Bachelor Arbeit von
Fabian Morón Zirfas

Inhalt

1 Einleitung

Diese Arbeit setzt sich mit der Automation von Design-Prozessen auseinander. Sie ist an Designer gerichtet, die einen rohen Überblick bekommen wollen, wie Arbeitsprozesse automatisiert werden können und soll als Hilfestellung dienen, einen Einstieg in das Schreiben von Skripten für Grafikanwendungen zu finden. Es ist keine komplette Anleitung zum Erlernen von Programmier-Grundkenntnissen. Diese können für viele verschiedene Sprachen im Internet gefunden werden. Vielmehr ist es eine Auseinandersetzung mit der Fragestellung, was Automation durch Programmierung für Designer leisten kann und welche Bereiche sie nicht abdeckt.

Im ersten Abschnitt soll dem Leser kurz erläutert werden, was er benötigt, um zu Skripten, um dann im zweiten Teil zu analysieren, wann dies sich lohnt. Hier wird an kurzen Beispielen erläutert, wo Probleme auftreten können und wie solche Probleme klug umgangen werden können. Im dritten Teil folgt ein Szenario an dem Schritt für Schritt die Logik und Funktionsweise eines Skriptes erklärt wird, sowie ein weiteres Szenario mit einer einzeiligen und doch nützlichen Lösung. Der vierte Teil behandelt reelle Anwendungsmöglichkeiten. Anhand dieser wird das Ausmaß, das diese annehmen können, gezeigt. Es folgt im fünften Teil ein Versuch die Frage zu beantworten, warum es bei Nicht-Programmierern/innen teils große Berührungsängste mit Computersprachen gibt. Im sechsten Teil, dem Fazit, wird der Erkenntniszuwachs aus all dem zusammengefasst. Am Ende der Arbeit folgt ein Abschitt mit Erläuterungen zur verwendeten Terminologie. Dieser Abschnitt sollte jedoch nicht linear gelesen werden, sondern als Erläuterung dienen.

1.1 Bevor Sie Einsteigen

Bevor sie weiterlesen, einige Hinweise. Wenn sie noch keine Erfahrungen mit Programmierung gemacht haben, lassen sie sich nicht abschrecken. Es werden einige für den Laien unverständliche Zeilen vorkommen und einige, die sich aus dem Kontext oder den Namen der Funktionen erschliessen lassen. Falls sie die Beispiele in den Abschnitten Eins und Zwei ausprobieren möchten, lesen sie den Abschnitt 7.08 «Was ist Hello World?» zuerst. Dort wird erklärt, wie und wo sie Skripte schreiben und ausführen können. Es werden Fachausdrücke vorkommen die, ebenfalls im Abschnitt 7 «Die kleine Terminologie» erklärt werden. Ab dem Abschnitt «3 Szenarien» steigen wir dann in das Lesen und wenn sie möchten, Schreiben von Skripten ein. Dafür sollte zumindest der eben genannte Abschnitt «Was ist Hello World?» einmal nachvollzogen worden sein. Wenn die dort dargestellten Zusammenhänge und Konstrukte zu komplex sind, verschaffen sie sich etwas Übung. Es existiert eine schöne Webseite, genannt codecademy.com, auf der sie in einigen lustigen Aufgaben durch die Grundlagen von JavaScript geführt werden. Wenn mal etwas nicht funktioniert, verzweifeln sie nicht. Kontrollieren sie ihre Schreibweise und suchen sie sich weiterführende Informationen im Netz. Die einfachste Möglichkeit ist den gewünschten Befehl samt JavaScript z.B. «alert() JavaScript» in einer Suchmaschine einzugeben. Da «das Internet» viel JavaScript benutzt, ergibt eine Suche in 99% der Fälle auch die richtige Antwort. Werfen sie ebenfalls einen Blick auf die Referenz-Liste. Dort finden sie viele Ressourcen für «Scripting», die ihnen den Einstieg erleichtern werden.
Diese Arbeit und alle Codebeispiele sind auch online zu finden.
http://fabiantheblind.github.com/MT4D/

http://www.codecademy.com/#!/exercises/0

2 Wann Soll Ich Skripten?

> *«Scripting languages assume that there already exists a collection of useful components written in other languages. Scripting languages aren't intended for writing applications from scratch; they are intended primarily for plugging together components.»*[1]
> *Scripting: Higher Level Programming for the 21st Century*
> *von John K. Ousterhout*

Auch wenn Aufgaben auf unterschiedliche Weise mit verschiedenen Programmiersprachen gelöst werden können, haben sich spezielle Anwendungsgebiete für die einzelnen Sprachen ergeben. Ganz unabhängig davon, dass sich in unserem Fall Adobe Anwendungen mit JavaScript ansprechen lassen, macht es Sinn, eine Skriptsprache zu verwenden, um die bereits in höheren Sprachen implementierten Funktionen zu verbinden. JavaScript ist unser «Kleber». Wir können es benutzen um unseren Arbeitsablauf durch gezielte Befehlsketten von repetitiven Aufgaben zu befreien. Ich bin mir sicher, dass ein Großteil aller Gestalter die vorgefertigte Software für ihre Arbeit verwenden, schon an den Punkt kamen, wo sie sich dachten: «Warum kann mein Programm DAS* nicht, es ist doch alles da. Der Knopf und danach diesen Knopf!». «Scripting» erlaubt es uns, diese beiden Knöpfe miteinander zu verbinden. Das bedeutet dann, dass wir unsere Arbeit um einen «Klick» reduziert haben. Wir haben zwei Knöpfe durch Verkettung auf einen neuen Knopf gelegt. Natürlich klingt die Reduktion um einen «Klick» vernachlässigbar. Wenn jedoch diese zwei «Klicks» 100-mal ausgeführt werden müssen und wir durch logische Anweisung diese ebenfalls auf nur einen Knopf zusammenführen können, ist der Zeitgewinn enorm. Ebenfalls muss hier erwähnt werden, dass viele der Probleme, die in einem Gestaltungsprozess auftreten, nicht zum ersten Mal bei eben dieser Person auftreten. Für den Bereich «Scripting» von Adobe-Anwendungen gibt es im Netz viele Seiten und Foren, die sich mit diesem Thema befassen.

DAS steht hier für eine gewünschte Funktionsweise

Im Bereich JavaScript gibt es noch viele mehr, da JavaScript auch verwendet wird beziehungsweise entwickelt wurde, um Browser zu steuern. Aufgrund dessen ist die Dokumentation mehr als ausgiebig. Es bedarf nur etwas Übung, um die gefundenen Beispiele zu lesen und auf die eigene Problemstellung zu abstrahieren.

Hierbei Sei Zu Beachten!

«Scripting» kann keine Design-Entscheidungen fällen. Es existiert kein Algorithmus, der Ästhetik simuliert.* Um eine spannende Komposition zu schaffen, braucht der Gestalter «nur» drei geometrische Grundformen zu erzeugen und diese im richtigen Verhältnis zu einander anzuordnen. Um dies programmatisch zu lösen, müsste ein Skript mehrere hundert Mal ausgeführt werden. Jedes Mal mit einer kleinen Veränderung der Koordinaten, der oben genannten drei Objekte. Irgendwann muss der Autor eintscheiden, welche Komposition spannend ist. Was das Skript leisten kann, ist anhand von bestimmten Rahmenparametern eine Fülle von Varianten zu liefern, die manuell ausgeführt Sehnenscheidenentzündungen hervorrufen würden. Programmieren ist wie eine neue Sprache zu erlernen. Stellen sie sich vor, sie sind in einem fremden Land, dessen Sprache sie nicht beherrschen. Sie werden zuerst Probleme haben, sich zu verständigen. Dann lernen sie, ihre Grundbedürfnisse zu decken. Ab einem gewissen Punkt können sie Tageszeitungen lesen, Inhalte erfassen und abstrahieren. Eines Tages werden sie feststellen, dass sie in der Sprache träumen. Der Vorteil an Computersprachen im Vergleich zu «Menschensprachen» ist, dass in der Computersprache kein Raum für Interpretation vorhanden ist. Jede Aussage ***muss***, im Gegensatz zur zwischenmenschlichen Kommunikation, eindeutig sein. In Quellcode ist kein Raum für Interpretation. Die Syntax muss valide sein. Dies ist ein Vorteil, da es Genauigkeit bedingt. Dennoch, eine Sprache lernen, ist keine leichte Aufgabe.

Ein Algorithmus ist ein logische Verkettung von Operationen siehe Abschnitt 7.03.

Daher sollten sie, bevor sie in die Tiefen von JavaScript abtauchen, um ein Problem zu lösen, entscheiden:

- Welche Operationen sollen ausgeführt werden?
- Wie komplex sind die Operationen?
- Wie groß ist das Zeitfenster und wann muss das Produkt fertig sein?
- Wie oft muss diese Tätigkeit ausgeführt werden?
- Lässt sich die Automation auch auf andere, ähnliche Bereiche anwenden oder mit geringem Aufwand abstrahieren?
- Wie sehr ist sie von Umgebungsvariablen abhängig?
- Soll die Automation von Dritten genutzt werden?
- Ist der Prozess linear oder bedarf es einer Rückkopplung zum Benutzer?

Und einige Punkte, die sich hauptsächlich durch Recherche, die bereits vor dem ersten Entwurf erledigt sein sollte, abarbeiten lassen:

- Existieren bereits Automationen in dem Sektor und, wenn ja, lassen sich diese abwandeln?
- Bietet die API direkten Zugriff auf die benötigten Funktionen oder braucht es eine Workaround?

Diese Analyse kann keine genauen Angaben über Zeit und Aufwand machen, da dies immer auch vom Erfindungsreichtum des Autors abhängig ist. Der Kreative kommt hier schneller zum Ziel.

2.0.1 Welche Operationen sollen ausgeführt werden?

Programme oder Skripte schreiben, ist nicht wie «Scribbeln». Wir können beim Telefonieren einen Stift in die Hand nehmen und drauf los kritzeln. Beim Programmieren muss das Ergebnis bereits definiert sein, bevor geschrieben wird. Natürlich ergeben sich auch während des Schreibens neue Ideen, dennoch muss eine konkrete Vorstellung existieren, was das Ziel sein soll. Auch dies ist mit Sprechen zu vergleichen. erst denken - dann reden.

2.0.2 Wie komplex sind die Operationen?

Wenn eine Fülle von unterschiedlichen Operationen ausgeführt werden soll, müssen auch entsprechend viele Anweisungen an das Programm erfolgen und der Autor muss sich auf eine längere Entwicklungszeit einstellen. Wenn es hingegen darum geht, einige Operationen 1000-mal auszuführen, kann es sein, dass sich der Kern des Skriptes auf zehn Zeilen reduziert. Dies bedeutet auch, dass die Zeit für Entwicklung und «Debugging» relativ gering sein und sich der Zeitaufwand lohnen kann.*

Ein Bug ist ein Fehler im Programm. Debugging ist der Prozess der Fehlersuche. Siehe Abschnitt 7.13 Was ist ein Bug? und 7.14 Was ist Debugging?.

2.0.3 Wie groß ist das Zeitfenster und wann muss das Produkt fertig sein?

Auch hier muss, abhängig von der Komplexität des Skriptes und dem eigenem Vermögen, geurteilt werden, ob dies in dem gegebenen Zeitraum recherchiert, entworfen, geschrieben und «debugged» werden kann. Hinzu kommt, dass das Ergebnis meist nur ein Teilergebnis ist und noch weiterverarbeitet werden muss. Es ist davon auszugehen, dass etwaige tiefere Fehler erst während des vollen Einsatzes auftreten. Wenn dies in dem entsprechenden Zeitraum nicht zu bewerkstelligen ist, sollte von einer Entwicklung abgesehen werden.

2.0.5 Wie oft muss diese Tätigkeit ausgeführt werden?

Wenn das Skript nur ein einziges Mal ausgeführt werden soll, sollte man sich fragen, wo darin der Nutzen liegt. Es kann natürlich sein, dass dies Sinn und Zweck hat und sollte nur bedingt ausschlaggebend sein. Hierbei gilt es zu unterscheiden, dass einmalig 1000-mal einen Knopf drücken, bereits eine Hilfe sein kann. Einmalig 1000-mal unterschiedliche Knöpfe drücken, ist eine Aufgabe, die doch besser manuell geschieht.

2.0.6 Lässt sich die Automation auch auf andere, ähnliche Bereiche anwenden oder mit geringem Aufwand abstrahieren?

Wenn dies der Fall ist, steigt der Nutzen der Arbeit. Der einmalige Aufwand ein Programm oder Skript für eine immer wieder kehrende beziehungsweise ähnliche Aufgabe zu schreiben, rentiert sich durch jede weitere Ausführung. Dies soll heißen: Durch das Schreiben des Programms, das ich immer wieder verwende, spare ich Zeit in der Zukunft.

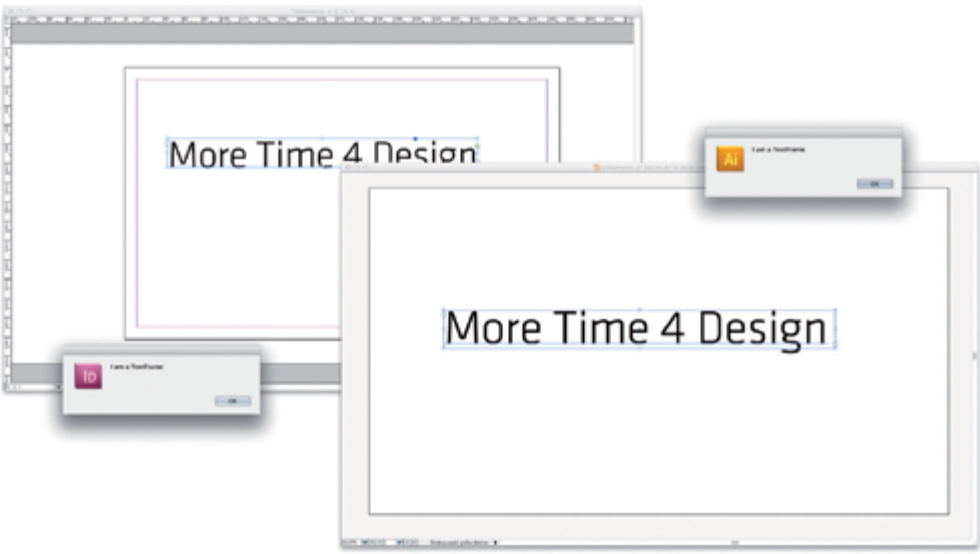
2.0.7 Wie sehr ist sie von Umgebungsvariablen abhängig?

Kann das Skript unabhängig von allen Variablen, die der Benutzer setzen kann, ausgeführt werden, vereinfacht das den Aufwand. Bei einer Abhängigkeit erfordert es immer erst einer Abfrage des «Ist-Status». Ein kleines Beispiel: In Illustrator oder InDesign wird die aktuelle Auswahl des aktiven Dokuments in einer Liste, genannt «selection», geführt. `app.activeDocument.selection;` In dieser Liste liegen einzelne Objekte, die beispielsweise Text, eine Vektor-Form oder ein Bild sein können. All diese haben gemeinsame Eigenschaften, aber auch spezielle. Es muss also, bevor eine Eigenschaft genutzt oder verändert werden kann, eine Abfrage stattfinden, welche Art von Objekt enthalten ist. Das nachfolgende Beispiel funktioniert in InDesign und Illustrator gleich. Es wird der Name des ersten Objekts in der Selektion abgefragt, wenn dies eine Textbox ist, gibt das Skript eine Meldung zurück.

```
// InDesign & Illustrator
var firstListItem = app.activeDocument.selection[0];
// first object in selection
if(firstListItem instanceof TextFrame){
// check the type
// and name it
    alert("I am a " + firstListItem.constructor.name);
};
```

https://gist.github.com/265464 5#file_find_textframes.jsx

Viele solcher Abfragen können ein Skript schnell komplex werden lassen. Oder anders ausgedrückt, je universeller der Nutzen sein soll, desto mehr Umgebungsvariablen müssen beachtet werden. Dies benötigt Zeit.



2.0.8 Soll die Automation von dritten Benutzt werden?

Dies ist ein wichtiger Faktor. Wenn nur der Autor selbst die Automation verwendet, kann er seine vorhergehenden Aktionen auf die Bedürfnisse und Beschränkungen des Skriptes anpassen. Wenn jedoch eine unbedarfte oder schlimmer noch eine dem «Skripten» nicht mächtige Person dieses Werkzeug nutzen soll, müssen wie bereits oben erwähnt, viele Umgebungsvariablen abgefragt oder selber bestimmt werden. Im Programmier-Slang sagt man: «Man muss vom DAU (Dümmster Anzunehmender User) ausgehen.» Dies ist nicht als Beleidigung gedacht. Es soll eher sagen, dass alle Benutzerfehler, die auftreten können, auftreten werden. In diesem Fall bekommt jemand, der nicht programmieren kann, wenn er Glück hat, nur eine Fehlermeldung. Im schlimmsten Falle führt es zu einem Programmabsturz. In beiden Fällen steigt die Hemmung des Nutzers ungemein, das Skript noch einmal zu verwenden. Wenn also Dritte mit ins Spiel kommen, erfordert es noch längere Test- und Debug-Phasen.

