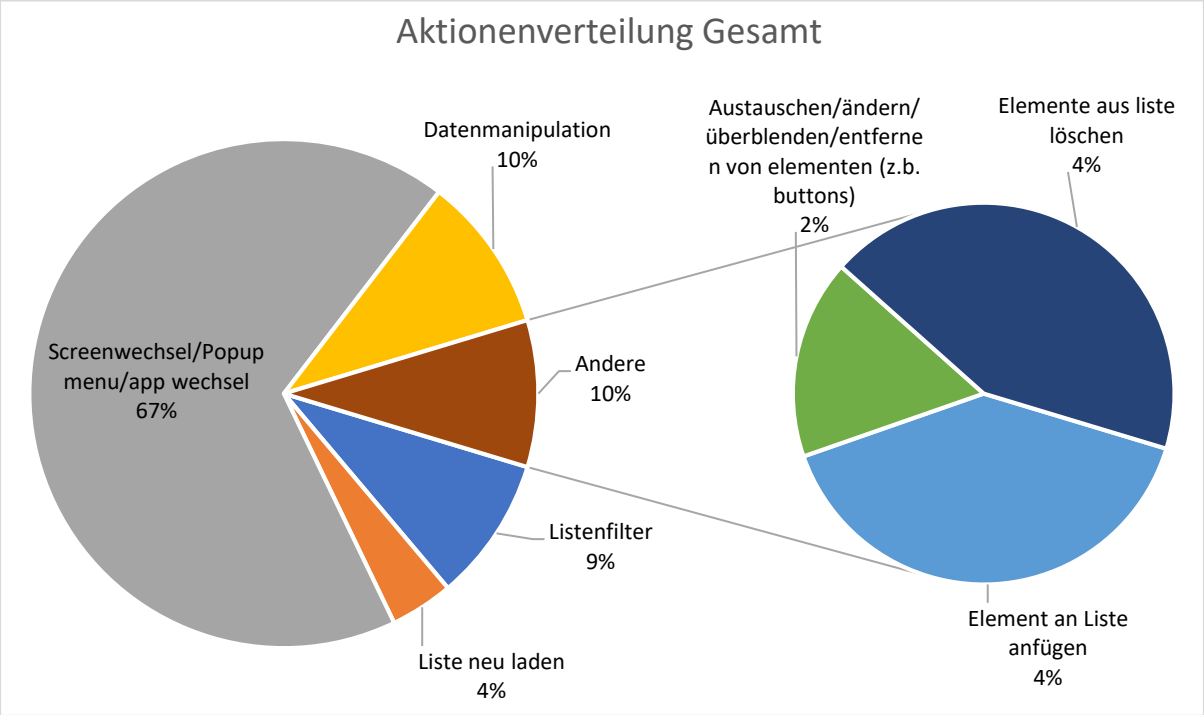


ABBILDUNG 9: GESAMTEVERTEILUNG DER AKTIONEN

Wenn die gesamte Verteilung der Aktionen über die Häufigkeit ihres Vorkommens auf beiden Systemen betrachtet, in Abbildung 9 dargestellt, wird schnell deutlich, mit welcher Priorität die Aktionen implementiert werden sollten. Der Wechsel oder die Änderung des Kontextes bzw. des Screens der Anwendung sollte als Folge der erlangten Verteilung als erste Aktion im Logik-Editor der Anwendung implementiert werden. Darauffolgen sollte die Implementierung der Änderung der Daten der Anwendung und die Listen spezifischen Aktionen. Das dynamische Verändern der Benutzeroberfläche sollte die geringste Priorität erhalten.

Als Fazit der Untersuchung der Häufigkeit der Aktionen auf Android 7.0 und Ubuntu Touch lässt sich zusammenfassen, dass trotz der deutlichen Unterschiede im Bereich der Elemente, im Abschnitt 2.1 beschrieben, die Aktionen sich auf den beiden Systemen eher geringfügig unterscheiden. Dies lässt sich auf die Navigation der mobilen Anwendungen zurückführen, welche unabhängig von der Gestaltung der Benutzeroberflächen auf den verschiedensten Betriebssystemen sehr konstant bleibt. Dies liegt an den Fehlen externer Human Interface Devices und der daraus resultierenden Konzentration auf die Touch-Eingabe als Hauptinterface zwischen System und Mensch.

3. Entwicklung der Anwendung

Der erste Schritt der Entwicklung von MockITUp war die Gestaltung der Oberfläche. Die Anwendung sollte so konzipiert werden, dass jeder Benutzer, auch ohne Kenntnisse in GUI-Design oder Programmierung von mobilen Anwendungen, MockITUp als interaktive Spielwiese der App-Gestaltung nutzen kann. Um dies zu ermöglichen wurde ein minimalistisches Design entworfen, welches visuelle Reizüberflutungen und dadurch eine schlechte Benutzerführung verhindern sollte. Da im Gegensatz zu dem in der Einleitung erwähnten AppInventor keine vollständigen Applikationen generiert werden sollen, sondern MockUps mit Interaktionsmöglichkeit, haben wir die Zahl der nötigen Dialoge auf ein Minimum reduziert und auf Werkzeuge wie einen Eigenschafteninspektor für die Elemente oder Screens vollkommen verzichtet. Der spätere Anwender sollte beispielsweise die Möglichkeit besitzen seinem Kunden das MockUp mitzugeben, damit dieser sich in Ruhe nicht nur ein Bild über die Benutzerführung auf seinem eigenen Gerät bilden kann, sondern hat zusätzlich selber ohne große Einarbeitungszeit verschiedene Möglichkeiten für die Optimierung der User Experience durchzuprobieren.

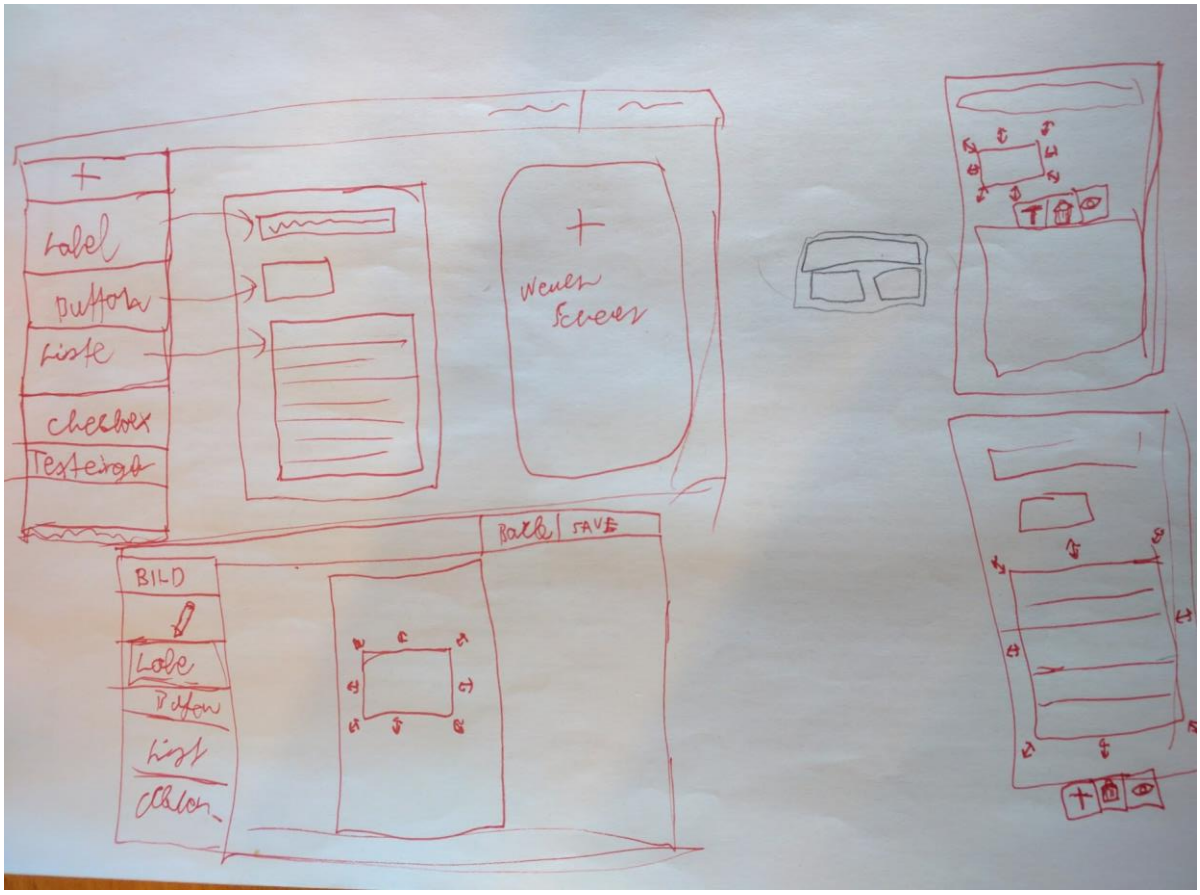


Abbildung 10: Scribbles der Oberfläche von MockITUp

Mit diesem grundsätzlichen Konzept, „so viel wie nötig aber so wenig wie möglich“, wurden die ersten Scribbles, Abbildung 9, entworfen. Diese Scribbles dienten zum einen als Vorlage für die

spätere Entwicklung der Oberfläche, allerdings konnten durch sie auch festgestellt werden ob alle Funktionen die Benutzer braucht in der Oberfläche vorhanden sind. Diesen Prozess, das Paper Prototyping, wurde so durchgeführt, dass versucht wurde zwei der untersuchten mobilen Applikationen aus Abschnitt 2., jeweils eine Android und eine Ubuntu Touch basierte, mit Hilfe dieser Handgezeichneten Benutzeroberflächen zu replizieren.

