Entwickler werden

-

Ratgeber für Einsteiger

(c) Jan Brinkmann - codingtutor.de
info@codingtutor.de

Wer bin ich?

Ich bin seit über 15 Jahren Software-Entwickler. Meine Schwerpunkte sind die App- und Webentwicklung. Eine besondere Leidenschaft verbindet mich mit der OpenSource-Community. Neben der beruflichen Tätigkeit als Entwickler betreibe ich seit 2011 die Webseite codingtutor.de, über die ich Einsteiger an die Software-Entwicklung heranführe.

Außerdem habe ich bereits für mehrere deutsche Fachzeitschriften Artikel verfasst. Dazu gehören das PHPMagazin, das Entwickler Magazin sowie das Linux Magazin.

Noch mehr?

Weitere Tipps, Trainings, Screencasts und Artikel findest Du auf codingtutor.de!

Noch Fragen?

Ich bin jederzeit für Dich da: info@codingtutor.de

Impressum


```
Jan Brinkmann
Alte-Rothe-Str. 40a
33189 Schlangen
codingtutor.de | info@codingtutor.de
```

Software Entwicklung ist keine Raketenwissenschaft. Jeder gesunde Mensch kann es Lernen. Die wichtigsten Fertigkeiten, die Du mitbringen solltest: Lesen, Schreiben und sicherer Umgang mit Grundrechenarten. Ansonsten hängt viel von Deinem Willen und Fleiss ab. Wunderkinder, die mit 18 Jahren bereits eigene Betriebssysteme schreiben, gelten oft als Stereotyp für Programmierer. In der Realität sind sie allerdings die Ausnahme. Und genau das ist einer der Gründe für dieses Buch. Ich will Dir Mut machen einfach anzufangen und auszuprobieren. Du sollst lernen an Dich zu glauben und Dich nicht verunsichern lassen.

Gleichzeitig ist dieses Buch ein Leitfaden für Einsteiger. Es gibt Bücher nach dem Schema “Programmieren lernen mit …”. Die haben aber alle eins gemeinsam: Sie betrachten die jeweilige Sprache als die beste Wahl. Und es gibt - ohne Recherche und Analyse - mindestens 15 bis 20 solcher Bücher. Die Lücke am Markt ist eine Übersicht, die eine Richtung vorgeben kann. Genau die möchte ich schließen.

Fragst Du Dich vielleicht welche Sprache Du wählen sollst? Welche für Einsteiger die richtige ist? An welcher Stelle steigst Du in die Programmierung ein? In welche Richtung kannst Du gehen? Welcher Bildungsweg ist optimal? Ich betreibe seit 2011 das Blog codingtutor.de und begegne dort häufig Lesern mit ähnlichen Gedanken. Ein ganz typisches Phänomen: Die Frage nach dem Anfang. Die Antworten auf solche und andere Fragen findest Du in diesem Buch.

Außerdem gibt es neben viel Hintergrundwissen auch ein

paar technische Details. Im Kapitel “Einblick in die Software Entwicklung” stelle ich Dir mit der Sprache JavaScript die Software-Entwicklung aus der Vogelperspektive vor. Du lernst Elemente und Konzepte kennen, die in heutigen Sprachen zu finden sind.

Und jetzt wünsche ich Dir viel Spaß beim Lesen und viel Erfolg auf Deinem weiteren Weg!

Dieses Buch ist ein Leitfaden für Entwickler. Ich möchte Dir zeigen, woraus es ankommt, in welche Richtung Du gehen kannst und wie Du den Einstieg findest.

Und danach?

Ich möchte Dich auf dem Weg in die Software-Entwicklung begleiten. Dieses Buch ist eine Orientierung, ein Wegweiser. Viele weitere Inhalte rund um die Programmierung findest Du auf meiner Webseite, codingtutor.de.

Warum solltest Du überhaupt Programmieren lernen? Selbst wenn der Entschluss feststeht noch eine berechtigte Frage! Ist es wichtig? Sollte es jeder können? Sollten vielleicht alle Menschen Grundkenntnisse haben? Auch jene, die gar nicht den Anspruch haben in “der IT” zu arbeiten? Aktuell wird sogar der Ruf nach Software-Entwicklung im Schulunterricht immer lauter. Vielleicht zweifelst Du an Sinn und Zweck? Lies weiter!

Programmieren weil es Spaß macht!

Du hast Deinen Weg zu diesem Buch gefunden. Ich werte das als Zeichen von Neugier! Du bringst also ein grundlegendes Interesse für Software Entwicklung mit? Viel besser geht es nicht! Wenn Dir etwas Spaß macht oder Dein Interesse weckt - halte daran fest! Egal was es ist. Solange Du damit weder Dich oder andere gefährdest, kann es nur gut sein. Wenn es zudem noch so wenig Geld kostet spricht gar nichts dagegen, oder? Die einzige nennenswerte Investition ist Zeit. Aus dieser Perspektive betrachtet dreh ich die Frage provokant um: Warum solltest Du nicht Programmieren lernen?

Warum solltest Du Programmieren lernen?

Ok. Formal betrachtet gibt es auch logische Argumente. Spaß haben und dabei auch noch etwas sinnvolles machen? Ganz sicher möglich! Die Gründe warum Menschen Programmierer werden sind sehr unterschiedlich. Jeder hat eine andere Motivation. Hier ein paar Motive zur Auswahl:

- Software Technologie ist wirklich faszinierend

- Software Entwicklung bietet viele Facetten
- eine Herausforderung eigener Fähigkeiten
- gute Programmierer werden händeringend gesucht
- Du möchtest eigene Ideen verwirklichen
- eigene Webseiten oder Apps erstellen
- auf lange Sicht gute Chancen auf dem Arbeitsmarkt
- Im Studium mit Software Entwicklung in Kontakt gekommen

Viele Entwickler die ich kenne haben einfach Ihr Hobby zum Beruf gemacht. Mit Deiner Leidenschaft Geld verdienen, viel besser geht es nicht.

Hobby zum Beruf. Aus Spaß wird Ernst

Es gibt tatsächlich eine weitere Sichtweise. Wenn Du Dein Hobby zum Beruf machst, verlierst Du in gewisser Hinsicht eine Freizeitbeschäftigung. Zumindest ist Programmieren ein schlechter Ausgleich, wenn Du es schon den ganzen Tag im Büro machst. Verlierst Du dadurch über kurz oder lang die Leidenschaft? Ich glaube nicht. Ich bin seit über 15 Jahren Entwickler. Bis heute ist die Leidenschaft vorhanden. Die Schmetterlinge im Bauch sind natürlich verschwunden. Aber es macht immer noch diesen großen Spaß. Wenn Du zum Beispiel den ganzen Tag Java-Anwendungen entwickelst, willst Du sicher Abends nicht genau dort weitermachen. Aber es gibt

endlos viele Bereiche, in denen Du Dich zusätzlich austoben kannst.

Ich kenne viele Entwickler die ganz bewusst in der Freizeit OpenSource-Projekte unterstützen. Dabei entwickeln sie - entweder in Eigenregie oder im Team - Code für quelloffene Projekte. Das ist eine völlig andere Welt. Du hast keine direkte Deadline und keinerlei Stress. Es gibt keinen Projektleiter der Dir auf die Füße tritt. Noch besser: Du musst keine zeitlichen Vorgaben einhalten, Dich möglichst genau an Vorgaben des Kunden halten oder ansonsten Ausreißer in der Zeiterfassung erklären. Du kannst Deine eigenen Ideen umsetzen und musst auch nicht Sinn und Unsinn bei Kundenprojekten hinterfragen. Du arbeitest an genau den Themen die Dir gefallen. Vergleich es mit einem Maler. Der kann sich in seiner Freizeit künstlerisch auslassen. Im Beruf ist das nicht immer gewünscht oder wirtschaftlich sinnvoll. Also: Die Leidenschaft für Software geht durch den Beruf sicher nicht verloren. Die Prioritäten verschieben sich einfach.

Die richtige Erwartungshaltung

Der Ursprung des Gedankens “Ich werde Programmierer” ist fast nebensächlich. Meist ist es ohnehin eine Kombination verschiedener Gründe, die den Ausschlag gibt. Zum einen ist es wichtig nichts unversucht als unmöglich abzuschreiben. Du wirst überrascht sein, was Du Dir selbst beibringen kannst. Lass Dich nicht einschüchtern und probier alles einfach aus. Du bereust im Nachhinein eher Dinge, die Du unversucht lässt. Das kann ich aus Erfahrung sagen. Mein Weg verlief sicher

nicht "gerade aus". Es war die ein oder andere "ungünstige Entscheidung" dabei, ich bin über Umwege ans Ziel gekommen und hab mehr als nur einen Rückschlag hinnehmen müssen. Positiv formuliert würde ich sagen, ich bin oft nicht den einfachen Weg gegangen. Unbewusst und sicher nicht mit Absicht. Ändern würde ich insgesamt aber nichts. Ich hätte nur gern mehr Mut gehabt und mehr ausprobiert.

Du solltest außerdem keine Wunder erwarten. Den Mythos vom schnellen Geld wirst Du als Entwickler sicher nicht finden, sei es in Form von Apps, Webseiten oder als Freelancer. Das führt früher oder später zu Ernüchterung oder Enttäuschung. Erfolg kommt zudem nicht über Nacht. Programmieren lernen ist nicht schwer. Die Thematik ist aber umfangreich. Daher ist mit Fleiss vieles möglich, nur eben nicht von heute auf morgen.

Programmieren ist eine Frage der Übung

Du kannst heute schnell erste Ergebnisse erzielen - dank umfangreicher Dokumentationen, Videotrainings, einer Vielzahl an Büchern und ausgereiften Programmiersprachen. Das wirkt sich positiv auf die Motivation aus. Die Grundlagen sind natürlich weiterhin wichtig. Aber schnell greifbare Ergebnisse - zum Beispiel in Form von Apps oder Webseiten - sind gerade am Anfang wirklich eine große Unterstützung. Es macht regelrecht süchtig.

Um ein erfahrener Programmierer zu werden bedarf es Praxis und Übung. Das bedeutet keineswegs, dass Du erst eine Unmenge an Software schreiben musst bevor

Du Programme entwickeln kannst. Das geht sehr viel schneller. Du gewinnst aber nach und nach an Erfahrung, lernst andere Blickwinkel kennen und findest auch neue Lösungswege für verschiedene Anforderungen.

Irgendwann kannst Du aus dem Gefühl heraus eine passende Lösung auswählen. Dafür musst Du aber die verschiedenen Lösungswege erst mal kennenlernen. Es wird unweigerlich passieren, dass Du Software schreibst die Du in Zukunft ganz anders aufbauen würdest. Das ist nicht schlimm und kein Fehler. Viel Schlimmer wäre es aus Angst vor einer Fehlentscheidung am Ende gar nichts zu machen. Geh nach bestem Wissen und Gewissen vor und halte Dich an die Erfolgsregel von Arnold Schwarzenegger: *Don't be afraid to fail*.

Einfach ausprobieren

Es gibt ein sehr bekanntes Sprichwort das besagt "Probieren geht über Studieren". Und das trifft es auf den Kopf. Der einfachste Weg um herauszufinden “Warum will ich Programmierer werden?” ist es die Gründe beim Lernen zu erforschen. Wie sagt man so schön? Der Appettit kommt beim Essen! Ich kenne Entwickler, die sich einfach für Computer interessiert haben. Erst durch den zufälligen Kontakt mit der Programmierung haben Sie sich damit identifiziert. Stempel Dich nicht zu früh selbst ab!

Und wenn Du es nicht sonderlich spannend oder interessant findest ist das gar nicht schlimm. Du verlierst ja nichts. Du

hast einfach einen kleinen Einblick in die Welt der Computer und Software gewagt, über den Tellerrand geschaut. Und das ist nun wirklich niemals ein Fehler.

Auch wenn sich dies nach einer langen und schwierigen Reise anhört, soll Dich das nicht entmutigen. Um auf den eigentlichen Gedanken zurückzukommen: Das Warum spielt aus meiner Sicht eine untergeordnete Rolle. Wichtig ist viel mehr, dass Du nicht erwartest innerhalb von 4 Wochen das nächste Facebook umzusetzen. Auch das erste Buch oder der erste Videokurs machen Dich nicht zum erfahrenen Programmierer. Aber: Es ist ein Anfang.

Selbst die längste Reise beginnt mit dem ersten Schritt. Einfach anfangen, Erfahrungen sammeln, Software schreiben, lernen, alternative Lösungswege kennenlernen, an Dir arbeiten. So hat jeder angefangen.

Wenn Du neu in der Welt der Software-Entwicklung bist, wirst Du von der großen Anzahl der Möglichkeiten, den unzähligen Buzzwords und den vielen Programmiersprachen regelrecht überrollt. Das kann einschüchtern - nicht nur Einsteiger! Kann ich wirklich Programmieren lernen? Die Informationsflut kann dafür sorgen, dass Du ernsthaft daran zweifelst. So viel Material zum Lernen, so viele Themen! Aber ich versichere Dir: Wenn Du grundlegend gesund bist, lesen und schreiben und mit Grundrechenarten umgehen kannst: Du schaffst es! Du brauchst Geduld und darfst Dich nicht entmutigen lassen. Fast noch wichtiger: Vergleich Dich nicht mit anderen.

Das Recht auf Deine Vergangenheit

In vielen Bereichen des Lebens wollen wir Menschen uns mit anderen vergleichen. Du willst ein Unternehmen gründen und es klappt nicht? Du wirst immer Gründer finden die Erfolg scheinbar mit Löffeln zu sich nehmen. Als Läufer findest Du bei jedem Wettkampf schnellere Teilnehmer. Du gehst ins Fitnessstudio? Da findest Du auch immer jemanden mit mehr Muskelmasse, einem kleinere Körperfettanteil oder höherem persönlichen Rekord (Gewicht) bei einer Übung Deiner Wahl. Und gleiches gilt auch beim Programmieren. Du musst nur, um für Dich das beste herauszuholen: Deinen Weg gehen. Was ich in den letzten Jahren beobachtet und auch in der Literatur wiedergefunden habe: Konzentrier Dich auf eine Sache und mach “Dein Ding”. Versuch nicht alles gleichzeitig. Schau Dich um. Erfolgreiche Fußballer? Klarer Fokus auf Ihren Sport.

Bestseller-Autoren wie Stephen King? Schreibt jeden Tag! Auch Rennfahrer, erfolgreiche Ärzte oder Blogger. Erfolgreiche Menschen konzentrieren sich auf ihre Sache.

Natürlich sollst Du Vorbilder haben. Du darfst nur Dich und Deine Ergebnisse nicht direkt mit anderen vergleichen. Jeder hat seine eigene Vergangenheit. Die total erfolgreichen Unternehmer? Vielleicht arbeiten Sie einfach schon länger an Ihrem Erfolg, haben mehr ausprobiert. Du siehst die ganzen Fehlversuche nicht. Thomas Edison hat die Glühbirne erfunden! Warum schaffe ich so etwas nicht? Bis zum ersten Prototyp brauchte er aber fast 2000 Versuche! Der Läufer mit dem Du Dich vergleichst? Der war vielleicht als Jugendlicher schon Sprinter und wiegt 15 Kilo weniger als Du. Vielleicht hat er auch als Sportstudent mehr Gelegenheit für das Training. Und die scheinbar erfolgreicheren Menschen im Fitnessstudio? Hatten am Anfang sicher ähnliche Gedanken wie Du jetzt - nur halt schon vor mehreren Jahren. Sie haben einfach einen großen Vorsprung. Jeder hat also eine andere Geschichte rund um den IST-Zustand zu berichten.

Vergleich Dich nicht mit den Linus Torvalds (Schöpfer von Linux), Dennis Ritchies (Vater von C und Unix) oder Bjarne Stroustrups (Erfinder von C++) dieser Welt. Eine wichtige Lektion, die ich durch den Sport und das Laufen gelernt habe: Dein einziger Gegner bist Du selbst. Versuch besser zu sein als Du selbst, besser als gestern oder besser als beim letzten Projekt. Klar, das sagt sich jetzt so leicht. Mir ist bewusst wie toll sich das anhört. Das logische Verständnis dafür hast Du

vermutlich sowieso. Das ändert am Gefühl herzlich wenig. Je öfter Du Dich aber bei solchen Vergleichen selbst erwischt und selbst korrigierst, desto eher wird das zu Deiner automatischen Reaktion.

Wirklich jeder kann!

Erfolg ist in jedem Bereich des Lebens viel weniger Zufall oder Resultat von angeborenem Talent. Die besten und erfolgreichsten Menschen der Welt haben eine ganz einfache Wahrheit verstanden: Du musst üben, üben und noch mal üben. Um wirklich ein Meister auf Deinem Fachgebiet zu werden, musst Du immer und immer weiter üben. Das soll nicht abschrecken. Du kannst auch ohne zur Elite zu gehören viel Spaß haben, erfolgreich sein und Software entwickeln.

Nutze die Talente die Du hast. Die Wälder wären still, wenn nur die begabtesten Vögel sängen. - Henry van Dyke.

Das Zitat drückt es besser aus als ich es könnte. Ich hoffe es macht Dir ein bisschen Mut und gibt Dir das Vertrauen in Dich selbst. Denn sei Dir sicher: Programmierer sind in den meisten Fällen keine übertalentierten Menschen. Auch magische Fähigkeiten spreche ich 99,9% davon ungesehen ab. Am Ende des Tages ist es eine Fertigkeit, die Du mit Fleiss, Ausdauer und den richtigen Gewohnheiten lernen kannst.

Komplexität als Chance

Die enorme Vielfalt und die zahlreichen Facetten schrecken Dich vielleicht ab. Dabei sind sie für Entwickler enorm positiv.

Durch die steigende Komplexität steigen auch Deine Chancen am Markt. Niemand beherrscht alle Programmiersprachen. Selbst innerhalb der Sprachen gibt es umfangreiche Frameworks, Anwendungen oder Fachbereiche. Bis im Detail kann ein einzelner Mensch die nicht alle kennen. Der Trend geht deswegen immer mehr zur Konzentration auf konkrete Nischen. Ein greifbares Beispiel sind Content Management Systeme. Während vor Jahren PHP-Entwickler noch die wenigen vorhandenen Systeme (mehr oder minder) beherrscht haben, gibt es heute Fachleute für WordPress, Magento, Typo3, Zend Framework, Contao und so weiter. Die mitgelieferten Bibliotheken und Klassen sind umfangreich. Sehr umfangreich. Und das ist im Moment erst der Anfang.

Selbst innerhalb der einzelnen Content Management Systeme gibt es Spielraum für spezielle Nischen. So gibt es bspw. Magento-Programmierer, die sich rein auf die Themes spezialisiert haben. Andere Entwickler kümmern sich um Erweiterungen der Shoplogik. Und dieser Trend wird nicht aufhören. Im Gegenteil. Die angebotenen Lösungen werden immer komplexer. Die Spezialisierung wird eher zunehmen. Da stehen Deine Chancen den passenden Fachbereich zu finden mehr als gut. Sicher geschieht es nicht über Nacht. Du musst erst mal ausloten wohin die Reise gehen soll. Du kannst aber stetig darauf hinarbeiten und neue Dinge ausprobieren. Egal was das Ziel auch sein mag: Selbst der längste Weg beginnt mit dem ersten Schritt. Es gibt leider keine genaue Karte, keine Wegweiser. Mit diesem Buch möchte ich Dir trotzdem helfen Dich zurecht zu finden. Egal wohin es geht. Aufbrechen lohnt

sich in jedem Fall! Wenn Du den Pazifik überqueren willst, musst Du auch erst mal vom Strand weg. Also warte nicht zu lange. Rein in das kalte Wasser!

Eine Frage der Motivation

Die Frage “Kann ich Programmieren lernen?” kannst Du auch anders stellen: “Wie sehr will ich wirklich Programmieren lernen?”. Sicher bedarf es einer grundlegenden Befähigung logisch und abstrakt zu denken. Für den Erfolg halte ich aber andere Fertigkeiten für wichtiger. Du musst Dich selbst motivieren können und eigenständig arbeiten. Jeden Tag auf’s Neue. Wenn Programmieren für Dich einfach “nice to have” ist, wird Dir der notwendige “lange Atem” fehlen. Auch rein logische Gründe wie Zukunftssicherheit und berufliche Chancen reichen nicht aus, um über lange Zeit am Ball zu bleiben.

Bei der Programmierung fallen Dir schon nach kurzer Zeit viele Dinge leichter. Und sobald die Grundlagen sitzen, baust Du schnell Momentum auf. Die Lernerfolge kommen, und somit Deine ersten Projekte ins Rollen. Die große Kunst ist früher oder später zu vielen Dingen “Nein” zu sagen. Es wird immer Ideen, Programmiersprachen und Einfälle geben, die Dich von Deinem derzeitigen Projekt ablenken. Und genau diesen Zustand musst Du aushalten können. Darauf vertrauen, dass Du gerade an der richtigen Idee festhältst.

Fazit

Kannst Du Programmieren lernen? Ich bin mir ganz sicher! Außerdem glaube ich an Dich. Du auch?

Wie lange dauert es bis Du eine Programmiersprache beherrscht? Das hängt von verschiedenen Faktoren ab. Es gibt keine allgemeine Zeitangabe. Jeder Mensch lernt unterschiedlich schnell. Auch Vorkenntnisse spielen eine Rolle. Eine zweite oder dritte Sprache lernst Du automatisch schneller als die erste Programmiersprache. Zudem ist auch jede Sprache unterschiedlich komplex. So ist es vermeintlich einfacher PHP zu lernen als in die Assembler-Entwicklung einzusteigen. Dynamische Webseiten entwickeln sich leichter als hardwarenahe Programme - zumal Assembler immer viel Kenntnisse über die verwendete Hardware erfordert.

Trotz der vielen Unbekannten versuche ich Dir einen möglichst genauen Ansatz zu liefern.

Wann sagst Du “Ich kann Programmieren”?

Ich kann Programmieren – Wann sagst Du das überhaupt? Oder: Wie definierst Du diesen Zustand? Ich verbinde mit der Aussage vielleicht etwas ganz anderes als Du. Ich finde es nicht so wichtig alle Aspekte, Funktionen, Methoden oder Klassen einer Programmiersprache auswendig zu kennen. Das Wissen könntest Du Dir aneignen, klar. Aber darum geht es bei der Programmierung gar nicht.

In der Praxis hast Du von Faktenwissen nicht sonderlich viel. Die Anwendung deutlich schwerer als in der Theorie. Lies ein ganzes Buch - oder gern mehrere. Allerdings ohne den Code jemals selbst auszuprobieren. Anschließend öffnest Du einen Editor und legst los. Schon merkst Du deutlich, das Wissen und

Können zwei verschiedene Paar Schuhe sind. Außerdem gibt es nicht ohne Grund zu jeder Programmiersprache Referenzen. In denen kannst Du einzelne Elemente nachschauen. Einziger Knackpunkt: Du musst wissen wo Du suchen musst.

Beim Lernen einer Sprache empfehle ich, Dich nicht zu sehr in Details zu verlieren. Leg stattdessen den Fokus auf die folgenden Stufen:

Syntax

Im ersten Schritt musst Du Dich mit der Syntax vertraut machen. Es ist wichtig durch Praxis mit der Sprache “warm zu werden”. Kleine Testprogramme, deren Funktion eher im Hintergrund steht, sind der optimale Einstieg. Anfangs ist es nicht unbedingt leicht. Du wirst oft etwas nachschlagen. Aber genau darum heißt es ja "Programmieren lernen" und nicht "Programmieren können".

Die Grundlagen gehen in der Regel schnell “in Fleisch und Blut über”. Das Wissen wandert ins implizite Gedächtnis. Das gleiche passiert zum Beispiel beim Binden der Schuhe, Schalten beim Auto fahren oder dem Schreiben mit der Hand – Du denkst nicht über jeder Schritt bewusst nach. Du handelst einfach. Genauso denkst Du nicht mehr nach wo eine Klammer gesetzt werden muss. Du weißt es irgendwann einfach.

Die Basics

Der zweite Schritt besteht darin die Basics zu verstehen. Das sind Dinge, die eigentlich in jeder Programmiersprache auftauchen. Du musst lernen wie Du etwas ausdrücken kannst.

Wie schreibst Du Zeichenketten, Arrays und ähnliches? Welche Kontrollstrukturen gibt es? Wie benutzt Du Schleifen? Wie definierst Du eigene Funktionen, Klassen oder Methoden? Wie erzeugst Du Objekte? Gibt es besondere Datenstrukturen (wie z.B. Tuples oder Dictionaries bei Swift)? Und so weiter. All dies ist wichtig, um nicht ständig benötigte Elemente in der Dokumentation nachzuschlagen. Sobald Du mit diesen Grundlagen sicherer umgehen kannst, wirst Du automatisch zuversichtlicher. Und gerade auf dieser Stufe gilt: Je mehr Kenntnisse Du schon hast, desto schneller geht es. Du kannst bei der zweiten oder dritten Sprache vieles mit vorhandenem Wissen in Beziehung setzen. Das beschleunigt ungemein.

Features kennenlernen

Erst im dritten Schritt begibst Du Dich auf die Reise durch die Bibliothek. Welche Features werden mitgeliefert? Welche Klassen und Methoden gibt es zu entdecken? Ein erster Überblick über die Standardbibliothek ist wichtig. Du lernst was die Sprache mit Bordmitteln lösen kann, wo externe Bibliotheken notwendig sind und wann komplett eigene Logik gefragt ist. Auswendig lernen bringt nicht viel. Erste Erfahrungen sammeln, experimentieren und herausfinden wo Du Ernstfall nachschauen kannst schon.

Am besten geht das – wer hätte es gedacht – in dem Du Software schreibst. Benutz mitgelieferte Features, erstell Testprogramm und experimentier damit. Es gibt zum Beispiel in vielen Sprachen mitgelieferte Lösungen für den Datenbankzugriff, Input/Output auf Dateiebene und viele

weitere Aufgaben.

Dieser Schritt macht in der Regel am meisten Spaß. Du kannst ungezwungen Deinem Spieltrieb nachgehen, hast aber bereits ein Gefühl für die Sprache entwickelt.

Best-Practices und Software-Architektur

Im vierten Schritt machst Du Dich mit Best-Practices und dem Entwurf vertraut. Gute Programmierung und sauberer Code entsteht vor allem wenn eine gute Architektur entworfen wird. Dies ist der eigentliche Knackpunkt der die Spreu vom Weizen trennt. Und genau dafür musst Du nicht nur Syntax und Sprachelemente kennen. In diesem Schritt rückt die Strukturierung der Software in den Vordergrund. Ein optimaler Ausgangspunkt sind die Design Patterns, die sogenannten Entwurfsmuster. Sie bieten sprachübergreifend Lösungen für verschiedene Problemstellungen. Sie können in nahezu jeder Sprache eingesetzt werden. Das hat mehrere Vorteile. Du nutzt Techniken die sich bewährt haben. Gleichzeitig können andere Entwickler den Code besser verstehen. Es handelt sich um standardisierte Musterlösungen.

Wie lange dauert es nun?

Wann genau Du das Gefühl hast eine Programmiersprache zu beherrschen, beantwortest Du individuell. Ich persönlich finde, dass man zumindest behaupten kann mit einer Programmiersprache umgehen zu können, wenn sich bei der Arbeit mit ihr sicher fühlt. Von beherrschen würde ich sprechen, wenn Du nur noch Referenzen lesen muss und mit

den angegebenen Informationen selbstständig arbeiten kannst (ohne Beispielcode). Die folgenden Faktoren solltest Du bei einer zeitlichen Einschätzung berücksichtigen.

Welche Vorkenntnisse hast Du?

Ganz entscheidenden Einfluss auf Dauer und Lernerfolg hat Dein vorhandenes Wissen. Wenn Du bereits andere Programmiersprachen beherrscht, lernst Du eine neue Sprache deutlich schneller. Du kannst vieles mit Erfahrungen in Verbindung bringen und Dich schneller mit dem Wissen "anfreunden". Gerade die Syntax und Basics sind irgendwann keine Herausforderung mehr. Du kannst Dich dann viel besser auf die Sprachfeatures, die Bibliothek und Dinge wie Architektur und Design Patterns konzentrieren. Du lernst nicht nur schneller. Mit der Zeit und mehr Vorkenntnissen wirst Du auch automatisch von Beginn an besseren Quellcode schreiben.

Wie viel investierst Du täglich / wöchentlich?

Ein maßgeblicher Faktor ist automatisch die Zeit. Wenn Du pro Tag vier Stunden investieren kannst, kommst Du schneller vorwärts als mit einer. Gerade Einsteiger warten mit dem Anfangen aber häufig auf die perfekten Umstände. Ein gemütlicher Winterabend, ein verregneter Herbsttag oder das lang ersehnte lange Wochenende. Sicher toll und gemütlich, aber keine Voraussetzung. Regelmäßig eine halbe Stunde konzentriert Lernen ist ebenso effektiv.

Wichtiger als sich lange am Stück hinzusetzen, oder ganze

Abende zu blocken, ist es daraus eine Gewohnheit werden zu lassen! Grundsätzlich ist mehr Zeit besser. Noch viel wichtiger ist, dass Du regelmäßig lernst.

Wie schnell verstehst Du einzelne Aspekte?

Der Lernerfolg ist auch von Deinen Fähigkeiten abhängig. Alles andere wäre gelogen. Manche Menschen lernen schneller, andere etwas langsamer. Das ist keine Frage der Intelligenz. Nicht jedes Gehirn funktioniert gleich. Jüngere Menschen brauchen z.B. in der Regel weniger Wiederholungen um das gleiche Wissen im Langzeitgedächtnis zu speichern.

Aber lass Dich nicht täuschen. Es wird gern als Ausrede benutzt, die wir aufgrund falscher Glaubenssätze sogar selbst einreden: Ich kann das nicht, ich bin dafür nicht geschaffen, darin war ich nie gut. Ich bin schon zu alt oder Abends zu müde. Das ist pures Gift und einfach nicht die Wahrheit. Wenn es nicht klappt, dann eher weil solche Gedanken zu einer sich selbsterfüllenden Prophezeihung werden können. Das darfst Du nicht zulassen. Es kommt nicht darauf an wie gut Du jetzt bist. Erfolg kommt, wenn Du es willst und bereit bist etwas dafür zu tun, egal wie schnell Du lernst.

Die besten der Welt, egal auf welchem Gebiet, sind in der Regel wegen ihrer Arbeitshaltung und durch Fleiß an der Spitze. Dirk Nowitzki, der Basketballspieler? Trainiert endlos viel. Lionel Messi, der Fußballer? Trainiert extrem diszipliniert. Linus Torvalds, der Erfinder von Linux? Programmiert schon seit mehreren Jahrzehnten, bereits als er ganz klein war hat er

für seinen Opa getippt.

Mathematisches / Logisches Denkvermögen

Software-Entwicklung ist in vielerlei Hinsicht der Mathematik sehr ähnlich. Es geht um Logik und um klar definierte Regeln. Natürlich rechnest Du nicht mehr ständig mit Bits, also mit eins und null. Auch mathematische Rechenoperationen sind nicht unbedingt der maßgebliche Anteil Deiner Arbeit. Trotzdem begegnen Dir viele Bestandteile wie bspw. Auslagenlogik oder Mengenlehre, zumindest in der ein oder anderen Form.

Versteh das nicht falsch. Du musst Dich nicht bereits mit den mathematischen Gesetzen und Regel perfekt auskennen, um Programmierer zu werden. In der Tat werden Dir komplizierte Berechnungen und höhere Mathematik eher im Rahmen der Spielentwicklung begegnen. Dort musst Du im dreidimensionalen Raum Berechnungen anstellen. Das können Abstände, Flugkurven oder Kollisionen sein. Gerade wenn eine realistische Physik emuliert werden soll, kommen auch physikalische Gesetze zum tragen.

In anderen Bereichen kommst Du zwar nicht direkt mit solchen Berechnungen in Kontakt. Es schadet aber nicht, wenn Du einen gesunden Umgang mit Mathematik pflegst. Das schult die genutzten geistigen Kapazitäten enorm. Das beweist die Realität: Jemand der erfolgreich Mathematik studiert kann sich in der Regel sehr schnell in die Software-Entwicklung

einarbeiten.

Abschließend bleibt festzuhalten, dass mathematisches Denkvermögen definitiv hilfreich ist. Das gilt fernab von den konkreten Inhalten. Selbst wenn das Wissen in der Praxis nicht angewandt wird, hilft es abstrakt zu denken, neue Blickwinkel auf Probleme zu finden und Probleme zu hinterfragen.

Lernen als Berufstätiger

Das Leben ist wie so oft nicht nur schwarz oder weiß. Es gibt dazwischen Grauzonen. So ist es auch als Entwickler. Ein wichtiger Faktor beim Lernen ist nicht nur wie viel Du lernst und wie gut Deine Voraussetzungen sind. Auch die Rahmenbedingungen spielen eine große Rolle! Wenn Du berufstätig bist und nur Abends Zeit findet, ist es schwieriger sich auf neue Themen einzustellen. Wenn Du tagsüber die Hochleistungsphasen des Gehirns nicht nutzt, weil Du im Büro bist und arbeitest, wirst Du vermeintlich mehr Zeit benötigen.

Auch hier der Hinweis: Es bedeutet nicht, dass es nicht geht. Es kostet vielleicht manchmal einfach mehr Kraft und Willen. Gerade wenn der Tag ohnehin anstrengend war, die Sporteinheit schon nicht leicht fiel und Du am liebsten nur noch auf dem Sofa liegen möchtest. Dafür ist der Erfolg am Ende umso schöner. Du musstest vermutlich einfach mehr dafür kämpfen, im Zweifel gegen Dich selbst.