2.0.9 Ist der Prozess linear oder bedarf es einer Rückkopplung zum Benutzer?

Wenn dem so ist, sollte der Prozess vielleicht in mehrere Skripte zerlegt werden. Für den Fall, dass Variablen von einem Skript an das nächste übergeben werden müssen, kann dies die Komplexität weiter erhöhen. Hierbei gibt es die Möglichkeiten, eigene Textdateien, in denen Werte abgelegt werden können, vom Skript kreieren zu lassen. Die Verwendung von Textdateien kann ihre Tücken haben. Auf unterschiedlichen Betriebssystemen werden Dateipfade unterschiedlich gehandhabt. Auch dies will abgefragt und getestet werden. Weiterhin ist es möglich, ein «Script Panel»* zu erzeugen, das solange es aktiv ist Werte beinhaltet. Eine weitere Option ist die Verwendung einer `targetengine`, in der, solange das Programm aktiv ist, Daten gespeichert werden. Die letzten beiden sind jedoch fortgeschrittene Lösungen, die ebenfalls viele Stolpersteine beherbergen können.

Ein Script Panel ist eine Erweiterung der grafischen Oberfläche, die es erlaubt, während das Skript läuft, weiterhin mit dem Programm zu interagieren.

2.1 Das Beispiel targetengine

Skript 1:

```
#targetengine "session01"
var myValue = 0; // new value
alert(myValue); // result is 0
myValue++; // increment by 1
```


Skript 2:

```
#targetengine "session01"
alert(myValue); // result is 1
```

Der Algorithmus lautet wie folgt:

Skript 1:

```
Start
Sitzung 1
definiere Variable meinWert und Speicher 0 in ihr
zeige Wert von meinWert
erhöhe meinWert im eins
Stop
```

Skript 2:

```
Start
Sitzung 1
zeige Wert von meinWert
Stop
```

Dies bedeutet, dass das Programm (nicht das Skript) sich den Wert für die Variable `myValue` gemerkt hat und ihn solange in `#targetengine "session01"` speichert, bis es beendet wird. Hier liegt die Gefahr darin, Bezeichnungen für Variablen doppelt zu vergeben. In vielen meiner Skripte speichere ich das aktive Dokument in der variable `doc`. Wenn diese nun in einer `targetengine` liegen, wird das Programm immer auf das zuletzt `doc` gleichgesetzte Objekt zugreifen. Dies geschieht auch wenn der Nutzer es nicht wünscht, sondern es lediglich nicht beachtet hat.

2.1.1 Existieren bereits Automationen in dem Sektor und, wenn ja, lassen sich diese abwandeln?

Wie bereits oben erwähnt, sind viele Probleme bereits einmal aufgetreten. Wenn das Problem nicht all zu speziell ist, kann es gut sein, dass es bereits ein Skript gibt, das entweder genau diese Funktion enthält und, wenn nicht, auf die eigenen Bedürfnisse angepasst werden kann. Meist is es der Fall, dass es mehrere Skripte gibt, die Teilprozesse der eigenen Idee beinhalten und als Referenz benutzt werden können. In den seltensten Fällen entsteht ein Skript nicht aus einem blanken Textdokument.

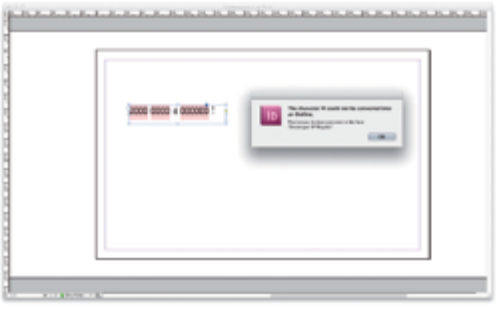
2.2 Bietet die API direkten Zugriff auf die benötigten Funktionen oder bedarf es eines Workaround?

2.2.1 Das Beispiel try char

Um Punkt 2.2 zu erläutern, möchte ich mich eines Beispiels bedienen. InDesign kann nicht erfragen, ob ein Zeichen in einer Schriftart enthalten ist. Ein Zeichen hat kein Eigenschaft, die `Font has Character` oder ähnliches beinhaltet. Um diese Abfrage zu simulieren, hat Peter Kahrel* die Funktion `try_char()`[2] geschrieben, die hier in einer etwas abgewandelten Form folgt. Um die folgende Funktion `try_char()` sinnvoll ausführen zu können, benötigen sie ein InDesign Dokument mit einer Textbox auf der ersten Seite. In dieser Box muss Text enthalten sein, der Zeichen beinhaltet, die in dieser Schriftart nicht dargestellt werden können. (siehe Bild S. 15)

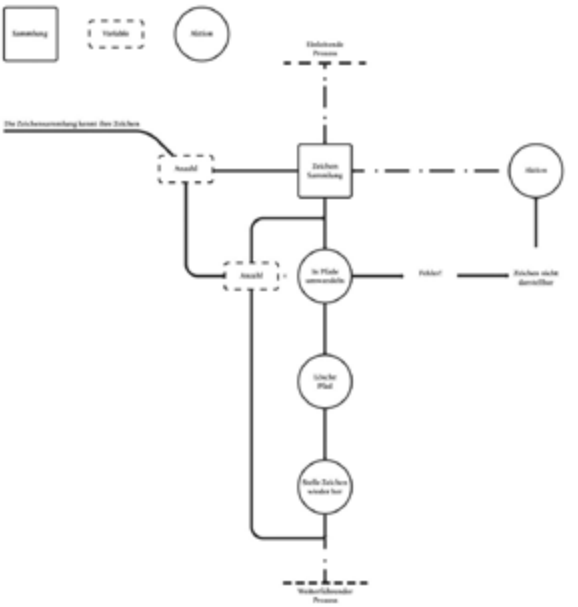
Peter Kahrel ist einer der präsentesten InDesign Skripter und Autor von «InDesign mit JavaScript automatisieren» erschienen im O'Reilly Verlag

```
var doc = app.activeDocument;
var tf = doc.pages.item(0).textFrames.item(0);
for(var i = 0; i < tf.characters.length;i++){
    var theChar = tf.characters.item(i);
    try_char (theChar);
    };
// original function by Peter Kahrel
// http://tinyurl.com/cku46vf
// edited by fabiantheblind
function try_char (theChar){
try {
    // save the character
    var storage = theChar.contents;
    // create outline
    theChar.createOutlines();
    // if we got here it worked, so delete the outline
    // if not the next line will be within the catch
    // block, if not it will cause an error
    theChar.remove();
    // insert the character (again)
    theChar.contents = storage;
}catch(e){
    alert("The character "+ theChar.contents +
    " could not be converted into an Outline.\n"+
    "That means he does not exist in the font:\n\""
    + theChar.appliedFont.name+"\"");
    }
}
```



Die Funktion `try_char` erledigt folgende Aufgaben.

Sie bekommt eine Zeichen als Parameter. Dieses speichert sie temporär in einer Variablen. Dann wird das Zeichen genommen und InDesign versucht es, in Pfade umzuwandeln. Dies kann nur passieren, wenn das Zeichen in der Schrift auch existiert. Sollte dem so sein, wird die neu kreierte Vektor-Form wieder verworfen und das Zeichen wird aus dem Zwischenspeicher wieder hergestellt. Wenn jedoch die Umwandlung einen Fehler erzeugt, wird dieser aufgefangen. Damit weiss das Skript ob das Zeichen darstellbar ist oder nicht.

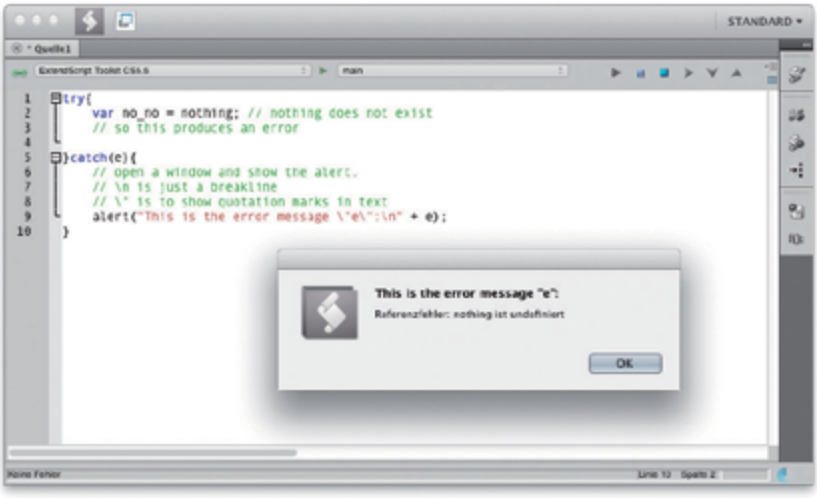


https://github.com/fabiantheblind/MT4D/raw/master/images/algorithmus_trychar.jpg

Dies passiert mit dem Konstrukt `try { } catch (e) { }`. Sollte der Inhalt der ersten geschwungenen Klammer, oder auch Block genannt, einen Fehler erzeugen, wird dieser abgebrochen und das Skript führt den Zweiten Block aus. Die Variable `e` ist in diesem Fall die Fehlermeldung. Probieren sie es mit dem folgendem kurzen Skript aus. Das Programm wird sie warnen, dass die Variable `nothing` nicht existiert.

https://gist.github.com/2654624

```
try{
    var no_no = nothing; // nothing does not exist
    // so this produces an error
}catch(e){
    // open a window and show the alert.
    // \n is just a breakline
    // \" is to show quotation marks in text
    alert("This is the error message \"e\":\n" + e);
};
```



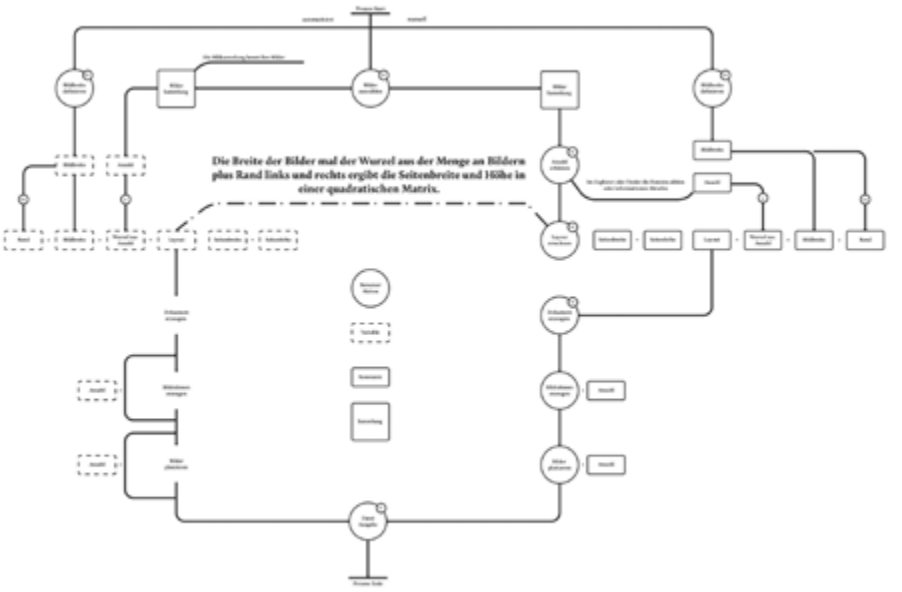
Solche Lösungen setzen nicht nur einen kreativen Umgang mit Code voraus, sondern auch ein tiefes Wissen über die Funktionsweise und Möglichkeiten innerhalb von, in diesem Falle, InDesign. Die oben genannten Hindernisse schrecken ab. Es ist jedoch sehr allgemein gehalten und manche Fragen stellen sich auch erst gar nicht. Bei Aufgaben die ein «Normal-Nutzer» niemals manuell machen würde, muss sogar ein Skript geschrieben werden, wenn sie erledigt werden sollen.

Auch wenn der Prozess des Skript-Schreibens eine kreative Arbeit ist, ist das Ziel des Skriptes nicht der kreative Output, sondern die Optimierung der eigenen Arbeitsabläufe. Das Skript oder Programm kann niemals die Idee liefern. Es unterstützt den Prozess indem es sich wiederholende Aufgaben erledigt, die entweder niemals gemacht worden wären oder aber vom eigentlichen, kreativem Prozess ablenken.

3 Szenarien

Um besser zu verstehen, was mit Automation erreicht werden kann, soll hier ein Fallbeispiel aufgemacht werden, vor dem jeder Layouter einmal stehen könnte. Stellen sie sich vor ihre Aufgabe ist folgende:

Erzeugen sie eine quadratische Bildmatrix in InDesign aus einer definierten Anzahl an quadratischen Bildern mit einer festen Grösse.



https://github.com/fabiantheblind/MT4D/raw/master/images/matrix_algorithmus_01.jpg

Dies würde bedeuten, dass wir zum Beispiel 4×4 also 16 Bilder darstellen wollen. Es folgt eine grafische Gegenüberstellung des automatisierten (links) und manuellen (rechts) Prozesses.

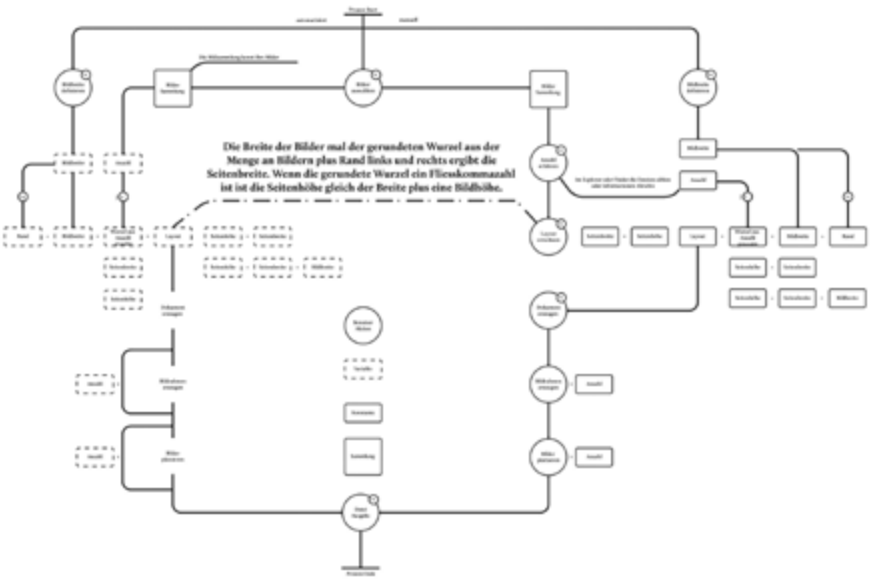
Bei 16 Bildern kommen wir auf 38 Aktionen. Mit einem Skript haben wir 3 Aktionen. Wenn es eine 10×10 Matrix sein soll, sind das 206 Aktionen. Im Skript sind es weiterhin 3. Der Kern der Aufgabe ist das errechnen des Layouts. Dies muss in der manuellen sowie in der automatisierten Variante passieren. Demzufolge kommt der Gestalter, der sich für die manuelle entschieden hat, nicht um das Errechnen des Layouts herum. Da wir von Prozessen im «Gestalter-Alltag» reden, kann es gut vorkommen, dass ein Kunde eine solche Aufgabe an sie stellt. Verändern wir die Aufgabe noch ein wenig:

Erzeugen Sie eine Bildmatrix in InDesign aus einer noch nicht definierten Anzahl an quadratischen Bildern mit einer festen Grösse.

Um dieser Anforderung gerecht zu werden, müssen wir in der automatisierten Variante nur eine Kondition einführen.

Wenn die Wurzel der Bildmenge eine Fließkommazahl ist, ist die Seitenhöhe die Seitenbreite plus eine Bildbreit.

https://github.com/fabiantheblind/MT4D/raw/master/images/matrix_algorithmus_02.jpg



3.1 Der Einsatz - das Beispiel image matrix

Die Fragen, die sich jetzt stellen, sind folgende: «Wie kommt dies zum Einsatz?», «Wie kann ich das nutzen?». Es folgt nun das In-Design-Skript ***image_matrix.jsx*** mit etwas mehr als 40 Zeilen Quelltext. Dieses Skript beinhaltet eine Benutzerinteraktion, Dateihandhabung, das Math Objekt*, Konditionen, eine Schleife, Funktionen, Variablen und es hat einen generativen Charakter. Das Skript erledigt folgende Schritte.

- Frage den Benutzer nach einem Ordner mit .jpg Dateien.
- Erzeuge ein quadratisches Dokument dessen Grösse auf der Quadratwurzel der Menge der Bilder basiert und alle Bilder fasst.
- Platziere alle Bilder.