Es konnte festgestellt werden, welche essentiellen Funktionen und GUI-Elemente in den Scribbles fehlen um zu verhindern, dass das Fehlen dieser erst später im Entwicklungsprozess auffällt und dort ein deutlich größerer Aufwand zu beheben ist.

Die Auswahl der Technologie mit der MockITUp realisiert werden sollte fiel auf eine moderne Web-Anwendung mit HTML, CSS und JavaScript. Eine Webtechnologie basierte Applikation versprach die schnelle Entwicklung einen funktionierenden Prototyp und gleichzeitig den Vorteil der Plattformunabhängigkeit, die im Rahmen der Funktion des Ausführens des MockUps auf einem mobilen Gerät gefragt ist. Die benötigten Sprachen sind weit verbreitet wodurch die Erweiterbarkeit bei genügend Strukturierung gegeben sein sollte.

ABBILDUNG 11: ERSTE ITERATION DER BENUTZEROBERFLÄCHE

Nachdem die aus dem Paper Prototyping entstandenen Scribbles in HTML und CSS zu der in der Abbildung 10 abgebildeten Benutzeroberfläche umgesetzt wurden startete die Implementierung der ersten Funktionen des Editors in JavaScript. Der erste Meilenstein in der Entwicklung war die Fertigstellung der Grafik-Ansicht des Editors, in welcher die verschiedenen Elemente auf die Screens der Applikation hinzugefügt werden können, inklusive der Implementierung der ersten beiden Elemente, dem Button und das Label. Während dieser Entwicklungsphase mussten wir uns zwischen zwei Möglichkeiten der Darstellung der Screens und Elemente entscheiden. Die erst Möglichkeit die

wir in Betracht gezogen haben war mithilfe von JavaScript die HTML Elemente zu generieren und diese in Verbindung mit CSS-Klassen zu benutzen. Die Screens wären dabei durch DIV-Objekte realisiert, die sich durch JavaScript Funktionen wie benötigt verhalten. Der Alternativansatz zur Implementierung der Elemente und Screens war ein Canvas für die Darstellung der Screens zu nutzen und auf diese Canvas die Elemente zu Zeichen. Das Ganze würde mithilfe der Bibliothek EaselJS[4] realisiert werden, welche das Arbeiten mit der HMTL 5 Canvas vereinfacht.

Ausschlaggeben für die Entscheidung war das minimalistische Grundkonzept der Anwendung, weswegen auf das Verwenden der Canvas verzichtet wurde. Die hohe Einarbeitungszeit in die Bibliothek hätte nicht nur den Entwicklungsprozess verlangsamt, sondern den Aufwand der Erweiterung der Anwendung zu einem späteren Zeitpunkt durch andere Entwickler deutlich erhöht.

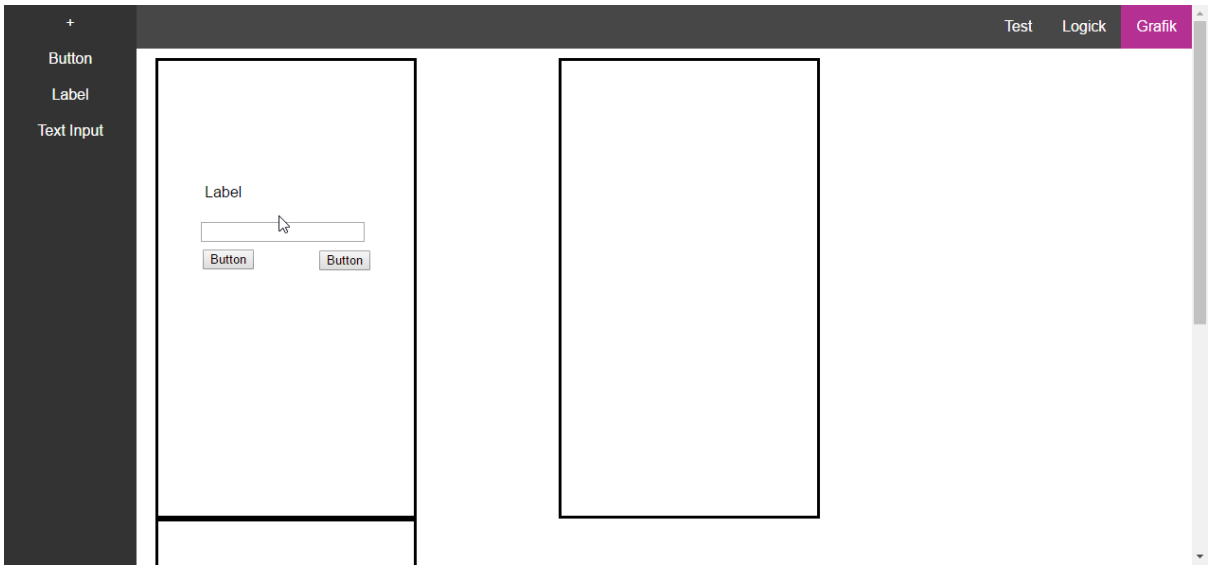


ABBILDUNG 12: IMPLEMENTIERUNG DES GRAFIK-EDITORS

Nach der Fertigstellung des Grafik-Editors folgte die Entwicklung des Logik-Editors. Dieser dient dazu die Interaktivität des MockUps zu ermöglichen. Zunächst sollte allerdings die Logikgenerierung des interaktiven Mock-Ups vollständig ohne Programmiererfahrung erfolgen können, daher haben wir auf eine visuelle Programmierschicht im klassischen Sinne wie bei der Anwendung App Inventor vom MIT oder der visuellen Programmiersprache SCRATCH, siehe Abbildung 13, verzichtet. Diese Sprachen bestehen aus Visuellen Programmblöcken und visuellen Objekten, welche per „Drag&Drop“ im Baukasten Prinzip zusammengestellt werden können.

Im Beispiel SCRATCH[5] kann jedem Objekt ein Programmblock aus vielen verschiedenen Bausteinen, bzw. Befehlen zugewiesen werden. Darunter fallen z.B. Die Bewegung des Objektes, visualisiert durch eine Computergrafik in der Entwicklungsoberfläche, das Ändern des Aussehens des Sprites,

[4] http://www.createjs.com/easeljs

[5] http://www.olinger.net/iWeb/berufliches/Scratch_files/scratch_referenzhandbuch.pdf

Steuerungselemente wie Abfragen oder Schleifen etc. Die Objekte in SCRATCH, oder aufgrund ihrer visuellen Darstellung auch Sprites genannt, haben ihre eigenen Programmblöcke und können daher autonom auf Ereignisse oder auch Nachrichten von anderen Sprites reagieren, ähnlich dem Actors Modell für Nebenläufigkeit[6].

Diese verhaltensorientierten Systeme, eine Mischung aus objektorientierter und constraints-orientierter Systemen mit visuellen Akteuren sind das zugrundeliegende Paradigma auf welchem auch MockITUp aufgebaut ist. Visuellen Objekten wird ein Verhalten zugewiesen, welches durch fertige Skript-Komponenten festgelegt werden kann. Attribute der Objekte können ihr Verhalten beeinflussen. Die Erstellung von eigenen visuellen Skripten für das Objektverhalten wie in den genannten Beispielen könnte als spätere Weiterentwicklung noch hinzugefügt werden um erfahrenen Nutzer zusätzliche Möglichkeiten für die Entwicklung der Mock-Ups zu bieten, wurde aber im Prototypen zunächst nicht implementiert.