Die Strategie entscheidet

Maßgeblich beeinflussen kannst Du Deinen Lernerfolg über

die gewählte Strategie. In den nächsten zwei Kapiteln lernst Du mehr dazu. Stichworte sind die 10.000-Stunden Regel und das sogenannte Pareto-Prinzip (auch als 80/20 Regel bekannt).

Festhalten lässt sich jetzt schon: Verlier Dich nicht zu sehr in den Details. Es ist gar nicht so wichtig alle geschriebenen Zeilen und Keywords wie ein Schwamm aufzusaugen. Das wirst Du ohnehin nicht schaffen. Außerdem bleiben die Inhalte mit der Zeit automatisch hängen. Geh entspannt an die Sache heran. Dann hast Du viel mehr Spaß. Konzentrier Dich darauf die Konzepte zu verstehen und Logik nachzuvollziehen. Dann geht es außerdem auch schneller.

Wenn Du Programmieren lernst, wirst Du an der ein oder anderen Stelle von der sogenannten 10.000 Stunden Regel lesen. Sie besagt, dass Du Dich mindestens 10.000 Stunden mit etwas beschäftigen musst, um zur Weltelite gehören zu können. Aufgestellt hat sie der Psychologe Anders Ericsson. Bekannt gemacht hat sie aber erst Malcolm Gladwell in seinem Bestseller *Überflieger: Warum manche Menschen erfolgreich sind - und andere nicht*.

10.000 Stunden Programmieren?

Bedeutet es, dass Du nun 10.000 Stunden Software-Entwickeln musst? Ganz sicher nicht. Die Regel wird häufig missverstanden. Zum einen musst Du nicht zur Weltspitze gehören, um ein erfolgreicher Entwickler zu sein. Nicht jeder Bäcker, Anwalt oder Finanzbeamte ist Weltspitze. Trotzdem kann er überdurchschnittlichen Erfolg haben!

Fast noch wichtiger finde ich, gerade im Bezug auf Software: Du musst nicht alle Disziplinen beherrschen. Du kannst Dich auch einfach mehrere Monate nur auf einen ganz konkreten Aspekt fokussieren. Schon dann wirst Du genau den vielleicht besser beherrschen als so mancher “Profi”.

Ein Rechenbeispiel

Wenn Du diese Regel in die Praxis übertragen willst, musst Du berücksichtigen wie viel Zeit Du pro Tag Software entwickelst. Es geht bei den 10.000 Stunden um reale Zeit, in der Du der Tätigkeit nachgehst. Abzüglich von Pausen,

Meetings und anderen Unterbrechungen schaffst Du es als Entwickler im Unternehmen von acht Stunden täglich vielleicht fünf Stunden produktiv zu nutzen. Bei fünf Arbeitstagen kommst Du auf **25 Stunden pro Arbeitswoche**.

Jedes Jahr hat 52 Wochen. Bei großzügigem Kontingent an Urlaubstagen (30) kannst Du sechs Wochen freinehmen. Mit Feiertagen und möglicher Krankheit rechnen wir einfach pro Jahr 44 Wochen. Somit entwickelst Du **jedes Jahr 1100 Stunden** Software.

Damit benötigst Du etwas mehr als neun Jahre Zeit, um zur Weltspitze zu gehören. Und das setzt voraus, dass Du Dich auf eine Sprache beschränkt hast. Um ein erfahrener Programmierer zu werden, ist es aber viel sinnvoller auch andere Technologien und Sprachen kennenzulernen. Die verschiedenen Blickwinkel und Lösungsansätze sind es, die Dich reifer machen. Daher kann ich aus Erfahrung berichten: Du kannst deutlich schneller mit Deinem Wissen in der Praxis etwas anfangen.

Fazit

Die 10.000 Stunden Regel kann aus meiner Sicht bei der Software-Entwicklung nicht einfach umgemünzt werden. Sie findet viel mehr bei Athleten und Sportlern Anwendung, die immer der gleichen Sportart nachgehen. Lass Dich also nicht abschrecken. Deutlich hilfreicher ist das sogenannte Pareto-Prinzip.

Programmierer sind im Schnitt intelligente Menschen. Dennoch wird ihnen liebevoll nachgesagt, sie sein faul. Dabei geht es aber eher um Anspielungen auf Paradigmen wie zum Beispiel das KISS- (Keep It Simple Stupid) und das DRY-Prinzip (Don't Repeat Yourself). Der Code soll möglichst schlicht, einfach und übersichtlich sein. Das gerade darin die Herausforderung steckt, übergeht die Verniedlichung bewusst.

Wenn Du Entwickler werden möchtest, kannst Du einfach nach dem Motto "work hard" vorgehen. Wenn Du Java lernst bedeutest es zum Beispiel ein möglichst umfangreiches Buch zu besorgen und dies von Cover zu Cover durchzuarbeiten. Die Taktik funktioniert, genügend Zeit und Durchhaltevermögen vorausgesetzt. Es ist aller Ehren wert. Schade nur, dass Du viel Streuung mit einkalkulieren musst. Du wirst Dir viele Themen anschauen, die Du nie oder erst sehr viel später in der Praxis verwendest.

Statt mit Scheuklappen durch den Lernstoff zu wühlen und dabei viel Energie, Fleiß und Zeitaufwand zu investieren, gibt es auch einen gezielteren Weg. Du musst Deine Ziele verfolgen, solltest aber strategisch vorgehen. Work smart! Und hier kommt das Pareto-Prinzip, benannt nach dem Ingenieur Vilfredo Pareto, ins Spiel.

Das Pareto-Prinzip

Pareto hatte die Besitzverhältnisse des vorhandenen Bodens in Italien untersucht. Dabei fand er heraus, dass zu dem Zeitpunkt nur 20% der Bevölkerung etwa 80% des gesamten

Bodens besaßen. Spannend ist, dass er diese Verteilung in vielen weiteren Bereichen vorgefunden hat. Daraus hat er später das Prinzip abgeleitet, das wegen der Verteilung auch als 80/20-Regel bekannt ist. Es lässt sich gewiss nicht immer anwenden. Aber gerade bei der Software-Entwicklung, dem Zeitmanagement und der Projektplanung entspricht es häufig der Realität. Nur die prozentuale Verteilung solltest Du nicht auf die Goldwaage legen.

Die 80/20-Regel in der IT

Eine etwas ungenaue Formulierung der gleichen Idee lautet: Der Teufel steckt im Detail. Und so ist es wirklich. Wenn Du Software-Projekte realisierst stellst Du schnell fest, dass der grobe Rahmen schnell steht. Die Knackpunkte sind gelöst. Was dann folgt sind Detailarbeiten. Damit verbringst Du einen Großteil Deiner Zeit.

Etwa 80% der Funktionalität einer Software ist oft schon nach 20% der Zeit fertig. Die restlichen 20% brauchen aber noch einmal 80% der Zeit. Wie kannst Du das nun zum Lernen benutzen? Zusammen mit der DiSSS-Methode von Tim Ferriss! Sie hilft Dir dabei Dich auf das Wesentliche zu fokussieren. So kannst Du viel schneller Programmieren lernen und Fortschritte erzielen. Die 80/20-Regel legt die theoretische Grundlage. Mit der DiSSS-Methode gibt es einen Weg, um sie beim Lernen in der Praxis anzuwenden.

Ein Beispiel

Es ist ein regnerischer Abend. Draußen ist es bereits dunkel. Getränke stehen auf dem Tisch. Einem gemütlichen Abend steht nichts mehr im Weg. Auf der Welt verteilt sitzen zwei Leute, jeweils in ihrem Zuhause, in diesem Szenario am Computer. Beide kommen gleichzeitig auf die Idee, die Programmiersprache Python zu lernen. Sie möchten beide mit dem Framework Django eine Webseite erstellen. Einer von beiden heißt Klaus. Er legt ohne zu zögern los und fängt an sich durch die Dokumentation und Tutorials zu arbeiten, von A-Z. Er befasst sich mit wichtigen Grundlagen und gleichzeitig mit vielen Themen, die er jetzt noch nicht kennenlernen muss.

Der andere Entwickler heißt Tim. Er geht systematischer vor und analysiert. Mit der DiSSS-Technik verschafft er sich einen Überblick. Er rechiert welche Themenbereiche es gibt und sammelt Stichpunkte. Dann setzt er Prioritäten. Er überlegt, welche der Themen er lernen muss um mit Django zu arbeiten. Er wird, dank der guten Vorbereitung, schneller Resultate erzielen und das Ziel erreichen. Er wird am Ende nicht die gleichen Fähigkeiten besitzen wie Klaus. Aber vielleicht 80% davon! Und wie schnell hat er die erreicht? Richtig, in etwa 20% der Zeit! Die Themen die er übersprungen hat, kann er jederzeit in der Zukunft lernen, wenn er das Wissen benötigt. Klingt sinnvoll, oder?

Was ist DiSSS?

DiSSS ist eine Sammlung von Arbeitsschritten. Genau genommen ist es keine bahnbrechende Innovation. Viel mehr sind es Methoden, die einfach strategisch eingesetzt werden.

Formalisiert hat sie Tim Ferriss in seinem Buch *4-Hour Chef*. Die Abkürzung DiSSS steht dabei für:

- Deconstruction
- Selection
- Sequencing
- Stakes

Das »i« hat keine eigene Bedeutung. Es wird verwendet um die Abkürzung auszusprechen.

Schnell programmieren lernen mit DiSSS

Eine gute Nachricht: Du kannst schnell Programmieren lernen! Wunder erwarten darfst Du allerdings keine. Ein Großteil dessen was einen professionellen Entwickler auszeichnet, ist Erfahrung. Die bekommst Du nur durch Praxis. Und dafür brauchst Du nunmal Zeit.

Bei DiSSS geht es auch nicht um die Wunderpille, die alles leichter macht. Es geht darum gewünschte Ergebnisse ohne Umwege zu erzielen. Im Beispiel oben gibt es das konkrete Ziel mit Django zu arbeiten. Klaus hat unmittelbar angefangen zu lernen. Er arbeitet ziellos die Dokumentation, Tutorials und Bücher durch. Dabei eignet er sich viel Wissen an, dass er für seine Ziele gar nicht braucht. Ich sage nicht, dass es falsch ist! Im Gegenteil. Bei gewissenhafter Arbeit wird er mittelfristig eine Menge Erfahrung aufbauen.

Tim wird aber seine gewünschte Software eher fertigstellen. Und genau das ist der Punkt. Denn es fühlt sich einfach gut an seine Ziele zu erreichen. Das ist ein großer Motivationsschub. Den wird Klaus nicht so schnell bekommen. Er wollte gern "Python lernen". Da gibt es aber gar keinen definierten Endpunkt.

DiSSS kann Dir helfen, schneller ans Ziel zu kommen. Es lohnt sich also, die notwendigen Schritte genauer anzuschauen.

Deconstruction

Zuerst verschafft sich Tim einen Überblick über die Programmiersprache und die einzelnen Themengebiete. Der Name »Deconstruction« deutet es an: Er zerlegt die Sprache in seine Bestandteile. Dazu arbeitet er die Dokumentation durch und umreißt Kernthemen. Wie kleinteilig das erfolgt hängt von der verfügbaren Zeit ab. Ein Beispiel:

Tim hat vier Tage Zeit, um zu lernen. Er unterteilt die Sprache Python in große, logische Lerneinheiten. Wenn er vier Wochen zur Verfügung hat, könnten kleinere Einheiten entstehen, um Details mehr Zeit zu widmen.

Der nächste Schritt besteht in der Auswahl der relevanten Module oder Lerneinheiten.

Selection

Der Schritt »Selection« besteht in der Auswahl der relevanten Lerneinheiten. Tim möchte eine Webanwendung

mit dem Framework Django entwickeln. Er muss mindestens grundlegende Themen wie Datentypen, Kontrollstrukturen, Objektorientierung und ähnliches anschauen. Zudem muss er eine Einheit zum Thema Datenbankabfragen einplanen. Viele andere Themen kann er am Anfang ignorieren.

Sequencing

Im Schritt »Sequencing« geht es dann um eine sinnvolle Reihenfolge. Wissen baut aufeinander auf. Die Grundlagen kommen logischerweise vor den Datenbankabfragen in den Lehrplan.

Eine Zeitplanung sollte ebenfalls erfolgen, da nur begrenzt Zeit verfügbar ist. Der Plan für die verfügbaren vier Tage könnte zum Beispiel zwei Tage Grundlagen, einen Tag Datenbanken und einen Tag Django selbst vorsehen.

Stakes

Der letzte Schritt in der DiSSS-Methode hat etwas mit Deiner Motivation zu tun. Die Idee von Tim Ferriss ist es etwas »auf's Spiel zu setzen«. Es soll den Druck erhöhen. Der hilft Dir auch wirklich zu handeln. Das kann zum Beispiel materieller Verlust sein. Wenn Du Deine eigene Vorgabe nicht erreichst, kannst Du Dich vorher verpflichten Geld zu spenden oder einen geliebten Pullover in die Kleidersammlung geben. Ich finde diesen Weg aber nicht sonderlich sinnvoll. Daher halte ich das für überflüssig. Was viel besser funktioniert, ist möglichst vielen Menschen von Deinen Zielen zu erzählen. Je mehr eingeweiht

sind, desto besser. Der Mensch versucht automatisch solchen Aussagen und Ankündigungen auch gerecht zu werden.

Fazit

Schnell programmieren lernen ist möglich. Wunder sind allerdings nicht zu erwarten. Trotzdem kannst Du mit etwas Strategie schon 80% des Ergebnisses mit nur 20% des Aufwands erreichen. Die DiSSS-Methode ist das Pareto-Prinzip in der Praxis!

Im Web gibt es zu jeder Programmiersprache eine regelrechte Flut an Informationen. Jede Sprache hat eine eigene Community, es gibt entsprechende Blogs, Foren, Wikis, Tutorials und How-Tos. Mit codingtutor.de biete ich einen Leuchtturm, der durch den Nebel hindurch Orientierung bietet. Ich will den Einstieg in die Software-Entwicklung erleichtern. Dabei empfehle ich Dir nicht einfach blind meine eigenen Favoriten. Du sollst Deinen eigenen Weg finden und gehen. Hier ein paar Anregungen zum Lernen, sortiert nach Priorität:

Lern Deine verfügbare Zeit zu schätzen!

Früher habe ich es für komplett überflüssig gehalten mir Hilfe zu holen. Ich dachte immer, ich kann das alles allein lernen. Immerhin leben wir im Informationszeitalter. Wir finden alles kostenlos im Internet. Das ist teilweise richtig. Als Schüler oder auch als Student hast Du auch noch ganz andere Prioritäten.

Irgendwann habe ich angefangen umzudenken. Ein großes Problem, wenn Du Dir alles selbst beibringen willst: Es kostet viel mehr Zeit. Zum einen musst Du selbst genau wissen was Du lernen musst. Dann musst Du qualitativ hochwertige Informationen finden. Wenn die kein didaktisches Konzept verfolgen, musst Du immer wieder neue Infos suchen um nicht erklärte Aspekte zu verstehen. Das kostet wertvolle Zeit. Und die kostet Dich sogar doppelt! Zum einen musst Du die Zeit zum Lernen und für die Recherche aufwenden, um Dir etwas beizubringen. Auf der anderen Seite hast Du auch sogenannte *Oppertunity Cost*. Dir geht die Zeit verloren etwas anderes in

der Zeit zu machen. Du könntest die Zeit zum Beispiel produktiv nutzen und an einem Projekt arbeiten. Verschenkte Zeit kostet Dich daher doppelt.

Gleichermaßen verschenkst Du auch Zeit, wenn Du auf eigene Faust lernen willst. Gerade in der IT kannst Du Dir den Einstieg in neue Themen um ein Vielfaches erleichtern. Du brauchst einen fähigen Mentor. Sei nicht überheblich und lass Dich auf Hilfe ein! Macht es Sinn kostenpflichtige Seminare, Workshops oder Videotrainings zu kaufen? Unbedingt! Ich zahle nicht in erster Linie für die Informationen, die ich am Ende vielleicht wirklich irgendwann selbst finde. Ich zahle für die Zeit, die es mir spart.

Ist ein Seminar für 500 Euro teuer, wenn es mir 4 Wochen Zeit spart in denen ich Kundenaufträge und Projekte umsetzen kann, die ich später in Rechnung stelle? Nein. Ist ein Videotraining für 100 Euro teuer, wenn ich dafür eine Woche weniger nach Informationen suchen muss und bei Problemen einen Ansprechpartner habe? Nein. Ist ein Buch für 40 Euro teuer, wenn ich dafür viele Codebeispiele bekomme und nicht Stunden in Dokumentationen wühlen muss? Nein.

Seminare und Workshops besuchen

Mit einem Seminar oder Workshop gelingt der Einstieg in neue Themen am besten. Du hast vor-Ort motivierte Teilnehmer und einen erfahrenen Trainer. Der kann Schwerpunkte setzen, wichtige Bereiche nennen und direkt bei Startschwierigkeiten helfen. Wie oben beschrieben weiß ich:

das kann Dir eine ganze Menge Zeit sparen. Und dabei rede ich nicht von ein paar Stunden. Es geht wirklich um Wochen und Monate.

Das gleiche gilt für Unterricht und Vorlesungen. Dir wird, bevor Du etwas umsetzt, gezeigt wie es funktionieren kann und welche Gedanken der Dozent oder Trainer bei der Entwicklung hat. Genau das ist in Büchern, Blogs und Tutorials so nicht möglich. Dort bekommst Du in den meisten Fällen fertigen Quellcode präsentiert. Die Überlegungen und getroffenen Entscheidungen werden nicht im gleichen Maß erläutert.

Mit Videotrainings lernen

Videotrainings sind mittlerweile für viele Entwickler, auch für mich, die erste Anlaufstelle für neues Wissen. Du bekommst nahezu alle Vorteile die auch ein Seminar bietet. Du kannst einem erfahrenen Entwickler “über die Schulter schauen”. Der erklärt live bei der Programmierung warum er bestimmte Wege geht und weshalb er Entscheidungen trifft. Auch zur Struktur und Architektur werden Überlegungen nachvollziehbar. Du lernst im Video außerdem Arbeitsabläufe kennen. Das Beste ist: Du kannst jederzeit eine Pause machen, zurückspulen und Videos später erneut ansehen.

Zudem sind Videotrainings didaktisch aufgebaut. Bei kostenpflichtigen Angeboten kannst Du Feedback bei Fragen erwarten und hast somit einen Ansprechpartner. Fast so gut wie ein Seminar vor-Ort.

Bücher lesen

Selbstverständlich sind auch Bücher nach wie vor wichtig. Egal ob Du eBooks oder gedruckte Bücher vorziehst. In der Regel gehen Bücher deutlich mehr in die Tiefe als ein Videotraining. Ein gutes Beispiel: Java ist auch eine Insel. Dort wird enorm viel Wissen auf über 1200 Seiten transportiert. Das ist in Videotrainings so gar nicht umsetzbar. Das wäre Videomaterial mit mehreren Tagen Laufzeit. Außerdem sind Bücher auf Dauer eine deutlich bessere Referenz, in der schnell etwas nachgeschlagen werden kann. Das ist bei einem Videotraining deutlich schwieriger. Gleichermaßen ist auch bei guten Büchern der didaktisch sinnvolle Aufbau sehr wertvoll.

Tutorials durcharbeiten

Es gibt für jede Programmiersprache eine Vielzahl an Tutorials. Die findest Du bei Google. Ich nutze solche Ressourcen gerne für den Einstieg in konkrete Themenbereiche. Für den allgemeinen Einstieg in neue Sprachen oder Frameworks finde ich mittlerweile Videotrainings angenehmer, da es einfach schneller geht und viel mehr Hintergrundwissen und Gedanken transportiert werden.

Offizielle Dokus nutzen

Die meisten Programmiersprachen werden vom Hersteller dokumentiert. So bieten z.B. Python, PHP oder Java sehr umfangreiche Dokumentationen der vorhandenen

Funktionalität. Hast Du erstmal ein Gefühl für die Syntax entwickelt, kannst Du in der offiziellen Dokumentation die Arbeitsweise und Methodenaufrufe der mitgelieferten Klassen nachvollziehen. Du wirst nur noch diese Dokumentation als Gedächtnisstütze benötigen. Spätestens dann wird es Zeit für eine neue Herausforderung.

Codingtutor.de besuchen!

Eigenwerbung stink? Ich hoffe nicht! Ich möchte natürlich noch mal auf das Blog hinweisen. Auf codingtutor.de findest Du viele Inhalte rund um die Themen App- und Webentwicklung. Ich möchte Dir beim Einstieg in die Software-Entwicklung helfen. Es bereitet mir große Freude meine Leidenschaft mit anderen Menschen zu teilen. Ich freue mich auf Deinen Besuch!

Erfolg mit eigenen Projekten erzielen

Sobald Du etwas Routine und Sicherheit gefunden hast, macht es Sinn den nächsten Schritt zu gehen. Es wird Zeit eigene Projekte umzusetzen. Die müssen anfangs nicht spektakulär sein. Wichtig ist viel mehr, dass Du selbst Lösungen suchst und findest. Dabei kannst Du Dich direkt mit der offiziellen Dokumentation vertraut machen. Von Projekt zu Projekt steigerst Du die Komplexität und setzt Ideen um, in denen Du immer wieder andere Teile der Programmiersprache verwendest.

Der große Vorteil von eigener Software ist, dass Du Dir um alles selbst Gedanken machen musst. Du musst eigene

Lösungen für Probleme finden und eine Architektur entwerfen. Das sind die größten Herausforderungen für Entwickler. Auch später werden Dir diese Schwierigkeiten begegnen. Sie zu meistern ist daher sehr wichtig. Außerdem lernst Du am besten bei die Fehlersuche.

Selbstständig arbeiten

Die Arbeit in Projekten verändert sich mit zunehmender Erfahrung nur unwesentlich. Lediglich die Komplexität und der Projektumfang ändern sich. Manches funktioniert vielleicht flüssiger. Du machst weniger kleine Fehler, musst weniger nachschlagen und in der Dokumentation recherchieren.

In den meisten Fällen wirst Du mit einem Team zusammen an Projekten arbeiten. Aber auch da ist der Workflow vergleichbar:

- Planungsphase
- Implementierung
- Fehlersuche
- usw.

Auch deshalb ist die Arbeit an eigenen Projekten von Anfang an sinnvoll. Der Praxisbezug bereitet Dich auf Deine Karriere als Programmierer vor. Neben vielen anderen Fertigkeiten ist die Fähigkeit selbstständig zu arbeiten und Probleme lösen zu können enorm wertvoll. Es weiß jeder Teamleiter zu schätzen,

wenn er sich auf Dich verlassen kann. Wenn nicht ständige Rückfragen notwendig sind ist das umso besser. Denn auch in einem Team wird nicht jede noch so kleine Entscheidung bei der Entwicklung abgesprochen.

Wenn Du durch die Fähigkeit gute Entscheidungen zu treffen glänzt, hast Du höhere Chancen auf Erfolg. Allerdings: um fundierte Entscheidungen zu treffen benötigst Du Programmierpraxis. Hier daher noch ein weitere Tipps für Deinen Weg:

- Viel Quellcode selbst schreiben.
- Viel Quellcode von erfahrenen Entwicklern lesen.
- Reflektieren.
- Eigenen Quellcode optimieren wenn Du Erfahrung gesammelt hast.
- Anderen bei der Fehlersuche helfen.
- Erklär Deine Erkenntnisse anderen, zum Beispiel in einem Blog.

Bis zum Ende gehen!

Anfangs macht es Spaß sich Gedanken über ein neues Projekt zu machen. Leidenschaftlich wird an Plänen oder sogar einem Geschäftsmodell gearbeitet. Bei der Implementierung bemerkst Du, dass es nur in kleinen Schritten in Richtung Ziel geht. Die Motivation sinkt automatisch. Der Zauber, der das

neue Projekt Anfangs umgeben hat, verschwindet. Schließlich hat Dich die Realität eingeholt: Oh je, wieder einmal sind Ausdauer und Fleiß gefragt!

Sehr viele Projekte versanden auf dieser Stufe. Auch weil das Vertrauen in die eigene Idee sinkt. Und deshalb liegt genau an dieser Stelle der Schlüssel zum Erfolg. Wer die Ideen und Projekte zu Ende bringt, hat deutlich bessere Chancen auf Erfolg. Das gilt auch - oder gerade - bei der Software-Entwicklung. Programmieren lernen bedeutet Ziele zu verfolgen - bis zum Ende. Du musst Motivation finden um auch mentale Durststrecken zu überwinden. Wer bereits an kreativen Projekten gearbeitet hat weiß: die werden kommen. Nur der feste Wille die gesetzten Ziele zu erreichen hilft. Wenn Du es schafft auch unspektakuläre (aber wichtige) Phasen eines Projekts durchzustehen, kannst Du alles erreichen. Daher: Alles eine Frage der Motivation! Oder um es anders auszudrücken: Die Frage lautet nicht “Kannst Du Programmieren lernen?”. Richtig ist: “Wie sehr willst Du es?”!

Sicher hast Du an Silvester schon mal gute Vorsätze für das neue Jahr gefasst? Die Euphorie war jedes Mal groß. Es herrschte Aufbruchstimmung. Scheinbar nichts konnte Deinen Willen erschüttern. Du warst fest entschlossen und bist voller Motivation in das nächste Jahr gestartet. Doch was ist aus den Vorsätzen geworden? Hast Du aufgehört zu rauchen? Bist Du weiterhin regelmäßig zum Sport gegangen? Hast Du angefangen wieder mehr zu lesen? Vielleicht wolltest Du auch einfach etwas sparsamer werden? Hat es am Ende funktioniert? Oder hat Dich der Alltag eingeholt?

Vielleicht hast Du schließlich das Ziel wieder aus den Augen verloren? Keine Angst: Es geht vielen Menschen so. Und das liegt nicht daran, dass Du einen schwachen Charakter hast. Dein einziger Fehler: Du bist ein Mensch! Der Erfolg entsteht weniger durch den unbändigen Willen. Es sind viel mehr die Gewohnheiten, die uns täglich steuern. Darum brauchst Du die richtige Taktik, um Deine Ziele zu erreichen.

Es geht nicht um Disziplin

Menschen denken häufig Veränderungen - die guten Vorsätze - hängen mit eiserner Disziplin zusammen. Das ist aber nicht der Fall. Wirklich nicht. Diese Annahme ist ein großer Fehler. Er stellt nicht nur erfolgreiche Menschen nahezu übermenschlich dar. Diese Sichtweise schmälert vor allem Deine Erfolgschancen ganz erheblich. Du verlierst das Gefühl Kontrolle und Einfluss zu haben. Der Erfolg wird zum Glücksspiel. Deine geplante Veränderung ist schon im Vorfeld

zum Scheitern verurteilt.

Deine Disziplin steht und fällt mit der Willenskraft. Es gibt Menschen mit starkem Willen, keine Frage. Es gibt auch Menschen mit weniger starkem Willen. Das lässt sich sogar, zumindest bis zu einem gewissen Grad, trainieren. Trotzdem haben alle etwas gemeinsam: Willenskraft ist und bleibt eine endliche Resource. Vergleich es einfach mit einer Autofahrt. Du bist auf der Landstraße unterwegs. Tankstelle um Tankstelle lehnst Du ab - mit der Hoffnung am Ziel zu sein bevor der Tank leer ist. Irgendwann musst Du doch nachgeben. Der Tank ist vorher leer.

Der Vergleich hinkt, aber so ähnlich funktioniert es. Du brauchst Deine Willenskraft im Laufe des Tages regelrecht auf. Wenn Du Dich nur auf Deine Disziplin verlässt, ist es ähnlich wie die Hoffnung es noch bis zum Ziel zu schaffen. Wenn Du den ganzen Tag neuen Versuchungen ausgesetzt bist, kannst Du nach jeder einzelnen der nächsten weniger gut widerstehen. Das ist auch ein Grund dafür warum in einer Diät häufig am Abend “das große Fressen” beginnt. Das hat natürlich noch weitere Ursachen. Aber ein Grund ist eben, dass es Energie kostet den ganzen Tag über “Nein” zu sagen.

Es gibt sogar Untersuchungen die zeigen, dass es mit jeder getroffenen Entscheidung schwerer wird Versuchungen zu widerstehen. Wirklich tolle Studien dazu findest Du in dem Buch *Don’t Eat the Marshmallow…Yet: Das süße Geheimnis von Erfolg*. Im Buch geht es um den berühmten Test, bei dem Kindern eine Süßigkeit gegeben wurde, zusammen mit der

Aussicht noch mehr zu bekommen. Einzige Vorraussetzung: sie dürfen die erste noch nicht essen. Anschließend wurde das Kind allein im Raum gelassen und beobachtet. Aber, zurück zum Thema.

Wenn Du Dich nur auf Deine Disziplin verlässt, musst Du ständig aktiv gegen einen inneren Dialog arbeiten. Du musst Dich überwinden und bewusst darauf achten Dich richtig zu verhalten. Soll ich Süßigkeiten kaufen, mir mal “etwas gönnen”? Heute doch schnell Fastfood essen, einmal kann ja nicht schaden? Vielleicht doch das leckere Eis? Geh ich doch eine Zigarette rauchen? Ein vergleichbarer innere Dialog läuft ständig ab. In allen Menschen! Wenn Du Dich nur auf Willenskraft verlässt, müsstest Du ständig in diesem Dialog auf korrekte Antworten achten. Denn die bestimmen am Ende wie Du handelst. Das Problem: Selbst bei klarem Verstand ist das nicht dauerhaft möglich. Und dann kommen noch Faktoren wie Müdigkeit, Stress und Dein eigenes Wohlbefinden dazu. Wie Du siehst: die Willenskraft ist kein guter Verbündeter.

Wenn Du Programmieren lernen willst wäre es also nicht schlau, Dich immer wieder zu überreden jetzt doch noch einmal "das Buch aufzuschlagen".

Gewohnheiten nehmen Entscheidungen ab

Stattdessen sind die richtigen Gewohnheiten wichtig. Die erste Zeit gehört natürlich etwas Disziplin dazu. Du musst bewusst darauf achten, Dich an Deine neue Routine zu halten. Nur so entsteht eine neue Gewohnheit. Auch deshalb solltest

Du immer nur an einer Verhaltensweise gleichzeitig arbeiten. Das steigert Deine Chance auf Erfolg ganz erheblich.

Der große Vorteil: Sobald etwas zur Gewohnheit wird, musst Du fast nicht mehr darüber nachdenken. Es ist einfach fest einprogrammiert. Denk einfach mal über Deine Routine jeden Morgen nach. Vieles wird ähnlich sein, jeden Tag. Und das unbewusst. Warum? Weil Du Dich daran gewöhnt hast. Auch oder gerade beim Einkaufen landet häufig das im Einkaufswagen, was zu unseren Gewohnheiten gehört. Ein anderes Beispiel: Vielleicht gehst Du ohnehin regelmäßig zum Sport? Oder hast das als Jugendlicher gemacht? Auch hier greifen ähnliche Mechanismen. Wenn Montags und Donnerstags der Sportverein auf dem Programm steht hinterfragst Du das nach kurzer Zeit nicht mehr. Hab ich überhaupt Lust? Gehe ich heute hin? Du machst es einfach, selbst wenn Du eigentlich viel mehr Lust auf das Sofa hättest. Der Tag ist programmiert und die Frage nach dem “Ja” oder “Nein” stellt sich gar nicht mehr. Das gilt gerade dann wenn Du innerlich stöhnst. Fühl beim nächsten Mal einfach mal “in Dich hinein”.

So formst Du Gewohnheiten

Im Buch *The One Thing* hat der Autor *Gary Keller* das Formen neuer Gewohnheiten sogar mit Zahlen belegt. Die decken sich in vielerlei Hinsicht mit meinen Erfahrungen. Es werden Studien und Untersuchungen zitiert, die von 66 notwendigen Wiederholungen ausgehen. Deine neue Gewohnheit ist jeden Tag lesen? Dann musst Du dies etwa 66

Tage wiederholen. Die Zahl wirkt aus der Luft gegriffen, vielleicht sogar etwas hoch. Aber es steckt viel Wahrheit darin. Du kannst Dich der Zahl auch auf anderem Wege nähern.

Wenn Du drei Mal pro Woche Sport treibst, finde ich die 66 Wiederholungen eher ungenau. Etwa 10 Wochen die gleiche Routine, also die gleichen Trainingstage und Zeiten einzuhalten, finde ich treffender. Danach wirst Du fast automatisch Sport machen - ganz von alleine, ohne inneren Schweinehund. Diese Zeitspanne deckt sich, wenn auch etwas anders, mit den Angaben. Denn 10 Wochen entsprechen 70 Tagen. Damit kommen wir den 66 Tagen sehr nah.

Wichtig ist, dass Du nur eine Gewohnheit gleichzeitig änderst. Es soll und darf nicht störend oder aufwendig wirken. Dann funktioniert es nicht. Du sollst Dir auch keinen Druck auferlegen. Die zweite Grundregel lautet: Die Regelmäßigkeit ist wichtiger als die konkrete Dauer. Fang einfach ganz klein an. Fünf Minuten reichen vollkommen aus. Das baust Du aus, Schritt für Schritt. Und wenn nicht ist das auch OK. Wenn Du jeden Tag fünf Minuten in einem Buch liest, kommst Du in der Woche auf mehr als eine halbe Stunde. Im Monat sind das zwei Stunden. Das ist besser als nichts zu tun, nur weil sich die perfekten Bedingungen und viel Ruhe zum Abschalten nicht ergeben haben.