Kopieren sie den nachstehenden Code und bereiten sie einen Ordner mit einigen Jpg-Bildern vor. Das Skript, das unten beschrieben wird, verarbeitet nur Dateien mit der Endung `.jpg` . Dateien mit `.jpeg` oder `.JPG` als Endung werden ignoriert. Es sollten um die 15 bis 30 sein, damit die Ausführung des Skriptes nicht zuviel Zeit in Anspruch nimmt. Es kann jedoch auch 100, 1000 oder mehr Bilder verarbeiten. Die genaue Menge ist irrelevant. Lesen sie nochmal den Abschnitt 7.08 «Was ist Hello World?» und führen sie das Skript aus. Wenn der Ordnerauswahl-Dialog sich öffnet, wählen sie den Ordner mit den Bildern.

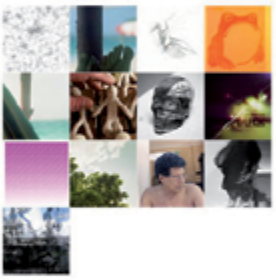
Das Math Objekt ist ein bereits bestehender Teil von JavaScript. Es erledigt solche Aufgaben wie Rundung von Werten oder das Berechnen einer Quadratwurzel.

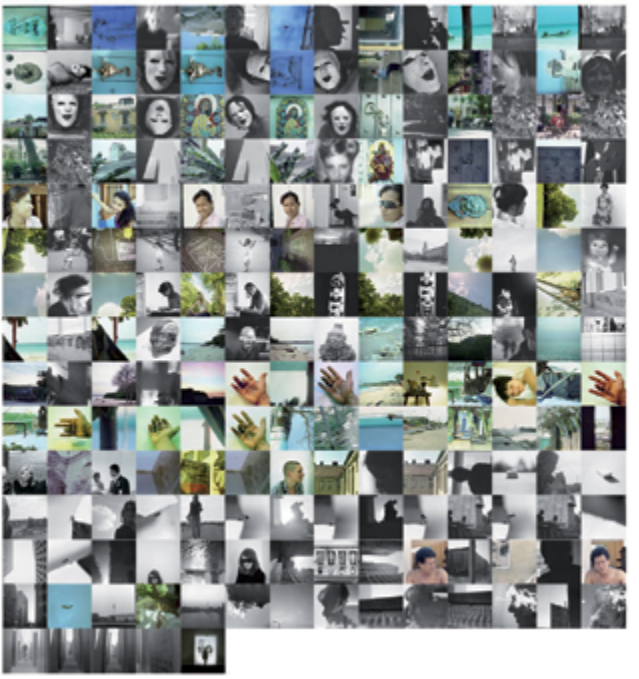
```
{ // START SCRIPT
main();
// you need a function to be able to cancel a script
function main(){ // all is in here
var w = 25; // the image sizes
// opens a prompt and lets the user choose a folder
var allImages = loadFiles("*.jpg");
if(allImages == null) return;
// if that what the function returns is nul
// cancelthe script
var pw = Math.round(
    Math.sqrt(allImages.length)
    ) * w + (w*2); // this will hold the page width
var ph = pw;
if(
    Math.round(Math.sqrt(allImages.length)) !=
    Math.sqrt(allImages.length)
    ){
    ph = pw + w;
    }
var doc = app.documents.add();
//build a basic document
doc.documentPreferences.pageWidth = pw;
// set the width
doc.documentPreferences.pageHeight = ph;
// set the height
var page = doc.pages.item(0);
// finally - get the first page
var y = w; // the upper left corner
var x = w; // the upper left corner
for(var i = 0; i < allImages.length;i++){
// loop thru all images
var rect = page.rectangles.add({
geometricBounds:[y,x,y + w,x + w]
}); // add a rectangle to the page
```

```
rect.place(allImages[i]); // place the image
rect.fit(FitOptions.FILL_PROPORTIONALLY);
// fit it to the frame
rect.fit(FitOptions.CENTER_CONTENT);
// fit it to the frame
x +=w; // increase x by an image width
if(x >= pw - (w)){
// if x is the width minus an image width
    x = w; // reset x
    y+= w; // and increment y by an image width
    }; //end increase x and y conditional
};//close allImages loop
};// end main function

function loadFiles(type){
// the function that loads the files
var theFolder = Folder.selectDialog ("Choose the
Folder");// user select a folder
if(!theFolder){
// if the folder is not a folder cancel the script
    return;
// this cancels the whole function image_loadFiles
}; // end folder check
var allImages = theFolder.getFiles(type);
// get the files by type
if((allImages.length < 1)||(allImages == null) ){
    // again if check if there are images
    alert("There are no images of the type: "+ type);
    // hm something went wrong? User error
    return null; // so we cancel the function
    }else{
    // give back the images. Success!
    return allImages;
    };// end all images check
    };// end function loadFiles
} // END OF SCRIPT
```

3.1.1 Ergebnisse - image matrix

Alle 3 nachfolgenden Bilder sind mit demselbem Skript erzeugt.





3.2 image matrix - Schritt für Schritt.

Ein Skript sollte immer in geschwungenen Klammern eingefasst sein. Dies ist nicht zwingend notwendig, jedoch nützlich, falls das Skript von einem anderem Skript evaluiert werden muss. Innerhalb dieser Klammern folgt der gesamte Quelltext.

```
{}
```

Als nächstes definieren wir eine Funktion, die einmalig aufgerufen wird. Hierbei ist es irrelevant, ob der Aufruf vor oder nach der Funktionsdefinition stattfindet. Aus Gründen der Übersichtlichkeit steht hier der Aufruf vor der Definition. Diese Funktion erhält keine Parameter. Später wird auch eine Funktion mit Parametern eingesetzt. Der Zweck, den gesamten Code in einer Funktion zu kapseln, ist folgender: Wenn es eine hierarchisch höhere Ebene gibt, unsere Klammern `{}`, kann eine Funktion mit dem Befehl `return;` beendet werden. Dies ist nützlich, um bei Fehlern oder falscher Benutzereingabe die Ausführung zu stoppen. Mit dem Befehl `return` kann nicht nur eine Funktion beendet werden, sondern auch ein Wert zurückgeliefert werden.
Wenn wir am Ende einer Funktion `return 1;` schreiben, wäre das Ergebnis 1. `var myValue = main();` Wenn die Funktion `main` nun diesen Wert zurückgeben würde, wäre `myValue` 1. In unserem Fall benötigt die `main()` keinen Rückgabewert.

```
{
main();
function main(){
}
}
```

Wir befinden uns nun im geschwungenen Block der Funktion `main(){}`. Es folgt die Deklaration der ersten Variable. Sie beinhaltet die Breite (und Höhe) unserer Bilder.

```
var w = 25;
```

Wie zu sehen ist, hat die Variable `w` (steht für width) nun den Wert 25. `w` wird im Verlauf des Skriptes mehrmals verwendet. Der Vorteil daran, Variablen zu verwenden, anstatt immer 25 zu schreiben ist, dass wenn wir die Breite verändern wollen, wir dies nur ein einziges mal an `w` erledigen müssen. Alle weiteren Verwendungen von `w` werden angepasst.

Als nächstes folgt eine Subfunktion mit einem Parameter und einem Rückgabewert. In JavaScript werden alle Variablen mit `var` deklariert. Die Unterscheidung des Typus findet erst bei der Verwendung statt. Deshalb können wir schreiben:

```
var allImages = loadFiles("*.jpg");
```

Damit die Funktion auch vorhanden ist, fügen wir sie hinter `function main(){}` an. Unser Code sieht also wie folgt aus.

```
{
main();
    function main(){
           var w = 25;
           var allImages = loadFiles("*.jpg");
    }
    function loadFiles(type){
    return 0;
    }
}
```

Den Rückgabewert werden wir noch verändern, denn wir wollen Bilder laden und nicht 0. Wie oben zu sehen ist, erhält `loadFiles` den Parameter `type`. Dies ist der Dateityp, der später geladen werden soll. `type` ist genau wie `w` und `allImages` eine Variable, nur dass wir im Parameterblock der Funktion das `var` weglassen können. Wenn wir `*.*` schreiben, würde unsere Funktion jeden Dateityp laden. Der Einfachheit halber beschränken wir uns auf `.jpg`, da wir wissen, dass InDesign .jpg-Dateien laden und platzieren kann.

Als nächstes begeben wir uns in die Funktion `loadFiles` und betrachten die Dateihandhabung und eine Benutzerinteraktion. Basierend auf dem Ergebnis der Interaktion fällt das Skript eine konditionale Entscheidung, ob es weiter arbeiten soll oder ob es die Funktion `loadFiles` beendet.

```
var theFolder = Folder.selectDialog ("Choose the Folder");
if(!theFolder){
    return;
    };
```

Bisher waren alle Befehle JavaScript Befehle. Der Befehl `Folder.selectDialog ("Choose the Folder")` jedoch ist ExtendScript. Das Objekt `Folder` hat eine eigene Funktion `selectDialog(parameter)`. Wir müssen diese Funktion nicht betrachten. Wir müssen nur wissen, dass sie ein Interface öffnet und dem Benutzer die Möglichkeit gibt, einen beliebigen Ordner auszuwählen. Der Parameter ist der Text, der auf dem Auswahldialog oben zu sehen ist. Der gewählte Ordner wird dann in der neuen Variable `theFolder` für eine spätere Referenzierung gespeichert. Dann folgt das Konstrukt der konditionalen Abfrage.

```
if(Statement){doSomething}
```

Sie besagt, dass, wenn die Aussage in der runden Klammer wahr ist, die geschwungene Klammer ausgeführt wird. In unserem Fall wird durch das `!` die Aussage negiert. Ausgeschrieben könnte dort stehen `if(theFolder == false)` oder `if(theFolder != true)`. Die Schreibweise `if(!theFolder)` ist lediglich verkürzt. Wenn wir `if(theFolder)` schreiben, würde das `if(theFolder==true)` entsprechen. Zu beachten ist nicht `=` zu schreiben. Dies bedeutet «wird zu», `==` bedeutet «entspricht». `=` ist eine Zuweisung von Werten, `==` ist ein Vergleich. Der Vollständigkeit halber soll noch erwähnt werden, dass es noch eine weitere Form gibt. Der typsichere Vergleich mit `===`, den wir jedoch nicht weiter betrachten wollen.

Gesprochen wäre die obere Aussage: «Wenn der Ordner nicht existiert - beende die Funktion».
Als nächstes betrachten wir was das Laden der Dateien aus unserem Ordner.

```
var allImages = theFolder.getFiles(type);
```

Ordnerobjekte (`Folder`) haben eine weiter eingebaute Funktion: `getFiles(parameter)`. Diese hat auch einen Parameter: den Dateityp. Hierbei wird nicht nur eine einzelne Datei zurückgegeben, sondern eine Sammlung aller Dateien des angegebenen Typs. Dies ist ein Array. Ein Array sollte man sich als eine Liste vorstellen, deren Positionen über einen Index aufgerufen werden können. An dieser Stelle ist zu beachten, dass das erste Objekt in der Liste nicht mit 1 sondern mit 0 angesprochen wird. Also ist `allImages[0]` das erste Bild unserer Liste.

Nun folgt ein weiterer Sicherheitscheck, ob wir Bilder gefunden haben und die Bestimmung des Rückgabewerts unserer Funktion. Ebenfalls wird der Nutzer, wenn es keine Bilder, gibt darauf hingewiesen.

```
if((allImages.length < 1) || (allImages == null) ){
alert("There are no images of the type: " + type);
return null;
}else{
return allImages;
};
```

Wie wir oben sehen, benutzen wir nochmals die konditionale Entscheidung `if()`, jedoch mit einer Erweiterung, dem `else`. Dies bedeutet, dass auf jeden Fall einer der beiden Blöcke ausgeführt wird. Um es im Pseudocode darzustellen:

```
Wenn die Aussage zutrifft mach dies
ansonsten mach dass
```

length ist eine Eigenschaft jedes Arrays und gibt, wenn wir ein Bild haben 1 zurück.

Innerhalb unsere runden Aussageklammer haben wir zwei Bedinungen verknüpft. In unserem Fall ist das einmal der Check, ob wir mindestens ein Bild haben `(allImages.length < 1)*`. Wir fragen, ob die Länge von unserer Liste kleiner als 1 ist. Diese Aussage verbinden wir mit `||`, dies bedeutet «oder». Wenn Aussage 1 oder Aussage 2 zutrifft, führe den ersten Block aus. Wenn beide Aussagen wahr sein müssen, schreiben wir dafür `&&`. Dies würde nur den ersten Block ausführen, wenn beide Aussagen zutreffen. In unserer zweiten Aussage fragen wir nach `null`, dies entspricht «nichts». Wenn wir einen Fehler beim Laden der Bilder hatten oder `allImages` nicht existiert, dann brich die Ausführung ab. Im ersten Block wird die Funktion beendet. Wenn eine der beiden Aussagen zutrifft, wird mit `return allImages` eine Liste mit Bildreferenzen an die Funktion `main` zurückgegeben.

Wir befinden uns wieder in der Funktion `main` und werden eine weitere Abfrage starten.

```
if(allImages == null) return;
```

Wenn `loadImages` `null` zurück gibt, bricht das Skript ab. Dies ist nötig, da `loadImages` in der Funktion `main` stattfindet. Also führt ein `return` in `loadImages` nur zurück nach `main` und nicht zum Beenden des Skripts. Hier ist ebenfalls eine verkürzte Schreibweise zu sehen. Die geschwungenen Klammern können bei nur einem einzigen Folgebefehl weggelassen werden. Das heißt:

```
if(statement){doSomething;}
```

entspricht:

```
if(statement) doSomething;
```

Ebenfalls könnte

```
if(statement){doSomething;}else{doSomethingElse;}
```

als

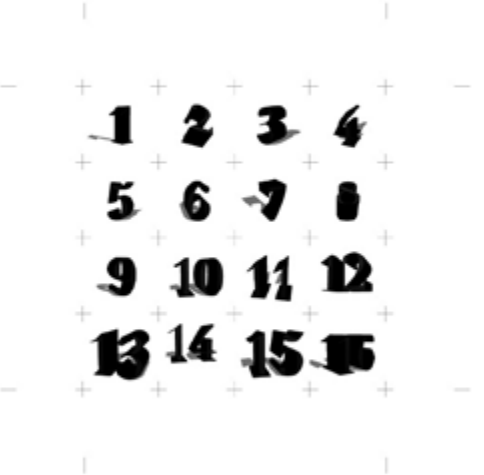
```
if(statement) doSomething; else doSomethingElse;
```

abgekürzt werden. Dies geht jedoch nur bei jeweils einem Befehl nach der Aussage.

Als nächstes folgt die Berechnung unserer Seitenbreite und Seitenhöhe. Die Höhe wird nochmals mit einer Kondition überprüft. Wenn die Menge an Bildern nicht in eine Matrix passt, also nicht 3×3, 4×4 oder 100×100 ist, muss die Höhe nochmals um eine Zeile erweitert werden. Hier kommt das JavaScript Objekt `Math` zum Einsatz.

```
var pw = Math.round(
    Math.sqrt(allImages.length)
    ) * w + (w*2);
var ph = pw;
if(
    Math.round(Math.sqrt(allImages.length))
    != Math.sqrt(allImages.length)
){
    ph = pw + w;
}
```

Dies wirkt kompliziert. Um es zu verstehen, lösen wir den ersten Satz in Werte auf. Betrachten sie dabei das Bild.




```
a ist gleich 16. Die Menge an Bildern.
b ist die Wurzel aus a. Wir wollen es rechteckig. Also 4.
c ist b gerundet. Falls es eine Fließkommazahl ist.
c bleibt in diesem Fall 4, wenn a 16 ist.
d ist 25. Die Breite der Bilder.
e ist c mal d.
Wir staffeln die Bilder nach rechts. 4 × 25 also 100.
f ist d mal 2. Das ist der linke und rechte Abstand zum
Seitenrand. Jeweils eine Bildbreite. Also 50.
```

Daraus ergibt sich eine Seitenbreite von 150.

Die Breite der Bilder multipliziert mit der auf- oder abgerundeten Wurzel aus der Menge an Bildern plus eine Bildbreite links und eine rechts. (Lesen sie den vorherigen Satz nochmals und versuchen sie sich ein geistiges Bild von der Berechnung zu machen.)
oder

```
g = [√a] × d + (d × 2)
```

oder

```
var pw = Math.round(
    Math.sqrt(allImages.length)
    ) * w + (w * 2);
```

Danach setzen wir mit `var ph = pw;` die Höhe der Seite der Breite gleich. Für den Fall, dass `b` (oder `Math.sqrt(allImages.length)`) eine Fließkommazahl ist und wir eine nicht quadratische Matrix erzeugen, also alle Bilder platziert werden sollen, müssen wir die Höhe der Seite um eine Zeile für Bilder erweitern. Deshalb wird in dem `if(statement)` verglichen, ob die gerundete Quadratwurzel nicht der Quadratwurzel entspricht.

```
(Math.round(Math.sqrt(allImages.length)) != Math.
sqrt(allImages.length)
```

oder

```
[√a] ≠ √a
```

oder

Wenn die Wurzel aus der Menge an Bildern nicht der gerundeten Wurzel aus der Menge an Bildern entspricht, ist die Seitenhöhe die Seitenbreite plus eine Bildbreite. Wenn a also 17 ist wäre die Seite 150 breit und 175 hoch.