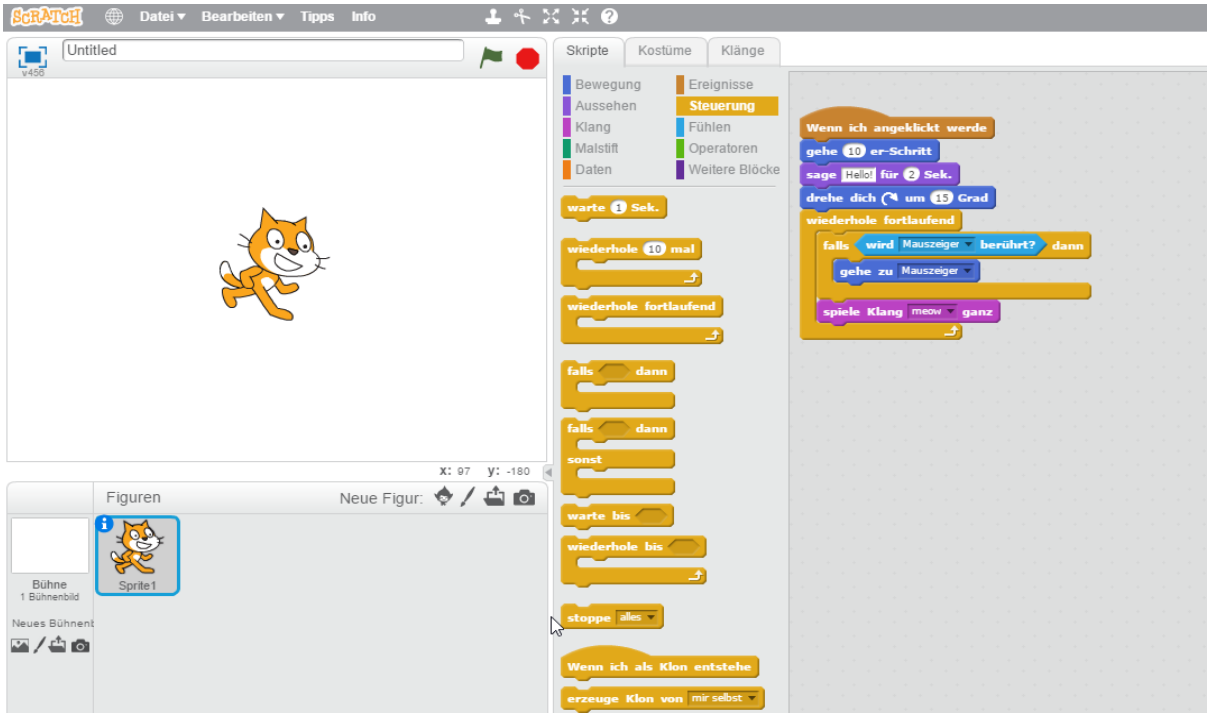


ABBILDUNG 13: VISUELLE PROGRAMMIERSPRACHE SCRATCH

Nach diesem finalen Meilenstein wurde durch einen Usability Test festgestellt wie vollständig die Benutzerführung ist und wo Optimierungen nötig sind. Im Folgenden sind die Ergebnisse dieses Tests kurz zusammengefasst.

Es wurde festgestellt das es für den Benutzer nicht intuitiv ist die Elemente von der Elementleiste auf die Screens zu Drag&Droppen, deshalb wurde eine Notifikation beim Klicken auf einen der Einträge der Elementleiste eingefügt welche dem Nutzer mitteilt das Drag&Drop angewendet

[6] Carl Hewitt. Peter Bishop, Richard Steiger

werden sollte. Außerdem konnte ermittelt werden, dass Drag&Drop für die Logik-Ansicht nicht geeignet ist, dort wurde die Interaktion auf einzelne Klicks umgestellt mit Anweisungen für den Nutzer was zu tun ist.

Wenn über ein klickbares Element mit dem Mauszeiger gefahren ist, war dieses nicht als solches zu erkennen, da der Mauszeiger sich nicht verändert hat, um dies zu beheben wechselt der Mauszeiger von dem klassischen Pfeil zu einer Hand, falls er auf etwas klickbares zeigt.

Einige Elemente wie zum Beispiel das Text Input Element hatte Probleme mit der Größenskalierung. Das Skalierungsverhalten war nicht immer wie vom Benutzer erwartet und wurde angepasst.

Es wurde sich ein Raster für die Elemente gewünschte, mit einer Art Snap-To-Grid Funktion, dieser Wunsch wurde jedoch nicht implementiert. Außerdem ist ein Bug aufgetreten der behoben wurde, einige Elemente konnten nicht gleichzeitig Auslöser und Ziel einer Aktion sein.

Angemerkt wurde, dass es bei einem Text Input Element intuitiver sei, wenn durch eine Bestätigung durch drücken der Enter-Taste der Dialog beendet werden könnte, dies wurde jedoch noch nicht implementiert. Die Orientierung von Text sollte geändert werden können, weswegen nun zwischen linksbündig, mittig und rechtsbündig gewählt werden kann.

Zuletzt kam der Wunsch nach einem Textfeld welches mehrere Zeilen umfassen kann, dies wurde so implementiert.

4. Technische Dokumentation

In diesem Abschnitt sind die wichtigsten Aspekte der technischen Seite des Projektes festgehalten um die Funktionsweise des Programms nachvollziehen zu können. Außerdem werden die Einstiegspunkte für die Weiterentwicklung der Anwendung dokumentiert, unter anderem wie neue Elemente oder Aktionen implementiert werden können.

4.1 Aufbau der Anwendung

Am Anfang dieses Abschnitts wird der grobe Programmablauf der Anwendung ab dem tatsächlichen Aufruf der Web-Anwendung erläutert. Im Anschluss findet sich eine Beschreibung der einzelnen Java-Script Module und die wichtigsten Funktionen derer.

4.1.1 Genereller Nutzungsablauf

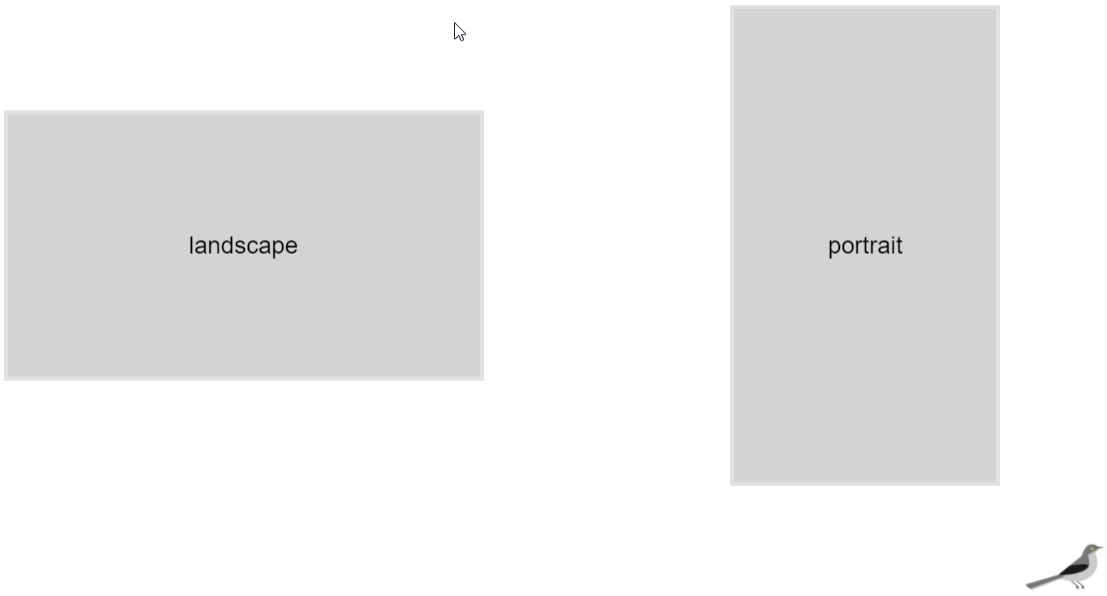


ABBILDUNG 14: START-SCREEN DER ANWENDUNG

In Abbildung 9 ist der Inhalt des Browsers zu sehen, wenn die Web Anwendung aufgerufen wird. Diese hier erkennbaren Elemente sind die einzigen, welche tatsächlich in der ind*ex.html* der Anwendung festgelegt sind, alle nachfolgenden Elemente und Screens werden durch das JavaScript generiert und dynamisch durch die Interaktion vom Benutzer modifiziert. Mit einem Klick auf einen der beiden Buttons wird dann der Editor für die MockUps mit der dementsprechenden gewählten Orientierung initialisiert. Dies erfolgt durch die *initialize* Funktion im dem JavaScript Modul *onLoad.js*. Die Funktion generiert die Navigations-Elemente, sowie den ersten, leeren Screen des MockUps und berechnet die Positionen für diese im Browser, siehe Abbildung 10.