Ich kann sogar an einem Beispiel aus eigener Erfahrung zeigen, was sich aus den fünf Minuten entwickeln kann. Ich bin seit Jahren begeisterter Läufer. Allerdings bin ich erst spät eingestiegen. Wirklich konkret wurde es sogar erst vor etwa vier

bis fünf Jahren. Ich konnte zu Beginn nicht mal einen Kilometer am Stück laufen. Aber: Immerhin ein Anfang. Ich habe mir keinen Druck gemacht. Ein umfassender Trainingsplan? Hätte mich sicher abgeschreckt. Ich musste erstmal den Spaß und die Lust am Laufen entwickeln. Der Zwang, plötzlich drei oder viermal pro Woche streng nach Plan zu trainieren, hätte das Aus bedeuten. Stattdessen bin ich einfach gelaufen, etwa zwei oder dreimal pro Woche. Je nachdem wie ich gerade Lust und Zeit hatte. Dabei habe ich die Leistung stetig ausgebaut. Irgendwann hab ich mehr Struktur und System gesucht. Erst dann gab es einen konkreten Plan. Aber da war Laufen bereits Gewohnheit.

Dieses Wissen habe ich auch auf das Krafttraining übertragen. Das brauchst Du als Läufer um Verletzungen vorzubeugen und um schneller zu werden. Aber auch dabei bin ich nicht von 0 auf 100 gegangen. Der größte Fehler: in ein Fitnessstudio gehen, einen Plan erstellen und Dich überfordern lassen. Eine viel zu gravierende Änderung in extrem kurzer Zeit. Ich habe mich, damals total unbewusst, langsam herangearbeitet. Zu Beginn hab ich dreimal pro Woche Liegestütze gemacht. Das hat jeweils weniger als fünf Minuten Zeit in Anspruch genommen. Irgendwann sind Griffe für Liegestütze hinzugekommen. Dann hab ich mir Kurzhanteln angeschafft. Später kam eine Langhantel hinzu. Schließlich kam die Idee eine Lösung zum Bankdrücken anzuschaffen. Danach kam noch ein Zugturm für Klimmzüge und ähnliches Equipment. Jetzt trainiere ich in meinem Homegym. Das ist Schritt für Schritt gewachsen, über Wochen und Monate.

Mittlerweile trainiere ich regelmäßig - aus Angewohnheit - nach einem systematischen Trainingsplan.

Gewohnheiten beim Programmieren lernen?

Die spannende Frage: Wie überträgst Du es auf das Programmieren? Was bringt Dir das beim Lernen? Eigentlich ganz einfach: Versuch nicht diszipliniert am Ball zu bleiben. Richte Dir systematisch Zeiten ein, an denen Du lernst. Schüler und Studenten haben noch mehr Spielraum und können einfach “drauf los” entwickeln. Aber selbst denen empfehle ich, systematisch zu planen. Du bist Schüler und kämpfst mit Hausaufgaben und zusätzlichem Lernen? Richte Dir feste Zeiten ein. Schon nach kurzer Zeit wirst Du die nicht mehr hinterfragen. Sie stehen einfach fest. Der Mensch ist wirklich ein Gewohnheitstier. Das solltest Du nutzen.

Als Angestellter oder jemand der im Berufsleben eingebunden ist, wirst Du es im Vergleich schwerer haben. Vielleicht hast Du auch noch eine Beziehung? Du bist verheiratet? Hast sogar Kinder? Das hilft nicht unbedingt bei der Suche nach Freiraum zum Lernen. Aber: Es geht trotzdem. Such einfach strategisch bspw. drei oder vier Abende pro Woche aus, an denen Du Dir Zeit nimmst. Und wenn es nur zwei werden ist das auch in Ordnung. Wichtiger ist, dass Du sie regelmäßig nutzt! Weich vielleicht auch etwas auf das Wochenende aus. Die Chancen auf Freiraum stehen da in der Regel deutlich besser. Um Erfolg zu haben solltest Du Dich streng an die Zeiten und Tage halten. Selbst wenn Dir nicht mehr nach Lernen ist. Setz Dich wenigstens eine Viertelstunde

an den Rechner. Es gibt gute und auch schlechte Tage. Wichtig ist, die Gewohnheit aufrecht zu erhalten. Du musst früher oder später Ausnahmen einplanen. Wichtig ist aber das sie genau das bleiben - gelegentliche Abweichungen vom eigentlichen Plan.

Mehr Informationen dazu

Wenn Dich dieses Thema interessiert empfehle ich unbedingt *The Power of Habit* von *Charles Duhigg* und *The One Thing* von *Gary Keller* als Lektüre. Es ist unglaublich wie sehr wir mit einfachen Mitteln und Verhaltensweisen unser ganzes Leben vereinfachen können. Das Gehirn ist unglaublich kompliziert, menschliches Verhalten in vielen Fällen aber trotzdem enorm berechenbar. Genau diese beiden Bücher sind daher nicht nur für Entwickler eine enorme Bereicherung.

Sport als Ausgleich zur sitzenden Tätigkeit - der Klassiker unter Hobbies und Interessen im Lebenslauf. Auch die logische Erklärung zur “Prävention” und “Vorbeugung von Krankheiten” wird häufig fallen gelassen. Inhaltlich gesehen sicher richtig. Aber animiert Dich das auch wirklich etwas zu unternehmen? Damit Du in 20 oder 30 Jahren “fit” bist? Das Gehirn funktioniert einfach anders. So abstrakte Belohnungen erreichen uns nur auf logischer Ebene. Und die reicht für eine Verhaltensänderung nicht aus.

Plakatives Beispiel: Wofür entscheidest Du Dich, wenn Du ohnehin gerade müde und schlapp bist: Dafür mit Sport möglichen Problemen vorzubeugen, die sich erst in einigen Jahren bemerkbar machen? Oder doch lieber für das Sofa, möglicherweise ein leckeres Eis oder eine andere "Belohnung"? Vielleicht zusammen mit einem Film oder Deiner Lieblingsserie auf Netflix? Vielfach ist die Antwort klar. Und genau deshalb ist dieses Kapitel in das Buch eingeflossen.

Perspektivwechsel. Weg mit abstrakten Vorteilen. Hin zu Verbesserungen im “Hier und Jetzt”. Es gibt ein schönes Zitat von Jürgen Klopp:

Ich glaube, dass Dich die Lust auf's Gewinnen eher zu einem Sieger macht, als die Angst vor dem Verlieren.

Und genau das ist in diesem Kapitel Programm. Denn Sport und Training haben ganz konkrete Auswirkungen auf das Hier und Jetzt! Auf Dein Wohlbefinden und die innere Zufriedenheit. Sicher, es hilft Dir vielleicht auch "im Alter".

Aber wer kann das schon im Detail belegen. Vergleiche sind nur schwer möglich. Was ich aber genau kenne sind meine eigenen Erfahrungen. Ich kenne beide Seiten der Medaille. Ich habe jahrelang täglich am Computer gesessen und Bewegung nahezu ganz vermieden. Gar nicht unbedingt bewusst. Es hat sich einfach so entwickelt. Anders ausgedrückt: Es ist zur Gewohnheit geworden, nur noch am Computer zu sitzen. Ich hatte viele Ideen, viel Arbeit, war vom Computer fasziniert und habe jede freie Minute im Internet verbracht. Nachdem ich die erste Flatrate bekommen habe, war ich kaum noch zu bremsen. Die guten Ratschläge wollte ich nicht hören. Lieber meine eigenen Fehler machen. Das hätte früher mein Motto sein können. Ein Dozent von mir formulierte es mal als "Lernen durch Schmerz" (er bezog sich klar auf psychischen, weniger den physischen Schmerz).

Nach der langen Zeit am Rechner kam ein Punkt, an dem etwas passieren musste. Weniger medizinische Notwendigkeit. Ich wurde immer unzufriedener, hatte zugenommen und fühlte mich unwohl. Mein Normalgewicht liegt bei etwa 75kg. Als ich schließlich 96,7kg auf der Waage sah, stand der Entschluss fest. Ich möchte niemals 100kg wiegen. Was dann folgte war Schritt für Schritt eine Transformation. Begonnen hat alles mit regelmäßigem Sport. Hinzu kam eine sinnvollere Ernährung und eine andere Lebensweise.

Alles was nun folgt erzähle ich, um auf messbare Ergebnisse in der Gegenwart hinzuweisen. Auf Vorteile und Verbesserungen für Deinen Alltag. Maßnahmen die Dich jetzt

gleich zufriedener, ausgeglichener und belastbarer machen können. Ich finde dieses Kapitel ist Pflichtlektüre für jeden, der sich langfristig an einen Computer binden will! Der gesundheitliche Aspekt schwingt natürlich immer mit, steht aber vollständig im Hintergrund.

Weniger Kopfschmerzen und Ibu Profen

Ein großer Vorteil, den mir regelmäßiger Sport gebracht hat: Mein Verbrauch von Ibu Profen tendiert mittlerweile gen Null. Sicher, ganz ausschließen lassen sich Kopfschmerzen nie. Aber früher hatte ich wirklich regelmäßig Verspannungen und Kopfschmerzen. Pro Monat eine Packung Ibu Profen ist nicht aus der Luft gegriffen. Im Durchschnitt traf das mit Sicherheit zu. Ich habe davon viel zu viel genommen.

Der physiologische Hintergrund ist vielfältig. Wichtigster Effekt: Durch die Haltung am Computer verkürzt - wenn Du nicht durch Krafttraining dagegen arbeitest - die Brustmuskulatur. Auch die Schultermuskulatur wird in Mitleidenschaft gezogen. Zudem verändert sich die hintere Beinmuskulatur in der Wade und dem Oberschenkel. Insgesamt wirkt die Haltung - bei fehlendem Training - ein wenig "in sich zusammengefallen". Es fehlt Grundspannung, die Dich automatisch in einer besseren Form hält. Durch diese Fehlhaltung gehen die Schultern nach vorn, die Brust nach hinten. Dabei muss es genau umgekehrt sein: Brust raus, Schultern nach hinten.

Und das ist ein wichtiger Punkt. Krafttraining ist, speziell im

Punkt Verspannungen, ein wichtiger Ausgleich. Das weiß ich aus schmerzhafter Erfahrung zu berichten. Im Rahmen von einem Ratgeber zum Thema "Entwickler werden" lässt sich keine konkrete Trainingsempfehlung geben. Aber wie zum Kapitel über Gewohnheiten beschrieben: klein Anfangen. Beginn mit Liegestützen und mach die regelmäßig. Mittelfristig wäre sicher ein Ganzkörpertraining zu empfehlen. Dazu reichen schon zwei Trainingstage pro Woche. Wer hier mehr Informationen benötigt oder Rat sucht, darf sich gern direkt an mich wenden. Schau einfach auf codingtutor.de vorbei. Dort findest Du alle Kontaktmöglichkeiten.

Keine Schmerzen mehr im Handgelenk

Weiterer Nebeneffekt: Wenn Deine Hände den ganzen Tag in einer weitestgehend starren Haltung bleiben, kann das zu verschiedenen Problemen im Bereich des Handgelenks führen. Ganz häufiges Problem bei Computerarbeit: das sogenannte "Karpaltunnelsyndrom". Es beginnt häufig mit einem gelegentlichen "einschlafen" der Hand. Ein Taubheitsgefühl breitet sich aus, ähnlich wie bei einem "eingeschlafenen Fuß". Die Ursache ist der leichte aber ständige Druck, der auf den sogenannten Karpaltunnel einwirkt. Die Folge kann eine Schädigung am Nerv sein. Soweit ist es bei mir zum Glück nie gekommen.

Es gibt aber weitere Probleme, die eine reale Bedrohung darstellen. Ich hatte häufig im Bereich der Gelenkkapsel Probleme. Die statische Haltung an der Tastatur hat regelmäßig Probleme verursacht. Teilweise so stark, dass ich nicht mehr

ohne stützenden Verband oder Kinesotapes weiterarbeiten konnte. Schon leichtes drehen der Hand hat Schmerzen bereitet. Eine physiologische Erklärung kann ich hier nicht liefern. Die Ursache wurde nie abschließend bestimmt. Aber: Die Physiotherapeutin vermutet, dass durch die fehlende Belastung Muskulatur fehlte und gleichzeitig die Durchblutung nicht ausreichend war.

Keine Rückenschmerzen mehr

Ähnliches gilt für Rückenschmerzen und Verspannungen. Wie erwähnt hatte ich oft Kopfschmerzen. Ebenso hatte ich häufiger das Vergnügen mit einer verkrampften Rückenmuskulatur. Besonders häufig trat das Morgens auf. Irgendwie falsch gedreht, zum Beispiel beim An- oder Ausziehen von T-Shirt oder einem Pullover. Schon war es passiert. Verspanungen im Bereich der Schultern und dem Nacken, den ganzen Tag lang.

Die logische Folge sind Schmerzen. Die Empfehlung der Ärzte und Physiotherapeuten ist ganz klar: eine Schonhaltung vermeiden. Leichter gesagt als getan. Auch hier war häufig Ibu Profen gegen die daraus folgenden Spannungskopfschmerzen notwendig. Abends half dann Wärmebehandlung mit ABC-Salbe und Heizdecke. Tagsüber waren es mitunter die schlimmsten acht Stunden, die ich jemals in einem Büro verbracht habe.

Dieses Problem hat regelmäßiges Krafttraining nahezu ausgeräumt. Die Medizin weiß zwar nicht genau, was in der

Muskulatur im Detail geschieht. Aber es hängt vieles mit der fehlenden Belastung und Beanspruchung. Durch regelmäßiges Training bleibt sie flexibel und wird dynamischer, wird besser durchblutet und ist weniger anfällig für diese Verkrampfungen, die auch durch Kälte und Zugluft begünstigt werden können.

Energie und Konzentration

Ein ganz entscheidender Vorteil ist wortwörtlich die Ausdauer. Joggen hat mir eine ganz neue Qualität von Konzentration und geistigen Leistungen ermöglicht. Während ich früher nach Feierabend häufig auf dem Sofa gestrandet bin, arbeite ich heute neben der normalen Arbeit im Büro auch Zuhause an Projekten. Sei es dieses eBook, das Blog codingtutor.de oder eines der Videotrainings. Auch auf Youtube bin ich aktiv und außerdem für Kunden als Freelancer tätig. Darüber hinaus habe ich trotzdem, oder gerade deshalb, noch Kraft für Sport. Und wenn das für mich gilt, dann mit Sicherheit auch für Dich.

Auch rein psychisch bringt Sport Vorteile. Es wird oft vom geistigen Ausgleich zum stressigen Alltag geredet. Das klingt fast wie ein Klischee. Es steckt aber unendlich viel Wahrheit darin. Durch den Wald oder an einer einsamen Straße endlang Joggen, es gibt nicht viel das ähnlich beruhigend wirken kann. Mit einem Spaziergang oder einer Wanderung ist das nicht vergleichbar.

Gerade in einem ruhigeren Tempo kann ich wunderbar abschalten. Aber auch intensive Läufe sorgen für ein tolles

Gefühl - gerade nach dem Lauf. Grenzerfahrungen bei denen die Lunge brennt, Atem knapper wird und die Beine langsam schwer werden - sicher gewöhnungsbedürftig. Speziell für Anfänger nicht zu empfehlen. Aber Du lernst mit der Zeit, dass der Zustand nur kurz andauert. Was anschließend hormonell im Körper passiert ist mit Worten nicht zu beschreiben. So haben alle Aspekte im Training einen eigenen Reiz. Welcher davon für Dich der richtige ist? Probier es aus! Mehrmals. Lass Dich von dem ersten Eindruck nicht täuschen.

Bestätigung zu meinen Beobachtungen bieten auch die Bücher *Your Brain At Work* sowie *The Winners Brain*. In beiden wird zur optimalen Unterstützung der Hirnfunktion dreimal pro Woche jeweils 30 Minuten Sport empfohlen. Das sichert die notwendige Durchblutung des Gehirns. Und die ist notwendig, damit es optimal mit Sauerstoff und Nährstoffen versorgt werden kann.

Mentale Stärke

Das wertvollste, das mir Sport gegeben hat, ist mehr mentale Stärke. Gerade das Marathon Training und die anschließenden Läufe über 42,195km haben mir eines vermittelt: aufgeben ist keine Option! Klingt pathetisch. Einmal selbst erlebt, verbindest Du damit aber nicht mehr nur Worte. Es wird ein Lebensgefühl. Das lässt sich nur schwer in Worte verpacken oder logisch vermitteln. Vermutlich muss man es einfach erleben. Tatsache ist: Du führst ab einem Gewissen Punkt einen Kampf gegen Dich selbst. Im Training können die wöchentlichen Läufe über 30 bis 35km am Ende vom

Trainingsplan an die Substanz gehen. Spätestens beim Marathon selbst kommst Du an den Punkt, an dem weiterlaufen zum Kampf wird. Es heißt nicht umsonst, dass der Marathon eigentlich erst bei Kilometer 35 beginnt. Es kann sogar passieren, dass Du Dich die letzten Kilometer ins Ziel zwingen musst. Weiter laufen. Weitere Schritte machen. Nicht aufgeben. Deshalb wird der Marathon auch der Mount Everest des kleinen Mannes genannt.

Klingt alles nicht nach relevanten Informationen für Entwickler? Weit gefehlt! Ich verspreche Dir: Wenn Du es ins Ziel schaffst, kommt dort ein anderer Mensch an. Früher oder später lernst Du diese Erfahrungen auf die Realität anzuwenden. Und das gilt nicht nur bei einem Marathon. Du kannst auf vielen Wegen solche Grenzerfahrungen sammeln. Wenn Du die durchstehst, wird das Dein Leben bereichern.

Der Effekt auf den Alltag übertragen: Ein anstrengendes Projekt braucht länger? Es geht irgendwann vorbei. Du musst Abends oder am Wochenende arbeiten, während andere Feiern? Warum aufgeben - jetzt wo Du schon so viel investiert hast? Eigentlich bist Du müde und möchtest gern auf das Sofa. Geht nicht, damit Aufgabe *xyz* fertig wird? Schalt im Kopf auf Kilometer 35. Pain is Tempory!

Du wirst merken, dass ein unbequemes Gefühl bei der Arbeit nicht mehr "das Ende der Welt" bedeutet. Aber gerade dann ist mentale Stärke gefragt. Du bist in der Lage durch pure Willenskraft wirkliche "Durststrecken” durchzustehen. Einfach weiter machen, auch wenn Du nicht mehr kannst. Es

durchziehen, obwohl Du wirklich keine Lust mehr hast. Nicht aufgeben wird, auch ganz ohne Marathon, schließlich zur Gewohnheit!

Schon Steve Jobs wusste Mindfulness-Meditation zu schätzen und hat sie regelmäßig praktiziert. Übersetzt bedeutet das soviel wie Achtsamkeits-Meditation. Eine unglaublich mächtige Übung, die für jeden Menschen eine Bereicherung sein kann. Du wirst nach kurzer Zeit sehr viel ausgeglichener. Außerdem bekommst Du einen Einblick in den inneren Dialog, den Du ständig mit Dir selbst führst. Du sollst und wirst Deine Gedanken dadurch nicht stoppen. Aber das regelmäßige Training hilft Dir, eine andere Beziehung zu ihnen aufzubauen.

Oft fühlen wir uns unruhig, unwohl oder laufen Emotionen hinterher. Uns ist gar nicht bewusst wie sehr uns Ängste, Ärger, Wut oder auch angenehme Gedanken beschäftigen. So unterschiedlich die Emotionen auch sein mögen, eines haben alle gemeinsam. Sie lenken Dich vom jetzigen Moment ab. Regelmäßige Meditation hilft Dir genau diese innere Unterhaltung bewusster wahrzunehmen. Schon 10 Minuten pro Tag reichen. Du wirst viel besser verstehen was Dich beschäftigt. Noch wichtiger ist, dass Du nachweislich Stress reduzierst und in schwierigen Situationen die Übersicht behältst. Die Effekte auf das Gehirn wurden sogar in sogenannten FMRI-Scans nachgewiesen. Ein Effekt ist auch, dass das Stresshormon Cortisol nach einiger Zeit reduziert wird. Du bist wortwörtlich weniger “gestresst”. Und das beste ist die schnelle Umsetzung. Positive Auswirkungen stellen sich bereits nach den ersten 60 Minuten Meditation ein. Bei 10 Minuten täglicher Meditation also schon nach wenigen Tagen.

Es hat nichts mit Religion zu tun

Wenn ich mit Leuten über Meditation spreche, denken viele direkt an religiöse Bräuche oder das bekannte Mantra “ommm”, das ständig wiederholt wird. All das hat mit Mindfulness nichts zu tun. Auch sehr häufige Reaktion: Die Vermutung totale Isolation und vollständige Ruhe sei notwendig. Das ist aber nicht der Fall. Im Gegenteil. Die Idee ist es nicht die Umwelt auszublenden oder künstliche Situationen zu schaffen. Das Ziel ist das Hier und Jetzt bewusst wahrzunehmen. Dazu gehören auch die Geräusche. Und darum hat jeder die Zeit und die passenden Umstände.

Du brauchst nur die Möglichkeit Dich für zehn Minuten zurückzuziehen. Theoretisch kannst Du sogar unter Menschen meditieren. Viele Leute machen dies im Bus oder in der Bahn. Du musst Dich aber in der Gegenwart anderer Menschen mit geschlossenen Augen wohl fühlen. Mir persönlich gefällt das weniger.

Volle Präsenz im Hier und Jetzt

Die Achtsamkeit, wie Mindfulness übersetzt heißt, bezieht sich auf Präsenz im aktuellen Moment. Wenn Du mit einigen oder allen Sinnen Deine aktuelle Situation spürst, bist Du achtsam. Das können Gerüche, Geräusche, Geschmack oder das Gefühl auf Deiner Haut sein. Ein Fixpunkt, auf den wir uns in jedem Moment beziehen können, ist unsere Atmung. Sie ist immer da. Außerdem kann der Körper ganz alleine atmen, ohne bewusste Steuerung. Perfekt für die Meditation.

Früher habe ich Einsteigern und Laien die Mindfulness-

Meditation als bewusstes Atmen beschrieben. Aber das ist nicht ganz richtig. Es geht nicht darum den Atem zu verändern. Du beobachtest ihn einfach nur. Das kannst Du ganz einfach ausprobieren. Setz Dich zunächst auf einen Stuhl. Komm einen Moment zur Ruhe. Spür wie Du sitzt und Dein Körpergewicht auf den Stuhl drückt. Fühl wie Deine Hände auf dem Schoß oder den Beinen liegen. Hör welche Geräusche um Dich herum existieren. Es geht weniger um bewusstes zuhören. Es geht viel mehr darum Geräusche wahrzunehmen, die Dich ohnehin erreichen. Nicht interpretieren oder überlegen wo sie herkommen. Einfach den Geräuschen erlauben gehört zu werden. Dann atmest Du dreimal bewusst tief ein und langsam wieder aus.

Nun schließt Du die Augen. Der Atem kehrt zu seinem natürlichen Rhythmus zurück. Das ist wichtig. Du atmest nicht bewusst. Achte einfache darauf, wie Dein Körper von alleine atmet. Anfangs ist das leichter gesagt als getan. Probier es einfach aus - ohne dabei zu verkrampfen. Achte beim Einatmen darauf wie sich der Brustkorb hebt und die Lunge sich mit Luft füllt. Beim Ausatmen fühlst Du wie sich der Brustkorb wieder senkt, die Luft entweicht und der Körper insgesamt wieder etwas entspannt. Nach ein paar Atemzügen machst Du die Augen wieder auf. Das ist die sogenannte Atem-Meditation (Breathing Meditation).

Um über fünf oder zehn Minuten die Konzentration zu unterstützen, kannst Du die Atemzüge zählen. Immer wenn Dein Körper einatmet zählst Du eins, beim ausatmen zwei.

Sobald Du wieder einatmest drei, beim ausatmen vier. Und so geht es weiter. Achte auch ganz bewusst auf die Atempausen, die Dein Körper von alleine macht. Sobald Du bis zehn gezählt hast, startest Du wieder bei eins. So hältst Du die Konzentration aufrecht.

Denn nun kommt die Achtsamkeit ins Spiel: Wenn Du auf den Atem achtest, wirst Du unweigerlich in Gedanken abschweifen. Das ist ein ganz normaler Prozess. Vielleicht erwischst Du Dich, wie Du unbewusst bist 30 gezählt hast? Oder vielleicht hast Du auch ganz aufgehört zu zählen? Wie Du den Faden verloren hast ist egal. Sobald Du es bemerkst, denkst Du Dir ganz locker "Oh, Ok. Ich war abgelenkt". Wichtig ist, Dir keine Vorwürfe zu machen. Das gedankliche Abschweifen ist ein ganz normaler Prozess. In der westlichen Welt trainieren wir unseren Geist in der Regel nie auf diese Art und Weise. Er ist es daher über Jahre gewohnt abzuschweifen.

Sobald Du Dich eingefangen hast, fängst Du bei eins wieder an zu zählen. So einfach, und doch so schwer. Aber keine Sorge: Mit genug regelmäßiger Übung merkst Du irgendwann immer schneller, wenn Du in Gedanken verloren bist. Und genau das überträgt sich schließlich auch auf Deinen Alltag. Du wirst häufiger im aktuellen Moment präsent sein.

Meditation gegen Screenapnoe

Es gibt ein Phänomen mit dem Namen "screen apnea", übersetzt "Bildschirm-Apnoe". Dabei hält Du für einige Sekunden am Bildschirm die Luft an. Speziell das Ausatmen

wird unbewusst verzögert, wenn Du Dich stark konzentrierst. Allgemein kommt es häufig vor, dass Du unbewusst verkrampfst und total angespannt am Rechner sitzt. Das merken wir oft gar nicht. Es gibt aber umfangreiche Untersuchungen dazu. Versuch einfach mal darauf zu achten.

Unter anderem dieses Problem bekämpfst Du mit Achtsamkeitsmeditation. Dir fällt einfach auf, wenn Du kurz nicht mehr atmest, in Gedanken bist oder verkrampft vor dem Rechner sitzt. Und sobald Du es merkst, kannst Du auch dagegen ansteuern.

Mit Mindfulness zu innerer Zufriedenheit

Achtsamkeit trägt zudem unglaublich zu innerer Zufriedenheit bei. Wenn Du im aktuellen Moment präsent bist, kannst Du den einfach wahrnehmen wie er ist. Du wirst merken, dass oft gar nicht die Umstände oder die Situation selbst schlimm ist. Viel schlimmer ist unsere Sicht auf die Dinge. Das klassische Beispiel: Du möchtest eine Frau ansprechen. In Gedanken hast Du schon viele Szenarien durchgespielt. Da gibt es Dialoge mit Gedankensprüngen wie den folgenden:

“Ich frage Sie - Sie lehnt mich vielleicht ab? - Sicher lehnt Sie mich ab - Warum lehnen mich nur alle Frauen ab? - Was habe ich an mir, dass mich so unattraktiv macht? - Warum bin ich so wenig attraktiv? - Ich muss mehr Sport machen? - Ja, ich hab zugenommen” ...

Aber auch das Gegenteil ist denkbar. In der positiven Variante hast Du Dir vielleicht sogar schon das erste Date, eine gemeinsame Zukunft, den ersten gemeinsamen Urlaub, eine Hochzeit oder gemeinsame Kinder vorgestellt. Klingt abwegig? Ist es wirklich nicht. Diese Denkprozesse laufen teilweise innerhalb von Sekunden ab.

Das Problem ist nicht, dass Du nun der fremden Person "Hallo" sagst und Dich vorstellst. Ein trivialer Vorgang. Schlimmer sind die vielen Gedanken, Deine Interpretation der Situation. Erst das macht Dich nervös. Das Achtsamkeitstraining hilft Dir, Dich auf den jetzigen Moment zu konzentrieren. Du gehst auf die Frau zu. Achte auf Deinen Atem, das Gefühl Deiner Schritte auf dem Boden, die Geräusche im aktuellen Moment. Vielleicht gibt es auch besondere Gerüche? Alles was Du mit Deinen Sinnen im aktuellen Moment wahrnehmen kannst, hilft Dir achtsam zu sein. Du bist voll im "Hier und Jetzt" präsent.

Geführte Meditation von Headspace

Zugegeben ist eine Erklärung der Technik über Text nicht wirklich optimal. Gerade zu Beginn ist es nicht leicht zu bemerken wenn Du gedanklich abschweifst. Ich möchte Dich in diesem Buch mit dem Konzept bekannt machen. Der beste Weg mit unmittelbaren Ergebnissen sind geführte Meditationen. Die sind für den Einstieg viel besser geeignet als ein Kapitel in einem Ratgeber wie diesem. Ich wünschte, mich hätte zu Beginn meiner Karriere jemand darauf hingewiesen.

Mein absoluter Favorit bei geführten Meditationen ist Headspace. Es handelt sich um eine App, in der Du verschiedene Programme durchlaufen kannst. Für den Einstieg reicht schon das Take10 Programm. Es läuft über 10 Tage und ist vollständig kostenlos. Das besondere ist der Gründer *Andy Puddicombe*. Er ist ein Experte auf dem Gebiet der Mediation und stammt ursprünglich aus England. Er hat aber mehrere Jahre in verschiedenen buddhistischen Klostern dieser Welt gelebt. Dort hat er von Mönchen gelernt und studiert, wie in den verschiedenen Traditionen meditiert wird. Unter anderem hat er auch den wunderbaren Vortrag *All it takes are 10 mindful minutes* im Rahmen der TED-Reihe gehalten. Außerdem hat er sein Wissen auf die westliche Welt übertragen. Das Ergebnis ist das Headspace Programm, ganz ohne religiöse Hintergründe.

Mindfulness bei Google

Das Thema Achtsamkeit ist noch nicht in der Allgemeinheit angekommen. Aber es hat viele berühmte Unterstützer. Wie ich eingangs erwähnt habe, war Steve Jobs großer Befürworter. Aber auch weitere Kandidaten in der IT-Welt sind aufgesprungen. Zum Beispiel Google! Der Empfangschef, Chade-Meng Tan, hat dazu ein tolles Buch verfasst: *Search Inside Yourself*. Darin geht es um eben genau diese Thematik, um Mindfulness.

Und bei Google geht es sogar noch weiter. Search Inside Yourself war bzw. ist ein Kursangebot bei Google. Viele Entwickler und Angestellte nutzen es. Wie Du siehst, ist das

kein Nischenthema für abgedrehte Menschen. Es gibt ganz ernsthafte Anwendungsfälle. Stell es Dir wie sportliches Training vor. In diesem Fall regst Du nicht Deine Muskeln, sondern Dein Gehirn zur Optimierung an.

Ich kann von Herzen empfehlen: Probier es einfach aus. Gib der Idee etwas Zeit. Damit erste Änderungen spürbar werden, braucht es ca. eine Woche. Du musst täglich nur 10 Minuten investieren, dafür aber regelmäßig und jeden Tag. Es würde mich freuen, wenn es Dein Leben genauso bereichert wie meins.

Die häufigste Frage, die mir über codingtutor.de gestellt wird: Welche Programmiersprache soll ich zuerst lernen? Oder etwas allgemeiner: Wo fange ich an? Ich stelle immer die Gegenfrage nach dem jeweiligen Ziel. Was möchte der Einzelne erreichen? Hat er konkrete Pläne? In diesem Kapitel findest Du einige Anregungen und Fragen, die Du Dir selbst beantworten kannst. Nach den folgenden Seiten kannst Du Dich mit Sicherheit besser orientieren.

Eins ist sicher: Der Einstieg in die Programmierung gelingt Dir mit nahezu jeder Sprache. Manche haben eine flachere Lernkurve. Andere hingegen sind etwas komplizierter. Erste Ergebnisse lassen vielleicht länger auf sich warten. Neben dem Schwierigkeitsgrad finde ich es aber wichtiger die richtige Richtung zu wählen. Wenn Du ein Rennauto fahren möchtest, lernst Du ja auch nicht zuerst wie Du Lastwagen fährst.

Am Ende hat jede Möglichkeit ihre Vor- und Nachteile. Daher empfehle ich unbedingt zu überlegen, bevor Du Dich blind in das Abenteuer stürzt.

Welcher Bereich soll es sein?

Es gibt eine ganze Menge Programmiersprachen. Der Bereich in dem Du eine Idee umsetzen willst, oder Deine Zukunft siehst, steht bei der Auswahl im Vordergrund. Das schränkt Deine Möglichkeiten weiter ein. Unterscheiden musst Du dabei im wesentlichen zwischen den folgenden Kategorien:

- Webentwicklung

- Appentwicklung (iOS oder Android?)
- Anwendungsentwicklung
- Serveranwendungen
- Systemnahe Entwicklung (Betriebssysteme, Treiber)

Die Grenzen verlaufen nicht immer starr, sondern teilweise sehr fließend. Webentwicklung kann auch mit der Umsetzung von Apps einhergehen und umgekehrt. Du kannst mit HTML/CSS und JavaScript mobile Anwendungen erstellen. Die werden in einer nativen App “eingebettet”. Plattformspezifischer Code ruft eine Webseite ab und stellt diese im Vollbildmodus dar. Fertig ist die hybride Anwendung. Vielleicht merkst Du gar nicht, dass sich auch auf Deinem Smartphone bereits solche Anwendungen verstecken.

Zudem können Sprachen für die Anwendungsentwicklung auch teilweise zur App-Entwicklung genutzt werden. Gute Beispiele sind Java, Objective-C oder auch Swift. Auch mit C# kannst Du Anwendungen unter Windows sowie Apps für das Windows Phone entwickeln.