Ich vermeide hier, um es nicht noch weiter zu verkomplizieren, jede Art von Maßeinheit. In der Standard-Einstellung in InDesign sind es Millimeter. Dies könnten wir definieren, müssen wir aber nicht, da InDesign immer eine Grundeinstellung hat.

Als nächstes folgen das Erzeugen des Dokuments, das Einstellen der Seite auf unsere errechneten Werte, das Überführen der ersten Seite im Dokument in eine Variable und die Definition unseres Startpunktes (x,y).

```
var doc = app.documents.add();
    doc.documentPreferences.pageWidth = pw;
    doc.documentPreferences.pageHeight = ph;
var page = doc.pages.item(0);
var y = w;
var x = w;
```

An dieser Stelle wird der Sammlung an Dokumenten ein neues hinzugefügt. Dabei ist irrelevant, ob bereits ein Dokument existiert. Dann stellen wir die Weite und Höhe der Seite ein. Danach nehmen wir aus der Sammlung an Seiten in dem neuem Dokument die Erste. Der Aufruf `item(0)` ist ebenfalls ein ExtendScript Befehl, der nur in InDesign funktioniert. Wir könnten auch `var page = doc.pages[0]` schreiben. Hier funktioniert beides gleich. Das ist jedoch nicht immer der Fall. Es gibt die Möglichkeit, `doc.pages.middleItem()` oder `doc.pages.lastItem()` abzufragen. Bei einem normalen Array wie unserem `allImages` würde `lastItem()` oder auch `item()` nicht funktionieren. Danach erzeugen wir den linken, obere Koordinate unseres ersten Bildes. x und y sind jeweils eine Bildbreite vom Rand der Seite entfernt. Es gibt noch die Möglichkeit, die ersten drei Zeilen abzukürzen. Dies ist eine komprimiert Schreibweise, die uns noch öfter begegnen wird.

```
var doc = app.documents.add({
    documentPreferences:{
           pageWidth:pw,
           pageHeight:ph
           }
    });
```

Aber zu dieser Schachtelung kommen wir noch. Wir sind bereits halb durch. Es folgt nochmals ein weiteres Konstrukt: die Schleife.

```
for(var i = 0; i < allImages.length;i++){
var rect = page.rectangles.add({
    geometricBounds:[y, x, y + w,x + w]
    });
rect.place(allImages[i]);
rect.fit(FitOptions.FILL_PROPORTIONALLY);
rect.fit(FitOptions.CENTER_CONTENT);
x +=w;
}
```

Mit dieser Schleife würden wir nur Bilder weiter nach rechts setzen. Den «Zeilenumbruch» fügen wir später ein. Die Aussage der Schleife ähnelt der Kondition `for(statment){doSomething}` nur mit `for` anstatt `if`. Es könnte als an «solange» übersetzt werden. Solange der Zähler kleiner als die Menge an Bildern in der Liste ist, führe etwas aus. Die Aussage

```
for(var i = 0; i < allImages.length; i++ )
```

ist eine verkürzte Schreibweise für

```
for(var i = 0; i < allImages.length; i = i + 1 )
```

. Es wäre auch möglich,

```
for(var i = 5; i < 100; i= i +5)
```

Bedenken sie, dass Arrays mit einem Index von 0 beginnen. Also ist myArray[5] das sechste Objekt der Liste.

zu schreiben. Das bedeutet: Zähler i ist 5. Solange i kleiner als 100 ist, erhöhe i um 5. Im ersten Durchlauf ist i 5, im zweiten 10, im dritten 15. Wir würden dann lediglich das 6te ,11te und 16te Bild erhalten*. Es folgt:

```
var rect = page.rectangles.add({
    geometricBounds:[y, x, y + w,x + w]
    });
```

Wir erzeugen auf der ersten Seite ein Rechteck, ähnlich wie bei unserem Dokument, und stellen die Außenkanten ein. Die geometricBounds. Wir könnten auch folgendes schreiben.

```
var rect = page.rectangles.add();
var y1 = y;
var x1 = x;
var y2 = y1 + w;
var x2 = x1 + w;
rect.geometricBounds = [y1, x1, y2, x2];
```

Die Eigenschaft geometricBounds erwartet immer ein Array mit 4 Stellen. Diese geben die linke, obere Ecke und die rechte, untere Ecke an. Immer von der oberen beziehungsweise linken Kante der Seite gemessen. Hierbei steht die y Koordinate vor der x Koordinate. Zu beachten ist auch, dass innerhalb von add() ein Doppelpunkt anstatt eines Gleichheitszeichens bei der Zuweisung des Wertes verwendet werden muss.
Dann folgen drei, einfache Befehle.

```
rect.place(allImages[i]);
rect.fit(FitOptions.FILL_PROPORTIONALLY);
// fit it to the frame
rect.fit(FitOptions.CENTER_CONTENT);
// fit it to the frame
x +=w;
```

Das Bild mit dem Index `i` wird in dem Rechteck platziert. Der Inhalt wird an die größe des Rechteckes angepasst, zentriert und wir erhöhen den x Wert um eine Bildbreite. Hier könnte auch `x = x + w` stehen. Es gibt weiter verkürzte Schreibweisen wie:

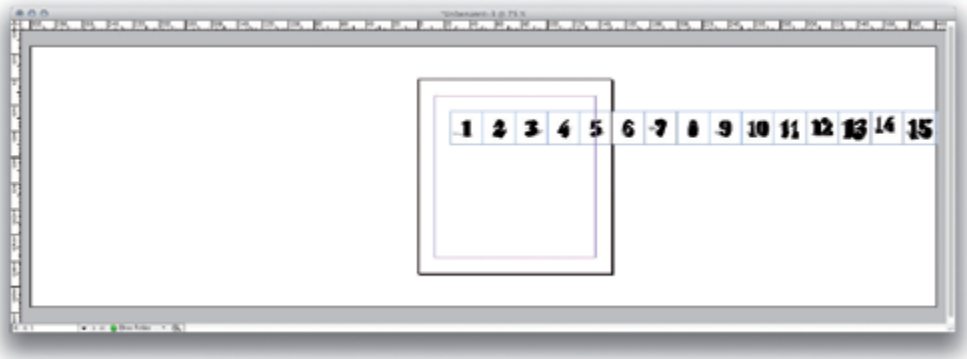
```
x-=5; //  equivalent to x = x - 5;
x*=5; //  equivalent to x = x * 5;
x/=5; //  equivalent to x = x / 5;
```

Da wir in unserer Schleife sind, geschieht dies für die komplette Menge der Bilder, unabhängig davon, ob 5, 100 oder 10000 vorhanden sind. Dies ist eine der großen Stärken von Skripten und Programmen. Die Iteration.

Damit die Bilder nicht nur nach rechts platziert werden, sondern auch umbrechen, wenn x (oder x1) größer oder gleich der Breite der Seite minus der Bildbreite (der Rand) ist, müssen wir eine weitere Kondition hinzufügen.

```
if(x >= pw - w){
x = w;
y+= w;
}
```

Wenn diese Kondition nicht da wäre, würde das Skript die Bilder wie in der folgenden Grafik platzieren. Achtung! Wenn zuviele Bilder über die Arbeitsfläche hinaus gehen, kann das InDesign zum Absturz bringen und/oder das Dokument unwiderruflich zerstören.



Die, die es bis hierher geschafft haben, sollten den vorherigen Code jetzt entschlüsseln können.

Wenn x grösser gleich 150 - 25 ist, um auf unser Beispiel mit der 16 als Menge der Bilder zurückzukommen, setze x zurück auf 25 und addiere auf y 25. Die Zahlenreihe ist dann:

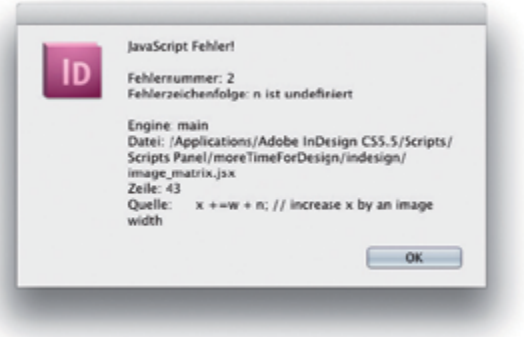
```
i = 0, x = 25, y = 25
```

danach folgt:

```
i = 1, x = 50, y = 25
```

Das passiert solange bis x 125 ist. Dort setzt die Kondition `if(x >= pw - (w))` an. `x` wird in der Kondition zurück auf 25 gesetzt und `y` wird auf 50 gesetzt. Dann ist `i 4`. Wir fangen bei null an zu zählen im Array, also ist `allImages[4]` das fünfte Bild. Und so weiter und so weiter.

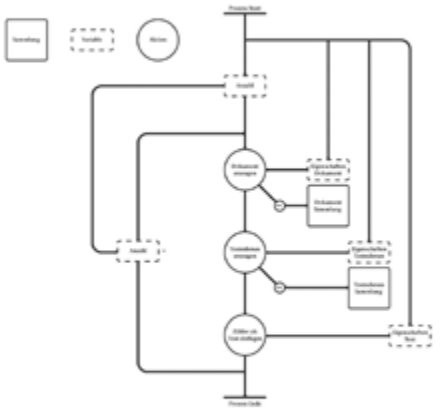
Wir sind am Ende. Probieren sie den Code mit unterschiedliche vielen `.jpg` Dateien aus. Verändern sie ihn, bis er nicht mehr funktioniert und lesen sie die Fehlermeldungen. Dies ist ein weiterverbreitetes Problem. Computer und Programme würden viel an ihrer Mystik verlieren, wenn einerseits die Nutzer die Meldungen lesen würden, anstatt nur auf «ok» zu drücken und andererseits die Meldungen verständlich geschrieben wären.



3.3 Das Beispiel greatPower

Ein weiters Skript, ***greatPower.jsx***, welches aus nur einer Befehlszeile besteht und dennoch mächtig und auch ein wenig gefährlich ist. Wenn bei der Entwicklung eines anderen Skripts immer wieder neue Dokumente erzeugt werden, kann es schnell passieren, dass 10, 20 oder sogar 50 Dokumente geöffnet sind. Die Oberfläche von InDesign bietet nicht die Möglichkeit, alle geöffneten Dokumente zu schließen ohne zu speichern. Das bedeutet, dass bei 50 Dokumenten 50 mal beim Schließen entweder gespeichert, oder das Dokument verworfen werden muss. Mit ***greatPower.jsx*** ist das möglich. Um dies auszuprobieren, erzeugen wir ein paar Dokumente mit dem Skript ***createDocuments.jsx***. Es hat eine stark verkürzt Schreibweise und verwendet eine andere Art von Schleife: die `while` Schleife.

https://github.com/fabiantheblind/MT4D/raw/master/images/algocreatedoc.jpg



https://gist.github.com/2651812

```
// createDocuments.jsx
var counter = 0;
var numberOfDocuments = 22;
while(counter <= numberOfDocuments){
app.documents.add({
    documentPreferences:{
        pageWidth:100,pageHeight:100
        }
    });
var tf = app.activeDocument.pages.item(0).textFrames.
add({
    geometricBounds:[10,10,90,90],
    contents:String(counter) + "Hello World"
    });
tf.characters.everyItem().properties = {
    pointSize:42
    };
counter++;
}
```

https://github.com/fabiantheblind/MT4D/raw/master/images/algogreatpower.jpg

Um diese 23 Dokumente wieder zu schließen, benutzen sie *greatPower-I.jsx*.

```
// Like Uncle Ben saz:
// With great power comes great responsability!
app.documents.everyItem().close(SaveOptions.NO);
```

https://gist.github.com/2651815#file_great_power_i.jsx

4Werkzeuge

Wo bleibt die extra Zeit für Gestaltung, die versprochen wurde? Wie bereits erwähnt, braucht das Lernen einer Sprache etwas Zeit. Die Grammatik und Rechtschreibung können aus einem Buch gelernt werden, die Nuancen und Umgangssprache kann jedoch nur durch das Sprechen geschult werden. Zum Glück sind wir dabei nicht alleine. Es gibt bereits viele Werkzeuge, die frei zur Verfügung stehen und zur Optimierung unserer Arbeitsprozesse genutzt werden können. Dies kann von kleinen Helfern wie ***greatPower.jsx*** bis zu voll ausgearbeiteten Systemen mit eigenen grafischen Benutzeroberflächen gehen.

4.1 AEMap.jsx & AEMap Utilites.jsx

In diesem Sinne habe ich die Skripte *AEMap.jsx* und *AEMap Utilities.jsx* für Adobe After Effects geschrieben, und unter einer Open-Source Lizenz ins Netz gestellt.

http://fabiantheblind.github.com/AEMap/

Viele Motion-Designer kommen irgendwann einmal an den Punkt, an dem sie eine Weltkarte oder die Außenkontur eines Landes benötigen. Bisher war es so, dass immer wieder eine Websuche begann nach benutzbaren Kartenmaterial. Hierbei entstehen einige Probleme. Die Rechte an Kartenmaterial, das «einfach so» aus dem Netz gezogen wurde, sind oft ungeklärt. Manchmal sind es Vektor-Dateien, wenn man Pech, hat sind es Pixelbilder. Diese Quelldatei liegt dann innerhalb eines Projektes und wird mit diesem archiviert. Wenn sie wieder benötigt wird, beginnt die Suche aufs Neue, was bei vielen absolvierten Projekten schnell zeitraubend werden kann. After Effects Projekte können schnell aus hunderten Quelldateien bestehen und sind gerne nach persönlichem Gusto sortiert. Was, wenn nicht nur eine Weltkarte benötigt wird, sondern zum Beispiel die Kontur von Ost-Timor und Papua Neu Guinea?

Selbst wenn eine vernünftige Vektor-Form einer Karte existiert und griffbereit ist wie können die beiden Länder darin gefunden werden? Was ist wenn eine Grenzänderung stattfindet? Und und und. Um diesen Problemen zu entgehen, kann *AEMap.jsx* eine Weltkarte in Rektangularprojektion erzeugen. Dabei entsteht eine After Effects Komposition in einer gewählten Skalierung (immer 2:1) in der 178 Prä-Kompositionen enthalten sind, die 286 einzelne Polygon-Formen beinhalten.

Der Nutzer kann zwischen verschiedenen Einstellungen wählen, zum Beispiel die Karte mit Kontur oder ohne zu zeichnen. Er kann entscheiden, ob alle Polygone auf eine Ebene gezeichnet werden sollen oder ob in die oben genannten Kompositionen gesplittet werden soll. Ebenfalls können 3D-Einstellungen definiert werden und ähnliches mehr. Die Daten bestehen auf einem GeoJson Datensatz, der zum freien Gebrauch ins Netz gestellt wurde. Der gesamte Funktionsumfang ist auf dieser Webseite (http://fabiantheblind.github.com/AEMap/) dokumentiert. Dieses Skript spart nicht nur mir Zeit, sondern auch anderen. Die Resonanz in der After Effects Community ist groß. Daher hat das Skript seit seiner Veröffentlichung auf aescripts.com am 10 April 2012 bereits über 700 Downloads gehabt (heute 23 Mai 2012). Das Tutorial und das Demo wurden bereits über 5000 mal auf Youtube geladen. Aber genug der Selbstbeweihräucherung. Der Vorteil an solchen und ähnlichen Werkzeugen, die auf aescripts.com bereit gestellt werden, ist, dass diese meist aus dem Zwang heraus entstanden sind einen Arbeitsablauf zu automatisieren, um Zeit zu sparen. Sie behandeln Probleme die in Designprozessen immer wieder auftreten.

4.2 createBook

Ein weiteres Werkzeug zum Multipublishing, das ich noch nicht umgesetzt habe und auch vielleicht niemals umsetzen werde, wäre folgendes. Zum Schreiben dieser Arbeit benutze ich eine Auszeichnungssprache genannt Markdown*, in einem Editor genannt iAWriter.

```
#Überschrift 1
##Überschrift 2
**Fett** oder __Fett__
*Italic* oder _Italic_
```

Der obere Text wird nach dem Übersetzen, durch mein CSS (Cascading Style Sheets) gesteuert, so dargestellt:

Überschrift 1
Überschrift 2

Fett oder **Fett**
Italic oder *Italic*

Mit iAWriter habe ich einen weißen Bildschirm und bin frei von jedweder Ablenkung. Mit diesem Set, die Markdown Auszeichnung und iAWriter, bin ich voll auf das Schreiben konzentriert und lasse mich nicht von Layoutaufgaben stören. Um diese Texte in ein lesbare und mit CSS gestaltbare Form zu bringen, benutze ich Jekyll. Jekyll ist ein in Ruby geschriebener Markup-Übersetzer, der statische .html Dokumente exportiert. Mit einem in Ruby geschriebenem Jekyll Plugin **könnte**, neben dem .html, auch eine für InDesign verständliche Datei erzeugt werden, ein .idml (InDesign Markup Language). Ebenfalls könnte ein .epub (Electronic Publication) oder ein RSS-Feed (Rich Site Summary) erzeugt werden. Alle drei sind offene Standards und basieren auf der .xml (EXtensible Markup Language) Formatdefinition.