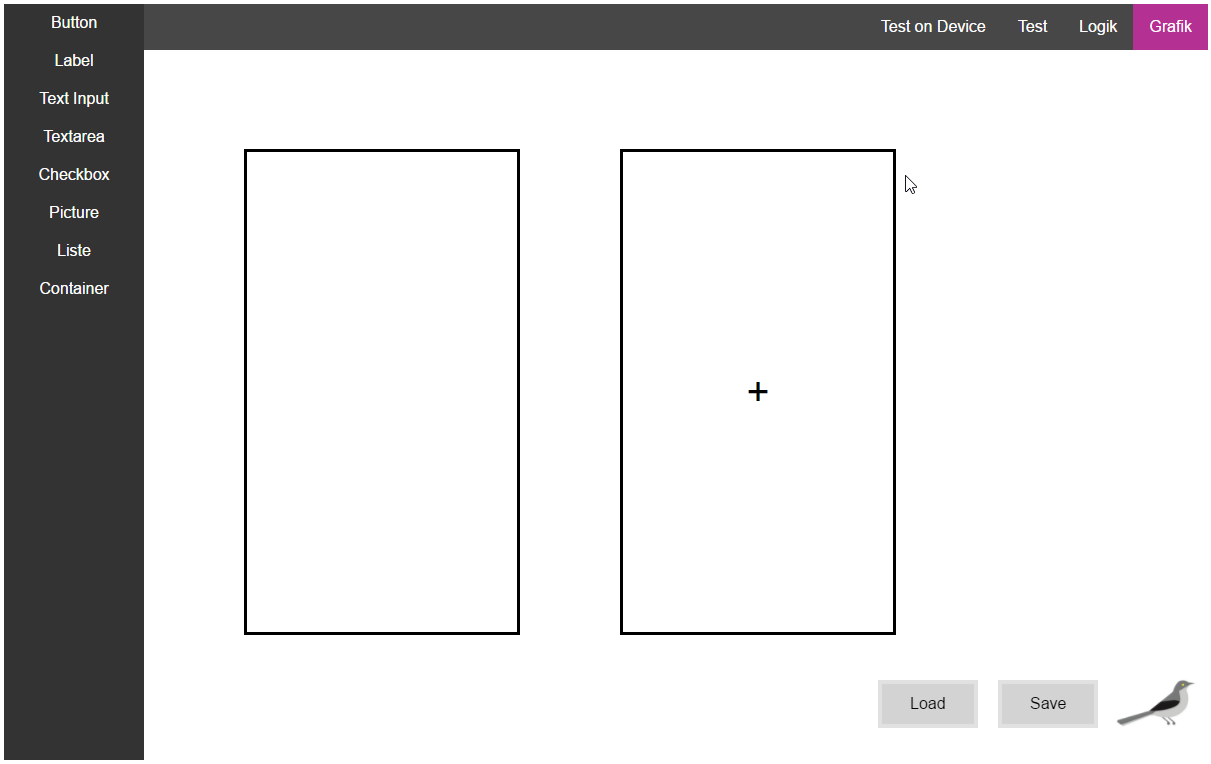


ABBILDUNG 15: MOCKUP-EDITOR MIT LANDSCAPE SCREENS INITIALISIERT

Legt man nun im MockUp-Editor, siehe Abbildung 10, seine ersten Elemente an, wird das von dem JavaScript Modul *elements.js* verwaltet. Es regelt die Generierung der HTML-Elemente auf den Screens des MockUps und das Laden dieser aus einem gespeicherten MockUp. Die Funktionen im Modul erstellen zuerst das jeweils benötigte DOM-Objekt für das ausgewählte Element, welches dann durch das Decorator-Pattern mit Funktionen, Variablen und Eigenschaften ausgestattet wird. Schließlich wird die *create* Funktion des fertigen Elements aufgerufen, die es auf den beabsichtigten Screen positioniert und eventuelle Standardwerte einfügt.

Das Anlegen neuer Screens wird von den Funktionen *new_Screen* und *create_Screen* in der main.js verwendet, die sich um die richtige Positionierung und alles Weitere kümmern um die Screens zu initialisieren und dem Dokument hinzuzufügen.

Wechselt man jedoch durch die obere Navigationsleiste in eine andere Ansicht, wird eine Funktion aufgerufen die den Wechsel auf die ausgewählte Ansicht übernimmt. Diese Funktionen rufen zu Beginn stets die Funktion *general_screenchange_cleanup* im *main.js* Modul auf, welche im Grunde genommen zuerst die ansichtsspezifischen Elemente aus dem DOM-Baum entfernt oder ansichtsübergreifende Elemente durch CSS-Modifikation ausblendet und generiert die neue Ansicht im Anschluss.

4.1.2 Beschreibung der wichtigsten Module

Im Folgenden werden die Module, die jeweils durch die `.js`, `.html` und `.css` Dateien gebildet werden, kurz beschrieben und die nennenswertesten Funktionen erwähnt.

Index.html

Die bereits 4.1.1 angesprochene `index.html` ist der wichtigste HTML Code im Projekt und beläuft sich mit 46 LOC zum größtenteils auf die Einbindung der JavaScript und CSS Dateien. Außerdem ist dort der erste Screen der Web-Anwendung implementiert, alles Weitere an HTML was der Benutzer im Browserdargestellt bekommt wird durch das Java Script generiert.

Testonmobile.html

Dieser HTML Code wird durch das PHP-Script `publish.php` automatisch generiert, wenn der Benutzer das MockUp auf seinem Mobilgerät testen möchte. Ein entsprechender eingerichteter Webserver stellt diese HTML-Datei dann zur Verfügung und dem Benutzer wird ein QR-Code in der Anwendung angezeigt, dessen URL auf den Server verweist. Die Verwendung dieser Funktion ist völlig optional, wenn kein Webserver eingerichtet ist, wird ein QR-Code generiert der einer Fehlermeldung entspricht. Der Name dieser HTML-Datei wird durch den aktuellen Timestamp des Systems bei seiner Generation bestimmt und alle Dateien, welche älter als einen Tag sind, werden automatisch gelöscht.

Main.css

Das einzige CSS Modul des Projektes enthält die Klassen die den in den JavaScript Modulen erzeugten HTML Elementen ihr Positionsverhalten und Aussehen geben.

Publish.php

Das `publish.php` Modul ist das einzige PHP-Modul dieses Projektes. Es enthält Funktionen für die Erzeugung der HMTL-Datei für den Webserver aus übergebenen HMTL-Code aus dem `testrenderer.js` Modul für das Testen den MockUps auf externen Geräten.

onLoad.js

Dieses JavaScript Modul ist dafür verantwortlich, dass nach der Auswahl der Orientierung der Screens des MockUps der eigentlich MockUp Editor initialisiert wird.

Der Funktion `initialize(selection)` wird die entsprechende als Parameter übergeben. Anschließend werden die Navigations- und Notfikationselemente und der `ScreenContainer`, welcher der Parent der aller MockUp Screen im DOM Baum ist, mit seinem ersten Screen generiert und dem Dokument hinzugefügt sowie die nicht mehr benötigten Elemente aus der vorherigen Ansicht ausgeblendet. Näheres zu diesen Elementen ist im Abschnitt zum Modul main.js zu finden da sich dort die Funktionen zu der Generierung dieser befinden.

Die zweite erwähnenswerte Funktion ist *portraitPosition()*, die als *onload*-Funktion des HTML-Bodys festgelegt ist, das bedeutet sie wird aufgerufen sobald der Browser den HMTL-Body der *index.html* lädt. Sie berechnet die Positionen einiger Elemente relativ zum Seitenverhältnis und der Auflösung des Browser-ViewPorts und fügt den beiden DIV-Elementen auf dem Startscreen das onClick-Verhalten mit Bezug auf die i*nitialize* Funktion.

Main.js

Das *main.js* Modul verwaltet alles was mit dem *ScreenContainer* und seinen Inhalt, die beiden Menüleisten, einmal links und einmal oben am Rand der Ansicht, und deren Inhalt zu tun hat. Außerdem sind die grundlegenden Editor-Funktionen der Anwendung hier implementiert wie das das Skalieren und die Menüleiste unter den bereits auf Screens platzierten Elementen.

Der *ScreenContainer* ist ein DIV-Element der als Parent für die Screens im DOM-Baum fungiert und für den Nutzer in der Oberfläche nicht ersichtlich ist. Durch die Verwendung des *ScreenContainers* können die Screens deutlich einfacher iteriert werden oder z.B. ausgeblendet werden indem lediglich das CSS des *ScreenContainers* geändert werden muss. Seine Kinder im DOM-Baum, die einzelnen Screens des MockUps, werden von den Funktionen *new_Screen()* und *create_Screen()* erstellt. Die Funktion *new_Screen()* ruft zu Anfang die *create_Screen()* Funktion auf, welche das DOM-Element erstellt mit Rücksichtnahme auf die gewählte Orientation, Landscape oder Portrait, und gibt das Element an *new_Screen()* zurück. Diese Funktion dekoriert das Element noch mit einigen Eigenschaften und Unterelementen und fügt es in den *ScreenContainer* ein.