Serveranwendungen sind nicht die besten Projekte für Einsteiger. Neben Sprachkenntnissen gehören in der Regel auch Netzwerkprotokolle wie HTTP und deren Definition dazu. Dadurch wird die Lernkurve deutlich steiler. Aber wenn Du in die Richtung gehen willst, können C/C++ oder Java eine gute Wahl für den Anfang sein. Auch Python wird hier eingesetzt. Mit allen Sprachen kannst Du richtig viel Spaß haben.

Außerdem sind alle Sprachen auch gleichermaßen für andere Szenarien einsetzbar.

Zu guter letzt kann auch systemnahe Programmierung aufregend sein! Assembler ist vergleichsweise trocken. Hier steht die Interaktion und Steuerung von Hardware im Fokus. Aber zum Beispiel in Verbindung mit Elektro-Bausätzen wie Arduino kann es sehr lebendig werden. Auch andere Mikrocontroller-Pakete versprechen viel Spaß zu machen. Du kannst über USB sogar Geräte steuern. Ein oft unterschätzter Tipp: Mit Lego-Mindstorms steht ein wirklich spannender Baukasten bereit, der auch für “große Kinder” interessant ist. Nicht umsonst wird er auch an vielen Universitäten eingesetzt.

Welche Sprache zuerst?

Diese erste Übersicht beantwortet Deine Frage vermutlich nicht. Vielleicht bist Du jetzt sogar noch mehr verwirrt als vorher? Ich stelle die Frage einfach mal anders. Was möchtest Du erreichen? Die Antwort führt Dich automatisch näher ans Ziel.

Die optimale Programmiersprache für den Einstieg ist vor allem eins: das richtige Werkzeug für Deine Ideen. Wenn Du Nägel versenken willst, lernst du ja nicht mit einem Schraubenzieher umzugehen. Es gibt auf die Frage keine simple Antwort. Es kommt maßgeblich auf Deine Ziele an. Auf den folgenden Seiten führe ich Dich nun Schritt für Schritt zu Deiner individuellen Antwort.

Die optimale Sprache

Der wichtigste Aspekt bei der Auswahl einer Sprache ist Spaß am Spiel! Du musst Freude bei der Arbeit haben. Ohne Spieltrieb und Spaß an Experimenten wirst Du nicht lange am Ball bleiben. Dein erstes Projekt wird nicht perfekt. Auch nicht das Zweite oder Dritte. Bis zur Umsetzung Deiner Millionen-Dollar-Idee braucht es viele Anläufe. Vorab stehen immer wieder kleine Projekte auf dem Programm. Du musst Dich Schritt für Schritt entwickeln. Von Projekt zu Projekt besser werden. Anders ausgedrückt ist der Weg das Ziel. Nur wenn der Spaß machst, kommst Du überhaupt an.

Die einzelnen Disziplinen

Auf den folgenden Seiten lernst Du die einzelnen Bereiche der Software-Entwicklung kennen. Besonders wichtig für Dich sind die Sprachen. Wenn Du bereits eine Projektidee hast, schau in welche Kategorie sie fällt und probier eine der genannten Programmiersprachen aus. Es kommt nicht auf Perfektion an. Schau ob es Dir gefällt. Verlass Dich auf Dein Gefühl.

Wenn Du Dich beruflich orientieren willst ist es natürlich schwieriger. Ich behaupte, dass es nicht möglich ist abzuschätzen wie interessant ein Beruf nach einigen Jahren noch ist. Dem kannst Du Dich nur annähern. Wenn Du zum Beispiel mit dem Bloggen anfangen möchtest, brauchst Du zuerst ein zentrales Thema. Bei mir steht auf codingtutor.de die Programmierung im Zentrum. Ob Du Deinem Blog über

lange Zeit treu bleibst und das Projekt aktiv bleibt, ist vorab nicht abzuschätzen. Aber es gibt eine Faustregel. Bevor Du ein Blog startest, solltest Du in ein oder zwei Stunden mindestens 50 Ideen für neue Artikel generieren können. Aus dem Bauch heraus. So ähnlich kannst Du auch bei der Wahl von Deinem Fachbereich vorgehen. Überleg Dir 20-30 Projekte, die Du interessant fändest. Bei Apps und Webprojekten habe ich nahezu endlos viele Ansätze. Bei hardwarenaher Entwicklung müsste ich mich wirklich von Idee zu Idee "quälen". Keine guten Vorzeichen.

Sammle einfach Einfälle. Überleg Dir welche Software Du schreiben möchtest. Es geht nicht darum Startup oder Geschäftsideen zu generieren. Es muss auch nicht einzigartig oder komplex sein. Du darfst Dir ein soziales Netzwerk wie Facebook vorstellen, die nächste Taschenlampen App oder ähnliches. Der Fokus liegt auf dem experimentieren. Wie baue ich mein eigenes Instagram? Ein eigenes Twitter?

Was hier zählt ist nicht der Launch von einem Unternehmen oder einer fertigen Software. Da gehören noch eine ganze Reihe anderer Fähigkeiten zu. Du sollst einfach mal Gedankenspiele betreiben, in die Theorie eintauchen. Wenn Dir die bereits Spaß macht, sind die Chancen auch für die Praxis sehr gut.

Eine relativ neue Facette ist die Entwicklung von Apps. Ermöglicht haben das Steve Jobs und Apple als sie den AppStore für iOS freigegeben haben. Eigentlich handelt es sich bei Apps auch “nur” um Anwendungen. Die Abkürzung steht für “Application” - zu deutsch Anwendung oder Programm. Es ist eine Anwendung wie jede Software auf dem Desktop auch. Es gibt für die Entwicklung aber spezielle Rahmenbedingungen. Darum ist die App-Entwicklung irgendwie ein Sonderfall.

Einstieg über die App-Entwicklung?

Für jeden Einsteiger sind schnelle Ergebnisse optimal. Der greifbare Fortschritt motiviert. Unheimlich! Darum halte ich Appentwicklung für einen perfekten Einstieg. Die verwendeten Sprachen sind vornehmlich Java, Swift und Objective-C. Und die sind gleichzeitig sehr ausgereift. Auch C# kann in der Windows-Welt ein Einstieg sein. Davon bin ich aber persönlich genauso wenig überzeugt wie vom Windows Phone. Die Microsoft-Welt kommt mir zudem bis heute unterkühlt und etwas emotionslos vor. Der Zauber der OpenSource-Community, Leidenschaft und die Liebe zum Spiel sind dort nicht so ausgeprägt vorhanden.

Für die App-Entwicklung generell gilt: Die Lernkurve ist sehr flach. Sobald Du die Syntax verstanden hast kannst Du erste Projekte umsetzen. Die kannst Du direkt Deinen Freunden, Bekannten und Verwandten zeigen, ganz einfach unterwegs. Das ist bei “normalen” Anwendungen auf dem Desktop schwieriger. Insgesamt ein tolles Gefühl, wertvolles Feedback

und Motivation! Fuel your ambition.

Besondere Anforderungen an die Software

Für Entwickler bedeutet Appentwicklung vorhandene Ressourcen effizient zu nutzen. Die stehen auf mobiler Hardware nämlich im Überfluss zur Verfügung. Markanter Unterschied ist zum Beispiel die Darstellung. Es gibt Smartphones mit hochauflösendem Display. Mit einer Desktopanwendung ist aber auch das kein Vergleich.

Auch Arbeitsspeicher ist knapp. Aktuelle Smartphones werden zwar weiter verbessert. Es ist auch schon viel geschehen. Aber im Vergleich zu einem Desktopcomputer sind die Möglichkeiten, aus nachvollziehbarem Grund, begrenzt. Effizientes Haushalten mit RAM ist also ein Muss. Gleiches gilt für die CPU-Zeit. Je mehr in Anspruch genommen wird, desto schneller ist der Akku leer. Darum belasten aufwendige Spiele die Akkulaufzeit deutlich stärker als zum Beispiel ein Browser.

Eine weitere Aufgabe: Die Anforderungen an Hard- und Software dürfen sich nicht am letzten Stand der Technik orientieren. Um ein großes Publikum anzusprechen, muss Deine App auf einem durchschnittlichen Smartphone lauffähig und nutzbar sein - nicht nur mit State-of-the-Art Geräten oder dem letzten Software-Update. Alles das sind Überlegungen, die bei der Veröffentlichung eingeplant werden müssen.

Großer Spaß garantiert

Die Einschränkungen sind kein großer Nachteil, gerade für

Anfänger. Sie schulen sogar den Sinn für Details. Wenn Du über nahezu unbegrenzte Ressourcen verfügst, lernst Du vermeintlich nicht sie verantwortungsbewusst einzuteilen. Zumindest ist der Leidensdruck dahinter nicht besonders groß. Dementsprechend wirst Du auch nicht unmittelbar mit wichtigen Designentscheidungen konfrontiert.

Zudem sind die Standardbibliotheken und Tools unheimlich ausgereift. Sie sorgen dafür, dass Du Deine Apps optimal auf die Plattform ausrichten kannst. Was vielleicht kompliziert wirkt und nach viel Arbeit klingt ist in der Praxis total spannend und macht eine Menge Spaß! Die gesamte Plattform ist ja bereits auf die besonderen Anforderungen abgestimmt. Du musst als iOS oder Android Entwickler nicht bereits ein Genie im Umgang mit Ressourcen sein. Das lernst Du unterwegs.

Die Entwicklungsumgebungen

Xcode wird in der Apple Welt als Entwicklungsumgebung verwendet. Was für ein geniales Stück Software! Das Interface wird grafisch erstellt und die Elemente direkt mit dem Quellcode bekannt gemacht. Der gesamte Entwicklungsprozess ist unheimlich flüssig. Spannend ist in diesem Umfeld auch die neue Sprache Swift. Die hat es mir wirklich angetan. Sie hat sich seit der Veröffentlichung im Juni 2014 rasant verbreitet. Auch wenn Objective-C schon gut war: Swift setzt der Entwicklung die Krone auf.

To be fair: Diesen Spaß bieten aber auch andere Plattformen.

Die Androidentwicklung hat - spätestens seit das Android Studio offiziell Eclipse als Umgebung abgelöst hat - stark aufgeholt. Es fühlt sich aber trotzdem nicht im gleichen Maße an wie “aus einem Guss”, so wie das bei Xcode und Apple der Fall ist. Trotzdem ist Android eine spannende Plattform. Nicht zuletzt weil darunter ein Linuxkernel arbeitet. Die Sprache der Wahl ist hier Java. Die ist mittlerweile unheimlich ausgereift und Erwachsen geworden. Ich lehne mich soweit aus dem Fenster und behaupte: Android-Apps sind in der Consumer-Welt der interessanteste Anwendungsfall für den Einsatz von Java. Nicht zuletzt dadurch hat die Sprache in den letzten Jahren einigen Aufwind erfahren.

Viele Facetten

Die Vermutung monotoner Arbeit liegt nahe. Aber: Jede App ist anders, nutzt andere Bibliotheken und stellt andere Herausforderungen und Aufgaben. Ähnlich wie die Webentwicklung wird es so schnell nicht langweilig! Sicher nicht. Selbst wenn Du Dich nur auf die iOS Plattform beschränkst, stehen Dir scheinbar endlose Möglichkeiten offen. Bis Du zum Beispiel alle Frameworks der iOS-Plattform genutzt hast vergeht eine ganze Zeit.

Fast noch wichtiger: Es gibt verschiedene Arten. Zum einen die *normalen Productivity-Apps*. Die nutzen die vorhandenen Klassen und stellen ein Interface mit Buttons, Menüs, Tabellen und so weiter bereit. Und damit meine ich nicht einfach Todo-Listen. Dazu gehören auch Apps wie der Musikplayer, die

Sportnachrichten oder der Kalender.

Zudem gibt es den Bereich der Spiele. Deine eigene Kreativität ist die einzige Grenze, zumindest fast. Es gibt nahezu endlose Möglichkeiten. Die Palette umfasst 2D- und 3D-Spiele. Der Komplexität sind in der Theorie keine Grenzen gesetzt. Allein auf diesem Gebiet kannst Du Jahre verbringen, selbst wenn Du auf eine existierende Game-Engine aufbaust. Die Gefahr der eintönigen Arbeit ist konzeptbedingt ausgeschlossen.

Neben nativen Apps kannst Du auch mobile Apps mit HTML, CSS und JavaScript erstellen. Die Kombination aus beiden Welten ist möglich und gängige Praxis. Einzelne Ansichten der App werden in Form einer mobilen Webseite realisiert und dann eingebettet. Ein weiteres Gebiet in das Du Dich einarbeiten könntest.

Zu guter letzt sei gesagt: Die Entwicklung für mobile Endgeräte steht noch am Anfang. Smartphones und Tablets sind gerade erst entstanden. Vom iPhone, dem ersten Smartphone überhaupt, gibt es gerade mal die sechste Generation. Es werden mit Sicherheit noch viele spannende Bereiche hinzukommen. Absolut heiß gehandelt wird zum Beispiel “Augmented Reality”. Dabei verschmilzt die virtuelle mit der realen Welt. Es gibt Apps die Text an Ort und Stelle übersetzen und einblenden: Du richtest die Kamera von Deinem Smartphone auf ein Straßenschild. Der Text wird erkannt, verarbeitet und im Livebild ersetzt. Das Straßenschild erscheint also z.B. in deutscher Sprache. Je leistungsfähiger die

Geräte werden, desto mehr Möglichkeiten bieten sich an.

Früher oder später wird Dir die Webentwicklung begegnen. Selbst wenn Du gar kein Programmierer werden willst. Jeder der im Internet Inhalte veröffentlicht, und seien es nur Texte auf Blogs, benötigt zumindest einige Grundkenntnisse. Es gibt kaum Bereiche die ganz am Web vorbeikommen. Vielleicht braucht Deine App ein zentrales Backend? Oder eine API? Du musst XML-Dokumente im Web abrufen und verarbeiten? Im einfachsten Fall brauchst Du vielleicht nur eine Webseite für ein Projekt?

HTML und CSS gehören daher unbedingt in Dein Portfolio. Es handelt sich allerdings nicht um Programmiersprachen. HTML ist eine Auszeichnungssprache. Es ist eine Beschreibung der Inhalte. So lassen sich Überschriften, Absätze, Listen und ähnliche Textelemente kennzeichnen. Erst JavaScript macht HTML-Dokumente dynamisch. Hier findest Du Elemente wie Kontrollstrukturen, Bedingungen, Funktionen und ähnliches. Gleichzeitig ist die Sprache ein wunderbarer Einstieg für Anfänger. In der Tat biete ich sogar auf codingtutor.de einen 7-tägigen Grundkurs auf JavaScript-Basis an.

Der Weg in die Webentwicklung

Webentwicklung ist ein toller Einstieg in die Programmierung! Erst recht wenn es später ohnehin in diese Richtung gehen soll. Die verwendeten Sprachen wie Ruby, Python und PHP haben eine flache Lernkurve. Erste Ergebnisse sind auch schnell erzielt. Aber: es gilt insgesamt mehrere Disziplinen zu meistern. Eine Skriptsprache ist nur ein Teil der Gleichung. Sie wird im sogenannten Backend eingesetzt, also

auf dem Server. Die Sprache im WorldWideWeb ist aber HTML. Sie wird zusammen mit CSS im Frontend genutzt, im Browser des Besuchers. Und genau das wird auch bei der Webentwicklung mit den oben genannten Sprachen erzeugt, ein HTML-Dokument. Deshalb musst Du für dynamische Webseiten neben HTML und CSS zusätzlich eine Skriptsprache lernen.

Starten als Webentwickler

Wenn Webseiten bzw. Webanwendungen auf Deinem Wunschzettel stehen, sollte hier Dein Fokus liegen. Du kannst mit Sprachen wie Swift, Java oder C/C++ einsteigen. Aber warum Wochen und Monate mit etwas anderem befassen? Verlass Dich auf Dein Bauchgefühl und folg Deiner inneren Stimme.

PHP, Ruby oder auch Python sind Handwerkzeug für Webworker. Sie erzeugen HTML-Code wenn eine Internetseite aufgerufen wird. Und genau den schickt der Webserver schließlich zurück an den Browser. Im Unterschied zu statischen HTML-Dokumenten, die in einer Datei abgespeichert werden, kannst Du Elemente wie Bilder und Texte *on-the-fly* verändern. Daher werden so erzeugte HTML-Dokumente als dynamische Webseite bezeichnet. Du siehst Dich in Zukunft als Webentwickler? Höchste Zeit HTML zu lernen und mit Ruby, Python oder PHP in das Rennen einzusteigen!

Das große Angebot

Spannend ist die große Vielfalt. Webentwicklung bedeutet mehr als nur Informationen für Benutzer aufzubereiten. Es gibt die klassische Webseite. Dabei wird meist mit einem Content Management System (CMS) der Inhalt verwaltet und dargestellt. Typischer Anwendungsfall sind Firmen- oder Imagewebseiten. Das Unternehmen stellt sich und seine Philosophie vor. Es gibt Bilder vom Team, Öffnungszeiten und einen Auszug aus dem Produkt- oder Serviceangebot. Prominente Lösungen in diesem Bereich sind die in PHP geschriebenen OpenSource Anwendungen WordPress oder Typo3.

In der zweiten Kategorie sind Webanwendungen zu finden. Auch hier wird HTML für die Beschreibung der Inhalte im Browser genutzt. Im Backend können sogar die gleichen Sprachen verwendet werden. Aber: Es steckt in der Regel eine ganze Menge Logik im Hintergrund. Bekannte Beispiele sind Facebook (mit PHP geschrieben), die unscheinbar wirkende Suchmaschine Google oder Twitter. Hinter vergleichsweise schlichten und aufgeräumten Benutzeroberflächen verbergen sich unheimlich komplexe Anwendungen oder ganze Rechenzentren, die auf der Welt verteilt betrieben werden. Und genau da steckt die eigentliche Herausforderung, in der Logik dahinter.

Welche Skriptsprache?

Welche der drei Sprachen solltest Du nun lernen? Jede hat

Ihre Vor- und Nachteile, eine eigene Story und eine mitunter leidenschaftliche Community. Ich möchte hier keinen Glaubenskrieg anzetteln. Ich finde grundsätzlich jede der Sprachen spannend und habe bereits mit ihnen gearbeitet. Im Schwerpunkt gehöre ich dem PHP-Lager an. Die Sprache empfehle ich auch Dir für den Einstieg. Der Vorteil ist die hohe Verbreitung. Du wirst immer wieder auf Lösungen stoßen, die in PHP geschrieben worden sind. Allein mit WordPress gibt es einen extrem berühmten Anwendungsfall. Weitere Beispiele sind Magento, Typo3, das Zend Framework, Symfony und Drupal. Mit PHP-Kenntnissen machst Du nichts falsch.

Dennoch möchte ich hier auch andere Sprachen nicht ungenannt lassen. Neben PHP gibt es z.B. auch noch Python als sehr erfolgreiche Skriptsprache. Sie wird für Webprojekte eingesetzt. Ein wirklich tolles Framework für die Webentwicklung gibt es mit Django ebenfalls.

Auch Ruby hat es verdient hier genannt zu werden. In der Startup Szene ist es durch die Firma 37Signals sehr bekannt geworden. Die hat das Framework Ruby on Rails geschaffen. Gerade im Silicon Valley ist es sehr beliebt. Es gibt eine ganze Reihe Firmen die darauf setzen, bspw. Buffer. Und die bewegen damit täglich massive Mengen an Informationen.

Auch JavaScript kannst Du theoretisch nutzen. NodeJS ist ein komplett anderer Ansatz, kann aber theoretisch auch verwendet werden um ein ähnliches Ziel zu erreichen. Empfehlen würde ich es einem Einsteiger aber nicht als erste Wahl für Backend-Lösungen. Neben NodeJS gibt es nicht viel

vergleichbare Anwendungsfälle.

Die Wahl steht Dir frei. Langfristig macht es unbedingt Sinn mehrere der Kandidaten auszuprobieren! Da PHP aber ein Schweizertaschenmesser unter den Sprachen für's Web ist, empfehle ich für den Einstieg weiterhin PHP. Außerdem wurden im Laufe der letzten Jahre viele der Kritikpunkte ausgeräumt.

Der Begriff Anwendungsentwicklung bezeichnet die klassische Entwicklung von Software für Desktopcomputer. Sie ist hochaktuell und begegnet Dir in allen Bereichen: Todo-Listen, Browser, Bildbearbeitung, Videoschnitt und sogar eine IDE wie Xcode oder Android Studio. Kurzum: Jegliche Software auf einem Desktop. Es gibt beliebig viele Möglichkeiten. If you can dream it, you can do it. Schon allein wegen der vielen Möglichkeiten sind Anwendungsentwickler nach wie vor gefragt.

Anwendungen entwickeln

Apps sind zu der hippen Spielwiese für Programmierer geworden. Das ist toll! Wenn sich dadurch noch mehr Menschen für die Software-Entwicklung begeistern, kann es nur gut sein. Aber auch Desktopanwendungen zu entwickeln macht nach wie vor wirklich Spaß. Du kannst nahezu endlos viele Programme umsetzen, genug Ideen vorausgesetzt. Und im Gegensatz zu früher ist es heute auch für den kleinen Entwickler ziemlich einfach Software zu verkaufen. Die Verbreitung und der Vertrieb von Anwendungen ist ähnlich leicht wie bei Apps. Gerade im Mac OS X Umfeld gibt es mit dem AppStore ein nahezu identisches System. Was sich immer mehr ändert, ist die Zielgruppe.

Für wen entwickelst Du?

Noch gibt es einen Markt für Desktoprechner beim Endverbraucher (Consumer). Zahlen belegen aber einen seit Jahren rückläufigen Trend. Für das klassische Onlineerlebnis,

der Mix aus Surfen und E-Mails, benötigst Du keinen Schreibtisch mehr. Es ist viel gemütlicher mit dem iPad auf dem Sofa einzukaufen. Beim Surfen ist ein Tablet mindestens ebenbürtig, beim Lesen längerer Texte oder eBooks sogar überlegen. Typische Consumer-Aufgaben werden immer mehr auf mobile Plattformen verlagert.

Anwendungen auf dem Desktop haben aber weiterhin ihren Platz, speziell im Business Bereich. Firmenkunden und professionelle Anwender können darauf nicht so schnell verzichten. Alle Menschen, die Inhalte erstellen oder pflegen werden zumindest mittelfristig weiter auf leistungsfähige Computer angewiesen sein: Videos schneiden, größere Datenmengen verwalten, Fotos professionell bearbeiten. Auch Entwickler nutzen weiterhin vollwertige Computer. Irgendwo müssen die Apps schließlich entwickelt werden. All dies sind klassische Aufgaben für einen leistungsfähigen Rechner. Es gibt zwar bspw. Tabellenkalkulation für das Tablet. Umfangreichere Dokumente erstellen sich aber eben doch leichter in gewohnter Umgebung.

Zusammengefasst sehe ich gerade für professionellere Anwendungen einen mittel- und langfristigen Markt. Und solange es die oben genannten Schwerpunktthemen gibt, bleiben auch die vielen Tools und Werkzeuge am Rand weiterhin wichtig. Ein *Mobilegeddon* sehe ich für Anwendungsentwickler nicht aufziehen.

Anwendungsentwicklung als Einsteiger?

Anwendungsentwicklung - im klassischen Sinne - ist wichtig und schafft einen tollen Background. Die eigentliche Magie und Kunst verbirgt sich ohnehin hinter dem Interface. Das ist wie bei der Entwicklung von Apps. Die Benutzerschnittstelle ist mehr oder weniger Fassade. Sie steuert die Logik nur. Gerade aus der Sicht ist es fast egal ob die Software auf einem Smartphone oder einem Desktopcomputer läuft.

Anwendungen sind nur einfach aus der Mode gekommen, sind nicht mehr so “sexy”. Gefühlt macht es viel mehr Spaß mobile Anwendungen in einem relativ geschlossenen Mikrokosmos wie der iOS-Welt oder dem Android-Universum zu schreiben. Mit der Pistole auf der Brust vor die Wahl gestellt - Apps oder klassische Anwendungen - würde ich mich als Einsteiger heute vermutlich aber auch zunächst für die Appentwicklung entscheiden.

Auf der anderen Seite ist die Konkurrenz bei Apps natürlich größer. Wenn Du Programmieren lernst um mit eigener Software Geld zu verdienen, ist Anwendungsentwicklung vielleicht die clevere Wahl. Zumindest unter Mac OS X gibt es sicher noch viele Nischen, in denen Du mit Deinen Lösungen etwas bewirken könntest.

Typische Programmiersprachen

Die Verdächtigen sind hier Swift bzw. Objective-C unter Mac OS X. Im Windows-Umfeld sind es eher C++ und C#. Plattformübergreifend steht Dir auch Java zur Verfügung, wobei aber die GUIs nur bedingt hübsch sind. Das Java Swing

Framework bringt nicht die beste Ästhetik mit.

Los geht es mit C++. Es gibt mächtige Frameworks für die grafischen Anwendungen. Dazu zählt unter anderem auch die OpenSource Lösung QT. Sie steht auf verschiedenen Plattformen bereit. Du kannst damit auch unter C++ portable Software entwickeln. Die funktioniert theoretisch unter Windows, Linux und Mac OS X. Plattformunabhängig wie Java ist diese Lösung aber nicht. Zumindest die grafische Oberfläche funktioniert dann mit verschiedenen Betriebssystemen. Hinzu kommt auch die native Windows-Entwicklung über Visual C++.

Java pocht auch auf das Recht hier genannt zu werden. Es gibt eine ganze Reihe Anwendungen, die Java wegen der Plattformunabhängigkeit ausgewählt haben. Speziell im Entwicklerumfeld musst Du gar nicht lange suchen. Viele IDEs (Entwicklungsumgebungen) wie Eclipse, Android Studio, IntelliJ IDEA und viele weitere basieren auf Java. OpenOffice hat ebenfalls einen Java-Anteil in seiner Office-Suite.

Wenn es um C# geht, muss über die starke Bindung an die Windows-Welt gesprochen werden. Die Sprache ist von Microsoft für das eigene Betriebssystem geschaffen worden. Das ist total Ok. Die Entscheidung sollte nur bewusst getroffen werden. Was spricht für C#? Eigentlich nur, dass Du wirklich voll bei der Entwicklung im Microsoft-Umfeld bleibst - Apps für das Windows Phone mit eingerechnet. Insgesamt erinnern viele Aspekte der Sprache an Java. Eine Menge von dem neu erlernten Wissen kannst Du vermutlich später direkt auf Java

übertragen.

Nicht fehlen dürfen Swift und Objective-C. In der Apple-Welt die dominanten Sprachen. Wenn Du neu einsteigst, kannst Du Objective-C hinten anstellen und direkt mit Swift beginnen. Die neue Sprache schiesst aktuell durch die Decke. Und wegen der Kompatibilität zu Objective-C kann Swift mit vorhandenen Bibliotheken zusammenarbeiten. Swift ist für Einsteiger klar die bessere Wahl - sowohl vom Spaß als auch im Bezug auf Zukunftssicherheit.

Interpretierte Sprachen wie Python unterstützen teilweise auch die Anwendungsentwicklung und integrieren grafische Bibliotheken. Auf dem Linuxdesktop gibt es in dieser Kategorie Spezialfälle wie zum Beispiel das Gnome Toolkit (GTK). Mit Python können so relativ schnell und einfach Desktopanwendungen entwickelt werden. Und da gibt es durchaus vorzeigbare Umsetzungen. Eine echte Alternative zu den oben genannten Möglichkeiten ist es aber nicht.

Es gibt neben der Anwenderebene einen weiteren Kosmos: die Serverwelt. Und da ist eine Menge los. Im Unix Umfeld findest Du Dämonen, Zombie-Prozesse, Kill-Befehle und Forkbombs. Harte Bandagen - die IT-Unterwelt! Spaß bei Seite. Das alles bezieht sich vornehmlich auf Begriffe aus dem Unixumfeld. Die Realität dahinter ist nur halb so gruselig wie es sich anhört.

Serveranwendungen entwickeln

Das Betriebssystem des Internets ist Unix. Es wird auf den meisten Servern eingesetzt. Und hier sind OpenSource-Lösungen nahezu Standard. Der sogenannte LAMP-Stack ist die Grundlage für einen extrem hohen Anteil aller Webserver. Die Abkürzung steht für Linux, Apache, MySQL und PHP. Großer Beliebtheit erfreut sich dieses Quartett unter anderem weil es sich um freie Software handelt. Und genau die ist für Einsteiger wie ein großer Schatz. Du kannst nachvollziehen wie eine Software aufgebaut ist und Dich sogar bei der Entwicklung mit einbringen.

Generell lässt sich zwischen zwei Kategorien unterscheiden. Es gibt zum einen Serveranwendungen. Ein klassischer Fall ist der sogenannte *Cron Daemon*. Das ist ein Serverprozess, der ununterbrochen im Hintergrund arbeitet. Er sorgt für die chronologisch gesteuerte Ausführung von Befehlen. Auf der anderen Seite gibt es Netzwerkanwendungen. Genau wie Serveranwendungen arbeiten sie im Hintergrund, also ohne grafische Ausgabe oder direkte Eingabeaufforderung. Netzwerkanwendungen lauschen außerdem auf bestimmten

Ports im Netzwerk und warten auf eingehende Verbindungen. Diese beantworten und bearbeiten sie gemäß festgelegter Protokolle. Der Apache Webserver ist eine solche Anwendung. Er lauscht bspw. Auf dem HTTP-Port 80 (TCP) und verhält sich konform zum HTTP-Protokoll.

Die Sprachen

Die vorherrschende Sprache ist, speziell im Linux und Unixumfeld, nach wie vor C. Große Teile von Unix sowie auch der beliebte Compiler "gcc", der Linuxkernel und endlos viele andere Werkzeuge sind darin geschrieben. Wenn Du in dieser Richtung aktiv werden möchtest, gehört C definitiv in Dein Portfolio. Ist es deshalb auch die richtige Sprache für den Einstieg? Sie hat sicherlich keine flache Lernkurve. Die Arbeit mit C ist aber unglaublich lehrreich! Außerdem hast Du dann eine solide Grundlage für eine ganze Reihe anderer Programmiersprachen. C++ wäre eine logische Konsequenz. Aber auch bspw. C#, Java, PHP, Swift und Objective-C haben eine ähnliche Syntax.

An dieser Stelle ist mittlerweile auch JavaScript zu nennen. Das Framework NodeJS wird für Server- und Netzwerkanwendungen genutzt. Es handelt sich dabei um autark laufende Netzwerkprozesse, die mit JavaScript geschrieben werden. Die Entwicklung geht daher wirklich unglaublich schnell. Insbesondere kleinere Webserver oder Prozesse zur Systemautomatisierung bauen auf dieser Technologie auf.

Python und Ruby haben ebenfalls jedes Recht hervorgehoben zu werden. C ist eine kompilierte Sprache. Im Vergleich gehen Ruby und Python in Punkto Performance als letzte durch's Ziel. Aber: Sie erlauben viel kürzere Entwicklungszyklen. Einige Arbeit wird Dir als Entwickler abgenommen, zum Beispiel Speicherverwaltung. Auch viele Sicherheitsmechanismen, die es in C nicht gibt, sind vorhanden. Unter anderem deshalb sind diese Sprachen sehr beliebt. Google hat in der Vergangenheit sehr intensiv auf Python gesetzt. Einige Werkzeuge auf der Unixshell wurden ebenfalls mit Python realisiert.

Serveranwendungen als Einsteiger?

Solltest Du hier den Einstieg suchen? Warum nicht. Gerade wenn Du ohnehin aus dem OpenSource-Umfeld kommst, eine Unixvergangenheit hast oder Linux-User bist. Dann ist es nur noch der nächste logische Schritt. Genauso bin auch ich in die Software-Entwicklung gekommen. Meine erste Sprache, die ich wirklich intensiv studiert habe, war C. Das hat mir einer wirklich solide Basis verschafft. Moderne Sprachen nehmen Dir zwar viel Arbeit ab. Damit nimmst Du Dir aber auch die Chance auf Hintergrundwissen rund um Konzepte wie Speicherverwaltung. Bei C bleibt das Deine Aufgabe. Auf der einen Seite gut, auf der anderen auch wieder nicht.

Ein großes Plus in dieser Gleichung ist die OpenSource-Community. Die ist das beste Umfeld für Entwickler, gerade für Einsteiger. Du bekommst Feedback, kannst Dich und Dein Wissen einbringen und kannst anderen helfen. In der Medizin

gibt es für Studenten ein tolles Motto:

See One, Do One, Teach One.

Du lässt Dir erst etwas erklären. Dann machst Du es selbst aus. Anschließend erklärst Du es jemandem. Für Einsteiger der beste Weg zum Lernen. Schön und gut, aber wir sind hier nicht im Krankenhaus? Stimmt, aber auch fernab der Medizin ist es relevant. Die OpenSource-Gemeinde funktioniert ähnlich. Erst lernst Du von jemandem etwas. Dann implementierst du es selbst. Später erklärst Du es wieder jemandem oder schreibst darüber. Du siehst, ein geniales Umfeld, auf jeder Erfahrungsstufe.

Für hardwarenahe Software-Entwicklung gibt es immer mehr Anwendungsfälle, auch fernab von der IT. Ein vielversprechender Bereich, der mittel- und langfristig noch mehr gefragt sein wird als er es heute bereits ist. Auch wenn viele Schlagwörter, wie das *Internet der Dinge,* schon seit Jahren herumgeistern. Langsam füllt sich die Technologie dahinter mit Leben. Den Kühlschrank, der automatisch neue Milch bestellen kann, gibt es bisher noch nicht. Aber das könnte vielleicht Dein großes Projekt werden? Es gibt natürlich noch viel mehr Möglichkeiten. Hardwarenahe Programmierung umfasst das gesamte Spektrum der Embedded-Devices, dem sogenannten “Internet der Dinge” und der intelligenten Steuerung von elektronischen Geräten.