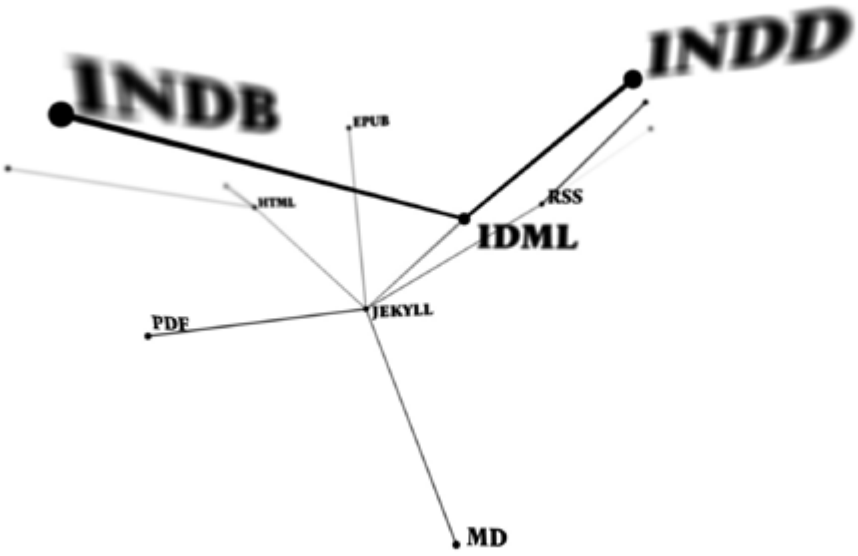
Die Auszeichnungssprache Markdown wurde von John Gruber entwickelt und erlaubt es in reinen Textdokumenten, Auszeichnungen wie Überschrift 1 oder Zitat zu verwenden, ohne dass der Lesefluss maßgeblich beeinträchtigt wird.

Diese Pipeline kann mit unterschiedlichen Templates für verschiedenste Formate gesteuert werden. Auch hier gilt, die erzeugten Ergebnisse der Automation sind nicht das Endergebnis, sondern müssen noch weiterverabeitet werden. Es würde jedoch die Datenübertragung erleichtern und mit einer Anweisung in der Kommandozeile `jekyll --server` Rohdaten für verschiedenste Medien erzeugen. Mit 3 Befehlen könnte dies schon auf einem Webserver bereitgestellt werden.

```
git add --all
git commit -m „This is the message describing the commit“
git push origin gh-pages
```

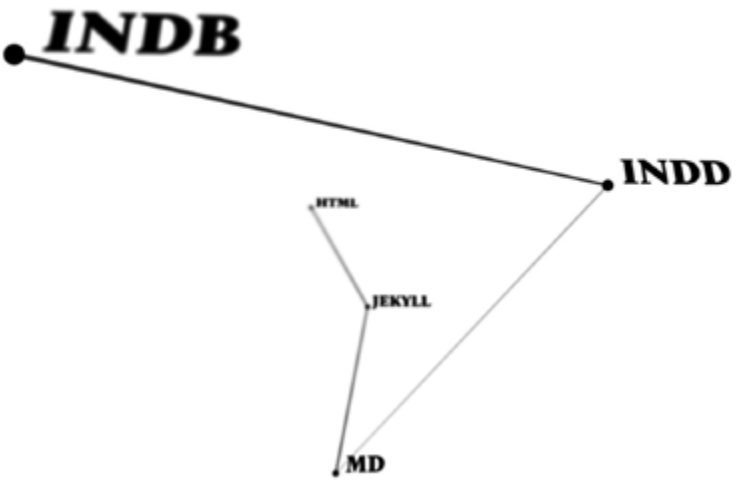
Damit sind die Daten im Netz verfügbar und können eingesehen werden. Hier könnte ein online Korrektursystem mit differenzierten Nutzerrechten eingebunden werden und und und…

Diese Daten müssten dann widerum einen Gestaltungsprozess durchlaufen. Leider ist dies nur eine Idee. Um es umzusetzen, müsste ich Ruby, das idml Schema, das epub Schema, mehr über xml, dtd, xslt, xPath, mehr über html und css und wie ich Plugins für Jekyll schreibe, lernen. Bis zum Ende dieser Arbeit ist das nicht zu bewerkstelligen. Aber toll wäre es.



Der aktuelle Prozess orientiert sich an dieser Idee, ist jedoch nicht komplett automatisert. Die Markdown Dateien werden mit einem Skript zu einem InDesign Buch zusammengeführt und die Auszeichnungen werden in Absatzformate übersetzt. Alle referenzierten Bilder werden auf den entsprechenden Seiten platziert. Ab diesem Punkt wird der «Feinschliff» des Layouts manuell durchgeführt. Das Übersetzen der Auszeichnung wird durch «Suchen und Ersetzen»-Routinen bewerkstelligt. Dies ist eine rohe Art der Transformation. Es wäre sinnvoller, die Übersetzung nicht mit einem Workaround zu erledigen. Wie bereits oben erwähnt, fehlt jedoch dafür die Zeit.

https://gist.github.com/2659939



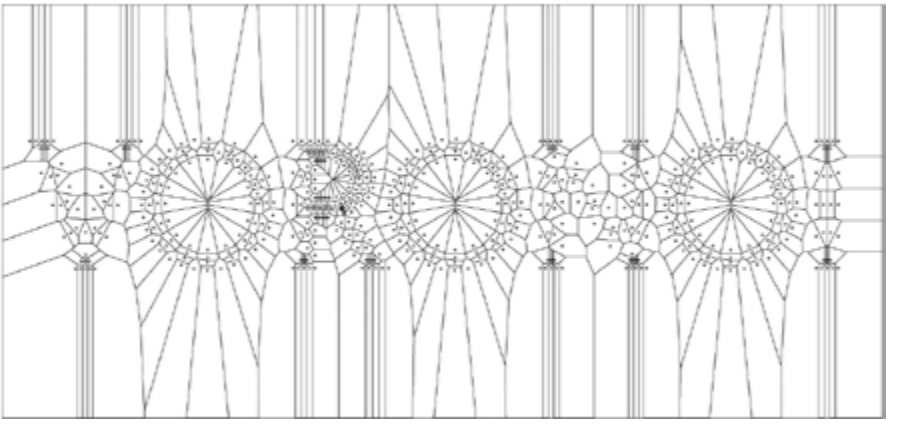
4.3 WAVEFRONT_objfrom AI

Um die filigranen Textgrafiken in dieser Arbeit zu erzeugen, habe ich mich eines weiteren Werkzeuges bedient: Plexus.

Plexus ist ein After Effects Partikelsystem, dass Punkte erzeugt die mit Linien verbunden werden können. Dieses Plugin kann Masken, Lichter und .obj Dateien verarbeiten um Partikel zu generieren. Besonders die letzte Möglichkeit weckte mein Interesse. Wavefront .obj ist ein offenes textbasiertes Format, dass 3D-Daten transportiert. Da ich keine 3D-Applikation beherrsche aber diese Option trotzdem benutzen wollte, habe ich ein Skript geschrieben, dass aus Adobe Illustrator Pfadpunkten .obj Dateien schreibt. Ebenfalls habe ich ein ähnliches Programm geschrieben das aus Processing diesen Dateityp erstellen kann. Dies ist ein Workaround um mein eigenes Unvermögen 3D-Applikation zu beherrschen.

https://gist.github.com/2103355

4.4 Illustrator Voronoi



Um Flächen und ihre optimale Ausnutzung berechnen zu können, kann sich des Voronoi Algorithmuses bedient werden. Dieser wurde bereits in einigen Sprachen implementiert. Ebenfalls in JavaScript. Der Vorteil an der von mir verwendeten Implementation von «gorhil» ist, dass sie nur die Berechnung übernimmt. Die Darstellung ist dabei offen. Dies ermöglicht es mit Illustrator die errechneten Daten zu visualisieren.

https://github.com/gorhill/Javascript-Voronoi
http://fabiantheblind.github.com/Illustrator-Javascript-Voronoi/

5 Die Angst

Wann immer ich gefragt werde, über was ich schreibe oder womit ich mich beschäftige und ich antworte: «Grafik und Programmieren», stoße ich auf die Aussage: «Das werde ich nie lernen.» Woher kommt diese Annahme? Auf der TEDxVancouver im November 2011 beschreibt Jer Thorp eine Situation, die gut unseren aktuellen Zustand darstellt: Wenn man den Menschen, die die Computer erfunden haben, sagen würde: «Es wird der Tag kommen an dem jeder einen Computer besitzt, aber keiner wird wissen wie man programmiert», würden die einen für verrückt erklären[3]. Computer haben sich von den Rechenmaschinen der Spezialisten zu einem alltäglichen Ding entwickelt, das wir nutzen und akzeptieren, dem wir aber nicht unter die Haube gucken möchten. Oder anders:

> *«Jede hinreichend fortschrittliche Technologie ist von Magie nicht zu unterscheiden.»*[4]
> *Sir Arthur C. Clarke's Drittes Gesetz***

Wir haben Computer und ihre Funktionsweise mystifiziert. Ein «Klerus», den wir Programmierer nennen, erzeugt «Ohs» und «Ahs», aber wie dieser Zauber wirkt, wissen wir nicht. Da ich keinen Einblick in die derzeitige Situation an unseren Schulen habe, kann ich das Folgende nur aus meiner eigenen Erfahrung beurteilen. Ein frühes vertraut machen mit Computern gab es in meiner schulischen Laufbahn nicht. Wer Interesse hatte, konnte in die Computer-AG eintreten. Freiwillig. Bei einem eher musisch begabtem Menschen rief dies natürlich wenig Begeisterung hervor. Aufgrund dessen waren Computer auch für mich mythische Wesen. Ich denke, damit stehe ich nicht allein.

*Jer Thorp alias @blprnt ist ein kanadischer Medienkünstler. Er ist derzeit Data Artist in Residence bei der New York Times und ist Lehrbeauftragtern an der Universität von New York im ITP (Interactive Telecommunications Program).

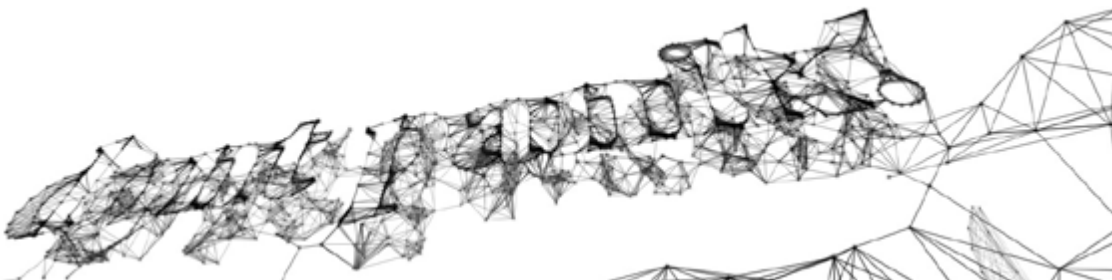
**Sir Arthur C. Clarke war ein britischer Science-Fiction Autor. Sein wohl bekanntestes Werk ist «2001: A Space Odyssey», welches von Stanley Kubrick verfilmt wurde und bei dem er ebenfalls Co-Autor war. Seine drei Gesetze sind folgende:

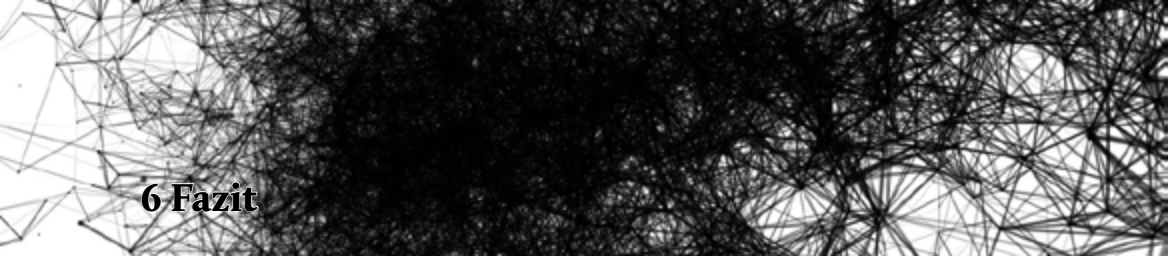
1. «Wenn ein angesehener, aber älterer Wissenschaftler behauptet, dass etwas möglich ist, hat er mit an Sicherheit grenzender Wahrscheinlichkeit Recht. Wenn er behauptet, dass etwas unmöglich ist, hat er höchstwahrscheinlich Unrecht.»

2. «Der einzige Weg, die Grenzen des Möglichen zu finden, ist ein klein wenig über diese hinaus in das Unmögliche vorzustoßen.»

3. «Jede hinreichend fortschrittliche Technologie ist von Magie nicht zu unterscheiden.»

Um auf den Vergleich des Erlernens von Fremdsprachen zurück zu kommen - natürlich haben wir ein unwohles Gefühl, wenn wir uns dort aufhalten wo wir uns nicht auskennen. Wir werden weiterhin versuchen, so oft wir können, unsere eigene Sprache anstatt der Landessprache zu nutzen. Dies ist menschlich und ein wenig töricht. Mit dem Erlernen einer Computersprache vollzieht sich ein Wandel. Es ist der Schritt vom Bediener zu einem Macher von Werkzeugen. Die Programme, die wir als Grafiker nutzen, sind nach der Idee eines Teams von Menschen geformt. Wir haben keinen Einfluss auf die Funktionsweise. Mit dem Schreiben von Programmen oder Skripten kann dies überwunden werden.





6 Fazit

Programmieren ist nicht leicht, aber leichter als man denkt.

Gerade der Anfang hat eine flache Lernkurve und der Lernende wird auch noch mit einem neuen und komplizierten Wortschatz konfrontiert. Ab einem gewissen Punkt ändert sich dies, aber der muss erst erreicht werden. Das Problem ist dabei, dass wir gerne von komplexen Systemen träumen, um solche umzusetzen, fehlt uns dann das Können. Denn zuerst gilt es, die einfachsten Schritte zu machen. Wir werden in ein Entwicklungsstadium zurückgeworfen, in dem wir noch nicht unsere Bedürfnisse formulieren können, aber möchten. Dies kann demotivierend sein. Wenn der flache Teil der Lernkurve endlich überwunden ist, kann diese Fähigkeit wie ein weiteres Werkzeug verwendet werden. Ab diesem Punkt eröffnet die Automation durch Programmierung für uns neue Welten in der Gestaltung. Aufgaben die einmal unmöglich schienen, können nun angegangen werden. Tausend Seiten mit tausend Bildern sind kein Problem. Hinzu kommt, dass beim Programmieren lernen sich ein neues Verständnis für die Logik von Computern bildet und deren Eigenheiten nicht mehr magisch sind. Dies nimmt die Angst vor der direkten Interaktion mit der Maschine. Ich denke nicht mehr, dass jeder programmieren lernen muss, um die Rechner, die wir haben, voll ausnutzen zu können, ist es jedoch unerlässlich. Denn mit den Programmen, die wir benutzen, kratzen wir nur an der Oberfläche ihrer Leistungsfähigkeit. Wir sind beschränkt auf die Funktionsweise, die ein anderer für uns entworfen hat, obwohl wir der Meinung sind, dass wir es eigentlich besser wissen. Ein weiterer Vorzug ist folgender. Nur wenn ich die Grundlagen einer Idee verstanden habe, kann ich darüber kommunizieren. Wie kann ein Gestalter eine Webseite planen, ohne ein wenig Verständnis für die Vorgänge und Funktionsweisen zu besitzen? Nur durch ein Grundverständnis können die Möglichkeiten, die dieses Medium bietet, gezielt eingeschätzt und ausgenutzt werden.

Der Idealfall zum Erlernen von Computersprachen ist unter Zwang, beispielsweise wenn sie vor einem Problem stehen, das manuell nicht gelöst werden kann, aber gelöst werden muss. Alle anderen müssen sich selbst überwinden.

Es kann vieles mit einer Automation durch Programmierung bewerkstelligt werden. Die Automation ist jedoch nicht die Lösung, sondern die Entschlackung unseres Arbeitsprozesses von geisttötenden Aufgaben. Jedes Programm oder Skript, welches wir schreiben, ist ein kreativer Prozess an dem wir unsere Fähigkeiten verbessern. Abschließen möchte ich mit einem Zitat von Marshall McLuhan um noch eine weiter Betrachtungsweise der Automation zu eröffnen.