Die beiden Menüleisten werden von den Funktionen *menubar(classname)*, *elementbar()*, *menubar_Item(…)* und *elementbar_Item(name, onclick)* generiert. Die *menubar(classname)* Funktion generiert, wenn unparameterisiert aufgerufen die obere Navigationsleiste, ansonsten wird die Menüleiste generiert, dessen CSS klasse als String der Funktion übergeben wird. Das nutzt die Funktion *elementbar()*, welche für die Generierung der Elementleiste links im Browser die *menubar(classname)* Funktion mit dem „elementbar“ Parameter aufruft. Die Elemente für die Einträge der Menüleisten werden von der menubar_Item(..) Funktion generiert. Analog wie bei den Funktionen für die Menüleisten kann dieser Funktion ein Klassenname übergeben werden um für verschiedene Leisten Einträge zu erstellen. Weitere Parameter sind die spätere Beschriftung des Eintrags und das *onClick*-Verhalten für dieses Element. *elementbar_Item(name,onclick)* nutzt die menubar_Item(…) mit entsprechenden Parametern um die Einträge für die linke Menüleiste zu erstellen, ein Aufruf ohne Klassenparameter erstellt die Einträge für die Navigationsleiste.

Settingsbar(parent) ist die Funktion welche die Elementspezifische Menüleiste unter den auf den Screens platzierten Screens generiert. Das im Parameter übergebene Element wird die Leiste

angefügt. In dieser Funktion sind einige weitere Funktionen definiert, welche unter anderem für das Skalieren der Größe der Elemente durch den Benutzer auf den Screens zuständig sind. Diese Funktion wird von den elementgenerierenden Funktionen in *elements.js* aufgerufen.

Neben diese essentiellen Funktionen sind in dem Modul noch einige Hilfsfunktionen z.B. *imageSelect(target,set)* in *main.js* enthalten, die dem Nutzer einen Dialog anzeigt welcher dem Benutzer Möglichkeiten bietet ein Bild an die Anwendung zu übergeben.

EditPicture.js

Dieses Modul beinhaltet alles, was für die Zeichen Ansicht verantwortlich ist. Diese Ansicht dient zum Zeichnen eigener Elemente oder dem verändern von Bilder die vom Benutzer hinzugefügt wurden.

Elements.js

Dieses Modul ist für die Elemente in der Grafik-Editor Ansicht zuständig. Es enthält die Funktionen um die Elemente zu erstellen und aus gespeicherten MockUps zu laden.

In dem Modul ist das globale Array *elements* definiert, in welchem die Funktionen für das generieren der Elemente gespeichert werden, welche vom Aufbau her an klassische OOP-Konstruktoren erinnern. Mithilfe dieses Arrays kann an anderer Stelle die Einträge für die Elementleiste ohne größeren Aufwand erstellt werden. Jedes registrierte Element hat zwei Funktionen, eine für die Erstellung einer neuen Instanz des Elements und eine um das Element bei dem Laden eines gespeicherten MockUp mit seinen Logikverbindungen wiederherzustellen.

Fast jede der elementgenerierenden Funktionen ruft nach Erstellung des DOM-Elements die Funktion *make_Container(elem, elemtyp)* auf. Diese Funktion nimmt das übergebene DOM-Objekt, der erste der beiden Parameter und stattet es mit Eigenschaften und JavaScript-Funktionalitäten wie z.B. dem Drag-and-Drop aus. Außerdem werden die Einträge für die möglichen Aktionen dieses Elements in der Logik-Ansicht erstellt. Diese Funktionen sind im Abschnitt Programmier-Pattern noch gründlicher beschrieben.

Logick.js

Analog zum Aufbau des *elements.js* Modul für die Elemente ist das *logick.js* Modul aufgebaut um die möglichen Aktionen zu verwalten die mit den Elementen möglich sein sollen. In dem Modul befindet sich das globale Array *logick_transaktions* in welcher die logischen Verbindungen der Elemente bekanntgegeben werden. Die Funktion function *logick_transaktion (evoker, evoking_aktion,target,name)* stellt bei einem Aufruf mit den korrekten Argumenten eine Instanz dieser logischen Verbindungen her. Das erste Argument *evoker* ist das auslösende Element der logischen Transaktion, wobei die evoking_aktion die Aktion mit diesem Element ist welche einen

Effekt haben soll, z.B. das Klicken auf dieses Element. Das *target* ist das Element welches durch den Effekt der logischen Transaktion beeinflusst werden soll. Das letzte Argument *name*, ist der Name unter dem diese logische Transaktion bekanntgegeben werden soll. Außerdem fügt die Funktion logick_transaktion(…) durch den Aufruf der Funktion *addActionToSettings(png, transaction)* den beiden Elementen die an der logischen Transaktion beteiligt sind, Auslöser und Ziel, in ihrer *Settingsbar*, die Menüleiste unter den einzelnen Elementen auf dem Screen, jeweils ein Icon, stellvertretend für jede logische Transaktion die mit diesen Elementen interagiert, hinzu. Das „Hovern“ mit dem Mauszeiger über diese Icons hebt jeweils den Auslöser und das Ziel der Aktion hervor und gibt dem Benutzer Informationen über die auslösende Aktion, sowie den Effekt dieser logischen Transaktion.

Für jede logische Transaktion wird eine Funktion in logick.js angelegt, welche dann in *onLoad.js* dem Logik-Menü bekanntgegeben wird um sie im Logik-Editor in die linke Menüleiste zu laden. Die Funktionen für den Effekt der logischen Transaktion und zum Auslösen der logischen Transaktionen befinden sich in test_logick.js. Wichtig ist das die Namen der entsprechenden Funktionen und der Transaktionen übereinstimmen. Näheres zu diesem Modul ist in dem Abschnitt über die verwendeten Programmier-Patterns und dem Abschnitt über die Einstiegspunkte für die Weiterentwicklung der Anwendung zu finden.

test_logick.js

Dieses Modul ist eng mit dem *logick.js* Modul verzweigt. Es beinhaltet zum einen die Funktionen die für die Effekte der logischen Transaktionen zuständig sind, aber außerdem auch die Funktionen welche die logischen Transaktionen auslösen. Die Funktionen welche für die Wirkungen der Transaktionen zuständig sind werden im Array *test_logick_effekts regestriert*, die Funktionen für die Auslösung der Transaktionen im Array *test_logikc_events*.

Außerdem ist der Dispatcher der Events bzw. Effekte in diesem Modul mit dem Namen *function test_do_execute(event,target,passedvalue)* implementiert. Dieser wird im Abschnitt Programmier-Patterns unter der Beschreibung des Observer-Patterns genauer betrachtet.

Im Array test_logik_transatkionen werden durch das testrenderer.js Modul die Zielelemente und die Effekte die auf sie wirken bei den Interaktionen registriert, da es in diesem Modul weniger wichtig ist welches Element die Transaktion auslöst, sondern nur welche Aktion und welche Wirkung diese auf das Zielelement haben soll.

testrenderer.js

Das testrenderer.js Modul beinhaltet alle Funktionen, die der Transformation der internen Datenstruktur, dem DOM-Baum mit JS-Funktionen, zu einer einfacheren Datenstruktur, reines HTML mit entsprechenden HTML-Datatags für JS-Funktionalität, dienen. Diese einfachere Datenstruktur kann anschließend genutzt werden um die Anwendung auf den Webserver zu übertragen und dient zusätzlich zur rekursiven Durchwanderung des DOM-Baums mit den Datatags, vergleichbar mit einem Compilervorgang, um die festgelegten Aktionen und deren Effekte Global zu registrieren um sie in der interaktiven Ansicht benutzen zu können. Außerdem nutzt dieses Modul qrcode.js um den QR-Code für das Testen der Anwendung auf einem externen Gerät zu ermöglichen.

Die wichtigsten Funktionen des Moduls testrenderer.js sind zum einen die Funktion `get_html()` und die darauf aufbauende Funktion function `TreeCompile(target)`. `get_html()` ist dafür verantwortlich das die festgelegte Logik der Elemente in HMTL-Datatags geschrieben werden, die den Elementen anhängen. Dadurch kann das gesamte MockUp ohne größere Mühen übertragen und anschließend die gesamte Logik aus den Datatags wiederhergestellt werden. Dafür ist die Funktion `TreeCompile(target)` zuständig. Sie ähnelt einem Compilervorgang, dabei wird der DOM-Baum des HTML-Dokuments systematisch nach Elementen durchlaufen. Die Datatags mit den Logik-Informationen werden für jedes Element untersucht und daraus die entsprechende JavaScript-Logik generiert und in test_logick.js registriert.

Der Grund für diese Vorgehensweise ist, dass wir dadurch nicht nur das ganze MockUp auf externe Geräte zu Verfügung stellen können, sondern es ebenfalls möglich ist lediglich Teile, wie einzelne Screens über den Webserver an andere Benutzer zu verschicken mithilfe der QR-Code Generation.

save.js

Das save.js Modul stellt alle Funktionen zum Speichern und Laden der MockUps auf dem lokalen System bereit. Dabei werden die MockUps in reinen HTML gespeichert mithilfe der Funktion `get_html()` aus `testrenderer.js`, welche die JavaScript-Logik auf HTML-Datatags zur Persistierung überträgt. Die Wiederherstellung der Logik-Verbindung erfolgt beim Laden ebenfalls analog zum `testrenderer.js` Modul. Das Laden der Elemente selber aus einem gespeicherten MockUp übernehmen die entsprechenden `elementname.createfromsave`() Funktionen aus dem elements.js Modul.

newElement.js

Dieses Modul enthält alle Funktionen die für die Erstellung eines benutzerdefinierten Elements nötig sind.

editPictures.js

Das Modul editPictures.js implementiert die Funktionen welche für das Bearbeiten von Bildern und der HTML-Canvas der Bilder benötigt werden.

qrcode.js

Dieses Modul ist von Kazuhiko Arase [7]entwickelt und unter der MIT Lizenz veröffentlicht worden. Es enthält die Funktionen die für die Generierung der QR-Codes.