Steuerungen aller Art

Die Automobilindustrie ist einer der wichtigsten Märkte für hardwarenahe Entwickler. Gerade im "Autoland Deutschland", Heimat einiger der renommiertesten Automobilhersteller der Welt. Noch sind Projekte wie das autonome Fahren einige Jahre von der Serienreife entfernt. Das kann sich aber schon bald ändern. Innerhalb kürzester Zeit werden die Hersteller noch mehr Bedarf an fähigem Personal haben.

Auch die sonstige Infrastruktur nutzt integrierte Systeme, Schaltungen und elektronische Steuerungen. Viele davon werden mit Assembler oder SPS-Steuerungen gelöst. Letztere findest Du z.B. oft hinter einer Ampellogik. Auch das Schienennetz nutzt solche Steuerungen für den Betrieb von

Weichen und beschrankten Bahnübergängen.

Das Smarthome

Das intelligente Zuhause ist sogar schon in Teilen Realität. Beim Joggen die Heizung im Bad vorheizen? Kein Problem! Das Wohnzimmer kurz vor Feierabend schon langsam auf Temperatur bringen? Bereits mit Apps möglich! Die Heizung automatisch abschalten, falls gelüftet wird? Auch das ist mit einfacher Programmierlogik bereits möglich. Rolladen aus der Ferne steuern? Mit einem durchschnittlichen Motor für Jalousien kein Problem mehr.

Hinter all diesen Dingen steckt Programm- und Steuerungslogik. Und genau Da kommst Du als hardwarenaher Programmierer ins Spiel. Es gibt noch endlos viele Bereiche, in denen die Geräte bisher nicht vernetzt sind. Wenn erst Mal eine solide Basis für das Smarthome vorhanden ist und stabile Schnittstellen feststehen, werden sicher deutlich mehr Geräte ans Netz angebunden. Ein Beruf mit Zukunft!

Die “Industrie 4.0”

Hast Du schon einmal vom Plan der “Industrie 4.0” gehört? Das ist die Hightech-Strategie der Bundesregierung. Ich möchte hier in aller Kürze gar nicht die Idee selbst, die realistischen Erfolgschancen oder die konkrete Umsetzung des Plans beleuchten. Das führt vom Thema zu weit weg. Aber grob gesagt umfasst der Begriff die Idee einer industriellen Revolution, ausgelöst durch digitalisierte Fertigungsprozesse.

Dabei müssen unter anderem die unterschiedlichsten Elemente, Maschinen und Arbeitsschritte durch Computer steuerbar und über das Netz verfügbar sein. Die Abfrage von Sensoren und Steuerung der Maschinen ist nur über hardwarenahe Software möglich. Wie Du siehst gibt es bereits jetzt Nachfrage. Ein Ende ist nicht in Sicht. Eher im Gegenteil.

Weniger Sexy, trotzdem wichtig

Diese Art der Software-Entwicklung ist zugegeben etwas weniger “sexy” als die Arbeit an Apps oder Webseiten. Trotzdem wird sie immer wichtiger und an immer mehr Stellen ein Thema.

Die meisten Leute in der hardwarenahen Programmierung haben gleichzeitig auch größeres Interesse an Elektronik und Elektrotechnik. Und das ist am Ende auch notwendig. Hardwarenah bedeutet nah an der Elektronik. Über die eigene Software werden Schaltungen gesteuert. Du musst genau wissen welche Kontakte und Schalter Aktionen auslösen. Bei der Produktentwicklung wirst Du vielleicht sogar mit am Entwurf der Schaltungen arbeiten. Immerhin ist auch das Feedback eines Entwicklers wichtig.

Mehrere Disziplinen sind zu meistern

Bei hardwarenaher Entwicklung führt der Weg über mehrere Disziplinen. Du musst nicht nur etwas über die Programmiersprache lernen. Die verwendeten Sprachen und Befehle sind vergleichsweise sogar relativ einfach gestrickt. Viel

wichtiger, oder mindestens ebenso wichtig, ist die Hardware dahinter. Die musst Du kennen um mit ihr zu kommunizieren.

Wirf einen Blick auf folgendes Beispiel aus der Welt der Mikrocontroller. Es gibt verschiedene Register in der CPU. Das sind Speicheradressen, über die Aktionen ausgeführt werden können. Über die befiehlst Du der CPU bestimmte Dinge zu tun. Oder Du speicherst Daten in einem Register. Dafür musst Du sie aber erst kennen. Ähnliches gilt wenn Du Schalter an- und ausschalten möchtest. Auch hier musst Du die Technik dahinter kennen.

Noch deutlicher wird es bei der Entwicklung von Treibern. Die konkreten Assembleranweisungen sind meist gar nicht so kompliziert. Die Spezifizierung der Hardware ist meist komplexer. Deine Aufgabe ist zu wissen was erforderlich ist. Wie kannst Du Daten auf der Netzwerkkarte versenden? Wie mit der Audiohardware umgehen? Welche Features bietet die Grafikkarte eigentlich an? Mit anderen Worten gehört ein solides Wissen rund um die Elektrotechnik dazu. Wie funktioniert eigentlich ein BUS-System? Welche Daten werden gesendet? Wie sehen binäre Daten eigentlich auf einem Oziloskop aus? Kannst Du darin STOP-Bits erkennen (bei serieller Datenübertragung)?

Natürlich spielt auch Mathematik eine Rolle - sogar mehr als in höheren Sprachen. Aber: es geht um weniger alltägliche Ausprägungen: Wie rechnest Du eine Zahl von Integer in das binäre System um? Wie kannst Du einen Binärwert im hexadezimalen System darstellen - und umgekehrt? Inhalte, die

Dir im Rahmen eines Studiums oder einer Ausbildung begegnen werden.

Welche Sprachen?

Der Klassiker ist natürlich Assembler. Aber es gibt auch viele andere Wege für hardwarenahe Programmierung. SPS-Steuerungen gehören auch dazu. Oder Programme für eine CNC-Fräse. Ein ganz schlichter Fall: Lego Mindstorms! Ein wunderbarer Einstieg in die Thematik.

Lego Mindstorms

Wenn Du Dich für dieses Thema interessierst, ist Lego Mindstorms wie geschaffen für Dich! Eine wunderbare Spielwiese. Du kannst viele Prinzipien kennenlernen und erste Steuerungen entwickeln und umsetzen. Die gelieferten Sensoren generieren Daten, die Du weiterverarbeitest. Du kannst bspw. einen Roboter bauen, der selbstständig eine Linie auf dem Boden folgt. Und weil es ein perfekter Einstieg ist, wird Mindstorms sogar an Universitäten im Unterricht genutzt.

Arduino

Etwas fortgeschrittener ist die Arduino-Plattform. Dabei handelt es sich um einen Mikrocontroller auf einem I/O-Board, das sowohl analoge als auch digitale Ein- und Ausgänge mitbringt. Die kannst Du in der Programmlogik ansprechen, Daten einlesen und auch wieder ausgeben. Da auch eine passende Entwicklungsumgebung (IDE) mitgeliefert wird, kannst Du ohne viel Zeit in die Hardware zu investieren in die

hardwarenahe Programmierung einsteigen.

Für Einsteiger?

Auch hier stellst Du Dir als Einsteiger vielleicht die Frage, ob es ein geeigneter Einstieg ist? Wenn Du entschlossen bist, Spaß am tüfteln hast und eine steile Lernkurve Dich nicht abschreckt, kannst Du natürlich direkt mit Assembler beginnen. Meine Empfehlung ist aber Lego Mindstorms oder ein Set wie Arduino für den Einstieg zu wählen. Es gibt auch Einsteigersets mit einem 8086-Controller, ebenfalls mit Erweiterungen wie einem USB-Anschluss und anderem Zubehör. Aber auf Dauer halte ich gerade im Bereich der Elektrotechnik einen qualifizierten Mentor für unersetzbar. Daher ist dies der einzige Bereich in dem ich das Selbststudium als Einsteiger für weniger sinnvoll halte. Ein Studium oder eine Ausbildung ist wirklich sehr zu empfehlen.

Viele Wege führen nach Rom. Nirgendwo gilt diese Weisheit so sehr wie in der Software Welt. Auf dem Weg zur Karriere als Programmierer führen die Wege aber auf drei Hauptstraßen. Und um genau die geht es in den nächsten Seiten.

Jede Option hat ihre Vor- und Nachteile. Zum einen möchte ich genau die herausarbeiten. Außerdem will ich Dir damit Mut machen und zeigen: Du brauchst als Entwickler nicht der typische Senkrechtstarter sein. Viele haben immer noch die Vorstellung vom klassischen Geek oder Nerd im Kopf - das Wunderkind. Es gibt solche Phänomene, ja. Aber diese Extremfälle sind Ausnahmen. Mir Fallen spontan auch nur eine Hand voll solcher Menschen ein.

Ausnahmen sind nicht die Regel

Linus Torvalds ist der Schöpfer und Vater von Linux. Bis heute betreut er den Linuxkernel als Chefentwickler und lenkt seine Geschicke. Er ist in der Tat ein Senkrechtstarter. Aber noch bevor er überhaupt wusste was er tippt, hat er auf dem Schoß von seinem Großvater die Tastatur für eben diesen bedient. Der Großvater war Professor an der Universität von Helsinki. Linus hat auf seinem VIC-20 später sogar angefangen seine ersten eigenen Programme zu schreiben. So eine Kindheit verschafft natürlich einen Vorsprung. Mit ein Grund, warum es durchaus sinnvoll wäre, schon in der Schule Grundlagenwissen zu vermitteln. Aber das soll jetzt nicht unser Thema sein.

Ein zweites Ausnahmetalent: John Carmack. Ein begnadeter Programmierer. Jeder Entwickler sollten diesen Namen

kennen. Selbst wenn Du gar keine Computerspiele spielst kann er ein Vorbild sein. Er ist das Gehirn der Firma "id Software". Das Unternehmen ist für Spielehits wie "Commander Keen", die Quake-Reihe, Doom, Wolfenstein und ähnliche Titel verantwortlich. Ein Wegbereiter heutiger 3D-Engines. Und das Genie dahinter ist John Carmack. Ein wahrlich großer Magier an der Tastatur.

Ein weiterer unglaublich begabter Mensch war Aaron Schwartz. Er ist mit 14 Teil des Kommitees gewesen, das den ersten Entwurf des RSS Standards verabschiedet hat. Die anderen Mitglieder wussten dies gar nicht. Irgendwann wurde er gefragt, wieso er nie bei Meetings vor-Ort anwesend sei. Der junge Aaron vermutete, dass seine Mutter dies wahrscheinlich nicht erlauben würde. Der Fall Aaron Schwartz ist sehr traurig und hätte vermutlich eine ausführliche Erklärung verdient. Ich verweise jedoch hier auf die Dokumentation dazu - The internets own boy. Er war eben ein solches Ausnahmetalent.

Es gibt sicher noch weitere Namen, die es verdient hätten genannt zu werden. Ich möchte hier aber auch nur etwas deutlich machen, keine Sammlung von Überfliegern anlegen. Es gibt die Ausnahmetalente. Aber auf einen Linus Torvalds kommen viele Tausende durchschnittliche Entwickler. Lass Dich von solchen Geschichten nicht entmutigen. Eher im Gegenteil. Es sollte Dich motivieren etwas zu bewegen! Sicher kann nicht jeder ein neuer Linus Torvalds werden. Aber das musst Du auch nicht. Alles was Du machen musst, ist Deinem Potential gerecht werden. Es gibt ein wunderbares Zitat, was es

verdeutlicht:

Der Wald wäre sehr leise, wenn nur die begabtesten Vögel singen würden.

In den nächsten Kapitel stelle ich Dir nun die drei Wege in Richtung Software-Entwicklung vor.

Der vermeintlich klassische Bildungsweg ist ein Studium der Informatik. Es stehen verschiedenste Inhalte auf dem Programm. Nur Details variieren je nach Studiengang und Form der Hochschule. Im Mittelpunkt stehen aber Mathematik, Software Entwicklung (Programmierung), Betriebssysteme und Theorie (häufig auf Unixbasis), Software Engineering (Software Entwurf) und weitere verwandte Themen.

Das Studium als fundierte Grundlage

Ein Studium ist eine wirklich geniale Grundlage. Das gilt natürlich gerade für den rein technischen Aspekt. Du lernst in verschiedenen Bereichen eine ganze Menge, entsprechendes Engagement vorausgesetzt. Aber auch auf sozialer Ebene ist ein Studium super. Du baust ein Netzwerk zu anderen Experten auf, die zumindest in Teilen in der lokalen Wirtschaft tätig werden. Neben den Verbindungen zu Kommilitonen sind auch externe Bekanntschaften die Regel, denn Du triffst auch auf Studenten anderer Studiengänge. Solche Kontakte in die Wirtschaft können später durchaus nützlich sein.

Fakt ist, dass viele erfolgreiche Menschen zumindest einige Semester studieren. Und noch wichtiger als Faktenwissen sind oft die geknüpften Kontakte und das aufgebaute Netzwerk. Solche Ansprechpartner sind nicht nur interessant um einen vermeintlich guten Job zu erhalten. Du könntest auch die möglichen Mitgründer Deiner Millionen-Dollar-Idee treffen. Siehe Mark Zuckerberg. Rein technisch war Facebook gerade beim Start nur eine schlichte Webseite. Der Erfolg hatte sehr

viel mit der außerordentlich guten Vernetzung zu tun.

Auslandssemester und Projekte

Ein weiterer spannender Aspekt sind Projektphasen bei praktisch orientierten Studiengängen. Hier lernst Du vielleicht bereits zukünftige Arbeitgeber kennen. Wirklich sehr häufig der Fall. Die Möglichkeit einer Festanstellung ist, bei entsprechendem Talent und Engagement, für Praktikanten eine reale Chance für die Zukunft.

Auch ein Auslandssemester kann sehr hilfreich sein, in verschiedener Hinsicht. Der sprachliche Aspekt ist immer ein großer Vorteil. Im Lebenslauf macht sich Auslandserfahrung zudem auch immer gut. Aber auch hier ist der Netzwerkeffekt nicht zu unterschätzen. Ich kenne Fälle, in denen Studenten durch ein Auslandssemester in Japan Fuß gefasst haben. Nach dem Studium in Deutschland sind sie ausgewandert und arbeiten dort nun an der Entwicklung von Apps. Sicher nicht für jeden eine Option. Was ich damit zeigen will ist wie sehr das Studium Dein Schicksal mitbestimmen kann.

Zeit zum Lernen

Neben allen Vorteilen ist im Studium vor allem ein Aspekt nicht zu ersetzen: der Zeitfaktor. Zeit ist die wertvollste Ressource überhaupt. Und wenn Du Spaß an der Software-Entwicklung hast, brauchst Du Zeit zum Lernen. Im Studium steht genau das im Mittelpunkt. Die Realität entspricht weniger der romantischen Vorstellung vom stressfreien Leben. Im

Gegenteil. Es kann stressig sein und ist auch mit viel Aufwand verbunden. Aber so viel Zeit zum Lernen und Ausprobieren wirst Du so einfach nie wieder bekommen. Die Chance solltest Du unbedingt nutzen.

Wenn Du erst einmal in Vollzeit arbeitest wird es schwerer Zeit freizuschaufeln. Klar, Programmieren ist Dein Hobby. Vielleicht sogar Deine Leidenschaft. Aber Du hast nun eine 40+ Stundenwoche. Rechne einfach mal einen 9-to-5 Job durch:

Du stehst um 7Uhr auf, machst Frühstück. Du bereitest etwas zu Essen für den Tag vor. Dann machst Du Dich fertig für's Büro und packst Deine Tasche. Insgesamt eine Stunde ist jetzt schon weg. Bei einer Anfahrt von 30-45 Minuten zuzüglich Berufsverkehr sind nun schon fast zwei Stunden vorbei. Anschließend bist Du mindestens acht Stunden in der Firma. Im besten Fall ist es also 17 Uhr wenn Du Feierabend machst. Dann die Rückfahrt. Wieder 30-45 Minuten, im Berufsverkehr. Ankunft Zuhause, Post durchgehen und "ankommen" - noch mal 15 Minuten.

Schon ist es 18 Uhr. Abendessen vorbereiten und Essen, kurz vom Tag abschalten, da kann es schnell 19 Uhr sein. Dann kommt der Faktor Sport hinzu. Ob nun vor oder nach dem Essen ist unerheblich. Bei dreimal pro Woche Sport bist Du, je nach Sportart, inkl. Duschen gut und gerne mal fix weitere 1,5 Stunden los. Schon ist es 20:30Uhr. Vielleicht muss noch der Fahrtweg einkalkuliert werden oder das ein oder andere Gespräch mit Kollegen. Schon bist Du erst um 21Uhr wieder Zuhause. Die verbleibenden ein bis zwei Stunden am Computer

werden sicher nicht die beste Zeit für geistige Höchstleistungen.

Natürlich gibt es trainingsfreie Tage. Selbst ganz ohne Sport oder nur 30 Minuten hast Du immer noch weitere Aufgaben. Du musst auch noch die Wohnung aufräumen, Wäsche machen, Staubsaugen und immer mal wieder einkaufen. Am Wochenende möchtest Du auch Deine Freunde, Verwandten und Bekannten sehen? Damit läuft es eigentlich jeden Tag ähnlich wie an Trainingstagen. Knapp zwei Stunden kannst Du schnell für "Alltägliches" einplanen.

Der Punkt auf den ich hinaus will: Als Student hast Du auch nicht unendlich viel Freizeit. Aber! Du hast für eine kurze Phase in Deinem Leben die Chance, sowohl im Unterricht/den Vorlesungen als auch durch zusätzliches Lernen, Dich intensiv mit Inhalten zu befassen. Solche Gelegenheit sind später nicht ausgeschlossen, aber der Freiraum ist in dem Maße fast ausgeschlossen. Bedenke, dass in dem Beispiel oben noch keine Beziehung, Familie, Kinder oder ein anderes Hobby eingerechnet sind.

Musst Du deshalb studieren?

Ganz sicher nicht. Aber ich empfehle es uneingeschränkt jedem, der die Chance hat. Wenigstens probieren kannst und solltest Du es. Und wenn dabei in jungen Jahren nach einem Jahr feststeht, es war nicht der richtige Weg? Dann ist rein gar nichts verloren! Die Bezeichnung Studienabbrecher ist total falsch. Sie spiegelt nur den Ausgang wieder. Viel besser wäre

die Bezeichnung “Student auf Probe” oder “Studienausprobierer”. Du hast etwas gewagt und Wissen mitgenommen. Hast Erfahrungen gesammelt. Du bist reifer geworden. Hast etwas probiert. Eine Möglichkeit ausgeschlossen. Das halte ich für viel sinnvoller als sich später zu fragen, ob es nicht doch interessant gewesen wäre. Ich habe sicher nicht alles richtig gemacht. Mit meinen Fehlern kann ich aber relativ gut umgehen. Was mich viel mehr stört sind Dinge, die ich nicht probiert habe. Das typische "Was wäre gewesen wenn ..." nagt am meisten an mir.

Später studieren

Aber selbst wenn Du nicht direkt das Studium wählst, kannst Du auch später immer noch darauf zurückkommen. Es wird aber mit der Zeit nicht unbedingt leichter. Immerhin schläft der Schulstoff ein. Außerdem wird es schwerer Zeit zum Lernen zu finden. Falls der Gedanke an ein Studium besteht, solltest Du auf finanzielle Unabhängigkeit wert legen. Binde Dich vertraglich nicht durch Abonnements. Finanzier kein Auto. Nimm keinen Kredit auf. Lösch Deinen Dispokredit und schaff Dir keine Kreditkarte mit Kreditlimit an. Das sind alles “Fallen”, die Dich später finanziell an ein festes Gehalt binden.

Ein Studium ist übrigens nicht nur im Präsenzstudium an einer Hochschule möglich. Du kannst mittlerweile aus vielen verschiedenen Möglichkeiten wählen, die auch später noch möglich sind. Dazu gehört zum Beispiel auch ein Fernstudium. Das kannst Du komplett neben der normalen Arbeitszeit absolvieren. Es ist sicher nicht unbedingt einfach. Die vielen

Absolventen beweisen aber, dass es eine echte Option ist.

Wenn Du später merkst Du möchtest Dich weiterentwickeln, kannst Du auch ein duales Studium anstreben. Dabei bist Du teilweise in einem Unternehmen tätig. Die restliche Zeit bist Du an der Universität oder Fachhochschule, beispielsweise in einem Quartalsrhythmus.

Eine Alternative zum Studium ist eine Ausbildung. Ist die im Vergleich also "2. Wahl"? Gewiss nicht. Rein inhaltlich gibt es Berufe, in denen Du mit einer Ausbildung sogar besser fährst. Ein Student lernt im reinen Informatikstudium nur wenig praktische Fertigkeiten eines Webentwicklers. Das passiert wenn nur in Eigenregie. Darum bereitet Dich eine spezifische Ausbildung vermeintlich besser auf die Praxis vor. Und da kannst Du zwischen zwei Varianten wählen: Es gibt die schulische und die betriebliche Ausbildung.

Schulische Ausbildung

Eine schulische Ausbildung wird an Berufsfachschulen absolviert. Wenn möglich würde ich diesen Weg immer vorziehen, vernünftige Bildungseinrichtung vorausgesetzt. Dort steht der Schüler und das Lernen im Vordergrund. Du sollst für die Praxis im Berufsalltag vorbereitet werden. In dieser Form kann viel intensiver mit den einzelnen Studierenden gearbeitet werden. Im Unterricht wird ein didaktisches Konzept verfolgt, an fünf Tagen der Woche. Im Betrieb steht und fällt Deine Tätigkeit in vielen Fällen mit den Aufgaben die zu erledigen sind.

Hier sei erwähnt, dass es auch große Firmen gibt, bei denen die Ausbildung in eigenen Ausbildungszentren absolviert wird. Das ist mit einer schulischen Ausbildung vergleichbar. Diese Variante ist tendenziell die bessere, gerade aus finanzieller Sicht. Du hast alle Vorteile der schulischen Ausbildung. Trotzdem bekommst Du zusätzlich eine Ausbildungsvergütung. Eine schulische Ausbildung bei einem privaten Bildungsträger

kostet sogar Geld.

Betriebliche Ausbildung

Bei einer betrieblichen Ausbildung landest Du schnell in der Praxis. Das kann gut sein. Soweit die Theorie. Ob die betriebliche Umgebung sinnvoll ist oder nicht, hängt maßgeblich vom Betrieb und dem Unternehmen ab.

Du lernst nicht in einer kontrollierten Umgebung mit Dozenten. In der Berufsschule ist die Zeitvorgabe eine ganz andere. Außerdem gibt es bei schulischen Ausbildungen Ansprechpartner, die nur für die Studierenden da sind. In einem wirtschaftlich orientierten Betrieb ist das in der Regel anders. Wenn Du als Auszubildender arbeitest, bist Du auf die Hilfe von anderen Angestellten und Vorgesetzten angewiesen. Das ist OK, denn auch dort bekommst Du freundliche Hilfe. Aber halt eher zu ganz konkreten Fragen und Problemen. Immerhin müssen Deine Kollegen ihre Arbeit zusätzlich erledigen. Du bist daher viel mehr auf Dich selbst gestellt.

Das gilt in verschiedener Hinsicht. Du wirst in der Regel in einem Betrieb schnell echte Aufgaben übernehmen müssen. Und die bestimmen somit unweigerlich auch Deinen Lehrplan. Zwar wird in der Regel die Problemstellung und die gewünschte Lösung genau erklärt. Trotzdem bist Du viel mehr auf Dich alleine gestellt. Du musst viel schneller Aufgaben mit ausreichender Qualität lösen. Die Schonfrist ist kürzer.

Außerdem ist es an Dir das große Ganze nicht aus den Augen

zu verlieren. Stell Dir vor, Du arbeitest als Azubi in einer Internetagentur. Dort löst Du immer wieder ähnliche Probleme. Du erstellt z.B. HTML-Formulare für Deine Kollegen. Du musst darauf achten, auch die anderen Möglichkeiten von HTML kennenzulernen. Das geschieht in der Regel Zuhause. Tagsüber hast Du in der Firma Deine betrieblichen Aufgaben.

Der Gehaltsfaktor

Das traurigste Argument gegen eine Ausbildung ist leider harte Realität. Es kann und wird Dir durchaus begegnen. Daher möchte ich es hier unbedingt ansprechen.

Für die gleiche Arbeit, die gleiche Anwesenheit und den gleichen Aufgabenbereich bekommt jemand mit einem abgeschlossenen Studium in der Regel mehr Gehalt. Es ist ähnlich sinnlos wie die Unterscheidung zwischen Mann und Frau - bei gleicher Leistung.

Du kannst darüber lamentieren oder es einfach akzeptieren. Ändern wirst Du die Firmenpolitik in dem Bezug ohnehin nicht. Das geht nur wenn Du in die Position kommst, in der Du über Gehälter bestimmst. Es gibt natürlich Unternehmen die eine Ausnahme machen. Aber die sind eben nicht die Regel, noch nicht.

Bedenke den Gehaltsfaktor bei der Entscheidung zwischen Ausbildung und Studium unbedingt, gerade wenn Du Deine Zukunft als Angestellter planst.

Der Prestige-Faktor

Auch das Ansehen einer Ausbildung ist nicht so hoch, wie das von einem abgeschlossenen Studium. Das ist grundsätzlich nachvollziehbar. Immerhin werden im Studium noch komplexere Themen behandelt. Auch höhere Mathematik ist nicht einfach und wird in der Ausbildung nicht in gleichem Maße abgedeckt. Doch was bedeutet "Prestige"?

Es geht mir nicht darum wie gut Du damit Freunde, Verwandte oder Deine Nachbarn beeindrucken kannst. Das Prestige ist bei der Bewerbung um eine Stelle wichtig. Der Personalverantwortliche wird, soweit die Theorie, von einem Studium eine höhere Meinung haben. An der Stelle musst Du mit einer abgeschlossenen Ausbildung mit Erfahrung und sonstigen Qualifikationen punkten. Denn auch viele Firmen wissen mittlerweile, dass ein Bachelor of Science der Informatik nur wenig über praxistaugliches Wissen aussagt.

Fazit zur Ausbildung

Du musst nicht studieren um in der IT-Welt erfolgreich zu sein. Sicher gibt es Firmen und Jobs, die ein abgeschlossenes Studium direkt oder indirekt voraussetzen. Das ist speziell dann wichtig, wenn höhere Mathematik eine Rolle spielt. Aber es ist nicht per se erforderlich.

Viele Bereiche wie Web- und Appentwicklung kannst Du Dir vollständig ohne den Besuch einer Universität erschließen. Vielleicht sogar besser! Die größten Karrierechancen bieten Dir

kleine und mittlere Betriebe. Dort wird häufig sehr unkompliziert gearbeitet. Es gibt viele Firmen, denen die Bewerbungsunterlagen nicht so wichtig sind. Was dort zählt sind Ergebnisse, eigenständiges Arbeiten und Fachwissen.

Meine Empfehlung: Wenn irgendwie die Chance besteht, solltest Du ein Studium ausprobieren. Informatik ist Dir an einer Hochschule zu trocken? Probier es mit einer Fachhochschule (FH). Es gibt auch duale Studiengänge. Da wechseln sich Theorie und Praxis in der FH und dem Betrieb ab. Sowohl eine Ausbildung als auch Studium an einer FH dauern in der Regel etwa 3 Jahre. Im Endeffekt hängt die Entscheidung vom gewünschten Ziel ab. Aber insgesamt kann eine Ausbildung mehr als ausreichend sein.

Gerade junge Menschen nutzen die klassischen Wege in die IT. Sie studieren oder machen eine Ausbildung. Es gibt gleichzeitig auch viele Quereinsteiger. Menschen mit Berufserfahrung, die sich Software-Entwicklung in Teilen oder vollständig selbst beibringen. Gerade die notwendige Praxis kannst Du auch auf Dich allein gestellt bekommen. Du wirst einfach nicht dafür bezahlt. Aspekte wie Teamarbeit kannst Du in OpenSource-Projekten lernen. Du brauchst wirklich nur einen Laptop mit Internetverbindung und Zeit. Denn Du musst eigentlich nur Programmieren. Üben, Fehler machen und lösen.

Als Quereinsteiger Programmierer werden

Manche Geschichten könnten selbst für einen Filme nicht besser geschrieben werden. Und doch gibt es sie wirklich häufiger als Du vielleicht denkst. Eine ganze Reihe solcher Quereinsteiger hat oft bereits in irgendeiner Form Kontakt mit Computern. Sie wechseln einfach von anderen Disziplinen in das *Entwicklerlager*. Genauso gibt es Leute aus ganz anderen Branchen, die den Übergang ebenfalls wagen und schaffen.

Es gibt keine formelle Anforderung an Deinen Bildungsweg. Klar, ein ehemaliger Ingenieur hat auf dem Papier bessere Argumente als jemand ohne Berufsausbildung. Auch ein themenverwandter Wechsel ist einfacher zu argumentieren. Ich habe selbst Jahre lang als Netzwerkadministrator gearbeitet und im Schwerpunkt Webserver betreut. Nebenbei habe ich aber immer Software entwickelt, OpenSource-Projekte unterstützt und entwickelt. Der Sprung zum hauptberuflichen

Entwickler war also vergleichsweise klein.

Aber es ist auch möglich wenn Du aus einem völlig anderen Wirtschaftszweig kommst. Ein interessanter Arbeitgeber legt Wert auf das Ergebnis, das Resultat und die Umsetzung. Eine tolle Ausbildung und Berufserfahrung ist nicht wegzudiskutieren, aber am Ende auch kein Garant für einen guten Job. Wichtig ist nur, dass Du Programmieren kannst. Du musst die Firma in ihren wirtschaftlichen Zielen unterstützen. Das erwartet das Unternehmen auch von den anderen Entwicklern.

Wo Du es gelernt hast sollte am Ende keine Rolle spielen. Der einzige Knackpunkt sind möglicherweise Referenzen. Wenn die Bildung und Erfahrung auf dem Papier nicht so einfach zu belegen ist, musst Du auf anderem Wege Punkten. Du kannst zum Beispiel wunderbar über ein Blog Autorität auf einem Fachgebiet aufbauen. Sinn und Zweck von Zertifizierungen alleine sind zwar fragwürdig. Sie tragen aber zu einem abgerundeten Gesamtbild bei. Zudem empfehle ich unbedingt aktiv an einem bekannten OpenSource-Projekt mitzuwirken. Auch das kannst Du unter Berufserfahrung nennen. Eine umfassende Anleitung ist im Rahmen von diesem Ratgeber nicht möglich. Dafür ist das Thema zu komplex. Das Stichwort für weitere Recherchen aber ist Personal Branding. Du musst Dich und Deinen Namen zu einer Marke machen. Das bedeutet nicht, dass Du Dich verkaufen oder verbiegen sollst. Im Gegenteil. Du musst authentisch andere Menschen auf Deine Fähigkeiten und Deine Werte aufmerksam machen.

Empfehlen kann ich als Einstieg das Buch *The Brand Called You* von *Peter Montoya*.

Außerdem kannst Du als Quereinsteiger eine Ausbildung zum Entwickler machen und ganz formal die Branche wechseln. Einziges Manko sind der zeitliche und finanzielle Einschnitt. Gleiches gilt für ein mögliches Studium. Nach einigen Jahren Berufserfahrung macht es Dich aber mittelfristig sogar interessanter. Alles eine Frage der Sichtweise! Spannende Projekte und innovative Ideen kommen oft nicht von den *Fachidioten* (positiv gemeint). Geniale Software-Ideen entstehen bei der Verbindung von zwei oder mehr Disziplinen. Wenn Du in einer Branche Experte bist und Software-Entwickler wirst, kannst Du Programme für eben genau Dein ehemaliges Fachgebiet schreiben. Du kennst bereits die Probleme und Nöte der Anwender. Beste Voraussetzungen für ein erfolgreiches Startup.

Über Praktika zur Festanstellung

Unternehmen suchen keine Qualifikation auf dem Papier. Jede Firma erhofft sich von einem Angestellten Wachstum. Dazu musst Du als Entwickler mit Deiner Arbeit an mehr Umsätzen beteiligt sein, als Du monatlich an Kosten verursachst. Soweit die Theorie. Wie überzeugst Du aber als Quereinsteiger einen Arbeitgeber, dass Du das kannst? Oben habe ich bereits das Personal Branding empfohlen. Wenn Du Dich als Experte positionierst, fragt am Ende niemand mehr nach Deiner Ausbildung. Das geht aber natürlich nicht von

heute auf Morgen.

Mach ein Praktikum! Das ist der Geheimtipp für Quereinsteiger. Du kannst sogar von Heute auf Morgen damit loslegen, wenn sich eine Chance ergibt. Es gibt keinen besseren Weg Deinen Beitrag im Unternehmen unter Beweis zu stellen. Wenn Du ein Team unterstützen kannst und Dich gekonnt präsentierst, gibt es keinen Grund warum Du nicht qualifiziert sein solltest. Sei einfach so gut, dass Sie Dich nicht ignorieren können.

Fazit für Einsteiger

Ganz offensichtlich ist Quereinsteiger kein bewusst gewählter Weg. Wenn Du Schüler bist und noch alle Möglichkeiten offen stehen, gehst Du automatisch direkt in die richtige Richtung. Ein Quereinsteiger bist Du per Definition bei einem Wechsel der Disziplin. Wie oben beschrieben ist das ein gangbarer Weg. Er macht Dich für die Wirtschaft sehr wertvoll. Du hast Wissen in einem bestimmten Wirtschaftsbereich gesammelt. Wenn Du anschließend IT-Kenntnisse aufbaust, kannst Du beide Welten verbinden und so vielleicht ganz neue Innovationen produzieren. An dieser Stelle bleibt mir nur noch ein eindeutiges Plädoyer für Quereinsteiger aus allen Bereichen. Die Chancen auf Erfolg sind, bei ausreichend Engagement, in jedem Fall hoch.