> *«Automation ist Information, und sie macht nun nicht nur den Spezialaufgaben im Bereich der Arbeit ein Ende, sondern auch der Auffächerung im Bereich des Lernens und Wissens. In Zukunft besteht die Arbeit nicht mehr darin, seinen Lebensunterhalt zu verdienen, sondern darin, im Zeitalter der Automation leben zu lernen. Das ist ein ganz allgemeines Verhaltensmuster im Zeitalter der Elektrizität. Es beendet die alte Dichotomie von Kultur und Technik, von Kunst und Handel und von Arbeit und Freizeit. Während im mechanischen Zeitalter der Fragmentierung Freizeit die Abwesenheit von Arbeit bedeutet oder bloßes Müßigsein, gilt im Zeitalter der Elektrizität gerade das Gegenteil.»*[5]
> *Marshall McLuhan*
> *«Die magischen Kanäle - Understanding Media»*

7 Die kleine Terminologie

Dieser Abschnitt ist zur Erläuterung gedacht und um den Lesefluss nicht durch Sekundärinformationen und Erklärungen von Fachbegriffen zu unterbrechen.

7.01 Was ist Code?

Als Code bezeichnen wir in der Regel Informationen, die verschlüsselt (encoding) werden, um dann an einer weiteren Stelle wieder entschlüsselt zu werden (decoding). Zum Beispiel stellt Morse-Code Buchstaben dar, indem ein einziges unmoduliertes Signal, zum Beispiel ein Ton, in kurze und lange Sequenzen unterteilt wird. Der Rezipient kann dann, wenn er des Systems mächtig ist, diese Informationen entschlüsseln und zu der originalen Nachricht wieder zusammensetzen. Ein weiteres Beispiel ist der Abakus. Dieser erlaubt es, wenn der Benutzer des Systems mächtig ist, Rechenoperationen auszuführen. Und als drittes die Knotenschrift der Inkas, die, wie der Name bereits besagt, aus einem System von Knoten auf einem Satz Schnüren bestand. Heutzutage findet das Wort «Code» im Computerbereich oft Verwendung als Kurzform des Ausdrucks «Source-Code», also Quelltext eines Programms.

7.02 Was ist ein Programm?

Das Bild, das viele im Kopf haben, wenn sie überlegen wie Programme enstehen, ist stark durch Film beeinflusst. Wir sehen junge, meist übergewichtig und verpickelte Menschen vor uns, die in abgedunkelten Räumen zwischen Monitoren, Kabeln und Pizzapackungen auf ewige grün leuchtende Zahlenkolonnen blicken, die für uns keinerlei Sinn ergeben. Diese oder ähnliche Bilder sind inspiriert aus einer Zeit, in der Computer nur einfarbige Pixel hatten und grafische Benutzeroberflächen, wie Windows, noch aus der Kommandozeile gestartet wurde. Als Hommage an diese Vorstellung hat «Duiker101» das Programm HackerTyper entworfen, welches mit bereits vorgegebenem Text, allein durch Tastendruck, den Bildschirm mit kompliziertem Quelltext füllt. Dabei ist es irrelevant welche Tasten der Benutzer drückt. Verwerfen sie diese überzogene Vorstellung. Es ist nicht so, dass es dies nicht gibt, dennoch entspricht es nicht der Regel.

http://hackertyper.net/

Ein Programm ist laut Duden: «die nach einem Plan genau festgelegten Einzelheiten eines Vorhabens»[6]. Unter dieser Betrachtungsweise ist jede Bauanleitung für Möbelstücke, eine Beschreibung des Weges von hier zum Bahnhof oder das Rezept für Sahnetörtchen ein Programm. Bloß, dass in letzterem Fall nicht ein Computer die Anweisungen ausführt, sondern ein Mensch. Die Sprache, in der dieses Programm geschrieben ist, ist Deutsch. Der große Unterschied zu einem «Computersprach-Programm» liegt hier in der Möglichkeit der Interpretation. Die ausführende «Maschine», in diesem Fall der Mensch, kann solche Angaben, wie «eine Priese Salz» oder «eine Messerspitze Meerrettich», verarbeiten. Ein Computer ist hierzu noch nicht fähig. Er bräuchte eine eindeutigere Angabe, wie zum Beispiel 50 Gramm. Das Programm muss an dieser Stelle noch von dem Begriff des Algorithmus abgetrennt werden.

7.03 Was ist ein Algorithmus?

Auch wenn sich die beiden Begriffe Algorithmus und Programm in ihrer Bedeutung überschneiden, sollten sie auf folgende Weise unterschieden werden. Der Algorithmus für Milch holen, wäre in Pseudocode:

```
wenn (Aussage (kein Milch ist im Kühlschrank) wahr ist):
hole neue Milch! wenn nicht: tue nichts!
```

Das Programm für Milch holen würde voraussetzen, dass alle Schritte und Notwendigkeiten bis zum Übergang der Milch in das Eigentum des Holenden bekannt und definiert sind. Also so etwas wie:

```
Person fabian ist gleich neu Person;
Kühlschrank Schrank ist gleich neuer Kühlschrank;
Kühlschrank Menge Milch ist gleich 1l;
jeden morgen fabian trinke Milch aus Kühlschrank;
Menge Milch reduziere um 0.2l;
jeden morgen fabian beobachte Menge Milch;
wenn (Milch kleiner gleich 0.2l ist)
fabian hole Milch im Supermarkt;
```

Und so weiter und so ähnlich. Diese Funktionsanweisungen müssten noch detaillierter ausgearbeitet werden. Hierbei sei zu beachten, dass solche Objekte wie Kühlschrank und Person bereits implementiert, also bekannt, sein müssen. Das Programm, im Vergleich zum Algorithmus, muss alle eingesetzten Mittel kennen und/oder selber beschreiben.

7.04 Was ist die Syntax?

Die Syntax ist die Form in der die Programmiersprache ausgestaltet ist. Die Syntax einer Programmiersprache besteht aus reservierten Worten, wie zum Beispiel in Java `new, while, null, true,` Operatoren wie `+,-,*,.` und Kontrollstrukturen wie `if(){}else{}` oder `for(int i = 0; i < x;i++)`. Lesen sie diese «Sätze» kurz. Wie würden sie es sprechen? Ausgesprochen wäre dies:

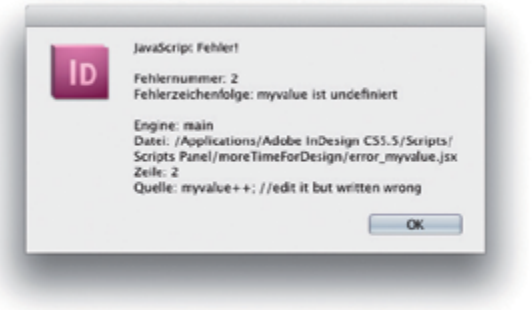
```
For int i gleich 0, i kleiner x, i plus plus
```

Es gibt in JavaScript einige reservierte Worte, die beim Schreiben nicht verwendet werden dürfen.

```
break, case, catch, continue, debugger, default, delete,
do, else, finally,for, function, if, in, instanceof, new,
return, switch, this, throw, try, typeof, var, void,
while, with, null, false, true[7]
Vgl.: Mozilla Developer Network, Reserved Words
```

Diese Worte, die die Syntax bilden, dürfen nur für bestimmte Aufgaben verwendet werden! Wenn ein Programmierer seine Variable `null` nennt, wird das Programm beim Ausführen auf einen Fehler stoßen. Die genannten Strukturen, Operatoren und reservierten Worte müssen erlernt werden. Eine vollständige Beschreibung aller würde jedoch den Rahmen dieser Arbeit sprengen. Weiterhin ist in der Syntax auch auf die Groß- und Kleinschreibung zu achten. Eine Variable, die mit dem Namen `myValue` initiiert wird, muss auch mit diesem Namen aufgerufen werden. Bei einer falschen Schreibweise `MyValue` würde das Programm warnen, dass die aufgerufene Variable nicht existiert.

```
var myValue = 5; // define a variable
myvalue++; //edit it but written wrong
```



Es gibt Sprachen, die versuchen ihre Syntax so weit wie möglich an unseren Sprachen zu orientieren. Zum Beispiel Applescript:

```
tell application "Safari" to activate
```

Und wieder andere, wie die esoterische Programmiersprache «Brainfuck», die mit ihren acht Zeichen `+ - < > [ ] , .` voll funktionsfähig, aber nicht für das Schreiben von Programmen gedacht ist, sondern eher ein Gedankenmodell darstellt.

Das Brainfuck Hello World:

```
>+++++++++[<++++++++>-]<.>+++++++[<++++>-]<+.+++++++..+++.
[-]>++++++++[<++++>-]<.>++++++++++[<+++++>-]<.>++++++++[
<+++>-]<.+++.------.--------.[-]>++++++++[<++++>-]<+.[-]+
++++++++.[8]
```

Andere Sprachen, wie C++, Processing oder JavaScript, sind zwischen diesen Extrema angesiedelt und vereinen, in einer für das geübte Auge lesbaren und dennoch kompakten Art, solche Befehlsaufrufe.

7.05 Was ist Pseudocode?

Im Verlaufe dieser Arbeit werde ich immer wieder auf die Darstellung von Algorithmen in «Pseudocode» zurückgreifen. Pseudocode besteht nicht aus einer bereits definierten Syntax und kann auch nicht von einem Compiler übersetzt werden, sondern dient nur zur Darstellung eines logischen Ablaufs. Wie oben in dem Milchalgorithmus zu sehen ist, versucht Pseudocode einen Programmablauf in menschenlesbarer Form darzustellen.

7.06 Was ist ein Compiler?

Der Compiler ist ebenfalls ein Programm, dass aus Hochsprachen wie C++ maschinenlesbaren Quelltext erzeugt (Assemblercode). Diesen Prozess bezeichnet man auch als Kompilierung.

7.07 Was ist eine IDE (Integrated Development Envoirement)?

Um Quelltext zu schreiben, bedarf es nicht viel. Der einfachste Texteditor reicht aus, um komplette Programme zu schreiben. Im Laufe der Zeit wurde jedoch viel Software programmiert, um das Schreiben von Quelltext zu erleichtern. Dies beginnt bei einfachem Syntax-Highlighting bis hinzu kompletten Entwicklungsumgebungen, die während des Schreibens die Syntax auf ihre Validität prüfen und gegebenenfalls Vorschläge machen, was gemeint sein könnte oder warnen bei nicht verwendeten Programmteilen. Wir werden die einfachste Möglichkeit nutzen, die sich uns bietet und die auf beiden Plattformen (Windows und Mac OS X) zur Verfügung steht. Das ExtendScript Toolkit.

Mehr dazu im Abschnitt «Was ist Hello World?».

7.08 Was ist Hello World?

Das «Hello World» Programm hat sich als Standardbeispiel etabliert, um die Syntax einer Sprache zu erklären. Es ist der erste Versuch in einer Sprache ein Programm zu schreiben, das eine Aussage trifft. Hello World. Exerzieren wir das einmal durch. Um für Adobe InDesign, After Effects, Illustrator, Photoshop, Photoshop Elements, Photoshop Elements Organizer, Bridge, Audition, Media Encoder und Premiere Pro Skripte zu schreiben, liefert Adobe eine eigene IDE mit.

Das ExtendScript Toolkit. Dies ist nicht der benutzerfreundlichste Editor. Er hat jedoch einige Vorteile, die die Entwicklung von Skripten sehr einfach macht. Die wichtigste Eigenschaft ist die, ein Skript, ohne es zu speichern, ausführen zu können. Das Toolkit wird bei der Installation von Adobe Produkten direkt mit geliefert. Suchen sie es in ihren Dienstprogrammen, dort sollten sie fündig werden. Wenn nicht, gehen sie auf diese Webseite http://www.adobe.com/devnet/scripting.html, laden und installieren sie es.

Wenn es dann installiert/gefunden und gestartet ist, geben sie folgende Text ein:

```
alert("Hello World");
```

und drücken sie auf den «Play/Run» Knopf oben rechts. Sie können dies auch über eine Tastenkombination ausführen. CMD-r oder CTRL-r, abhängig von ihrer Plattform.

Herzlichen Glückwunsch. Ihr erstes JavaScript.

Um dieses Skript in InDesign oder Photoshop auszuführen, muss die Ziel Applikation aus dem PullDown Menü auf der oberen Leiste gewählt werden. Beim Öffnen ist es auf ExtendScript Toolkit gestellt. Wählen sie dort InDesign aus. Das Toolkit wird sofort fragen, ob InDesign auch gestartet werden soll. Bestätigen sie das und führen sie das Skript noch einmal aus. Sie werden sehen, dass der Computer zu InDesign überwechselt und den gleichen Hinweis gibt. Probieren sie weiter Skripte und Kalkulationen aus. Zum Beispiel:

```
var h = "Hello",w = "World";
var calc = (10*50)/23 - 1;
alert(h + " " + w +"! Your result is: "+ calc );
```

Oder speziell für InDesign:

```
var doc = app.documents.add();
doc.pages.item(0).textFrames.add({
geometricBounds:[13,13,23,50],
contents:"Hello InDesign"});
```

Oder gehen sie auf die Seite codecademy.com um mehr zu lernen.

7.09 Was ist Syntax-Highlighting?

Syntax-Highlighting ist eine Hilfestellung für Programmierer, um ihren Quelltext übersichtlicher zu gestalten. Hierbei werden bestimmte Teile wie Operatoren, Kommentare, Funktionsdeklarationen oder reservierte Worte farblich hervorgehoben beziehungsweise zurückgenommen. um das Lesen zu erleichtern.

7.10 Was ist eine API

(Application Programming Interface)?

Dies ist die Schnittstelle, die ein Programm bietet, um auf seine Funktionen zugreifen zu können. Dies wird beim Schreiben von Programmen wie InDesign von den Programmierern definiert. Diese Schnittstelle ist wie ein Baum mit Querverweisen aufgebaut.

Um in InDesign einem bestehenden Dokument auf der ersten Seite eine Textbox hinzuzufügen, muss durch diesen Baum manövriert werden.

```
app.activeDocument.pages.item(0).textFrames.add();
```

Diese Zeile erzeugt in der linken oberen Ecken eine Textbox.

All diese Befehle und Eigenschaften müssen nicht auswendig gelernt, sondern können nachgeschlagen werden. Im ExtendScript Toolkit kann unter dem Menüpunkt «Hilfe / ObjektModell Viewer» ein Hilfsprogramm aufgerufen werden, das alle Befehle mit einer kurzen Erklärung enthält. Oder es kann auf der Seite von jongware eine .chm Datei heruntergeladen werden, die die gleichen Informationen enthält und mit einem .chm Viewer durchsucht werden kann. http://www.jongware.com/idjshelp.html

7.11 Was ist Objektorientierung?

Als Objektorientierung, auch als «OO» abgekürzt, versteht man eine bestimmte Art, wie Programme aufgebaut sind. Ein Objekt ist ein gekapselter Teil des Programmcodes, der Schnittstellen und Methoden bietet und seine Eigenschaften an weitere Objekte vererben kann. Um dies besser zu verstehen, möchte ich ein hervorragendes Beispiel aus «Processing: A Programming Handbook for Visual Designers and Artists» von Casey Reas and Ben Fry zur Hilfe nehmen[9].

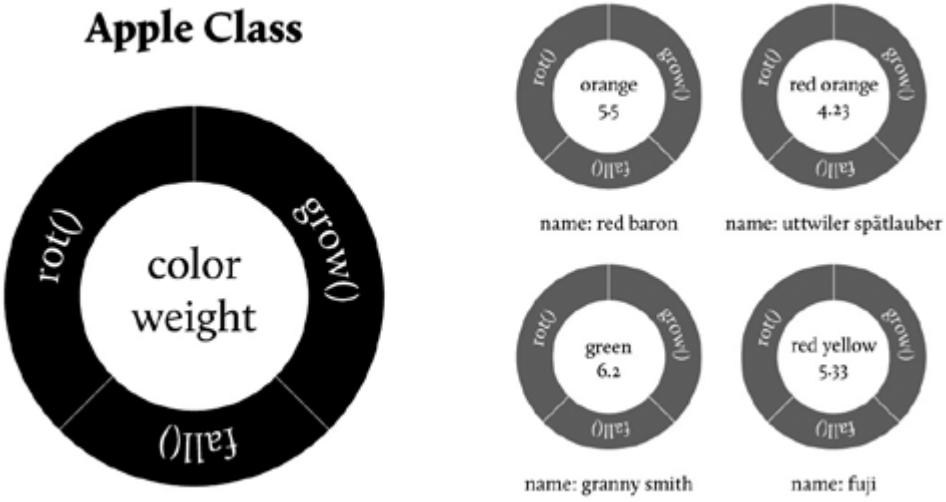
Klammer auf!