4.1.3 Verwendete Programmier-Patterns

Decorator-Pattern

Das Decorator-Pattern ist ein Design-Pattern welches eine Alternative zu klassischen Unterklassenbildung der objetkorientierten Programmierung bildet. Eine Klasse wird durch einen sogenannten Decorator mit zusätzlicher Funktionalität ausgestattet um eine spezialisierte Instanz dieser Klasse zu erhalten. Der Benutzer kriegt von diesem Umweg nichts mit und erhält sein Objekt so wie er es erwartet.

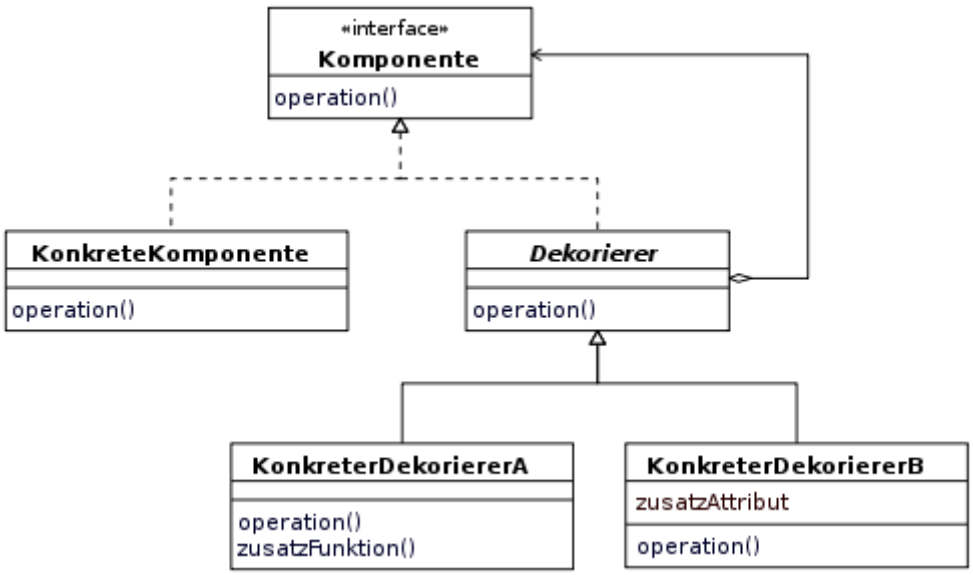


ABBILDUNG 16: DECORATOR-PATTERN[8]

Der Vorteil dieses Vorgehens besteht darin, dass mehrere dieser Decorator hintereinandergeschaltet werden können und sogar zur Laufzeit oder selbst nach der Instanziierung von Objekten ausgetauscht werden können. Da wir nicht nur viele ähnliche Funktionen für verschiedene Elemente benötigen, sondern es dem Nutzer unter anderem Erlauben seine eigenen Elemente zu erstellen oder zu Gruppieren, benutzen wir dieses Design-Pattern um unsere Elemente mit ihren Menüs,

[7] *http://www.d-project.com*
[8] *https://de.wikipedia.org/wiki/Decorator*

Interkationen und Funktionen auszustatten. Im Folgenden ist ein Beispiel aus *elements.js* zu erkennen in welchem dieses Pattern verwendet wird um das Element Button zu dekorieren.

```
elements["Button"] = function element_button(e,x,y)
{
        b = document.createElement("button");
        b = make_Container(b,"Button");

    b.jsoncreate = function(target)
    {

this.settingsbar.add(settings_Icon("textedit.svg",function(){text_input_ove
rlay(this.parentElement.parentElement,function(value){this.target.parent.in
nerHTML = value;})}));
    }
        b.appendChild(document.createTextNode("Button"));
    b.logick_menu.add(logick_menu_item("Text
Link",logick_button_textlink));
    b.logick_menu.add(logick_menu_item("Change
Text",logick_button_change_text));
        b.create(e,x,y)
}

function make_Container(elem,elemtype)
{
        /*Saves the type of element in the data tags*/
        elem.dataset["elementtype"] = elemtype;

    elem.style.position = "absolute";
    elem.settingsbar = settingsbar(elem);
    elem.jsoncreate = function(){};

    /*Defines if the element ist visible in the test renderer*/
    elem.isVisible = true;
    elem.dataset.isVisible = elem.isVisible;

    /*Menu containing the logic funktions of the element*/
    elem.logick_menu = logick_menu(elem);

elem.logick_menu.add(logick_menu_item("Abort",function(){reset_transaktion(
this)}));
    elem.logick_menu.add(logick_menu_item("Hide",logick_button_hide));
    elem.logick_menu.add(logick_menu_item("Make
Visible",logick_button_unhide));
    elem.logick_menu.add(logick_menu_item("Toggle
Visibility",logick_button_toggle_visibility));



    elem.togglevisible = function()
    {
        if(this.isVisible) this.style.opacity = "0.3";
        else this.style.opacity = "1";
        this.isVisible = !this.isVisible;
        this.dataset.isVisible = this.isVisible;
    }
.....
.....
```

Jede Funktion in *elements.js* die für die Erstellung einer neuen Instanz eines Elements zuständig ist erstellt zunächst das entsprechende HMTL-Element und fügt es dem DOM-Baum des HTML-Dokumentes hinzu. Anschließend wird die Funktion *make_Container(elem, elemtyp)* aufgerufen, welche das entsprechend übergebene Element so mit Funktionen und Eigenschaften ausstattet wie es für dieses Element festgelegt wurde. Anschließend wird dem Element noch seine spezifischen Menüfunktionen angehängt über die *Settingsbar* und die logischen Operationen die von diesem Element ausgeführt werden können.

Observer-Pattern

Das Observer-Pattern ist ein Softwareentwicklungsmuster, das für die Strukturierung von Verhalten genutzt werden kann. Es ermöglicht die Weitergabe von Informationen über die Änderung bzw. Interaktion eines Objektes an die verbundenen Objekte oder Funktionen dieses Objekts. Das Ganze findet Anwendung, wenn die Änderung eines Objektes Änderungen an anderen Objekten hervorruft oder andere Objekte über diese Änderung benachrichtigt werden sollen ohne im direkten Kontext mit dem Objekt zu stehen.

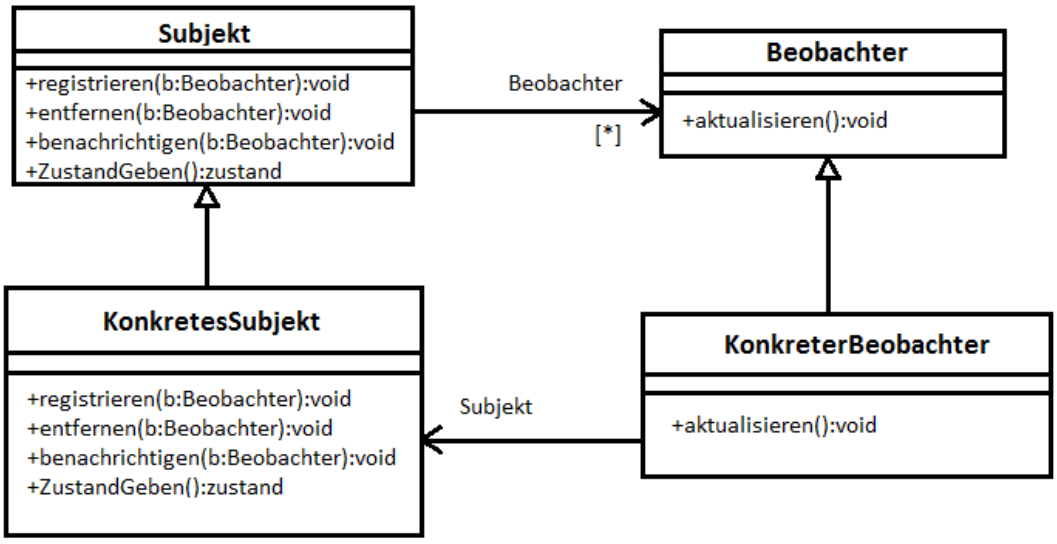


ABBILDUNG 17: OBSERVER-PATTERN[9]

Die Vorteile dieses Vorgehens in der Entwicklung liegt darin, dass die Subjekte und die Beobachter getrennt voneinander getauscht werden können. Die beiden sind auf eine lose Art über Abstraktion miteinander verbunden, wodurch das Subjekt keinerlei Informationen über seine Observer benötigt.

[9] *https://de.wikipedia.org/wiki/Beobachter_(Entwurfsmuster)*

Dabei erhalten die vom Subjekt abhängigen Objekte ihre Änderungen automatisch durch den Beobachter oder Dispatcher.

Im Gegensatz zu dem Decorator-Pattern, bringt das Observer-Pattern einige nennenswerte Nachteile mit sich. Umso höher die Anzahl der Observer eines Subjektes steigt, desto größer werden die Kosten bei einer Änderung. Es wird jeder Beobachter eines Subjektes bei jeder Änderung informiert, unabhängig davon, ob dieser die Information benötigt. Außerdem können die Folgeänderungen an den gekoppelten Objekten unerwartet hohe Rechenaufwand hervorrufen. Diese Folgeänderungen können außerdem bei schlechter Umsetzung Änderungsmethoden des beobachteten Subjekts auslösen wodurch es zu unerwünschten Schleifen kommen kann. Durch die inhärente lose Kopplung der Strukturen kann beim Betrachten des Quellcodes dieses Musters bei Subjekten oft nicht festgestellt werden, welche Ereignisse bestimmte Zustände auslösen werden.

Das Observer-Pattern nutzt MockITUp um die Logik-Transaktionen in der Test-Ansicht zu realisieren. Das *testrenderer.js* Modul kompiliert den DOM-Baum des HTML-Dokuments so, dass jede Änderungsmethode der Subjekte bei einem Dispatcher im *testlogick.js* Modul registriert werden. Sobald ein Beobachter eine Änderung feststellt, werden alle vorher registrierten Änderungen durch diesen ausgelöst. Durch diese lose Kopplung wird vermieden, dass die Elemente in direkten Kontext zu den anderen Elementen stehen müssen mit denen sie logische Verbindungen besitzen. Dies ermöglicht nicht nur die einfachere Änderung dieser Verbindungen im Editor, sondern ermöglicht komplexere logische Aktionen, welche über mehrere Elemente hinweg agieren. Im Folgenden Codesnippet ist der Dispatcher der Änderungsfunktionen zu sehen, welcher alle bekannten Änderungsfunktionen für dieses Objekt und die beobachtete Änderung iteriert und diese dementsprechend umsetzt.

```
alle Funktionen zum Auslösen von Effekten. beinhaltete den Dispatcher
test_do_execute(event,target,passedvalue) => observer-pattern
wirkt bei auslösen eines events alle registrierten Wirkungen für dieses
element (registrierung in test_logik_transaktions)

/*Executes all actions, which are linked to the event identifier (e.g.
click)*/
function test_do_execute(event,target,passedvalue)
{
    for(m of target.tans_out)
    {
        if(m.name == event)
        {

test_logik_transaktions[m.id].effekt(test_logik_transaktions[m.id].target,p
assedvalue,m.id);
        }
    }
}
```

4.2 Einstiegspunkte für die Weiterentwicklung

In diesem Abschnitt wird beschrieben, wie neue Elemente, Auslöser für Logiktransaktionen und Effekte dieser zu MockITUp hinzugefügt werden können, um die Anwendung um gewünschte Funktionalität zu erweitern.

Für alle Erweiterung gilt zu Beginn das gleiche Vorgehen, zunächst muss eine neue JavaScript-Datei erstellt werden in welcher der Code für die Erweiterung hineinkommt. Dieses neue Modul muss anschließend in der *index.html* im *head* des HTML eingebunden werden.

```
<html>
<head>
    <meta charset="utf-8">
    <title>MockITup</title>
    <link rel="stylesheet" type="text/css" href="main.css">
    <link rel="shortcut icon" href="Mockingbird_Illustration_Color4.png"/>
    <script src="main.js"></script>
    <script src="onLoad.js"></script>
    <script src="elements.js"></script>
    <script src="testrenderer.js"></script>
    <script src="logik.js"></script>
    <script src="test_logick.js"></script>
    <!--<script src="newElement.js"></script>-->
    <script src="save.js"></script>
    <script src="qrcode.js"></script>
    <script src="editPictures.js"></script>
</head>
```

In dem nun eingebundenen Modul kann nun der Code eingefügt werden für die gewünschten Erweiterungen. Im den folgenden Abschnitten wird beispielhaft erklärt, wie ein neues Element, ein neuer logischer Transaktionsauslöser, eine neue Wirkung einer logischen Transaktion und eine neue Option in der *Settingsbar* von Elementen hinzugefügt werden kann.

Elemente

Um eine neues Element zu erstellen muss zunächst die objektgenerierende Funktion erstellt werden.

```
elements["elementname"] = function() { ... }
```

Hierbei sollte „elementname“, der Index des Arrays elements, der Name des neuen Elements darstellen. Der Funktionskörper kann anschließend analog zum Aufbau der anderen elementgenerierenden Funktionen in *elements.js* erstellt werden. Wichtig dabei ist dabei zu Beginn das Erstellen eines DOM-Objekts und dessen Weitergabe an den Decorator.

Der zweite Schritt in der Erstellung neuer Elementfunktionalitäten ist die Erstellung der Ladefunktion für dieses Element. Diese dient dazu aus einem gespeicherten MockUp die dort verwendeten Elemente wiederherzustellen. Hier können als Referenz ebenfalls die Funktionen aus elements.js betrachtet werden, welche sich nur wenig von der elementgenerierenden Funktion unterscheiden.

```
elements["elementname"].createfromsave = function() { ... }
```

Mit diesem Schritt ist die eigentlich Erstellung des neuen Elementes abgeschlossen und es muss lediglich noch in die Elementleiste der Grafik-Ansicht des Editors eingebunden werden. Dies erfolgt über den folgenden Funktionsaufruf:

```
grafic_elements.add(elementbar_Item("elementname", notifikation_to_drag_and_drop));
```

Hierbei zu beachten ist, dass dieser Funktionsaufruf nur nach der *onLoad*-Funktion des HTML-Bodys ausgeführt werden kann, weswegen empfohlen wird das Event *initialise* abzufangen, welches beim Beenden der *onLoad*-Funktion geworfen wird.

Logische Transaktionsauslöser

Zunächst muss eine Funktion erstellt werden, die das neue Event global bekanntgibt. Dabei sollte in dem String in der letzten Zeile und der Funktionsname entsprechend an die neue Funktion angepasst werden.

```
function setclickevent(e)
{
    general_event_stuff();
    evoker = this.parentElement.eventtarget;
    evoking_aktion = "click";
}
```

Der zweite Schritt zur Erstellung eines neuen logischen Transaktionsauslösers ist, dass Hinzufügen eines Eintrages in die Transaktionsmenüleiste in der Logik-Ansicht des Editors. Dies ist über den folgenden Funktionsaufruf möglich, wobei „eventname“ und eventfunktion mit entsprechenden Werten ersetzt werden müssen.

```
logick_elements.add(menubar_Item("eventname",eventfunktion);
```

Hierbei, ähnlich wie bei der Elementleiste, den Funktionsaufruf erst nach Beendigung der *onLoad*-Funktion des HTML-Bodys durchführen. Dafür kann das *initalise* Event abgefangen werden, was von der *onLoad*-Funktion bei ihrer Terminierung geworfen wird.

Als letzter Schritt muss eine Funktion geschrieben werden, die für das tatsächliche Auslösen verantwortlich ist. Dabei kann es sich um eine simple HMTL-Event Verknüpfung handeln, oder eine komplexere Benutzerdefinierte Funktion. Im Folgenden ist ein Beispiel für die logische Transaktionsauslösung des „click“-Events. Dabei wurde lediglich die Übergebene Funktion, execute, welche in diesem Fall der Dispatcher der Transaktionseffekte ist, an das HTML-Click Event gebunden.

```
test_logick_events["click"] = function(target,execute)
{
    target.onclick = function(){execute("click",this);};
}
```

Wichtig ist hier, dass der Name der Transaktionsauslösung in allen drei Funktionen gleichbleibt.

Wirkungen logischer Transaktionen

Um einen neuen Effekt einer logischen Transaktion hinzuzufügen, muss zuerst eine Funktion erstellt werden, die dafür zuständig ist bei der Instanziierung dieses Effektes zusammen mit seinem Auslöser die Transaktion in ein globales Array zu schreiben. Um das folgende Beispiel zu übernehmen müssten der Funktionsname und der übergebene String, der den identifizierenden Namen des Effektes bildet, abändern.

```
function logick_button_toggle_visibility()
{
    logick_transaktions.push(new
logick_transaktion(evoker,evoking_aktion,this.parentElement.parent,"togglev
isibility"));
    reset_transaktion(this);
}
```

Im zweiten Schritt muss die gerade erstellte Funktion entweder in die objektgerneriende Funktion des Elemnts oder der Elemente in *elements.js*, oder wenn der Effekt für alle Elemente zu Verfügung stehen soll in den Decorator make_container(…), ebenfalls in *elements.js*, hinzugefügt werden.

```
elem.logick_menu.add(logick_menu_item("Toggle Visibility",logick_button_toggle_visibility));
```

In diesem Beispiel sieht man wie der Effekt „Toggle Visibility“, der für alle Elemente zur Verfügung steht dem Decorator hinzugefügt wird. Der Funktionsaufruf bei einzelnen Elementen bleibt der gleiche, bis auf den Kontext.