{

Der Proto-Apfel* hat bestimmte Eigenschaften, wie Gewicht und Farbe. Hinzu kommen bestimmte Methoden, wie Fallen, Wachsen und Verrotten, die durch die Eigenschaften beeinflusst wer-

den können. Wenn nun ein Baum wächst und Äpfel produziert, erzeugt er nach dem Bauplan des Proto-Apfels neue Äpfel und jedem werden bestimmte Werte übergeben.

Der Baum erzeugt den Apfel und ruft kontinuierlich die Methode «Wachsen» auf. Hierbei wird der Wert des Gewichtes inkrementiert. Sind die Früchte dann reif, wird, abhängig vom Gewicht des einzelnen Apfels, die Methode «Fallen» ausgelöst. Wenn der Apfel dann auf dem Boden liegt und die Methode «Fallen» beendet ist, beginnt die Methode «Verrotten» ihre Arbeit. Die Farbe des Apfels verändert sich und das Gewicht wird wieder dekrementiert.



Das bedeutet, der Proto-Apfel selber wird nicht angerührt, sondern Instanzen von diesem. Um dies noch auf die Spitze zu treiben, haben wir nicht nur eine Art Apfel, sondern verschieden Sorten. Also ist die Klasse Granny Smith eine «Kindklasse» der «Basisklasse» Apfel. Granny Smith erbt alle Eigenschaften der Klasse Apfel, ohne dass sie neu implementiert werden müssen und bekommt noch eine weitere Eigenschaft. Den Namen.

Diese Konstruktionsweise spiegelt sich nicht nur im «Scripting» oder bei Äpfeln wieder. Die Benutzung einiger Komponenten in unseren Werkzeugen ist genau nach diesem Prinzip organisiert.

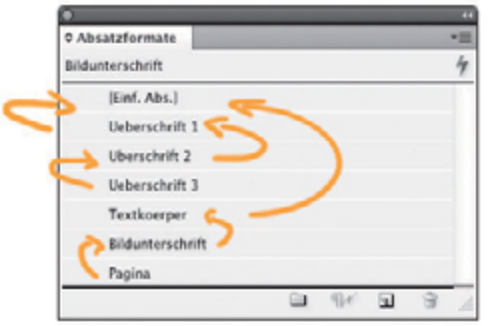
Eine weitere Klammer auf

{

In InDesign hat der Benutzer die Möglichkeit, Absatz- und Zeichenformate anzulegen, ohne die das Setzen eines Buches eine wirklich zeitraubende Arbeit wäre. Eine einfache Gruppe von Absatzformaten kann wie folgt aufgebaut sein.

- [Einfacher Absatz]
- TextKörper
- Überschrift 1
- Überschrift 2
- Überschrift 3
- Bild Unterschrift
- Pagina

In dem Format «[Einfacher Absatz]» wird eine Schriftart und eine Schriftgröße definiert. Dieses Format ist das Proto-Format, von dem sich alle weiteren Formate ableiten. Sie erben alle die Eigenschaften Schrift (appliedFont) und die Schriftgröße (pointSize). Das Format «Überschrift 1» (Ü1) bekommt dann eine eigene Schriftgröße und einen fetten Schnitt der Schriftart. In dem Format «Überschrift 2» (Ü2) wird dann festgelegt, dass nicht mehr das «[Einfacher Absatz]» Format als Basisklasse benutzt wird, sondern Ü1. Damit erbt Ü2 alle Eigenschaften von Ü1. In Ü2 wird dann nur noch eine neue Schriftgröße festgelegt. Das Gleiche kann dann mit «Überschrift 3» (Ü3) passieren. Ü3 basiert auf Ü2 und bekommt ebenfalls eine eigene Schriftgröße. Jetzt basiert Ü3 auf Ü2, welches auf Ü1 basiert. Ü1 basiert auf «[Einfacher Absatz]». Ähnlich verfahren wir mit den weiteren Formaten. «Textkörper» basiert auf «[Einfacher Absatz]», «Bildunterschrift» auf «Textkörper», nur kleiner, und «Pagina» auf «Bildunterschrift», aber mit 70% Deckkraft. Wenn der Gestalter nun entscheidet, dass eine andere Schriftart vonnöten ist, ändert er sie nur in «[Einfacher Absatz]» und alle Kinder werden entsprechend angepasst. Dies ist ebenfalls objektorientiert.




```
        }
}
```

7.12 Was sind Funktionen/Methoden?

Eine Funktionen oder eine Methoden sind gekapselte Programmteile, an die Parameter übergeben werden können und die an den gegebenen Parametern eine Kalkulation durchführen. Sie können dann Werte zurückgeben. Dies ist nützlich, wenn bestimmte Tätigkeiten an unterschiedlichen Positionen eines Programms oder Skripts mehrmals aufgerufen werden müssen.

7.13 Was ist ein Bug?

Ein Bug ist ein Fehler, der das Programm von seiner einwandfreien Ausführung abhält. Dies kann nur ein Rechenfehler sein oder ein Fehler, der das gesamte Programm zum Absturz bringt.

7.14 Was ist Debugging?

Debugging ist der Prozess des Findens und Korrigierens von Fehlern.

7.15 Was ist ein Workaround?

Ein Workaround ist eine kreative Lösung, um ein Problem zu umgehen. Wenn zum Beispiel eine API eine bestimmte Funktion nicht anbietet, muss ein Weg um dieses Problem herum gefunden werden. Lesen sie den Abschnitt «2.2.1 Das Beispiel try char».

7.16 Welche Konventionen gibt es?

Neben den Bestimmungen der Syntax gibt es verschiedene Konventionen, die das Lesen von Skripten/Programmen einfacher machen sollen. Zum Beispiel ist es üblich, dass die Funktionsweise am Beginn eines Skripts in einem Kommentar erklärt oder die Variable i (für Iterator) als Zähler in Schliefen verwendet wird.

7.17 Die Herkunft von JavaScript

JavaScript orientiert sich in seiner Syntax an C und C++.

Zur Veranschaulichung folgt hier ein Programm, das diese Aufgaben erledigt:

- Zwei Zahlen in Variablen definieren
- Ein Objekt mit folgenden drei Eigenschaften erzeugen
- Einem Namen
- Einer Funktion, die zwei Zahlen miteinander vergleicht und feststellt, welche die größere von beiden ist.
- Ein Ergebnis zurück geben
- Einen Satz aus allen Variablen zusammenfügen
- Diesen Satz darstellen

Dieses Programm liegt in verschiedenen Sprachen vor.

Ich werde kurz auf kleine Unterschiede eingehen und versuchen an drei Beispielen zu erläutern, warum JavaScript eine einfache und mächtige Sprache ist. An dieser Stelle ist es nicht zwingend, den Programmcode komplett zu verstehen. Beachten sie nur die syntaktischen Unterschiede. Vergleichen sie die Funktion oder Methode `int Object::compare(int a, int b)` in C++ mit `int compare(int a, int b)` in C und betrachten sie danach die Funktion this.`compare = function(a,b)` in JavaScript. Es gibt kleine Unterschiede, aber die gemeinsame Herkunft ist unübersehbar. Eine tiefere Erklärung des Quelltextes findet sich in den Kommentaren der einzelnen Programme und des Skriptes.

Assembler (anObject.s)

Dies ist die Variante, die die Maschine versteht. Es wurde jedoch nicht von mir geschrieben, sondern ist das Ergebnis des Kompilierungs-Prozesses des C++ Programms. Näher wäre nur noch Nullen und Einsen zu schrieben. Ja, es gibt auch diesen Aspekt der Programmierung, der genau dem entspricht, was Menschen davon abhält, sich mit ihr auseinander zu setzen. Um dort an zu gelangen, ist es ein weiter Weg, den wenige gehen und auch wir in dieser Arbeit nicht beschreiten werden. Aus Platzgründen werde ich hier nur die ersten 23 Zeilen zeigen. Das gesamte Programm ist in Assembler über 700 Zeilen lang.

```
.section   __TEXT,__text,regular,pure_instructions
.globl     __ZN6Object7compareEii
.align     1, 0x90
__ZN6Object7compareEii:
Leh_func_begin1:
pushq      %rbp
Ltmp0:
movq       %rsp, %rbp
Ltmp1:
movq       %rdi, -8(%rbp)
movl       %esi, -12(%rbp)
movl       %edx, -16(%rbp)
movl       -12(%rbp), %eax
movl       -16(%rbp), %ecx
cmpl       %ecx, %eax
jle LBB1_2
movl       -12(%rbp), %eax
movl       %eax, -24(%rbp)
jmp LBB1_3
LBB1_2:
movl       -16(%rbp), %eax
movl       %eax, -24(%rbp)
LBB1_3:
```

Gruselig oder nicht? Zu unserem Glück müssen wir so etwas weder schreiben noch lesen können. Von 1969 bis 1973 wurde die Sprache C von Dennis Ritchie im «Bell Labs Computing Sciences Research Center» entwickelt. Diese Sprache wird beim Kompilieren in Assembler Code übersetzt. Dies ist eine Automation zum schreiben von Automationen. Der Vorteil, an C im Vergleich zu Assembler, ist eine viel einfachere Syntax, die es erlaubt den Schwerpunkt der Aufmerksamkeit auf das Konzept des Programms zu richten.

ANSI C noObject.c

Im Unterschied zu C++ und JavaScript ist die Sprache C nicht für eine Objektorientierung ausgelegt. Deshalb hier einmal das Beispiel noObject.c, welches die gestellte Aufgabe des Vergleichs löst, aber kein Objekt erzeugt. Die Syntax ähnelt sehr der von JavaScript. Sobald jedoch Pointer hinzukommen, wird es komplizierter.
Aber ich schweife ab. Keine Sorge mit Problemen wie: «In welchem Speicher lege ich meine Variable ab» und «Habe ich auch den Müll rausgetragen» (Das löschen eines unbenutzten Objekts wird «Garbadge Collection» also «Müll sammeln» genannt), werden wir in JavaScript nicht konfrontiert. All diese Funktionalität wird von der Maschine im Hintergrund erledigt. Somit bleibt Zeit sich mit der Funktionsweise und dem Konzept zu beschäftigen. Stellen sie es sich vor, wie es wäre, das Auto vor dem Fahren zusammen zu bauen, anstatt in das fertige Auto einzusteigen und loszufahren. Um die Abstammung von JavaScript verstehen zu können, folgt die Vergleichs-Funktion in C.

Objektorientierung hat den Vorteil des Versteckens beziehungsweise Kapselns von Daten. Es kommt die Klasse hinzu. Sie erlaubt ganze Programmteile auszulagern und nur eine Schnittstelle für ihre Benutzung zu liefern. Für genauere Erklärungen lesen sie den Abschnitt 7.11 Was ist Objektorientierung?.

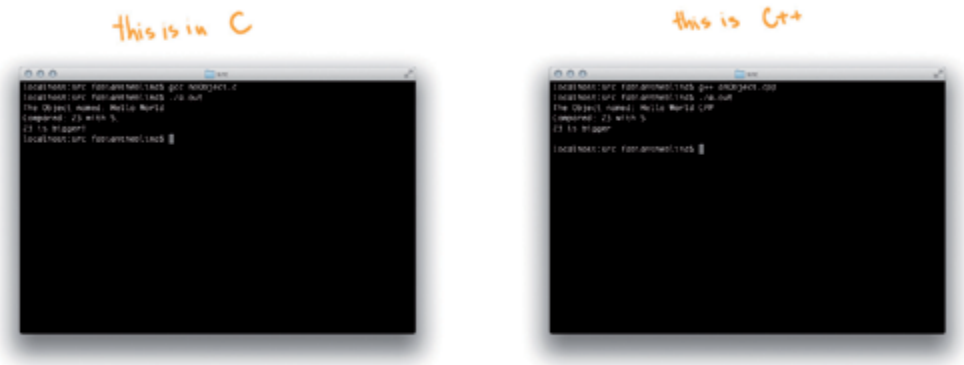
```c
// include the io lib
#include<stdio.h>
// the compare function takes integer as arguments
int compare(int a, int b){
    if(a>b){return a;}else{return b;}
}
int main(void){// the main function
    char name[11] = "Hello World"; // our string
    int a = 23,b = 5; // the values to compare
    // now print all that stuff
    // with %s you insert string arguments
    // with %d digits
    printf("The Object named: %s\nCompared: %d with
%d.\n%d is bigger!",name,a,b,compare(a,b));
return 0;// everything went fine return 0
}
```

C++ anObject.cpp

Basierend auf C wurde zu Beginn der Achtziger Jahre des letzten Jahrhunderts im letzten Jahrtausend die Sprache C++ von Bjarne Stroustrup in den AT&T Labs entwickelt. Diese Sprache ist nach dem Konzept der Objektorientierung aufgebaut. In diesem Beispiel ist bereits zu sehen, dass die Menge an Code, die geschrieben werden muss, geringer wird (im Vergleich zum Assembler Code). Wobei hier, wie auch im C-Beispiel, bedacht werden muss, dass bereits vorhandene Programmteile mit weiteren hunderten oder tausenden Zeilen Code hinzugefügt werden.

```
#include <iostream>
#include <string>
```

Mit den oberen zwei Zeilen werden fertige Klassen eingebunden. «iostream», um die Ein- und Ausgabe des Programms zu handhaben und «string», um Zeichenketten zu verarbeiten. Der Unterschied ist enorm. Während in C eine Zeichenkette noch eine Liste einzelner Zeichen ist, wird in C++ eine Zeichenkette als ein einziges Objekt gehandhabt und bringt viele Funktionen zur Verarbeitung dieser mit.

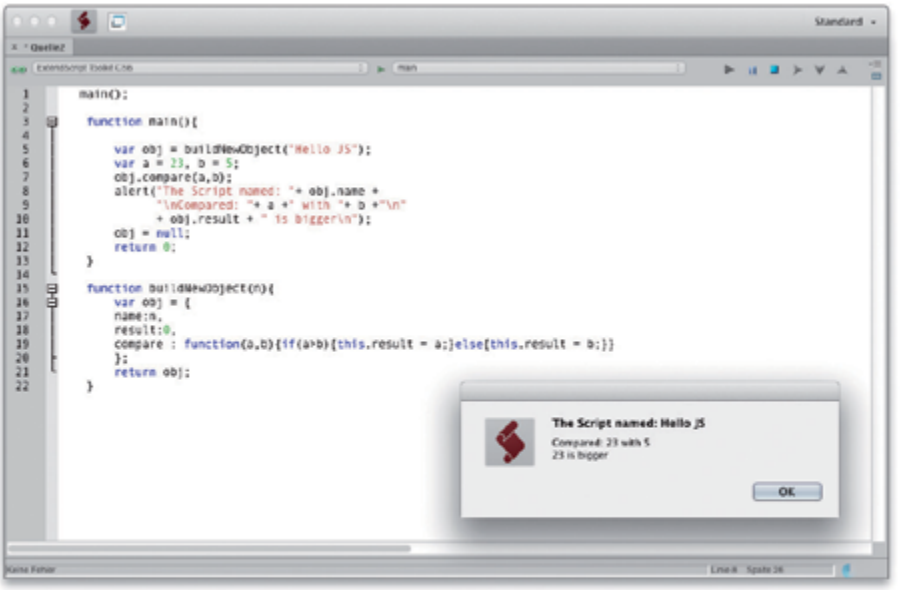


```
// the string and io classes
#include <iostream>
#include <string>
// using namespace std;
// if you use this you can remove all std::
class Object{ /* our object*/
    public:
    std::string name; // its name
    Object(){}// basis constructor
    // another constructor
    Object(std::string in){name = in;}
    //Prototype for comparsion
    int compare(int a, int b);
    // to set the name
    void setName(std::string in){name = in;}
};
/* the compare function */
int Object::compare(int a, int b){
    if(a>b){return a;}else{return b;}
 }
 int main(){ /* now the main program*/
// make a new object
    Object* myObject = new Object("Hello World CPP");
    int a = 23,b = 5;// declare some values
    // now the output directly with comparsion
    std::cout << "The Object named: "<<
    myObject->name << "\nCompared: "
    << a << " with "
    << b << "\n"
    << myObject->compare(a,b)
    <<" is bigger\n"
    << std::endl;
    delete myObject; // remove the object from memory
    return 0; // everything went fine return 0
}
```

JavaScript (anObject.js)

Im Jahre 1995 wurde die Sprache LiveScript zusammen mit der 2.0 Version von Netscape (ein Web-Browser) veröffentlicht und bald in JavaScript umbenannt. Wie in vielen Büchern, die sich mit JavaScript oder Java auseinandersetzt, möchte auch ich hier sagen und es dabei belassen:*Java is to JavaScript like ham to hamster*

Mit JavaScript gehen wir einen Schritt weiter als mit C++. Hier sind das Objekt, die Ein-/Ausgabe, der String bereits existent und müssen weder neu implementiert noch eingebunden werden. Genaugenommen ist in JavaScript fast alles ein Objekt. Der maßgebende Unterschied ist, dass die vorherigen Programme vollwertige Programme sind, die nach dem Kompilieren aus der Kommandozeile ausgeführt werden können. Die JavaScript Variante benötigt ein Programm, in dem sie ausgeführt wird. Es ist alleine nicht lauffähig. Werfen sie einen Blick auf die Funktion `this.compare = function(a,b)` in `buildNewObject(n)`. Dies ist eine kleine Abwandlung der `compare` Funktion, die wir in der C++ Variante sehen. Die verbesserte Funktion kann die gegebenen Variablen vergleichen und in sich selbst, mit `this.result`, das Ergebnis festlegen. Um eine solche Funktionalität in C++ zu erzeugen, bedürfte es einiger Zeilen mehr.