Im letzten Schritt muss die Funktion geschrieben werden, welche die eigentlichen Änderungen an dem Zielelement oder Zielelementen durchführt. Diese wird vom Dispatcher an entsprechender Stelle aufgerufen und ausgeführt.

```
test_logick_effekts["togglevisibility"] = function(target)
{
    if(target.style.display == "none") target.style.display = ""; else
target.style.display = "none";
}
```

Wichtig ist das auch hier der String, der den identifizierenden Namen des Effektes ausmacht, in den Funktionen gleichbleibt.

5. Quellen

1 http://ai2.appinventor.mit.edu| ***zuletzt aufgerufen am 10.05.2017***

2 https://wiki.ubuntuusers.de/Ubuntu_Touch/ | ***zuletzt aufgerufen am 07.06.2017***
https://de.wikipedia.org/wiki/Ubuntu

3 https://de.wikipedia.org/wiki/Android_(Betriebssystem) | ***zuletzt aufgerufen am 07.06.2017***

4 http://www.createjs.com/easeljs | ***zuletzt aufgerufen am 07.06.2017***

5 http://www.olinger.net/iWeb/berufliches/Scratch_files/scratch_referenzhandbuch.pdf | ***zuletzt aufgerufen am 07.06.2017***

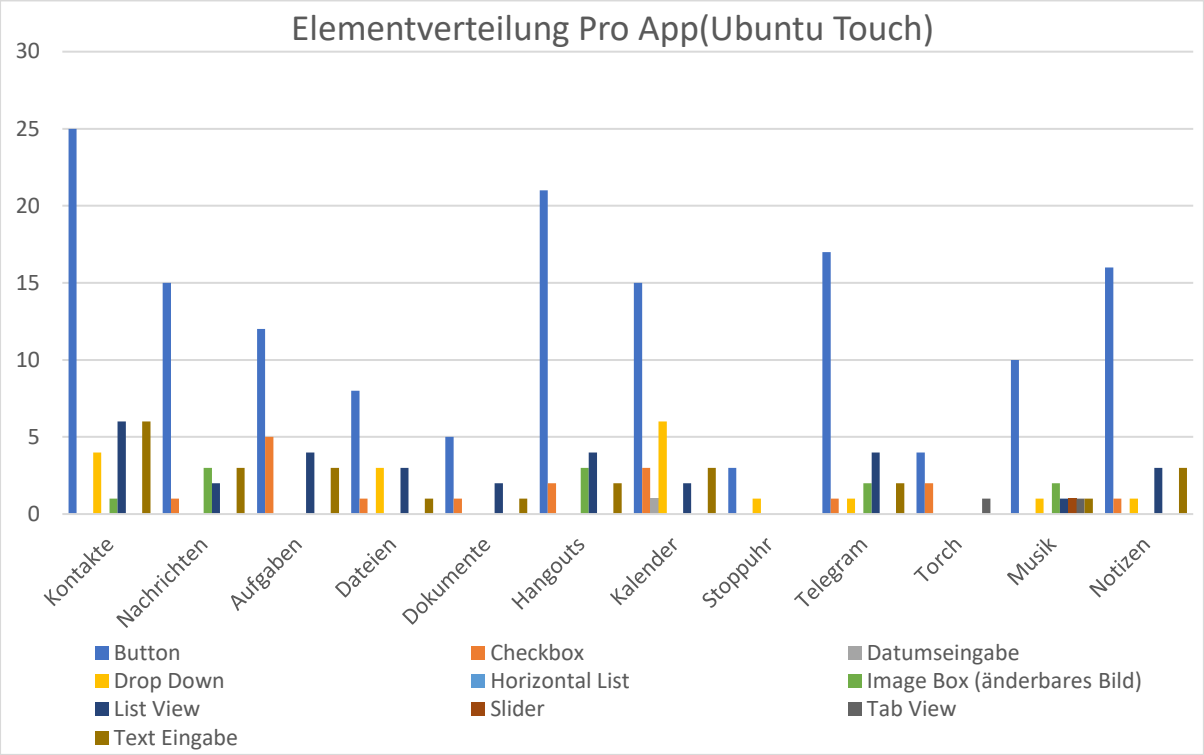
6 Carl Hewitt, Peter Bishop, Richard Steiger: *A Universal Modular Actor Formalism for Artificial Intelligence*. In: *Proceeding IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence*. 1973, S. 235–245

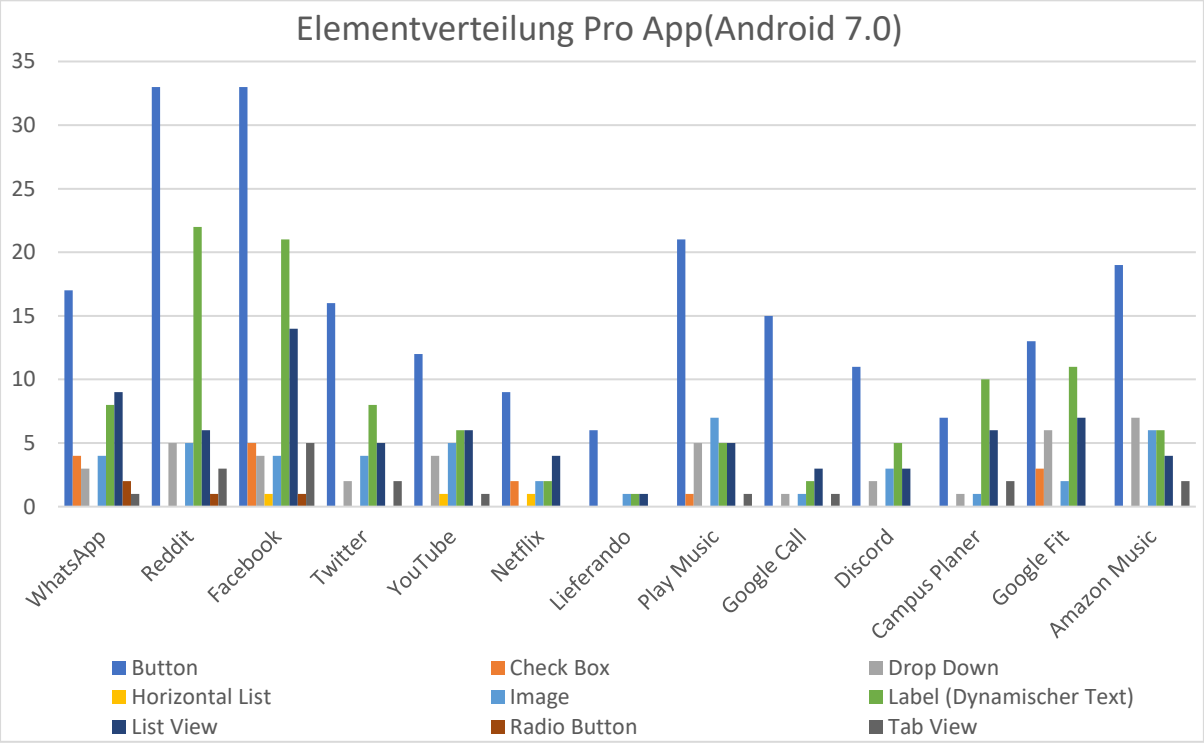
7 http://www.d-project.com | ***zuletzt aufgerufen am 07.06.2017***

8 https://de.wikipedia.org/wiki/Decorator | ***zuletzt aufgerufen am 07.06.2017***

9 https://de.wikipedia.org/wiki/Beobachter_(Entwurfsmuster) | ***zuletzt aufgerufen am 07.06.2017***

6. Anhang


Elementverteilung Pro App(Ubuntu Touch)
30
25
20
15
10
5
0
Kontakte
Nachrichten
Aufgaben
Dateien
Dokumente
Hangouts
Kalender
Stoppuhr
Telegram
Torch
Musik
Notizen
Button
Checkbox
Datumseingabe
Drop Down
Horizontal List
Image Box (änderbares Bild)
List View
Slider
Tab View
Text Eingabe


Elementverteilung Pro App(Android 7.0)
35
30
25
20
15
10
5
0
WhatsApp
Reddit
Facebook
Twitter
YouTube
Netflix
Lieferando
Play Music
Google Call
Discord
Campus Planer
Google Fit
Amazon Music
Button
Check Box
Drop Down
Horizontal List
Image
Label (Dynamischer Text)
List View
Radio Button
Tab View