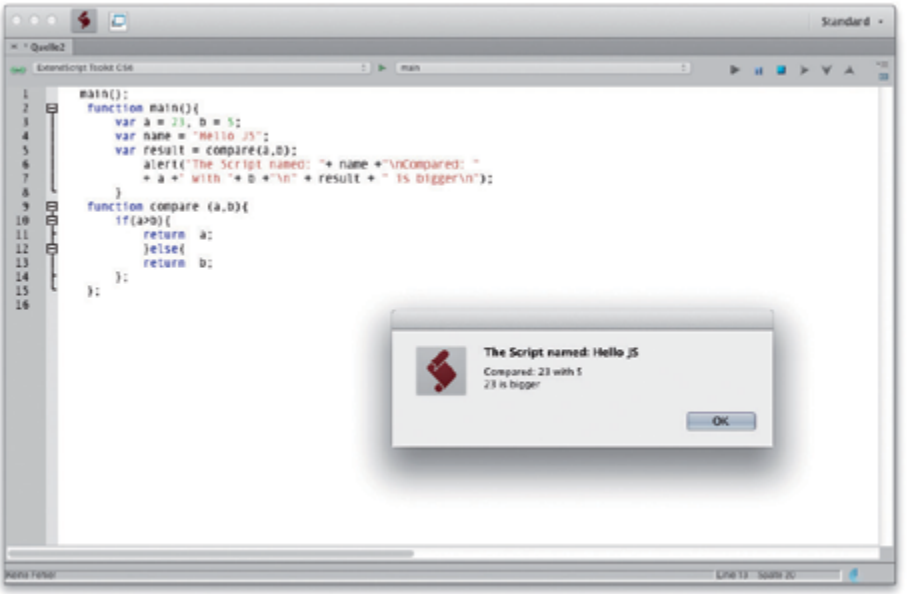


```
main(); // call the main function

function main(){ /* here wee go */
    // create a new Object
    var obj = new buildNewObject("Hello JS");
    var a = 23, b = 5; // define the values
    // let the object compare the values
    obj.compare(a,b);
    // and make the message
    alert("The Script named: "+ obj.name +
          "\nCompared: "+ a +" with "+ b +"\n"
          + obj.result + " is bigger\n");
    obj = null; // delete the object
    return 0; // everything went fine return 0
};

/* this is our object builder*/
function buildNewObject(n){
    /* its name is the incoming value*/
    this.name = n;
    /* this will be set by the compare function*/
    this.result = 0;
    this.compare = function(a,b){
          if(a > b){
                this.result = a;
          }else if(a < b){
                this.result= b;
          }
    };
};
```

Ohne diese Erweiterung und ohne das Objekt würde das JavaScript wieder der C Variante ähneln.

https://gist.github.com/2660661#file_no_object.js

```
main();
function main(){
    var a = 23, b = 5;
    var name = "Hello JS";
    var result = compare(a,b);
    alert("The Script named: "+ name +"\nCompared: "
          + a +" with "+ b +"\n" + result + " is
          bigger\n");
};
function compare (a,b){
    if(a>b){
          return  a;
    }else{
          return  b;
    };
};
```



7.18 Was ist der Unterschied zwischen JS & JSX?

JavaScript, oder EcmaScript, ist das, was uns täglich in Webbrowsern solche Dinge beschert, wie Scrollen auf Knopfdruck, Warnhinweise und ähnliches. ExtendScript ist ein «Dialekt» von JavaScript, der von Adobe entwickelt wurde. Das bedeutet: InDesign versteht JavaScript und ExtendScript, aber ein Browser kann mit ExtendScript Befehlen nichts anfangen.

7.19 Was sind Pointer?

Ein Pointer ist der Verweis auf die Speicheradresse, unter der eine Variable abgelegt ist. Das kann dann so aussehen:

> *«Beispeilsweise besagt:*
>
> ```
> double *dp, atof(char *);
> ```
>
> *daß[sic] in einem Ausdruck **atof()** und ***dp** Werte vom Typ **double** liefern»*[10]
> *aus Programmieren in C von Brian W. Kernighan & Dennis M. Ritchie. Published by Prentice-Hall in 1988*

8 Referenzen

Dies ist eine Liste der Referenzen, die bei Skriptentwicklung genutzt werden können um Befehle nachzuschlagen. Sie müssen sich jedoch etwas einlesen. Zum Beispiel ist im InDesign SDK (Software Development Kit) nicht sofort ersichtlich, dass eine riesige Sammlung an Beispiel Skripten darin enthalten ist. Ebenfalls werden sie sich wundern, dass für After Effects seit Version CS3 kein neuer «Scripting Guide» geschrieben wurde. Leider ist dem so.

IDJS Help von jongware

Diese Seite ist nützlich für InDesign, InDesign Server, InCopy, Illuatrator und Extendscript. Hier ein kurzer Auszug was sie hinter diesem Link erwartet.

> *(...) Unfortunately, starting with InDesign CS3, Adobe chose not to include the JavaScript reference as a PDF, but have users access it through their ExtendScript editor instead. They probably assumed everyone would use that to write and edit scripts. Well, they were wrong. I'm so much more used to plain text editors (my current favourite has been TextPada for a while), which have oodles more handy functions than the ExtendScript one. But if you use another editor, you'll miss out on the help!*
>
> *No sweat. Amongst the literally hundreds of files of InDesign, I found a large one with the rather weird name «omv$indesign-5.0-en_us.xml». Curious as ever, I opened it, and – joy! – it appeared to contain each and every scriptable object, class, name, function and enumeration for JavaScript.(...)*
>
> *(...) The results of all of this work can be downloaded freely from the following links. All of the files are packed as ZIP files; download, save it somewhere, then unpack. Your operating system should be able to do so automatically – if not, search the web for a PkZip compatible unpacker.*
>
> *Note that the Windows Help file format CHM is not usable on Macintosh OS X without additional software.(...)*

http://www.jongware.com/idjshelp.html](http://www.jongware.com/idjshelp.html

Adobe SDK's

Für die unterschiedlichen Adobe Applikationen gibt es verschieden Sets, für die Plugin Entwicklung gedacht sind. Es sind jedoch auch einige «Scripting» Ressourcen enthalten.

AI SDK	http://www.adobe.com/devnet/illustrator/sdk.html
AE SDK	http://www.adobe.com/devnet/aftereffects.html
BR SDK	http://www.adobe.com/devnet/bridge.html
ID SDK	http://www.adobe.com/devnet/indesign.html
PS SDK	http://www.adobe.com/devnet/photoshop.html

Script UI for Dummies

Falls sie sich mit der Entwicklung von grafischen Benutzeroberflächen beschäftigen möchten, ist dieses PDF die ausführlichste Referenz, die sie finden können.

Script UI http://www.kahrel.plus.com/indesign/scriptui.html

Javascript

Dies ist eine umfassende Referenz aller JavaScript Befehle.

w3schools http://www.w3schools.com/js/

InDesign mit JavaScript automatisieren

Dieses Buch behandelt die CS2 und CS3 Versionen von InDesign, ist jedoch ein grossartiger Einstieg in JavaScript für InDesign.

> «*In jedem Layoutprozess fallen routinemäßige Aufgaben an, die sich zwar durch Handarbeit lösen lassen, aber in der Ausführung zeit- und nervenaufreibend sind. Skripten können viele dieser Aufgaben übernehmen, doch wie stellt man es an, Aufgaben zentral zu bewerkstelligen, wenn man noch nie geskriptet hat? Dieses TecFeed gibt Ihnen sowohl das Wissen als auch die Skripten an die Hand, mit denen Sie den Layoutprozess in InDesign optimieren können.(...)*»[11]

InDesign mit JavaScript automatisieren von Peter Kahrel - Deutsche Übersetzung von Martin Fischer 1. Auflage August 2007
ISBN 978-3-89721-624-2

9 Weblinks

Dies sind nützlichen Weblinks, die sich über die Jahre bei mir angesammelt haben.

Foren

AE aenhancers
- http://www.aenhancers.com/

AE Adobe After Effects Scripting
- http://forums.adobe.com/community/aftereffects_general_discussion/ae_scripting

ID Hilf Dir Selbst
- http://www.hilfdirselbst.ch/foren/Adobe_InDesign_Skriptwerkstatt_Forum_61.html

ID Adobe InDesign Scripting
- http://forums.adobe.com/community/indesign/indesign_scripting

PS ps-scripts
- http://www.ps-scripts.com/bb/

Script Sammlungen

AE aescripts
- http://aescripts.com

AE aenhancers - General Scripts Library
- http://www.aenhancers.com/viewforum.php?f=3&sid=1c6eb50d9f703a54e425be7c30f27617

AE crgreen
- http://www.crgreen.com/aescripts/

AE redefinery
- http://www.redefinery.com/ae/

AI johnwun/js4ai
- https://github.com/johnwun/js4ai

ID indesignscript
- http://www.indesignscript.de/

ID indesignsecrets
- http://indesignsecrets.com/category/secrets/pluginsscripts

ID kahrel.plus
- http://www.kahrel.plus.com/indesignscripts.html

ID Scripting Guide Scripts
- http://www.adobe.com/devnet/indesign/documentation.html#idscripting

ID fachhefte
- http://www.fachhefte.ch/

Tutorial Sammlungen

AE motionscript
- http://www.motionscript.com

AE redefinery
- http://www.redefinery.com/ae/fundamentals/

AI js4ai
- http://js4ai.blogspot.de/

ID indesignscript
- http://www.indesignscript.de/

ID indiscripts
- http://www.indiscripts.com/

ID indisnip
- http://indisnip.wordpress.com/tag/javascript/

ID JavaScripting InDesign
- http://jsid.blogspot.de/

ID Scripting Guide Scripts
- http://www.adobe.com/devnet/indesign/documentation.html#idscripting

JS codecademy
- http://www.codecademy.com/#!/exercises/0

Verschiedenes

ID idgrephelp
- http://www.jongware.com/idgrephelp.html

JSX boethos - script panel creation
- http://www.crgreen.com/boethos/

10 Quellen

Primär

1. Ousterhout, John, «Scripting: Higher-Level Programming for the 21st Century,» IEEE Computer, Vol. 31, No. 3, März 1998, pp. 23-30 / Tcl Developer Xchange, http://www.tcl.tk/doc/scripting.html (August 8, 2000)

2. Vgl.: Kahrel, Peter, Script for «Entering characters with diacritics and other special sorts», http://www.kahrel.plus.com/indesign/compose_cs3.jsx (Oktober 2011)

3. Vgl.: Minute 3, Sekunden 30, Thorp, Jer, «Make data more human» , TEDx-Vancouver November 2011, http://www.ted.com/talks/lang/en/jer_thorp_make_data_more_human.html (März 2011)

4. Vgl.: Zimmerman Jones, Andrew, «What Are Clarke's Laws?» http://physics.about.com/od/physics101thebasics/f/ClarkesLaws.htm (2012)

5. Seite 520f, McLuhan, Marshall, Deutsch Übersetzung: Amann, Meinrad, «Die magischen Kanäle - Understanding Media»,(Originalausgabe 1964) Basel 1995, Verlag der Kunst Dresden

6. Duden online, «Programm, das» http://www.duden.de/rechtschreibung/Programm#Bedeutung1c (2012)

7. Vgl.: Mozilla Developer Network, «Reserved Words» https://developer.mozilla.org/en/JavaScript/Reference/Reserved_Words (2012)

8. Tilman, «Die Programmiersprache Brainfuck» http://tilmanb.junetz.de/brainfuck.php4 (April 2002)

9. Vgl.: Seite 396f, Reas, Casey & Fry, Ben, «Processing: a programming handbook for visual designers and artists», Massachusetts 2007, MIT Press

10. 10, Seite 98, Ritchie, Dennis & Kernighan, Brain, Deutsche Übersetzung: Dr. Schreiner, A.T. & DR. Janich, Ernst «Programmieren in C», München 1983, Carl Hanser Verlag

11. Klappentext, Kahrel, Peter, Deutsche Übersetzung: Fischer, Martin, «InDesign mit JavaScript automatisieren», Köln 2008, O'Reilly Verlag

Sekundär - Bücher

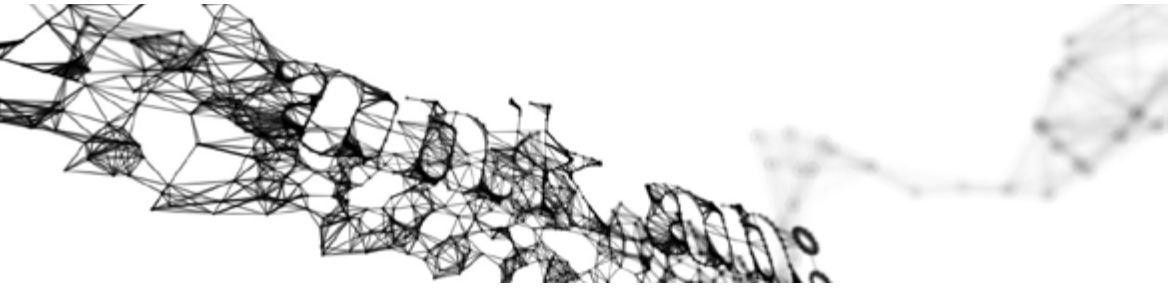
- Lazzeroni, Claudius (Hrsg.), Bohnacker, Hartmut & Groß, Benedikt & Laub, Julia «Generative Gestaltung Entwerfen Programmieren Visualisieren» Mainz 2009, Verlag Hermann Schmidt Mainz
- Maeda, John, «Creative Code, Ästhetik und Programmierung am MIT Media Lab», Basel 2004, Birkhäuser
- Noble, Joshua, «Programming Interactivity», Sebastopol 2009, O'Reilly Media, Inc.
- Reas, Casey & McWilliams, Chandler & LUST, «Form + Code, in design, art and architecture», New York 2010, Princeton Architectural Press
- Stroustrup, Bjarne, «The C++ Programming Language 3rd Edition», New Jersey 1997, Addison-Wesley Publishing Company
- Ullenboom, Christian, «Java ist auch eine Insel», Bonn 2009, Galileo Press
- Wenz, Christian, «JavaScript Das umfassende Handbuch», Bonn 2009, Galileo Press

Sekundär - Internet

- Beck, Ian, «Learning to code» http://beckism.com/2012/04/learning-to-code/ (April 2012)

Impressum

Korrektur : Sebastian P. & M., Steven N., Dr. Dagoberto M. L.
Webdeployment : Jekyll, Twitter-Bootstrap & Github.com
Fonts : Calluna & `DejaVu Sans Mono Book`
Druck und Herstellung : Schaltungsdienst Lange oHG, Berlin
Papier : Munken Pure Alster
Redaktionelle Bearbeitung & Satz & Layout : Fabian Morón Zirfas



Frau Müller und Herrn Krohn für die Betreuung, Sebastian P. für die stundenlange Telefonkorrektur und mein Freund sein, Sebastian M. fürs korrigieren und warten bis ich fertig bin, Steven für die Revision des Quellcodes und immer schnellen Hilfen bei Programmierproblemen, Dago fürs korrigieren und Fabian machen, Heidrun fürs Fabian machen, Rafa fürs ewige aufpassen, ich weiss es war nicht leicht, Pete, dafür dass du Pete bist, Sebastian Z. fürs hier sein, Mona fürs mögen meiner Arbeit auch wenn du es oft nicht verstehst aber bereit bist das zuzugeben, Tom fürs auf Mona aufpassen und mich großer nennen obwohl du größer bist, Oliver für die langen Diskussionen warum er skripten lernen soll und erklären warum er es doch nicht macht, Johan fürs Freund bleiben, das Netz im Ganzen für die Millionen Zeilen Code, die Open Source Coder, ohne euch würde das alles nicht gehen, die Codefee, die über nacht meine Probleme lösst, Tom Preston Werner für das grossartige Jekyll, das ich nicht verstehe aber trotzdem gern benutze, das Twitter-Bootstrap Team für das tolle Tool, Github.com für den Platz in der Cloud, die AE Community für das grossartige Feedback zu AEMap.jsx, Lloyd Alvarez fürs kontaktieren und an alle die ich vergessen habe.

Super extra spezial Dank mit Sahne an alle die mich mit Zuspruch zum weitermachen animieren.