Software Entwicklung ist ein spannendes Thema. Es gibt viele Facetten, viele Programmiersprachen und unterschiedliche Plattformen. Das gesamte Angebot kannst Du als einzelner Entwickler gar nicht mehr beherrschen. Und trotz der scheinbar unendlichen Vielfalt gibt es eine gemeinsame Basis. Nahezu alle Programmiersprachen unterstützen bestimmte Konzepte. Manchmal sehen sie in der konkreten Ausprägung etwas anders aus. Verstehst Du sie jedoch in Sprache “X”, ist es oft leicht das Wissen auf die Sprache “Y” zu übertragen. In den folgenden Kapiteln lernst Du einige dieser Elemente kennen.

Einblicke in die Software Entwicklung

JavaScript kommt dabei als Sprache zum Einsatz. Die Voraussetzungen bringt jeder handelsübliche Desktopcomputer bereits mit. Du benötigst keine spezielle Entwicklungsumgebung. Du hast sehr wahrscheinlich einen Browser und einen Texteditor auf Deinem Computer. Außerdem funktioniert JavaScript unabhängig von der eingesetzten Plattform. Optimal für ein solches Tutorial. Unmittelbares Ziel ist es nicht, Dir JavaScript im Detail zu vermitteln. Das geht mit separaten Büchern oder Videotrainings besser. Es geht viel mehr um grundlegende Konzepte. Betrachte JavaScript also eher als Mittel zum Zweck.

Anfänge der Software Entwicklung

Die Grundidee der Software-Entwicklung war es nicht Spiele zu entwickeln, jedem Bürger ein Smartphone in der Tasche zu

ermöglichen oder Ähnliches. Im Gegenteil. Zu Beginn sollten Computer mathematische Berechnungen erleichtern. Der Name »Computer« bedeutet übersetzt sogar »Rechner«. Die ersten Modelle waren dabei alles andere als mobil. Sie haben ganze Räume gefüllt.

In den Anfängen der Software Entwicklung wurden verschiedene Ansätze verfolgt. In den frühen Jahren wurde sogar auf physischer Ebene programmiert, mit Lochkarten! Eine genaue Erklärung der Funktionsweise führt hier zu weit. Wichtig ist nur, dass sie bis um 1980 herum weit verbreitet waren. Die klassischen Programmiersprachen, wie wir sie heute kennen, entstanden erst relativ spät. Ihre Verbreitung hat das Aus der Lochkarte besiegelt und die Grundlage für heutige Programmierer gelegt. Wir schreiben wirklich relativ junge Geschichte. Viele der Sprachen gibt es noch heute. Ein bekanntes Beispiel ist die Sprache C. Auch Weiterentwicklungen und Erweiterungen wie Objective-C oder C++ erfreuen sich großer Beliebtheit.

Die Konzepte der Software Entwicklung

Software besteht letztendlich aus einer Reihe von Tests und Aktionen, damals wie heute. Es gibt unterschiedliche Konzepte und Herangehensweisen bei der Strukturierung von Quellcode und komplexen Softwareprojekten. Gutes Beispiel sind objektorientierte Programmiersprachen. Auch die Planung von Software und die Herangehensweise an Projekte ist anders. Die agile Software-Entwicklung findet immer mehr Zuspruch,

steckt aber selbst noch in den Anfängen.

Egal was bisher unternommen und versucht wurde. Zum Schluss läuft es darauf hinaus, dass Maschinencode produziert wird. Teilweise bereitet ein Compiler ihn vor. Bei einer Skriptsprache wie Python muss diese Aufgabe der Interpreter übernehmen. Am Ende steht jedoch immer die CPU. Die arbeitet einfach Maschinencode Schritt für Schritt ab. Für das Verständnis dieser Hintergründe ist ein Exkurs in die Welt der Assembler Programmierung hilfreich und spannend. Für Dich ist in diesem Augenblick aber nicht wichtig, dass nicht der menschenlesbare Quellcode, sondern binärer Maschinencode verarbeitet wird.

Ziel der Programmierung

Bei der Programmierung geht es fast immer um die Verarbeitung, Aufbereitung, Sammlung oder Darstellung von Daten. Im Kern tut jede Software genau dies. Egal ob Spiel, E-Mail-Programm, Smartphone-App oder Online-Shop. Überall werden Daten gesammelt, aufbereitet und dargestellt. Das ist das sogenannte EAV-Prinzip: Eingabe, Ausgabe, Verarbeitung.

Das ist in der Theorie so trivial wie es klingt. Die Realität ist allerdings komplizierter, da es sehr viele Möglichkeiten gibt um diese Schritte zu erledigen. Gerade der Schritt der Verarbeitung kann beliebig komplex werden. Zudem gibt es auch sehr viele Datenquellen, diverse Formate, unterschiedliche Wege der Darstellung und unterschiedliche Geschmäcker. Die Kunst besteht darin, die Anforderungen umzusetzen und

zielgerichtete Software zu entwickeln. Da unsere Welt sehr komplex ist und Daten in sehr unterschiedlichen Formen daherkommen, gibt es verschiedene Programmiersprachen und Plattformen.

Facetten der Software Entwicklung

Die Programmierung hat unterschiedliche Facetten, schon bedingt durch die unterschiedlichen Bereiche in denen Software eingesetzt wird. Da gibt es bspw. die hardwarenahe Entwicklung. Typischer Anwendungsfall sind derzeit Autos. Der einfache KFZ-Mechaniker hat nahezu ausgedient. Er wird Zug um Zug vom Mechatroniker ersetzt, der sich auch mit den elektrotechnischen Bestandteilen heutiger Automobile auskennt. Das ist eine notwendige Entwicklung, denn ohne die eingesetzte Elektronik und die Steuergeräte kommen die Autos nicht mehr aus. Gleichzeitig erschließt das den Markt für Entwickler. Denn es gibt nicht nur den Mechaniker in der Werkstatt. In der Kette vorher stehen Programmierer und Ingenieure, die solche Steuerungen und die passende Elektronik entwickeln.

Ein großer Markt ist zudem die Software-Entwicklung für mobile Endgeräte. Dort werden fortlaufend Experten für die unterschiedlichsten Anwendungsbereiche gesucht. Das geht von mobilen Lösungen mittels HTML, CSS und Javascript bis hin zur Entwicklung von nativen Anwendungen für iOS und Android. Auch Spielentwickler werden gesucht. An die werden natürlich besondere Anforderungen gestellt, da unter anderem die Entwicklung von 3D Software alles andere als trivial ist.

Speziell die notwendigen mathematischen Berechnungen sind anspruchsvoll. Aber selbst im Bereich der Spieleentwicklung gibt es ein sehr breites Spektrum.

Ebenfalls nach wie vor gefragt sind Webentwickler. Der Markt ist auch hier groß, da Webentwickler viele verschiedene Bereiche abdecken. Sie entwickeln Online-Shops, normale Firmenauftritte, mobile Webseiten und vieles mehr. Ohne Webentwickler wäre das Internet nicht das, was es heute ist. Dabei musst Du zwischen zwei Disziplinen unterscheiden. Auf der einen Seite stehen die Frontendentwickler. Die setzen mit HTML, CSS und JavaScript die Darstellung der Webseite um. Die Logik dahinter wird vom sogenannten Backendentwickler realisiert. Beide Wege sind gerade für Quereinsteiger ein heißer Tipp!

Es gibt noch mehr Fachgebiete, auf die Du Dich spezialisieren kannst. Zum einen kannst Du Dich auf einzelne Plattformen spezialisieren, Dich auf bestimmte Software-Lösungen fokussieren oder Dich auf bestimmte Technologien stürzen. Die Möglichkeiten sind deutlich größer als die zur Verfügung stehende Zeit.

Fazit

Die Software Entwicklung ist spannend. Wenn Du grundlegend Interesse hast, kannst Du Dir aus nahezu unendlich vielen Möglichkeiten Deine Favoriten heraussuchen. Schon jetzt ist es unmöglich als einzelne Person alle Technologien zu beherrschen. Und das ist erst der Anfang. Die

Software-Welt wird immer komplexer. Dieser ständige Wandel ist ebenfalls spannend. Dadurch entstehen regelmäßig neue Technologien, in denen auch Einsteiger die etablierten Profis jederzeit überholen können. Es ist nicht möglich Schritt zu halten. Und trotz der großen Vielfalt hat sich am Kern seit der Einführung der ersten 80386 Prozessoren nur wenig geändert. Alles wurde schneller und leistungsfähiger, aber die heutige Hardware ist in weiten Teilen noch kompatibel.

Nach diesem theoretischen Ausflug geht es nun im nächsten Kapitel um ein erstes Beispielprogramm – Hallo Welt.

Eines der ersten Programme, das Du als Einsteiger in eine Programmiersprache implementierst, ist das »Hallo Welt« Programm. In der Regel ist es auf die Ausgabe der Zeichenkette »Hello World« bzw. »Hallo Welt« beschränkt. Diese Tradition wurde im Klassiker »*The C Programming Language*« ins Leben gerufen. Seitdem hat es sich als Standardprogramm eingebürgert.

Warum »Hallo Welt«?

Das simple Programm zeigt erste Elemente der Syntax und ist ein erstes Beispielprogramm. Es reicht um den typischen Programmaufruf zu zeigen. Und der steht am Anfang jeder Erklärung. Bei einer kompilierten Sprache wird der Compiler-Aufruf und der Programmstart gezeigt. Bei einer Skriptsprache muss dir klar sein wie der Interpreter angesprochen wird. Gleichzeitig wird mit einem minimalen Programm getestet, ob die Entwicklungsumgebung funktioniert.

Die Geschichte von »Hallo Welt«

Die Einführung in eine Programmiersprache mit dem »Hallo Welt« Programm zu beginnen, reicht bis zu den Anfängen der Programmiersprache C zurück. Das erste Buch, das »Hello World!« Beispielcode enthielt, war »*The C Programming Language*« von Brian Kernighan und Dennis Ritchie. Die beiden Urväter der heutigen Programmierung haben die Sprache C entwickelt und die Grundlagen für heutige Unixsysteme gelegt. Wie man sieht, ist eine weitere ihrer Erfindung Teil des Lebens eines jeden Programmierers

geworden: Das »Hallo Welt!« Programm.

»Hallo Welt« in verschiedenen Sprachen

Die »Hallo Welt« Programme sind Teil einer langen Tradition. Hier einige Beispiele für »Hallo Welt« Programme in unterschiedlichen Programmiersprachen. Ich habe besonders bekannte Vertreter ausgewählt:

C:

```
#include <stdio.h>
int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

C++:

```
#include <iostream>
 int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Hi!</title>
</head>
<body>
    <p>Hello, world!</p>
</body>
</html>
```

Java:

```
public class HelloWorld
{
   public static void main(String[] args)
   {
       System.out.println("Hello, world!");
   }
}
```

JavaScript:

```
// Ausgabe in der Konsole
console.log('Hello world!');

// Dialog-Fenster
alert('Hello world!');

// im HTML Dokument:
document.write('Hello world');
```

Objective-C:

```
#import <stdio.h>

int main(void)
{
    NSLog(@"Hello, world!\n");
    return 0;
}
```

PHP:

```
<?php echo 'Hello, world!' ?>
```

Python (>= 3):

```
print("Hello, world!")
```

Bei Interesse empfehle ich unbedingt helloworldcollection.de! Da werden mittlerweile 481 Hello World Beispiele gezeigt (bei Veröffentlichung dieses Buches).

Fazit

Selbst die längste Reise beginnt mit dem ersten Schritt. Wenn Du Programmieren lernst begegnet Dir das obligatorische »Hallo Welt« Programm zuerst. Die Implementierung solltest Du nicht als lästigen Einstieg betrachten. Sieh es vielmehr als eine Art Tradition, die von Generation zu Generation weitergegeben wird! Die Implementierung zu Ehren von Brian Kernighan und Dennis Ritchie fühlt sich auch gleich viel besser an, oder?

Variablen sind Platzhalter, die Du vielleicht aus der Mathematik kennst. Dort werden mit ihrer Hilfe Funktionen definiert. Nicht bekannte Werte kannst Du durch Platzhalter ersetzen. Beispiele sind zum Beispiel x, y, z und so weiter. Auch der bekannte Satz des Pythagoras ($a^2+b^2 = c^2$) greift auf Variablen zurück. Und ähnlich wie in der Mathematik kannst Du sie auch bei Computerprogrammen verwenden. Das Konzept ist nur etwas umfangreicher, da Daten zwischengespeichert werden müssen.

Variablen in der Software Entwicklung

In der Welt der Software sind Variablen komplexer als in der Mathematik. Wird in einer Variable ein Wert gespeichert, wird er im Arbeitsspeicher abgelegt. Es muss also Speicher dafür reserviert werden. Das übernimmt in manchen Sprachen wie C/C++ der Entwickler, andere Sprachen nehmen Dir die Speicherverwaltung ab. Außerdem kennt die Mathematik keine Konzepte wie zum Beispiel Zeichenketten. Die sind aber in einem Computerprogramm ständig zu finden.

Zwischenspeichern von Werten

Variablen sind flüchtig. Der Inhalt, der in ihnen gespeichert wird, geht bei Programmende verloren. Wenn wichtige Daten dauerhaft gespeichert werden müssen, zum Beispiel eine Benutzereingabe in Deiner Software, dann musst Du das über separate Programmlogik realisieren. Der Wert kann beim jedem Programmstart wieder in eine Variable geladen werden.

Aber auch das musst Du als Entwickler so vorsehen.

Stell Dir als gedankliches Modell einen Schrank oder eine Kommode mit vielen Schubladen vor. Jede Schublade ist eine Variable, in der ein Wert abgelegt werden kann. Du hast jederzeit Zugriff auf den Inhalt, kannst neue Werte hineinlegen oder die Schublade wieder vollständig freigeben. Der Name der Variable ist eigentlich nur ein Alias für die Adresse des Speicherbereichs. Im Beispiel vom Schrank ist das beispielsweise die »dritte Schubladen von oben«, beim Computer sind es hexadezimale Zahlen. Sobald das Programm beendet wird, sind die Schubladen wieder leer. Alles was nicht gespeichert wurde, geht verloren.

Datentypen von Variablen

Variablen haben einen sogenannten Datentyp. Damit der Wert in einer Variable gespeichert werden kann, muss der Datentyp kompatibel sein. Das wird von Sprache zu Sprache unterschiedlich gehandhabt. Sprachen wie zum Beispiel C oder Java verlangen vom Entwickler die explizite Definition der Datentypen. Dies nennt man statische Typisierung. Es muss genau feststehen welcher Datentyp vorliegt. Das kann eine Ganzzahl, eine Fließkommazahl oder auch eine Zeichenkette sein.

Dynamische Typisierung

Es gibt mittlerweile auch viele Sprachen mit »dynamischer Typisierung«. Vornehmlich Skriptsprachen wie PHP, Python

oder JavaScript fallen in diese Kategorie. Variablen haben weiterhin einen Datentyp. Der wird aber nicht vom Entwickler angegeben. Außerdem kann er sich im Programmverlauf jederzeit ändern. Initialisiert wird die Variable zum Beispiel mit einer Ganzzahl. Anschließend wird ein Wahrheitswert hinterlegt und dann ein Objekt. Bei einer Sprache wie JavaScript funktioniert das genau so. Um den Typ kümmert sich der Interpreter im Hintergrund. Der Programmierer muss ihn nicht angeben:

Ein Beispiel auf Basis von Javascript:

```
var x = 1;
x = "eine Zeichenkette";
x = 1.5;
x = 42;
```

Du sieht hier, dass unterschiedliche Arten von Werten zugewiesen werden, ohne das etwas beachtet werden muss.

Statische Typisierung

Die Arbeit mit statischer Typisierung klingt aufwendiger. Ist sie grundlegend auch. Entwickler müssen genau wissen welchen Datentyp eine Variable haben soll. Der wird schon im Quelltext festgelegt. Treten Fehler auf, werden sie bevor das Programm übersetzt wurde abgefangen. Wenn Du in einer Variable vom Typ Integer (Ganzzahl) eine Fließkommazahl speicherst, wird der Versuch einen Fehler erzeugen oder mindestens einen Warnhinweis ausgegeben. Das ist ein großer Vorteil, da solche Fehler zur Laufzeit manchmal schwer zu entdecken sind.

Hier ein Beispiel für statische Typisierung in Java:

```
int a = 17;
int b = 3;
int summe = a + b;
double i = 1.5;
double j = 3.1;
```

Es ist klar erkennbar, dass die Variablen einen eindeutigen Typ erhalten haben. Der wird bei der Deklaration der Variable festgelegt.

Fazit

Variablen sind in jeder Programmiersprache präsent und sehr wichtig. Erst sie ermöglichen die dynamischen und komplexen Programme überhaupt. Du wirst schnell feststellen, dass Variablen in jeder Software unverzichtbar sind. Daher sind sie fester Bestandteil jeder Programmiersprache.

Im Kapitel zu Variablen habe ich es bereits angesprochen. Es gibt verschiedene Datentypen. Welche Dir zur Verfügung stehen hängt von der verwendeten Programmiersprache ab. Insgesamt gibt es jedoch, angelehnt an die Mathematik, nur eine überschaubare Anzahl. In diesem Kapitel lernst Du die grundlegenden Typen kennen.

Wo begegnen Dir Datentypen?

Fragst Du Dich nun, warum es überhaupt Datentypen gibt? Meist gibst Du in einem Programm einfach Daten ein. Als Anwender merkst Du von der Art Deiner Daten nicht viel. Dir ist egal ob es sich um Buchstaben, Zahlen, Fließkommazahlen oder Zeichen handelt. Hinter den Kulissen der Software sieht es anders aus. Als Entwickler musst Du Dir dazu sehr wohl Gedanken machen.

Warum Datentypen?

Datentypen werden verwendet, um Speicher für Variablen im Arbeitsspeicher zu reservieren. Jede Variable benötigt Speicherplatz, um Werte abzulegen. Wie viel das ist bestimmt unter anderem der Datentyp. So lässt sich der Bedarf genau berechnen. Manche Programmiersprachen nehmen Dir heutzutage die Entscheidung über den Datentyp ab. Sie wählen ihn aber im Hintergrund weiterhin aus, nur eben automatisch. Das bedeutet nur, dass der Datentyp dynamisch verwaltet wird. Er existiert trotzdem.

Neben der Berechnung von Speicherbedarf gibt es eine

weitere Aufgabe für Datentypen. Sie werden genutzt, um die Werte zu interpretieren. Auf unterste Ebene werden alle Daten als Kombination aus 1 und 0 gespeichert. Die Bits repräsentieren Zahlen, dargestellt im binären System. Auch Text wird notgedrungen als Zahl abgespeichert. Es gibt spezielle Übersetzungstabellen, wie die ASCII-Tabelle, anhand derer ein Buchstabe einer Zahl zugeordnet wird und umgekehrt. Die heutigen Computer zeigen bisher keine Anzeichen von eigenständiger Intelligenz. Es muss also bspw. deutlich gemacht werden, wie die Daten zu interpretieren sind. Handelt es sich um Zahlen oder um Text? Und eben diese Unterscheidung ermöglichen die Datentypen ebenfalls. Auch ob eine Zahl ein negatives Vorzeichen haben kann oder nicht, wird über den Datentyp festgelegt.

Welche Datentypen gibt es?

Welche Datentypen im Detail existieren, und wie diese verwendet werden, ist von Sprache zu Sprache verschieden. Es gibt jedoch, in der einen oder anderen Form, die grundlegenden Typen wie:

- Integer (ganze Zahlen)
- Fließkommazahlen (Zahlen mit Dezimalstellen)
- Zeichenketten / Strings (Textinhalte)
- boolesche Werte (Wahrheitswerte ja/nein)

Wie gesagt gibt es grundlegend unterschiedliche

Ausprägungen in den verschiedenen Programmiersprachen. So kennt Java zum Beispiel nicht nur den Datentyp Integer. Es gibt viele Abstufungen für ganze Zahlen. Der Typ »byte« kann nur Werte von -128 bis einschließlich 127 umfassen. Der Datentyp »long« kann, als Beispiel am oberen Ende der Integer Werte, extrem lange Zahlen speichern:

9.223.372.036.854.775.808

Das ist schon wirklich hoch und zum Beispiel für astronomische Berechnungen interessant. Dazwischen gibt es selbstverständlich noch weitere Abstufungen. Wichtiger als die Ausprägungen in Java ist an dieser Stelle aber eher, dass es überhaupt verschiedene Wertebereiche geben kann. Andere Sprachen wie JavaScript sind schlichter und kennen nur den Typ “Ganzzahl”.

Fazit

Welche Datentypen Dir konkret zur Verfügung stehen, hängt stark von der Programmiersprache ab. Dennoch orientieren sich die elementaren Datentypen immer an den Anforderungen unserer Umwelt. Und mit den oben genannten Datentypen können die meisten Aufgaben bewältigt werden. Seien es Zeichenketten wie Namen, Preisangaben, das Alter einer Person oder Wahrheitswerte, die mit Ja oder Nein beantwortet werden können.

Nach dem Überblick über die Datentypen geht es nun um Operationen in Programmiersprachen. Du lernst ein paar Grundlagen zu Operatoren, erfährst welche Arten es gibt und lernst Beispiele kennen. Ich beschränke mich dabei bewusst auf die Elemente, die in den meisten Programmiersprachen angeboten werden.

Operatoren in der Programmierung

Ähnlich wie in der Mathematik werden bei der Software Entwicklung Operatoren verwendet, um eine Operation zu kennzeichnen. Das lässt schon der Name vermuten. Ich behaupte, dass Du viele Operatoren bereits aus der Mathematik kennst. Die brauchst Du auch als Programmierer. Ohne die grundlegenden Rechenoperationen kommt keine Software aus. Am Ende bedeutet der Begriff Computer übersetzt nicht umsonst »Rechner«.

Es gibt Zuweisungsoperatoren, Vergleichsoperatoren für den Vergleich von Werten, arithmetische Operatoren zum Rechnen und auch die logischen Operatoren für die Aussagenlogik. Die folgenden Abschnitte behandeln die verschiedenen Typen im Detail.

Der Zuweisungsoperator

Der vermutlich am meisten verwendete Operator ist der Zuweisungsoperator, das Zeichen »=«. Durch den mathematischen Hintergrund wird eine Zuweisung schnell als »ist gleich …« gelesen. Das ist nicht ganz korrekt. Vollständig

lautet die Aussage, dass dem linken Operanden der Wert des Ausdrucks auf der rechten Seite zugewiesen wird. Eine sprachliche Bezeichnung könnte »x wird auf den Wert von … gesetzt« sein. Ein Beispiel:

```
var x = 1;
var y = 1 + 1;
```

Die erste Zeile müsste also sprachlich als »x wird auf den Wert von 1 gesetzt« ausgedrückt werden. Noch deutlich wird es in der zweiten Zeile. Hier nimmt die Variable das Ergebnis des Ausdrucks an. Es wird nicht »1 + 1« gespeichert, sondern der errechnete Wert »2«.

Manche Programmiersprachen bieten zudem kombinierte Operatoren an. Die vereinfachen die Zuweisung, in dem sie direkt Operationen wie die Addition, Subtraktion, Multiplikation und Division mit der Zuweisung verknüpfen:

```
var x = 1; // x ist 1
x += 2; // x ist 3
x -= 1; // x ist 2
x *= 5; // x ist 10
x /= 2; // x ist 5
```

In den Beispielen wird der alte Wert mit dem Wert rechts vom Gleichzeichen verrechnet. Die Rechenarten stehen links vom Gleichzeichen. Das Ergebnis wird direkt der Variable zugewiesen. Ausgeschrieben würde dort bspw. statt »x += 2« stehen: »x = x + 2«. Es ist nur eine verkürzte Schreibweise.

Vergleichsoperatoren

Ähnlich wichtig sind Vergleiche. Ohne sie kannst Du keine Fallunterscheidung realisieren, keine Bedingungen definieren

und auch nicht auf Benutzereingaben reagieren.

Die Ähnlichkeit zur Mathematik ist natürlich auch hier spürbar. So kennt man die folgenden Operatoren aus dem Schulunterricht, teilweise mit etwas anderer Schreibweise:

- »<« – kleiner
- »>« – größer
- »<=« – kleiner oder gleich
- »>=« – größer oder gleich

Hinzu kommen noch weitere Vergleiche, die in der Praxis quasi täglich genutzt werden:

- »==« – ist genau gleich
- »!=« – ist ungleich

Die letzten beiden Vergleichsoperatoren sind besonders interessant, da nicht nur Zahlenwerte damit verglichen werden. Auch boolesche Werte lassen sich so überprüfen.

Hier einige Beispiele:

```
var i = 1;
var j = 2;

if (i < j) {
  // wahr;
}

if (i > j) {
```

```
  // Nicht wahr;
}

if (i == j) {
  // Nicht wahr;
}

if (i != j) {
  // wahr;
}
```

Arithmetische Operationen

Ohne Rechenoperationen würde ein Rechner ziemlich alt aussehen. Daher ist jede Sprache in der Lage zu rechnen. Genau dafür nutzen wir die bekannten arithmetischen Operationen wie bspw.:

- »-« – Subtrahieren
- »+« – Addieren
- »*« – Multiplizieren
- »/« – Dividieren

Hinzu kommen in manchen Sprachen noch besondere Operationen, zum Beispiel die Inkrement- und Dekrement-Operatoren:

- »variable++« – Wert der Variable um eins erhöhen (Inkrementieren)
- »variable–« – Wert der Variable um eins verringern (Dekrementieren)

Beispiele:

```
var meinWert = 1;
meinWert++; // meinWert ist nun 2
meinWert--; // meinWert ist nun wieder 1
```

Logische Operationen

Den Abschluss bilden die logischen Operatoren. Du kannst mit ihnen Aussagen verknüpfen. Auch hier orientiert sich die IT-Welt an der Mathematik. Genau genommen handelt es sich um Aussagenlogik. Die grundlegenden Operatoren sind folgende:

- »||« – Das logische »Oder«
- »&&« – Das logische »Und«
- »^« - Das exklusive »Oder«

Die Verwendung ist denkbar einfach. Wenn zwei Bedingungen verknüpft werden müssen, kannst Du die oben genannten Operatoren verwenden:

```
var a = 3;
var b = 4;
var c = 5;
var d = 6;

// beide bedingungen müssen zutreffen. gelesen:
// - Wenn a kleiner als b UND c kleiner als d...
if (a < b && c < d) {
  // trifft zu
}

// nur eine Bedingung muss zutreffen. gelesen:
```

```
// - Wenn a größer als b ODER c kleiner als d...
if (a > b || c < d) {
  // trifft auch zu!
}
```

Das “logische Und” erzwingt, dass beide Aussagen wahr sein müssen. Hier müsste also “a” größer als “b” und gleichzeitig “c” kleiner als “d” sein.

Durch das “logische Oder” setzt Du durch, dass mindestens eine der beiden Aussagen wahr sein muss. Allerdings können ebenso beide Aussagen zutreffen.

Über ein “exklusives Oder” prüfst Du, ob entweder die eine oder die andere Aussage wahr ist. Der Unterschied zum logischen Oder: es dürfen nicht beide gleichzeitig wahr sein.

Fazit

In diesem Artikel gab es einen grundlegenden Überblick über Operatoren, die in vielen Programmiersprachen verwendet werden. Das nächste Kapitel behandelt ein weiteres wichtiges Konzept, die Dokumentation von Quellcode mittels Kommentaren.

Nach einigen praktischen Elementen steht dieses Kapitel im Zeichen der Dokumentation von Quellcode. Es geht um Kommentare!

Warum Kommentare schreiben?

Sobald Du etwas Praxiserfahrung gesammelt hast, werden Dir viele Dinge leichter fallen. Programmieren wirkt zwar kompliziert, ist es aber nicht! Die Kunst ist der Blick für das große Ganze, für aufgeräumten Code und gut strukturierte Software. Genau dafür brauchst Du Erfahrung. Die einzelnen Zeilen zu implementieren ist, wenn erst mal ein Konzept steht, relativ einfach. Und da lauert eine große Gefahr.

Während der Implementierung Deiner Klassen und Methoden bist Du »voll im Thema«. Das sieht aber schon nach einigen Tagen oder Wochen ganz anders aus. Sicher ist nicht jede Stelle im Code unverständlich. Aber es gibt sie, die etwas undurchsichtigen Stellen an denen Dein spezielles Hintergrundwissen gefragt war. Zusammenhänge sind nach einiger Zeit oft selbst für den Urheber nicht mehr direkt ersichtlich. Gut, wenn Du für diesen Fall mit Kommentaren vorgesorgt hast!

Lesbarkeit von Quellcode

Lesbarer, aufgeräumter und guter Quellcode ist eine Kunst. Dazu wurden bereits viele Bücher verfasst. Das lässt sich nicht in einem Absatz erklären. Grundlegend lässt sich aber sagen, dass guter Quelltext fast wie ein Text gelesen werden kann.

Dazu sind zum Beispiel sprechende Bezeichner wichtig. Die Aufgabe von Funktionen, Methoden, Objekten, Klassen, Parametern und Variablen sollte schon am Namen abgeleitet werden können. Auf kryptische Abkürzungen kannst Du also verzichten! Es gibt darüber hinaus noch weitere Faktoren, auf die ich jetzt aber nicht weiter eingehen werde.

Unsere Aufmerksamkeit gilt nun gezielt den Kommentaren! Denn die setzen die ein oder andere Codezeile erst in das rechte Licht. Eine Analogie dazu: Wenn eine Funktion oder Methode das Kapitel von einem Text wäre, kannst Du Dir den Kommentar als eine Überschrift vorstellen. In wenigen Worten wird die Aufgabe dokumentiert und zusammengefasst.

Außerdem werden mit Kommentaren knifflige Stellen kenntlich gemacht. Auch übergebene Parameter werden dokumentiert, erwartete Datentypen angegeben und andere Metadaten dokumentiert.

Was genau sind Kommentare?

Ein Kommentar ist eine Anmerkung im Code. Er dient dem menschlichen Verständnis. So kannst Du direkt im Quellcode Hinweise hinterlegen, die bei der Programmausführung ignoriert werden. Je nach Sprache gibt es unterschiedliche Wege um Kommentare zu kennzeichnen. Hier einige Beispiele:

```
/*
  Block Kommentar
  in mehreren Zeilen

  Java, C, C++, Objective-C, PHP, ...
```

```
*/

// Kommentar in einer einzelnen Zeile, weit verbreitet

# manche Sprachen erlauben Kommentare mit dem "#"

; auch Semikolon am Anfang kommt vor (z.B. SQL)
```

Wie genau die Kommentare in der Programmiersprache Deiner Wahl markiert werden, entnimmst Du am besten der Dokumentation.

Richtlinien zum Kommentieren?

Schwieriger zu definieren ist die richtige Menge von Kommentaren. Eine goldene Regel dazu lautet:

»So viel wie nötig, so wenig wie möglich«.

Das ist zugegeben schwammig. Anders ausgedrückt würde ich sagen, die Menge macht das Gift. Zuviel und die Übersichtlichkeit leidet. Zu wenig, und man erkennt vielleicht nicht auf Anhieb die Bedeutung oder die Funktionalität an konkreten Stellen.

Ein guter Vergleich ist das gezielte Markieren von einzelnen Wörtern oder Sätzen in einem Text, zum Beispiel mit Fettschrift oder durch Unterstreichen. Dort wird der optische Hinweis auch nur gesetzt, wenn die Stelle oder Textpassage für das Verständnis wichtig ist. So kannst Du dem Leser das Überfliegen von Texten erleichtern. Einige Konventionen zur Verwendung von Kommentaren, findest Du im nächsten Abschnitt.

Aufgaben von Kommentaren

Es gibt standardisierte Aufgaben für Kommentare, die mehr oder weniger in jeder Programmiersprache anzutreffen sind. Zu Beginn eines Dokuments kann zum Beispiel die generelle Aufgabe der Datei hinterlegt werden. Wie schon angesprochen kann Quellcode über Kommentare unterteilt oder gegliedert werden. Auch die konkrete Bedeutung einzelner Zeilen wird über Kommentare deutlich gemacht.

Darüber spielen Kommentare in der Entwicklungsphase noch eine wichtige Rolle. Sie werden vom Compiler oder dem Interpreter ignoriert und nicht ausgeführt. Das machst Du Dir bei der Programmierung zu Nutze. So kannst Du ganze Bereiche oder einzelne Zeilen deaktivieren. Du machst aus ihnen einfach einen Kommentar.

Fazit

Kommentare gestalten Deinen Quellcode übersichtlich. Es gibt jedoch neben der reinen Funktion als Hinweis noch eine ganze Reihe weiterer Aufgaben. Wann ein Kommentar gesetzt wird, und wann besser nicht, gehört mit zum individuellen Stil und der kreativen Freiheit des Entwicklers. Vieles ergibt sich durch Programmierpraxis. Auch durch das Lesen von Quellcode anderer Entwickler bekommst Du ein Gefühl dafür, wann Kommentare sinnvoll sind.

Software ohne Kontrollstrukturen wäre ziemlich unflexibel. Wenn Du Programmieren lernst, begegnen sie Dir daher relativ schnell. Sie gehören zu den elementaren Grundlagen jeder Programmiersprache.

Was sind Kontrollstrukturen?

Nach dem Programmstart wird das Programm von Anfang bis Ende abgearbeitet. Das Programm wird Zeile für Zeile ausgeführt. Mit Kontrollstrukturen beeinflusst Du dieses Verhalten. Und dazu gibt es verschiedene Möglichkeiten.

Über Schleifen, die Du im nächsten Kapitel kennen lernst, können Zeilen oder ganze Codeabschnitte wiederholt ausgeführt werden.

In der Praxis mindestens genauso wichtig sind Bedingungen. Die werden verwendet um Code nur dann auszuführen, wenn bestimmte Voraussetzungen erfüllt werden. Das kann der Wert einer Variable, der Rückgabewert einer Funktion oder auch eine Kombination aus mehreren Bedingungen sein.

Bedingte Ausführung von Quellcode

Bei der Entwicklung von Software greifst Du täglich auf »if«-Bedingungen zurück. Sie werden in unterschiedlichsten Situationen verwendet. Für das Verständnis sind zwei Bestandteile wichtig, die Syntax und die Bedingungen selbst. Die Syntax richtet sich nach der jeweiligen Programmiersprache. Die folgenden Beispiele zeigen Bedingungen in JavaScript. Die Syntax ist unter Java, C/C++,

Objective-C, PHP und so weiter aber sehr ähnlich. Hier ein Beispiel:

```
// Basisvergleich
var alter = 18;

if (alter < 18) {
    document.write(“Nicht volljährig, unter 18!");

} else {
    document.write("Volljährig! 18 oder älter!");
}
```

Hier passieren gleich mehrere Dinge. Zum einen siehst Du die Syntax, die mit dem Schlüsselwort »if« eingeleitet wird. Übersetzt bedeutet das Wort »wenn«. Zusammen mit der Bedingung und dem Code ergibt sich gelesen die Aussage »Wenn die Bedingung ... erfüllt ist, führ folgendes aus«. Wird der Ausdruck als wahr ausgewertet, wird der Quellcode zwischen den geschweiften Klammern ausgeführt.

Außerdem gibt es hier einen sogenannten »else«-Zweig. Übersetzt bedeutet »else« soviel wie »ansonsten«. Das spiegelt wunderbar die Natur dieser Anweisung wieder. Der Code in diesem Zweig wird immer dann ausgeführt, wenn die Bedingung nicht erfüllt wurde. Gelesen wird der Bereich also als »ansonsten führe folgendes aus«.

Fazit

Bedingungen sind sehr wichtig. Auch sie sind deshalb fester Bestandteil jeder Sprache. Sie werden verwendet um Vergleiche durchzuführen und Code an Bedingungen zu knüpfen. Aber sie sind nicht die einzigen Kontrollstrukturen. Ein weiteres

Element sind die Schleifen. Die sind Thema des nächsten Kapitels.

Auch Schleifen gehören zu den Kontrollstrukturen. Sie sind mindestens genauso alltäglich wie Bedingungen. In diesem Kapitel lernst Du deshalb das grundlegende Konzept kennen.

Schleifen in der Programmlogik

Codeabschnitte mehrfach durchlaufen ist ein großartiges Hilfsmittel. Schleifen machen genau das, sie ermöglichen es den starren Programmfluss »auf Wiederholung zu stellen«. Prominenter Anwendungsfall ist die Verarbeitung von Arrays. Da muss eine flexible Lösung her. Du weißt zum Zeitpunkt der Entwicklung nicht immer, wie viele Elemente zur Laufzeit enthalten sind. Und selbst wenn, möchtest Du die gleiche Logik gemäß dem DRY-Prinzip (Don’t Repeat Yourself) nicht erneut implementieren.

Typische Eigenschaften

Es gibt unterschiedliche Schleifenarten. Je nach Sprache ist unterschiedlich welche genau Angeboten werden. Teilweise verhalten sie sich auch etwas anders. Wesentliche Eigenschaften haben sie aber alle gemeinsam.

Schleifenkopf

Im Schleifenkopf wird eine Bedingung für die Ausführung definiert. Ein Großteil der Schleifen arbeitet nach diesem Schema. Die Bedingung wird dem auszuführenden Code vorangestellt. Beispiele sind z.B. die while- oder for-Schleife. Diese Art bezeichnet man als kopfgesteuerte Schleifen. Der Schleifenkörper wird nur ausgeführt wenn die Bedingung wahr

ist.

Das Gegenteil sind fußgesteuerte Schleifen. Dort wird die Bedingung nach dem Code definiert. Es handelt sich um eine Abbruchbedingung. Ein wesentlicher Unterschied ist, dass der Schleifenkörper mindestens einmal ausgeführt wird. Erst dann wird die Bedingung zum ersten mal geprüft. Diese Technik wird unter anderem bei der do-while- Schleife angewandt.

Bedingungen zur Ausführung

Die Anzahl der Wiederholungen muss kontrolliert werden. Ansonsten handelt es sich um eine Endlosschleife. Das kann gewollt sein. Aber der Entwickler muss es auch so vorgesehen haben.

Die Wiederholungen werden über die Bedingung kontrolliert. Je nach Schleifentyp wird diese entweder vor oder nach einem Durchlauf ausgewertet. Bei den kopfgesteuerten Schleifen wird vor dem Durchlauf geprüft, ob der Ausdruck wahr ist. Nur dann wird die Schleife ausgeführt.

Handelt es sich um eine fußgesteuerte Schleife, wird der Schleifenkörper mindestens einmal ausgeführt. Im Anschluss wird die Bedingung für den Abbruch ausgewertet. Dies kann in verschiedenen Situationen eine Lösung vereinfachen.

Schleifenkörper

Der Quellcode, der bei jedem Durchlauf ausgeführt wird, steht im sogenannten Schleifenrumpf oder Schleifenkörper. Der wird entweder nach dem Schleifenkopf oder vor dem Fuß der Schleife definiert. Bei JavaScript und vielen anderen

Sprachen wird er zum Beispiel von geschwungenen Klammern umschlossen.

Schleifenarten

Es gibt unterschiedliche Schleifen. Welche Dir zur Verfügung stehen, hängt von der jeweiligen Programmiersprache ab. Zwei der Grundarten sind die while- und die for-Schleife. Die meisten Sprachen unterstützen mindestens diese beiden Varianten.

for-Schleifen

Das Schlüsselwort, mit der diese Schleife eingeleitet wird, lautet »for«. Daher stammt auch der Name. Nach dem Keyword folgt die Definition vom Schleifenkopf. Der besteht aus drei Segmenten. Im ersten Abschnitt wird ein sogenannter Schleifenzähler definiert. Der wird genutzt, um die Anzahl der Wiederholungen zu zählen. Das ist zum Beispiel bei der Verarbeitung von Arrays wichtig. Mit Hilfe dieser Hilfsvariable kannst Du Schritt für Schritt auf jedes Element im Array zugreifen. Ein allgemeines Beispiel:

```
    for ( var i = 1; i <= 5; i++ ) {
        document.write(i);
    }
```

Der Schleifenkopf einer »for«-Schleife besteht aus drei Segmenten. Die werden durch ein Semikolon getrennt:

```
for (… ; … ; …) {
    // …
}
```

Das erste Segment wird nur beim ersten Durchlauf evaluiert.

Es erzeugt die Variable »i« und weist ihr den Wert »1« zu.

Das zweite Segment enthält die Bedingung. Solange sie wahr ist, wird der Körper weiter ausgeführt. Im Beispiel wird geprüft ob der Wert in der Variable »i« kleiner oder gleich fünf ist. Die Antwort ist bei der ersten Prüfung »wahr«. Eins ist kleiner als fünf. Dann folgt der erste Durchlauf. Ausgegeben wird der Inhalt von »i«, die Zahl eins.

Nach dem Durchlauf wird dann das dritte Segment ausgeführt. Dies modifiziert den Schleifenzähler. Im Beispiel wird die Variable »i« inkrementiert, der Wert also um »1« erhöht. Anschließend startet der Prozess von vorne. Die Bedingung wird erneut geprüft. Ist zwei kleiner oder gleich fünf? Das ist der Fall. Die Schleife wird erneut ausgeführt. Nachdem der Schleifenkörper erneut abgearbeitet wurde, word ebenfalls der Schleifenzähler wieder inkrementiert. Weiter geht es mit der erneuten Prüfung der Bedingung.

Der letzte Durchlauf ist schließlich der Fünfte. Die Bedingung wird das letzte Mal erfüllt. Ist fünf kleiner oder gleich fünf? Einen sechsten Durchlauf gibt es nicht, da sechs nicht kleiner oder gleich 5 ist.

while-Schleifen

Eine »while«-Schleife ist der »for«-Schleife sehr ähnlich. Beide Schleifen sind kopfgesteuert und es gibt eine Laufbedingung. Unterschiedlich ist der Aufbau. Es gibt nur ein Segment, die Bedingung:

```
var i = 1;
```

```
while (i <= 5) {
    document.write(i);
    i++;
}
```

Ein Schleifenzähler muss vorab erzeugt werden. Zudem musst Du ihn als Entwickler selbst erhöhen. Es gibt verschiedene Szenarien, mit denen dieses Konzept die bessere Lösung darstellt. Wenn vorab nicht berechnet werden kann wie viele Durchläuft benötigt werden, bietet sich eine while-Schleife an.

Ein Beispiel könnte folgendes Szenario sein. Dein Programm fragt Dich nach einer Zahl zwischen 50 und 100. Anschließend erzeugst Du Zufallszahlen in genau diesem Bereich und vergleichst sie mit der Benutzereingabe. Du möchtest herausfinden wie viele Versuche der Computer braucht, um die Zahl zu erraten. Vorab lässt sich nicht sagen, wie viele Anläufe in der Schleife notwendig sind. Du könntest einfach eine Endlosschleife nutzen und diese mit dem Keyword *break* beenden, sobald Deine Zahl gefunden wurde.

Schleifen beenden

Das Schlüsselwort break kann in einer Schleife verwendet werden, um deren Ausführung komplett zu beenden. Das Beispiel oben könnte in JavaScript wie folgt aussehen. Ich verzichte bewusst auf die Zufallszahl und setze die auf einen festen Wert. Ich denke, dass dies das Verständnis erleichtert. Im Endeffekt musst Du sie nur mittels *Math.random()* erzeugen und aufbereiten. Wie das geht, lernst Du unter

anderem in den Videotrainings von codingtutor.de:

```
var zufallszahl = 56;
var benutzereingabe = 57;

var i = 1;
while (true) {
    if ( zufallszahl == benutzereingabe ) {
        document.write(“Gefunden in Durchlauf: ” + i);
        break;
    }

    zufallszahl = 56
    i++;
}
```

In diesem Beispiel würde die richtige Zahl immer beim zweiten Durchlauf entdeckt.

Fazit

Schleifen sind in der Programmierung nicht zu ersetzen. Welche zur Verfügung stehen, hängt von der eingesetzten Sprache ab. Die beiden Grundarten, die »for«- und »while«-Schleife, werden jedoch meist unterstützt. Ebenfalls wichtig sind die Funktionen. Sie begegnen Dir heute in vielen Sprache auch in der Objektorientierung. Dort bezeichnet man sie als Methoden.

Funktionen sind in der Entwicklung ein wichtiges Mittel zur Strukturierung von Quellcode. In manchen Sprachen werden sie Subroutinen genannt. Auch im Objektkontext gibt es ein verwandtes Konzept, die Methoden. Dieses essentielle Tool für Entwickler lernst Du nun kennen.

Warum überhaupt Funktionen?

Bei einer Funktion handelt es sich um Unterprogramme - daher auch die Bezeichnung Subroutine. Ganz alte Sprachen kannten zwar Sprungbefehle. Im wesentlichen bedeutete Programmausführung aber die Abarbeitung aller Anweisungen von oben nach unten, eine Zeile nach der anderen. Wenn Du mehrfach die gleiche Logik brauchtest, z.B. für bestimmte Berechnungen, musstest Du die gleichen Anweisungen mehrfach hinterlegen. Eine separate Funktion löst genau dieses Problem.

Ein Beispiel ist die Umrechnung von Dollar nach Euro. Stell Dir vor, Du musst in einer Software an mehreren Stellen die Währung umrechnen. Heute schreibst Du dafür eine zentrale Logik in einer Funktion. An allen Stellen wo die Berechnungen notwendig sind, greifst Du auf sie zurück. Und zwar beliebig oft. Ändert sich der Dollarkurs, änderst Du ihn nur an einer Stelle. Genau, nur in Deiner Funktion.

Merkmale von Funktionen

Funktionen haben unabhängig von der gewählten Programmiersprachen grundlegende Eigenschaften. Das ist in

den meisten Fällen mindestens der Name. Außerdem kannst Du an Funktionen beim Aufruf Parameter übergeben. Das sind Variablen, in denen Du der Funktion Werte zur weiteren Verarbeitung überreichst. Die Details sind von Sprache zu Sprache unterschiedlich. Teilweise legst Du die Anzahl im Code statisch fest. Manche Sprachen erlauben Dir Parameter dynamisch zu erzeugen. Manchmal kannst Du sogar Standardwerte definieren. Die werden genutzt, falls Du den Parameter beim Funktionsaufruf nicht mit übergibst.

Ein letztes Merkmal sind Rückgabewerte. Eine Funktion kann Werte an die aufrufende Stelle zurückgeben. Diesen Wert kannst Du in einer Variablen speichern, auf dem Bildschirm ausgeben oder direkt weiterverarbeiten. Die Rückgabewerte kannst Du so nutzen wie einen statischen Wert, den Du selbst angibst.

Logik in Funktionen auslagern

Das Konzept der Funktionen klingt schon mal gut? Fantastisch! Wenn Du bereits den Wert solcher Vereinfachungen erkennst, ist das eine Menge Wert. In der Praxis ist vieles nicht schwarz oder weiß. Es ist nicht immer so einfach zu entscheiden, wann eine eigene Funktion Sinn macht. Jede Anforderung, jede Aufgabe und Situation kann unterschiedlich gelöst werden. Viele Wege führen nach Rom. Genau das macht Software-Entwicklung so komplex. Aber, es gibt einige Richtlinien.

Eine Funktion sollte nicht zu lang werden. Als Richtlinie gilt

ein Maximum von ca. 50 bis höchstens 100 Zeilen. Klingt ziemlich pauschal? Ist es tatsächlich, da es sich ja nur um einen Richtwert handelt. Die zweite Gedankenstütze ist das *Single-Responsibility-Prinzip*. Jede Funktion soll nur eine Aufgabe haben. Du hast zum Beispiel Mitarbeiter in einer Datenbank. Die willst Du in einer HTML Tabelle ausgeben. Sinnvoll wäre eine Funktion zum Verbindungsaufbau zur Datenbank. Eine weitere liest die Mitarbeiter aus den Tabellen. Die dritte Funktion bereitet die Daten auf. Noch eine andere Funktion erzeugt die HTML-Tabelle. So hat jede Funktion nur genau eine Aufgabe.

Beispiele für Funktionen

Es gibt eine ganz Reihe sinnvoller Beispiele, per Definition endlos viele sogar. Die folgenden Funktionen habe ich mit der Sprache JavaScript umgesetzt. Sie sollen die grundlegende Idee demonstrieren. Es werden klar definierbare Abläufe in einer Funktion implementiert:

Addieren:

```
function addieren(a, b) {
    return a + b
}
```

Währungsumrechnung:

```
function euro2Dollar(euroBetrag) {
    return euroBetrag * 1.14
}
```

Minimalwert bestimmen:

```
function min(a, b) {
    if (a <= b) {
        return a
    }

    return b
}
```

Volljährigkeit prüfen

```
function isVolljaehrig(alter) {
    if (alter >= 18) {
        return true
    }

    return false
}
```

Funktion ohne Rückgabewert

```
function logError(message) {
    window.alert(message)
    console.log(message)
}
```

Es gibt wie gesagt noch viele weitere Möglichkeiten. Wichtiger als die Aufgaben ist die generelle Idee dahinter. Es wird ein Namen festgelegt und Parameter definiert. In der Funktion werden Aktionen ausgeführt. Anschließend gibst Du eventuell einen Wert zurück.

Programmieren gegen Schnittstellen

Ein wichtiges Mittel ist die Programmierung gegen Schnittstellen. Du musst bei der Arbeit mit Funktionen nicht wissen, wie sie ihr Ziel erreichen. Für Dich als Entwickler zählt nur der Aufruf und die Daten, die zurückgegeben werden.

Wenn Du zum Beispiel Zufallszahlen mit Math.random() erzeugst, ist Dir egal wo sie herkommen. Was zählt, ist der Wert den die Methode zurückgibt.

Das macht die Entwicklung im Team viel einfacher. Sobald feststeht welche Funktionen es geben muss, können mehrere Programmierer am gleichen Projekt arbeiten. Werden einzelne Funktionen bereits vorab benötigt, kannst Du mit sogenannten *Mock-ups* arbeiten. Dabei werden einfach statische Daten im passenden Format zurückgegeben. Die Implementierung der Funktion kann später erfolgen.

Funktionen in der Objektorientierung

Die Grundidee wurde in die Objektorientierung übertragen. Methoden sind in vielerlei Hinsicht Funktionen. Der große Unterschied: Eine Methode gehört zu einer Klasse. Und darum geht es im nächsten Kapitel.

Das Konzept der *Objektorientierung* sollte Dir als Entwickler ein Begriff sein. Denn Objekte sind aus der Programmierung nicht mehr wegzudenken - aus gutem Grund! Wieso, weshalb und warum? Lies weiter!

Ein Konzept der physischen Welt

Hast Du schon mal etwas von Objekten gehört oder gelesen? Wenn Du es zum ersten Mal hörst, klingt die Bezeichnung *objektorientierte Programmierung* kompliziert und abstrakt. Dabei ist es das genaue Gegenteil. Das Konzept ist der physischen Welt nachempfunden. Das menschliche Verständnis der Welt wurde auf Software übertragen. Dieses gedankliche Modell macht die Bestandteile für Dich besser greifbar. Das ist, gerade wenn eine Software komplexer wird, ein großer Vorteil.

Die grundlegende Idee: Alle Gegenstände, Personen und Tiere unterscheiden sich. Bei direkter Betrachtung ist es schwer Gemeinsamkeiten von einem Fisch und einem Auto zu erkennen. Doch genau das ermöglichen Objekte. Der Fisch und das Auto werden zu einem Objekt. Alle Objekte haben bestimmte **Eigenschaften**. Sie beschreiben das Objekt. Außerdem können sie unterschiedliche Aktionen ausführen, die sogenannten **Methoden**. Ein Attribut des Autos könnten die Farbe oder seine Motorleistung in PS sein. Beim Fisch ist es die vielleicht die Fischart. Beim Auto gibt es eine Methode beschleunigen. Der Fisch kann schwimmen.

Was ist ein Objekt?

Ein Objekt kann theoretisch alles sein. In Deiner Software repräsentieren Objekte alle Gegenstände, Personen, Werkzeuge oder andere Dinge. Genau wie im echten Leben haben die Objekte Eigenschaften. Die beschreiben alle Merkmale. Ein Auto hat bspw. eine Farbe, eine bestimmte Leistung, eine Höchstgeschwindigkeit, Hersteller und so weiter. Außerdem kann es Aktionen ausführen: lenken, beschleunigen, bremsen und viele weitere. In der Softwareentwicklung nutzt Du dafür Methoden.

Gleiches gilt nicht nur für Autos. Auch bei Fischen, einer Waschmaschine oder einem Rasenmäher gilt die gleiche Idee. In Deinen Programmen werden sie als Objekt dargestellt. Du beschreibst sie über Eigenschaften. Aktionen, die sie ausführen können, werden als Methoden implementiert.

Was sind Klassen?

In der Objektorientierung spielen Klassen eine zentrale Rolle. Sie sind die Baupläne für Objekte. Die Programmiersprache erstellt und initialisiert Objekte auf Basis von Klassen. Formal korrekt ausgedrückt: Ein Objekt ist die Instanz einer Klasse zur Laufzeit. Einfacher umschrieben: Eine Klasse ist der Quellcode den Du schreibst. Ein Objekt entsteht, wenn das Programm ausgeführt wird und die Kopie einer Klasse erzeugt wird. Wie es passiert ist von Sprache zu Sprache unterschiedlich. Eine Programmiersprache bei der Du dieses

Konzept bis in die Haarspitzen kennen lernst, ist Java.

Eine Java Klasse als Beispiel

Die Syntax zur Definition von Klassen ist abhängig von der Programmiersprache. Auch Instanzen der Klasse, also Objekte, erzeugst Du in jeder Sprache etwas anders. Die Basis ist aber ähnlich. Immerhin wird das gleiche Konzept verfolgt. Es ändern sich im wesentlichen nur Details.

Die folgende Java-Klasse könnte einen Apfel beschreiben, zumindest rudimentär:

```
// Apfel.java
public class Apfel
{
    public String farbe = "grün";

    public void entkernen()
    {
        System.out.println("Apfel entkernt");
    }
}
```

Wichtig ist hier weniger die Bedeutung der Keywords wie "public" und "void". Interessanter ist der allgemeine Aufbau. Eingeleitet wird die Klasse mit dem Schlüsselwort "class". Dann folgt der Klassenname "Apfel". Alle Details rund um Java, die Grundlagen und Konzepte wie Objektorientierung findest Du zum Beispiel auf codingtutor.de im Videotraining Programmieren lernen mit Java.

Eigenschaften

Oben in der Klasse siehst Du die Eigenschaft "farbe". Sie ist

vom Typ String. Es handelt sich um den Datentyp String, eine Zeichenkette. Sie wird genutzt um die Farbe des Apfels zu speichern. Es wären natürlich noch weitere Eigenschaften denkbar. Sinnvoller Kandidat wäre die Sorte, ebenfalls als Zeichenkette dargestellt. Auch möglich: süß ja/nein als Wahrheitswert. Welche Eigenschaften benötigt werden, hängt immer vom Kontext und Deiner Software ab. Eine automatisierte Lösung zur Sortierung von Äpfeln würde zum Beispiel mit dem Durchmesser arbeiten. Eine ERP-System für Obst- und Gemüsehändler würde sicher unter anderem das Gewicht oder den Lieferanten erfassen.

Methoden

Methoden sind Aktionen die ausgeführt werden können. Sie orientieren sich an Verben, die mit dem Objekt in Verbindung stehen. Das Beispiel oben bietet die Methode "entkernen". Ein Objekt für eine Datei könnte als Methoden bspw. "laden", "speichern" oder "löschen" anbieten.

Wie entsteht ein Objekt?

Ein Objekt entsteht auf Basis einer Klasse. Innerhalb von Java gibt es dafür das Keyword "new". Dies erzeugt eine neue Instanz der Klasse, das Objekt. Im Fall der Klasse Apfel wäre folgende Zeile notwendig:

```
Apfel meinApfel = new Apfel();
```

Das Objekt wird im Speicher abgelegt. In der Variable "meinApfel" wird eine Referenz gespeichert, ein Verweis auf die Speicheradresse des Objekts. Da Klassen sehr komplex werden

können, ist es weniger aufwendig Adressen statt Daten zu kopieren.

Was ist Objektorientierung?

Eine objektorientierte Sprache verwendet Objekte zur Strukturierung der Software. Die gesamte Architektur wird in einzelne Objekte unterteilt. Ein Paradebeispiel dafür ist Java. Hier ist bereits der Einstiegspunkt der Software in einem Objekt. Es gibt einige primitive Datentypen wie Ganzzahlen (Integer), Fließkommawerte (double) oder auch Wahrheitswerte (boolean). Ansonsten sind alle Bestandteile Klassen und Objekte, die komplexen Datentypen.

Bei Java muss die Hauptklasse der Software eine Methode mit dem Namen "main" enthalten. Die wird beim Start ausgeführt. Sie bietet Dir als Entwickler die Gelegenheit den Programmfluss von dort aus aufzubauen und zu verzweigen. In Verbindung mit der Klasse Apfel könnte eine solche Hauptklasse wie folgt aussehen:

```
public class HalloWelt
{
    public static void main(String[] args)
    {
        Apfel apple = new Apfel();
        apple.entkernen();
        System.out.println(apple.farbe);
    }
}
```

Grundsätzlich zeichnet sich objektorientierte Software durch die Architektur aus. Den genauen Ansatz des Entwurfs zu erläutern ist in Kürze nicht möglich. Auch zu diesem Thema

gibt es viel eigenständige Literatur. In Kurzform besteht die Kunst darin die richtige Unterteilung zu finden. Es müssen möglichst sinnvoll eingeteilte Klassen implementiert werden. Dabei musst Du darauf achten, dass diese möglichst voneinander unabhängig bleiben. Ein wichtiges Instrument in dem Zusammenhang sind die sogenannten Design Pattern (Entwurfsmuster). Aber auch da gilt: eine detaillierte Ausführung sprengt an der Stelle komplett den Rahmen. Es gibt separate Literatur zu diesen Themen.

Welche Objekte brauchst Du?

Wenn Du eine Software planst, besteht die größte Herausforderung in deren Architektur. Das Thema Software-Entwurf füllt eigene Bücher, Vorlesungen und Seminare. Es gibt also keine pauschale Antwort.

Aber es gibt eine Leitlinie. In einer Software realisierst Du häufig die Hauptwörter, die im Kontext einer Software vorkommen. Du entwickelst eine App zum Verleih von Büchern? Hier wirst Du mindestens die Bücher selbst als Objekt finden. Auch der Verleih selbst muss erfasst werden. Eine eigene Klasse bietet sich an. An wen verleihst Du die Bücher? An Personen. Richtig. Auch die sind heißer Kandidat für eine eigenständige Entität.

Wenn Dich das Thema Software-Entwurf interessierst empfehle ich Dir Lektüre zu Entwurfsmustern in Verbindung mit der Sprache Deiner Wahl. Als allgemeinen Einstieg in das Thema finde ich zum Beispiel das Buch Design Pattern mit Java

hervorragend.

Welche Programmiersprachen sind Objektorientiert?

Eigentlich ist heute fast jede Sprache in der Lage mit Objekten umzugehen. Was sich unterscheidet ist das Maß, in dem Objekte eine Rolle spielen. Gerade Java und Python pochen stark auf das Recht hier genannt zu werden. Ebenfalls komplett objektorientiert ist C++. Immerhin ist es die Erweiterung von C um genau dies: Klassen und Objekte. Auch im Rahmen der iOS Entwicklung mit Swift und Objective-C wird stark auf Objekte gesetzt.

Fazit

Objektorientierung ist eigentlich ganz einfach. Die notwendige Syntax verstehst Du an einem Tag. Die große Kunst liegt in der Abbildung komplexer Strukturen und Zusammenhänge. Eine pflegeleichte, übersichtliche und saubere Strukturierung erfordert viel Erfahrung.

Und trotzdem beginnt auch hier die Reise mit dem ersten Schritt. Es gibt als Programmierer keinen echten Weg, der an Objekten vorbeiführt. Es gibt nur wenige Sprachen, die ohne auskommen. Irgendwann wirst Du mit Sicherheit auf Objektorientierung stoßen.

Fragst Du Dich, wie Du nun loslegen sollst? Der folgende 10-Punkte-Plan ist ein möglicher Weg in die Software-Entwicklung. Es ist mindestens eine Inspiration. Anschließend bist Du definitiv in der Lage Dich allein zurecht zu finden. Im nächsten Kapitel zeige ich Dir außerdem wie Du Java lernen könntest.

1. Einsteigen und Grundlagen lernen

Wie überall im Leben gilt auch hier: Bevor Du Laufen lernst, musst Du erst gehen können. Du solltest Dir durchaus Zeit nehmen und die Grundlagen verinnerlichen. Dazu gehören bspw. generelle Konzepte der Programmierung wie:

- Variablen
- Datentypen
- Kontrollstrukturen (z.B Bedingungen und Schleifen)
- Code-Blöcke
- Funktionen und Methoden

Tendenziell ist es weniger wichtig, in welcher Sprache Du diese Grundlagen verstehst. Wichtig ist, dass Du sie wirklich ausgiebig nutzt verinnerlichst.

Es spricht allerdings einiges dafür, Dir diese Grundlagen mit einer “zielführenden” Sprache beizubringen. Das gewährleistet langfristiges Lernen. Es bringt nichts wenn Du Dich täglich mit einer Sprache befasst, aber gar keinen Spaß dabei hast. Such

Dir eine Richtung aus, wähl danach eine Sprache und befass Dich intensiv damit. Grundlagen in einer Sprache lernen, die Du später gar nicht unbedingt brauchst ist wenig sinnvoll. Wieso ein halbes Jahr mit C++ befassen, wenn Du eigentlich Webanwendungen umsetzen willst?

2. Eine objektorientierte Sprache lernen

Grundsätzlich gilt: Ein erfahrener Entwickler beherrscht mehr als nur eine Sprache. Du solltest in unterschiedlichen Bereichen experimentieren und Erfahrungen sammeln. Nur so kannst Du aus verschiedenen Blickwinkeln hinter die Kulissen von Software blicken.

Mindestens eine objektorientierte Sprache solltest Du aber beherrschen. Meine Empfehlung ist an dieser Stelle ganz klar Java. Anfänger können daraus großen Nutzen ziehen:

- plattformunabhängig (Programmieren für auf Windows, Linux und den Mac)
- viele praktische Anwendungen (Paradebeispiel Android)
- strenge und mächtige Objektorientierung
- Entwickler muss sich im Detail mit Datentypen befassen
- mehr als umfangreiche Standardbibliothek
- Syntax ähnlich bei C/C++, Objective-C, PHP und weiteren Sprachen

Das sind mit die wichtigsten Aspekte. Es gibt ggf. auch Szenarien, in denen eine andere Sprache optimal wäre. Wenn z.B. Steuerung von Robotern oder Hausautomatisierung ein Traum ist, kommen doch wieder hardwarenahe Sprachen in Betracht. Es gibt immer Ausnahmen. Die pauschale Empfehlung ist jedoch *Java*.

3. Praxis, Praxis, Praxis

Du lernst nur Programmieren indem Du Software schreibst. Du brauchst Praxis. Viel Praxis. Der ideale Einstieg am Anfang sind zum Beispiel Seminare, Videotrainings oder Vorlesungen. Auch Bücher können ein Einsteig sein. Aus Erfahrung ist die Lernkurve bei der Arbeit mit Büchern jedoch deutlich steiler. Es ist von unschätzbarem Wert, wenn Du jemanden bei der Programmierung beobachten kannst. Die Gedankengänge bei der Entstehung von Code kann ein Buch nicht transportieren.

Anfangs ist es wichtig minimale Programme umzusetzen. Hier kommt es nicht auf Perfektion an. Wichtiger ist es Dich mit den Grundlagen auseinanderzusetzen. So entwickelst Du ein Gefühl für die Syntax. Oft findest Du auch Übungsaufgaben und Anregungen für Software. Optimal um Grundkenntnisse zu erlangen. Die Komplexität kannst Du Schritt für Schritt steigern.

Wirkliche Erfahrungen sammelst Du dann am besten durch eigenständige Projekte. So lernst Du im Laufe der Zeit immer mehr Teile und Aspekte einer Programmiersprache kennen. Vor allem machst Du dabei unweigerlich Fehler. Die

aufzuspüren und zu beheben schult unheimlich. Der Lerneffekt ist bei vollständigen Projekten, die mit einem “leeren Blatt” beginnen, am größten. Zudem macht es auch einfach großen Spaß an eigenen Ideen zu arbeiten.

4. Fremden Quellcode lesen

Genau so wichtig wie Software zu schreiben, ist es Software von anderen Entwicklern zu lesen. Daran führt später ohnehin kein Weg vorbei. Je eher Du damit beginnst, desto besser.

Das Positive ist: Du kannst von der Erfahrung anderer Programmierer profitieren. Software Architektur, Organisation von Quellcode sowie Code-Style (Einrückung, Namensgebung für Variablen, und so weiter) sind hier spannende Aspekte.

Wenn Du die Logik von anderen Programmierern analysierst und die einzelnen Methoden- oder Funktionsaufrufe logisch nachvollziehst, lernst Du immer etwas neues. Du solltest nicht einfach ziellos herumsuchen. Wichtig ist, bewusst zu hinterfragen:

- Was würde ich selbst als Programmierer anders machen?
- Warum würde ich Änderungen vornehmen?
- Welche kreativen Lösungswege wurden vom Programmierer gewählt?

Wichtig zudem: Bist Du unerfahren, darfst Du nicht einfach ohne zu hinterfragen Vorbilder suchen. Es ist wichtig, dass Du

Code nicht einfach blind als gutes Beispiel ansiehst. Hilfreich ist in diesem Bezug OpenSource-Software. Die Chance ist hoch, dass auf Qualität wert gelegt wird. Zudem kannst Du im OpenSource-Umfeld immer jemanden Fragen, wenn Du etwas nicht verstehst.

5. Spezifische Anwendung der Sprache

Der nächste Schritt ist es die Sprache in der Praxis einzusetzen. Lern die Möglichkeiten der Sprache kennen. Scheu Dich außerdem nicht etablierte Lösungen wie Frameworks zu verwenden. Das Rad immer wieder neu zu erfinden ist eine Krankheit, die gerade neue Entwicklern häufig entwickeln.

Ein Beispiel aus dem Webumfeld: Wenn Du einen Onlineshop betreiben möchtest, brauchst Du keine eigene Shoplösung mehr zu entwickeln. Hier kannst Du Dich als Entwickler wunderbar in einem vorhandenen Ökosystem einbringen und auf vorhandene Ressourcen aufbauen. Es gibt verschiedene Frameworks wie Oxid, Magento oder Shopware. Sie stellen die Basisfunktionalität bereit. Du kümmerst Dich um spezielle Anforderungen. Als Programmierer kannst Du so mit Deinem Wissen wunderbar Geld verdienen. Du kannst Dienstleistungen anbieten, fertige Erweiterungen für die Systeme verkaufen oder individuelle Aufträge für Kunden umsetzen.

Gleiches gilt in ähnlicher Form für das Java-Umfeld. Auch hier gibt es verschiedene Ökosysteme, in denen man sich einbringen kann. An erster Stelle ist Android zu nennen. Als

Entwickler kannst Du Geld mit der Umsetzung von Apps für Auftraggeber verdienen. Gleichzeitig lernst Du dabei. Ein finanziertes Selbststudium, wenn Du so willst.

6. HTML/CSS

Generell tust Du als Entwickler auch gut daran, Dich mit der Sprache des Web zu befassen. Du muss nicht unbedingt Designer werden, solltest aber durchaus in der Lage sein eigene Webseiten mit HTML und CSS umzusetzen.

Früher oder später sollte daher unbedingt HTML auf dem Plan stehen. Kompliziert ist die Syntax generell nicht. Die kannst Du an einem Wochenende lernen. Schwieriger ist es die Dokumente gekonnt zu strukturieren. Auch das Verhalten der verfügbaren Browser am Markt ist unterschiedlich. Es ist eine Herausforderung HTML-Dokumente mit CSS zu gestalten, die überall gleich aussehen.

Ein weiterer Aspekt der für HTML/CSS spricht: Mobile Webseiten und Apps, die darauf aufbauen. Es gibt viele Apps für Android und iOS, die im Hintergrund eigentlich “nur” aus einer Webseite aus HTML, CSS und JavaScript bestehen. Die Webseite wird in einer App eingebettet und dem Benutzer gezeigt.

7. Unix als Entwickler

Wenn Du als erfahrener Entwickler gelten möchtest, führt an Unix kein Weg vorbei. Als Programmierer kannst Du Dich nirgendwo so gut austoben wie unter Unix. Zum Lernen

optimal: Ein Linuxsystem, dass einfach mal nach Herzenslust untersucht und “auseinander genommen” werden kann.

Ich hatte früh in meiner Karriere das Glück, den Weg in die Unixwelt zu finden. Durch der Game- und Webserver bin ich damit in Berührung gekommen und war schnell fasziniert. Es gibt endlos viele Werkzeuge, Bibliotheken und Anwendungen für Entwickler. Software entwicklen, kompilieren und mit einem Debugger untersuchen macht enorm viel Spaß.

Das eigentliche Unix, ein BSD System, wurde in C geschrieben. Diese Sprache bildet auch heute noch die Grundlage für viele moderne Betriebssysteme und Anwendungen. Wenn Du mehr darüber wissen möchtest wie das Internet aufgebaut und strukturiert ist, solltest Du ein gutes Verständnis für Unixsysteme entwickeln.

Ein Exkurs zum Thema Unix ist alles andere als eine Geschichtslehrstunde. Es ist vielmehr der Eintritt in die Plattform für Entwickler schlechthin.

8. Software-Architektur

Fortgeschrittene Themen sind auf dem Weg zum Profi ebenso unerlässlich. Dazu gehören maßgeblich auch der Software-Entwurf sowie die Architektur von Programmen.

Die Architektur der Software umfasst die Strukturierung und Unterteilung der Logik. Welche Klassen, Pakete und Methoden gibt es? Wie hängen diese Teile zusammen? Wie entwirfst Du ein Design, dass später erweiterbar und übersichtlich ist? Ein

guter Einstieg in dieses Thema sind die Design Patterns, die sogenannten Entwurfsmuster. Die liefern für viele einzelne Szenarien Musterlösungen, unabhängig von der verwendeten Sprache.

Nur eine saubere Trennung von Verantwortlichkeiten, Datenhaltung, Darstellung und Logik sorgt für nachhaltig übersichtlichen Quellcode. Dieser ist nicht das Produkt von Zufall oder viel Erfahrung bei der Arbeit mit Syntax. Nein. Du musst Dich gezielt mit der Struktur befassen. Die Folge davon ist aufgeräumter Quellcode.

9. Software-Qualität

Mindestens genauso wichtig ist das Thema Software Qualität. An erster Stelle stehen hier automatisierte Tests, z.B. Unit-Tests. Ein Thema, mit dem Du Dich früher oder später befassen solltest. Im Kern geht es darum, dass jede einzelne Klasse und Methode genau getestet wird, automatisch. Der generell Ansatz bei der Entwicklung ist folgender:

- ein Feature wird implementiert, das Programm neu übersetzt
- anschließend wird es ausgeführt und getestet

Das birgt Gefahren, die man mit Unit-Tests umgehen kann. Zum einen kann eine Änderung Seiteneffekte an ganz anderer Stelle produzieren. Die werden bei diesem Vorgehen in der Regel übersehen. Niemand testet immer wieder eine komplette Anwendung durch. Das wird gerade bei komplexeren Projekten

ein Problem. Der Überblick geht einfach verloren. Ab einer gewissen Komplexität ist nicht mehr jedes Feature ausreichend prüfbar. Entweder machst Du Abstriche bei der Qualität oder automatisierst diese Tests.

Wenn es die Möglichkeit gibt, dank automatisiert ablaufender Tests, jede Zeile Code unter verschiedenen Bedingungen zu testen, kannst Du Seiteneffekte ausschließen. Zudem kannst Du, theoretisch nach jeder Änderung, die komplette Software neu überprüfen. Zugegeben, Du musst die Tests selbst implementieren. Das spart Dir aber mittel- und langfristig viel Zeit bei der Fehlersuche.

Ein guter Ansatz, um das nachträglich sehr umständliche Erstellen von Tests sicherzustellen: *Test-Driven-Development*! Noch bevor Du eine Methode schreibst, implementierst Du den entsprechenden Test. Ein spannendes Thema, dass ich zur weiteren Recherche empfehle!

10. Jedes Jahr eine neue Sprache

Eine allgemeine Richtlinie empfiehlt, dass Entwickler jedes Jahr eine neue Sprache lernen. Du solltest Dich nicht (zu lange) auf dem ausruhen, was Du bereits erreicht hast. In der Komfortzone findet nur wenig Wachstum statt.

Gerade zu Beginn ist die Gefahr eigentlich eher gering. Aber auf Dauer solltest Du es nicht bei einer Sprache belassen. Lern irgendwann eine zweite, dritte und schließlich eine vierte Programmiersprache. Wie gesagt, generell lautet die

Empfehlung sich jeweils ein Jahr mit einer Sprache auseinander zu setzen. Perfekt geeignet ist dafür der einsetzende Herbst. Die Tage werden kürzer, das Wetter draußen ungemütlich. Perfekt um mehr Zeit am Rechner zu verbringen.

Allerdings solltest Du diese etwas ältere Empfehlung mittlerweile unter anderen Gesichtspunkten betrachten. Die IT-Welt ist deutlich komplexer geworden. Du könntest Dir also zum Beispiel auch pro Jahr ein umfassenderes Framework vornehmen. Wenn Du Dich intensiv mit Java befasst hast, kannst Du Dich locker ein weiteres Jahr mit Android-Apps befassen. Streng genommen wäre das auch “nur” Java-Entwicklung. Es gibt aber in der Android-Welt viel zu entdecken. Das ähnelt fast einer weiteren Sprache.

Insgesamt ist die Idee Dich ständig weiterzuentwickeln. Ein großer Vorteil, wenn Du mehrere Sprachen beherrschst: Jede Sprache ist anders. Du lernst verschiedene Lösungsansätze kennen. Nachdem Du Dich detailliert mit C/C++ beschäftigt hast, kannst Du sicher noch den ein oder anderen neuen Ansatz von Objective-C lernen. Als PHP-Entwickler wird Dir sicher Python oder Ruby noch weitere spannende Eindrücke vermitteln. Die Summe solcher Erfahrungen ist es es schließlich, die einen “vollständigen Programmierer” aus Dir macht.

Eine Programmiersprache lernst Du nicht an einem Tag. Es ist kein Sprint. Du läufst einen Marathon. Das gilt um so mehr wenn Du Java lernen willst. Die Standardbibliothek ist wirklich umfangreich und Java sehr ausgereift. Es gibt viele Facetten und mit Lösungen wie Android weitere umfangreiche Anwendungsgebiete. Dieses Kapitel ist der ultimative Ratgeber zum Programmieren mit Java. Es geht weniger darum ein gutes Buch oder Videotraining zu ersetzen. Du lernst viel mehr wie Du vorgehen kannst.

Überlegungen zu Java vorab

Grundsätzlich halte ich Tatendrang nicht auf! Das gilt für die Studenten in meinen Videotrainings genauso wie für mich selbst. Es gibt ab und zu Themen, auf die ich mich regelrecht Stürze. Bestes Beispiel ist die Sprache Swift. Apple hat sie im Juni 2014 vorgestellt. Ohne groß zu überlegen habe ich viele Projekt direkt nach der Keynote auf der WWDC einfach liegen lassen und habe mir die neue Sprache angeschaut. Diese Einstellung in allen Ehren, empfehle ich für eine nachhaltige Zukunft als Entwickler aber trotzdem etwas Planung. Versteh diese Anregungen also nicht als mahnende Einleitung. Du kannst auch ausprobieren und später reflektieren - überlegen wie es weitergehen soll.

Eignet sich Java als erste Programmiersprache?

Eine spannende Frage, die gerade von Einsteigern oft gestellt wird! Immerhin sollte die Reise in die richtige Richtung gehen. Wer sich über Wochen und Monate mit einem Thema befasst, möchte, dass diese zeitliche Investition irgendwann Früchte

trägt. Es gibt zwar keine Programmiersprache, die für den Einstieg ein Fehler wäre. Die Grundprinzipien lernst Du, mal mehr und mal weniger, überall kennen. Wer aber Android-Apps entwickeln möchte, sollte nicht mit C# oder C++ anfangen. Gleiches gilt für die iOS-Welt. Lieber gleich mit Swift beginnen, wenn Du iPhone-Apps schreiben möchtest. Zielgerichtet lernen. You get the idea.

Unabhängig von der Zielsetzung ist Java wunderbar als erste Sprache geeignet. Sie ist ausgereift, Erwachsen und bietet eine ganze Menge spannender Konzepte. Die Syntax bietet eine ganze Reihe Elemente, ist sehr aufgeräumt und gut durchdacht. Noch wichtiger: Java bietet eine erstklassige Objektorientierung. Zum Lernen ist sie wie geschaffen. Das es mit Android unheimlich prominente Anwendungsfälle gibt, rundet das Paket noch ab.

Gerade für Einsteiger ist interessant, dass die fertigen Anwendungen direkt auf allen großen Plattformen genutzt werden können. Java-Anwendungen laufen auf Linux, Mac OS X und Windows. Dazu sind theoretisch keine Anpassungen notwendig. Das Motto lautet *Write once, run anywhere*. Übersetzt bedeutet es "einmal entwickeln, überall nutzen". In der Realität ist das nicht immer so einfach. Entwickler verniedlichen die Aussage etwas: *Write once, test anywhere* - "einmal entwickeln und überall gründlich testen".

Anwendungsgebiete

Egal welche Sprache Du wählst, Du wirst keine ultimative Lösung für alle Probleme finden. Die wäre dann ein "Jack of all

trades, master of none". Frei übersetzt ein "Tausendsassa" mit vielen Talenten, der aber keines davon wirklich in Perfektion beherrscht. Er salopp formuliert: für alles zu haben und zu nichts zu gebrauchen. Wenn Java Deine erste Sprache wird, solltest Du vorab Ihre Einsatzgebiete und Stärken kennenlernen. Hier sind einige.

In den letzten Jahren hat sich Android als wohl bekanntester Anwendungsfall herausgestellt. Android-Apps werden mit Java entwickelt. Aber auch im Umfeld "größerer" Webanwendungen ist Java sehr beliebt. Große Seiten wie Twitter, LinkedIn oder Tumblr setzen (mittlerweile wieder) vermehrt auf Java. Es ist also auch eine Grundlage für Anwendungen im Web. Das Spring Framework bspw. ist innerhalb des Java-Lagers ein sehr bekanntes Webframework.

Auch Anwendungen auf dem Desktop sind unbedingt bei Anwendungsfällen zu nennen. Hier versteckt sich die Technologie hinter sehr bekannten Titeln wie der Eclipse Plattform, dem Android Studio oder (in Teilen) auch OpenOffice und LibreOffice. Das Spiel Minecraft ist Dir vielleicht ein Begriff? Auch dies wurde mit Java umgesetzt. Wenn Du Minecraft-Mods (Modifikationen - Erweiterungen des Spiels) schreibst, tust Du dies ebenfalls mit Java. Übrigens auch eine mögliche Motivation für den Einstieg in die Software-Entwicklung!

Es gibt, zumindest in der Theorie, auch noch Java-Applets. Die haben sich jedoch, wie Flash, nie vollständig durchsetzen können. Ein großes Problem: Das Web soll auf möglichst jedem

Endgerät funktionieren. Darum sind vollständig offene Standards wie HTML enorm wichtig. Für Java-Applets ist ein spezieller Interpreter zu installieren. Der steht aber, gemessen an den möglichen Endgeräten für das Web, nur auf einer Handvoll der Geräte bereit. Außerdem "fühlen" sich Applets eher wie ein Relikt vergangener Zeiten an. Für die möglichen Lösungen gibt es heute modernere Alternativen.

Welches Ziel verfolgst Du?

Noch bevor Du mit dem Java lernen beginnst, solltest Du Dir über Deine Ziele klar werden. Das muss kein Aufsatz werden. Im besten Fall reicht ein Leitsatz wie "Ich will Android-Apps entwickeln". Das schafft ganz andere Prioritäten als der Wunsch Desktopanwendungen umzusetzen. Vielleicht hast Du ja auch eine ganz konkrete Idee? Formulier sie möglichst genau: "Ich will eine Software zur Verwaltung des Fuhrparks schreiben" ist spezifisch. Das spielt bei der Auswahl der notwendigen Themen später eine bedeutende Rolle.

Vielleicht möchtest Du auch ganz allgemein Programmieren lernen? Oder Java ist eine weitere Sprache die Du lernen willst? Du suchst einen Überblick über die Möglichkeiten. Möchtest Dich orientieren und ein wenig experimentieren? Dann brauchst Du natürlich weniger Planung. Um die Zeit effizient zu nutzen solltest Du trotzdem schon vorher die Schwerpunkte auswählen und nicht blind arbeiten.

Lernen mit System: Java mit der DiSSS-Methode

Genau das ist die Idee hinter der DiSSS-Methode von Tim

Ferriss. Das System beschreibt er selbst als einen Weg, das Dir ein optimales Maß an Minimalismus beschert. Einfacher ausgedrückt kannst Du eine große Menge an möglichen Lerninhalten auf das Minimum reduziert. Und genau dafür ist eine Zielsetzung so wichtig. Nur wenn Du exakt weißt wo Du hin willst, kannst Du auch eine geeignete Route berechnen. Der Weg klingt fast zu einfach. Und genau darin liegt das Geheimnis. Tim Ferriss hat die Schritte zwar nicht erfunden. Er hat aber den Weg definiert. Genau wie in der Software-Entwicklung liegt darin die größte Schwierigkeit. Unnötige Komplexität beseitigen.

Im Kern geht es um einen praktischen Weg, der das Paretoprinzip anwendbar macht. Der auch 80/20 Regel genannte Effekt besagt, dass 80% der Ergebnisse eines Projekts schon mit 20% der Zeit erreicht werden. Die restlichen 20% Projektfortschritt benötigen noch einmal 80% der Gesamtzeit. Das ist nicht aus der Luft gegriffen. Am Anfang des Buches gehe ich da etwas näher drauf ein. Im Umfeld der Programmierung kann ich diese Regel aus Erfahrung als Gesetzmäßigkeit unterschreiben. Der grobe Rahmen steht häufig schnell. Der Teufel steckt dann aber im Detail.

Die Abkürzung DiSSS steht für:

- **D**econstruct
- **S**election
- **S**equencing
- **S**takes

Diese vier Schritte helfen eine passende Themenauswahl zu

treffen.

Eine Programmiersprache beherrschen?

Das Ziel steht fest. Du möchtest Java lernen. Warum ist eigentlich fast egal, zumindest an dieser Stelle. Ich habe eine spannende Frage für Dich: Wann hast Du denn Dein Ziel erreicht? Wann sagst Du, jetzt "habe ich Java gelernt"? Gibt es einen Punkt, an dem Du das Gefühl bekommst Herr der Lage zu sein und die Reise durch das Java-Universum für beendet erklärst? Ich bin mir nicht so sicher. Du lernst nie aus. Insofern könntest Du Dich also endlos mit Java befassen. Aber, Hilfe ist auf dem Weg! Ich behaupte es gibt im wesentlich vier Stufen, über die Du früher oder später stolperst.

Syntax

Egal ob Du Java oder eine andere Sprache lernst. Die Syntax steht zuerst auf dem Lehrplan. Du musst ein grundlegendes Gefühl für die Sprache entwickeln. Werden Klammern genutzt? Gibt es besondere Zeichen für die Nutzung von Variablen (bspw. ein Dollar-Zeichen wie bei PHP)? Werden Zeilen oder Anweisungen mit einem Semikolon beendet? Sicher lernst Du nicht gleich jegliche Sonderzeichen und deren Bedeutung kennen. Die ersten Programme beantworten jedoch meist solche Frage fast automatisch. Daher geschieht dieser Schritt häufig unbewusst.

Grundlagen

Nun geht es an die Grundlagen. Damit Du die ersten Programme ohne Anleitung und Vorgabe schreiben kannst,

musst Du einige Grundlagen verstehen und auch verinnerlichen. Das sind vornehmlich die Punkte, die ich Dir bei dem *Einblick in die Software-Entwicklung* vorgestellt habe. Die lernst Du nur durch Praxis. Gerade dieser Schritt wird häufig als unangenehm wahrgenommen. Die Programme sind noch nicht aufwendig oder komplex. Außerdem musst Du noch häufiger in die Dokumentation schauen. Du kommst nicht richtig in einen Schreibfluss. Als Motivation kann ich Dir versichern, dass es häufig schnell vorbei geht. Halte durch! Aber, ein stabiles Fundament ist wirklich wichtig. Du musst erst lernen zu Gehen bevor Du Laufen kannst.

Zu den Grundlagen zählt z.B. das Deklarieren und Verwenden von Variablen. Außerdem solltest Du wichtige Operatoren kennenlernen und Kontrollstrukturen verstehen. Wenn diese Themen fest sitzen, kannst Du Dich bereits sehr viel sicherer bewegen. Gerade wenn Du Java lernst ist an dieser Stelle zusätzlich die Objektorientierung zu nennen. Du musst unbedingt das Konzept selbst, aber auch die Umsetzung mittels Java verstehen. Die Sprache ist durchweg von Objekten geprägt. Also führt auch daran kein Weg vorbei.

Sprachfeatures und Standardbibliothek

Nach einiger Arbeit, mit den grundlegenden Inhalten, folgt eine wirklich spannende Phase. Du kannst nun mit der Sprache bereits umgehen. Für einfache Bedingungen, Schleifen, Variablen und Objekte brauchst Du die Dokumentation nicht mehr oder eher selten. Das gibt Selbstvertrauen. Noch besser: Du kannst nun so gut mit der Dokumentation arbeiten, dass Du

einfach neue Features aus der Standardbibliothek ausprobieren kannst. Bestens geeignet für eine solche Entdeckungsreise sind die Bücher *Java ist auch eine Insel* und der Nachfolger, *Java SE: Standard-Bibliothek - Insel 2*. Damit bekommst Du über 2000 Seiten Beschäftigung. Mit Sicherheit ausreichend für die nächsten Monate!

Du solltest aber nicht Deine Zielsetzung aus den Augen verlieren. Gerade an dieser Stelle bekommen die richtigen Prioritäten eine besondere Rolle. Du kannst Dich wie gesagt mehrere Monate nur mit den Möglichkeiten aus der Standardbibliothek befassen. Wenn Du aber z.B. Android-Apps entwickeln willst, brauchst Du vieles davon gar nicht.

Best Practices und Architektur

Früher oder später rücken inhaltliche Themen in den Hintergrund. Du wirst merken: Es muss noch etwas anderes geben? Was ist das Geheimnis hinter guter Software? Mindestens werden Dir bei immer komplexer werdender Software auch Probleme rund um die Architektur und die Strukturierung begegnen. Welche Klassen brauche ich? Wie teile ich die geschickt auf? Welche Methoden sind notwendig? Welche Pakete lege ich im Java-Projekt an? Wie verknüpfe ich die möglichst sinnvoll?

Die große Kunst ist häufig nicht, eine Software oder ein Feature einmal zu implementieren. Die Magie beginnt, wenn die Software einfach erweitert, angepasst und gewartet werden kann. Und genau dafür ist eine gute Architektur notwendig. In der Objektorientierung helfen Dir dabei die Entwurfsmuster

(Design Pattern). Die bieten für verschiedene Probleme oder Aufgabenstellungen eine Lösung an. Ein Thema, dass in dem Buch Design Patterns - Elements of Reusable Objekt-Oriented Software wunderbar behandelt wird.

Fazit

Gerade die letzten beiden Punkte werden Dich immer wieder beschäftigen. Es gibt inhaltlich immer wieder etwas neues zu entdecken, neue Framework oder Bibliotheken. Und außerdem steht der IT-Markt nicht still. Also gibt es auch immer wieder neue Ansätze zu Best Practices, verbesserte Architekturen und Weiterentwicklungen.

Wege zum Java lernen

Der Markt hat sich in den letzten 10 Jahren unwahrscheinlich stark entwickelt. Davon betroffen ist natürlich auch die Industrie rund um Lehrmaterial und das Angebot an Dokumentationen. Wenn Du Java lernen willst, ist das für Dich eine erfreuliche Entwicklung. Der Zugang zu Wissen rund um die Programmieren wird einfach erleichtert. Viel mehr Menschen finden so ihren Weg in die Software-Entwicklung. Hier sind meine persönlichen Favoriten beim Lernen. Die folgenden Wege bevorzuge ich.

Seminare

Seminare und Workshops vor-Ort sind unglaublich wertvoll. Du wirst dabei nie eine Sprache umfassend lernen. Aber Du wirst den perfekten Einstieg in die Thematik finden. Viele Fragen, die Dich am Anfang automatisch beschäftigen, werden

durch einen guter Trainer beantwortet. Und wenn nicht fragst Du ihn und bekommst Antworten.

Meist dauern Seminare ein Wochenende oder fünf Werktage (Montag bis Freitag). Durch die Kürze der Zeit ist der Trainer gezwungen die wichtigsten Themen herauszupicken. Er übernimmt also (eher unbewusst) den DiSSS-Schritt für Dich. Größtes Problem bei Seminaren vor-Ort ist der Preis. Der liegt sehr schnell jenseits von 1000 Euro und mehr.

Videotrainings

Gleiches gilt für Videotrainings. Ich habe mich bei der Reihenfolge von Videotrainings und Seminaren sehr schwer getan. Zum einen ist ein Videotraining vergleichsweise geschenkt. Der Preis kann nur durch die Auflage gegenfinanziert werden. Die Kosten für die Produktion und die viele Zeit die investiert werden muss, rechnen sich nur durch die Anzahl der verkauften Exemplare. Für den Kunden ist das jedoch unerheblich.

Ein weiterer Vorteil: Ein Seminar läuft einmal ab. Du musst Notizen machen und Quellcode schreiben. Du kannst in der Regel die Vorträge und Präsentationen nicht erneut anhören. Sicher, es gibt mittlerweile Trainer die Videomaterial bereitstellen. Das ist aber eher die Ausnahme. Solches Material wird schnell im Internet illegal veröffentlicht. Schon bricht die wirtschaftliche Grundlage des Trainers unter ihm zusammen.

Gegenüber dem geschriebenen Buch ist ein Videotraining unheimlich viel mehr Wert. Im Buch siehst Du den fertigen

Quellcode. Der wird zwar erklärt. Aber die Zwischenschritte und die Entstehung des Codes bleibt häufig außen vor. Gerade das ist bei einem Seminar und Videotraining so genial. Ein guter Trainer erzählt welche Gedanken er sich macht und warum er etwas implementiert. Das ist wie Malen lernen oder Lernen Gitarre zu spielen. Mit einem bewegten Bild und Kommentaren geht es viel direkter.

Bei der Umsetzung besonders genial: Du kannst einen Abschnitt mit Erklärung hören und sehen. Anschließend drückst Du Pause und setzt es um. Viel direkter geht es gar nicht.

Bücher

Das soll aber nicht heißen, dass Bücher ausgedient haben. Im Gegenteil. Sie sind zentraler Bestandteil in meinem Lernprozess. Während Seminare und Videotrainings besonders für Einsteiger wunderbar sind, eignen sich Bücher perfekt für fortgeschrittenere und speziellere Themen. Sobald Du etwas sicherer mit Java umgehen kannst, sind sie die beste Wahl um ein breitgefächertes Wissen aufzubauen.

Es gibt eine Menge Literatur zu Java und Themen rund um die Software-Entwicklung. Es ist eine andere Art des Lernens. Außerdem kann in ein Buch mehr Wissen auf engem Raum untergebracht werden. Die Inhalte der Java-Inseln würden mit Sicherheit viele Tage oder sogar Wochen mit Videomaterial füllen. Das würde deutlich über das Ziel hinaus schießen.

Tutorials

Auch die vielen Tutorials im Web sind ein enorm wertvoller Schatz. Es mag der Eindruck entstehen, dass ich den nicht zu schätzen weiß. Stimmt aber nicht. Ich würde nur Einsteigern empfehlen sich gerade zu Beginn auf andere Ressourcen zu verlassen. Ein Blogartikel ist schnell überholt. Gleiches gilt auch für Videomaterial. Hier ist sogar das ständig sichtbare Interface eine besondere Herausforderung. Schon bei kleineren Änderungen der Anordnung oder im Workflow kann schnell ein ganzes Produkt als veraltet gelten. Bei einem Buch kann mit relativ wenig Aufwand Quellcode korrigiert werden. Wenn es um eBooks geht, ist ein Update quasi ein Kinderspiel.

Ein Argument dagegen ist, dass auch Bücher und Videotrainings nur begrenzt aktuell bleiben. Das stimmt. Aber ein kommerzielles Interesse am Produkt ist, wenn es am Markt angenommen wird, immer ein Garant für ein Interesse an hoher Relevanz. Der Hersteller wird schnell für Abhilfe sorgen. Nur so bleibt er schließlich konkurrenzfähig.

OpenSource Software

Eine etwas fortgeschrittenere Quelle für weitere Informationen sind OpenSource-Programme und Bibliotheken. Wer sich bereits etwas sicherer fühlt, findet hier einen Fundus an Informationen, Beispiele aus der Praxis und meist auch guten Stil. Das gilt nicht pauschal. Der Anspruch an Qualität ist im OpenSource Umfeld allerdings generell sehr hoch. Immerhin wird hier oft in der Freizeit gearbeitet. Es gibt keine Deadlines und den Zwang neue Versionen zu liefern. Die Arbeit ist eine ganz andere und für viele Entwickler ein Ausgleich zum

eher stressigen Berufsalltag. Da darf bei einer Lösung der Stil in den Vordergrund rücken und die Wirtschaftlichkeit in den Hintergrund. Insofern: fast immer eine tolle Ressource.

Grundlegende Java Inhalte

Dieser Ratgeber kann und soll nicht die perfekte Ressource zum Java lernen werden. Mir geht es um das "Drumherum", zu zeigen wie Du vorgehen kannst. Aber da ich oben schon die DiSSS-Methode erwähnt habe, macht es Sinn mein Ergebnis für die grundlegendste Schnittmenge an Inhalten vorzustellen. Egal in welche Richtung später die Reise gehen soll, folgende Konzepte sollten unbedingt verinnerlicht werden. Sie begegnen Dir immer wieder.

Entwicklungsumgebung, JDK und Programmaufrufe

Es geht im ersten Schritt um die Vorbereitung der eigenen Workstation auf die Entwicklung. Im wesentlichen sind drei Komponenten notwendig: ein Editor, der Java-Compiler sowie die Laufzeitumgebung. Der letzte Bestandteil wird im JDK von Oracle geliefert. Aber gerade beim Editor gibt es eine ganze Reihe an Möglichkeiten. Ich empfehle Dir unbedingt die kostenlose IntelliJ IDEA von Jetbrains. Dabei handelt es sich um eine vollwertige IDE, die Dich bei Deiner Arbeit unterstützt. Sie ist sehr fortschrittlich, modern und deutlich besser nutzbar als zum Beispiel Netbeans oder Eclipse. Das sieht mittlerweile sogar Google so. Das Android Studio basiert auf IntelliJ IDEA. Wenn Du später Android-Apps entwickeln möchtest, kennst Du gleich die Entwicklungsumgebung.

Syntaxgrundlagen

Du musst nicht alle Schlüsselwörter auswendig lernen. Mach Dich einfach mit ersten Beispielen und einigen Grundlagen vertraut. Java ist streng. Du brauchst zum Beispiel die main()-Methode als Einstiegspunkt Deiner Java-Anwendungen. Auch erste Richtlinien wie die Klammersetzung, oder das Semikolon am Ende einer Anweisung, solltest Du kennenlernen. Die Ausgabe von Daten steht vermutlich bereits ohnehin schon im obligatorischen HelloWorld Programm auf jedem Lehrplan. Aber auch solche Übungen sind wichtig, um mit der Sprache warm zu werden.

Variablen und Datentypen

Das zweite wichtige Kapitel sind Variablen und Datentypen. Ein Grundverständnis für die Vorgänge im Hintergrund ist eine große Hilfe. Du kommt bei Java nicht unmittelbar mit Pointern, also Referenzen auf Adressen im Arbeitsspeicher, in Kontakt. Aber ein ähnliches Konzept gibt es mit Referenzen im Rahmen der Objektorientierung ebenfalls.

Wichtig ist, dass Du weißt was Datentypen sind. Wieso gibt es die? Welche Typen gibt es? Welche Wertbereiche kannst Du in den jeweiligen Datentypen hinterlegen? Wie viel Speicher belegen Variablen eines bestimmten Typs? Ein solides Verständnis hilft Dir später effizientere Software zu schreiben. Gewöhn Dir so früh wie möglich an eine sinnvolle Deklaration Deiner Daten. Es ist später viel schwieriger falsche Gewohnheiten wieder abzulegen. Also Werte vorab genau beurteilen. Du kannst zum Beispiel das Alter einer Person zum wunderbar im kleinsten Integer Datentyp ablegen. Älter als 127

wäre schon eine echte Sensation.

Operatoren und Ausdrücke

Ohne Operatoren und Ausdrücke kommst Du nicht weit. Schon für das Zuweisen von Werten benötigst Du den Zuweisungsoperator (=). Darüber hinaus gibt es noch eine ganze Reihe weiterer. Die mathematischen Operationen werden selbstverständlich unterstützt. Aber auch Themen wie das Inkrementieren und Dekrementieren sind wichtig. Darüber hinaus gibt es auch Aktionen wie die Modulo-Operation. Du solltest ein gutes Verständnis haben und die Basisoperationen fließend beherrschen. Sie begegnen Dir nicht nur in Java, sondern auch in vielen anderen Sprachen.

Kontrollstrukturen in Java

Kontrollstrukturen beeinflussen den Programmablauf. Gemeint sind ganz konkret Bedingungen und Schleifen. Es gibt mit den Keywords for(), while() und do...while() bereits drei wichtige Vertreter der Schleifen. Auch die Fallunterscheidung mit switch() ist unbedingt zu nennen. Bedingungen werden mit dem Schlüsselwort if eingeleitet. Auch die benötigst Du in jedem Projekt häufiger, unabhängig von der gewählten Sprache. Ohne sie wäre jede Software nahezu statisch und einfach eine Kette von auszuführenden Anweisungen.

Ein Vorteil von Java ist die Nähe zur Syntax von C/C++. Es gibt noch weitere Sprachen die sehr ähnlich aufgebaut sind. Du kannst also Java lernen und das Wissen auf andere Programmiersprachen übertragen. Auch deshalb ist Java eine

perfekte Sprache für Einsteiger.

Grundlagen der Objektorientierung

Eines der Hauptargumente für Java ist die tolle Objektorientierung. Mir gefällt die Umsetzung wirklich sehr gut. Unter anderem deshalb wird Java häufig bei der Ausbildung oder im Studium als erste Sprache eingesetzt. Der Themenbereich ist komplex. Du wirst nicht von Heute auf Morgen ein Meister. Aber einmal verinnerlicht, wirst Du in jeder Sprache objektorientiert entwickeln können - und wollen.

Objekte in Java ziehen sich konsequent durch das Gesamtkonzept. Das ist in Java unglaublich ausgereift und ermöglicht eine nahezu perfekte Programmstruktur. An Objekten führt in der Java-Welt wirklich kein Weg vorbei.

Erste Programmideen

Wenn Du Deine ersten Programme schreibst, können häufig sinnvolle Projektideen fehlen. Hier einige Inspiration und Anregungen.

99 bottles of beer

Sehr beliebt ist für den Einstieg nach den ersten Lerneinheiten ein sehr bekanntes Lied: 99 bottle of beer on the wall. Der Songtext zieht sich weitestgehend gleichbleibend durch den Sing. Er zählt aber von 99 bis 0 herunter und gibt zum Schluss einen besonderen Text aus. Perfekt, um ihn mit Java zu generieren. Die Zeilen folgen bis kurz vor dem Ende immer diesem Schema:

99 bottles of beer on the wall, 99 bottles of beer.

Take one down and pass it around, 98 bottles of beer on the wall.

98 bottles of beer on the wall, 98 bottles of beer.

Take one down and pass it around, 97 bottles of beer on the wall.

...

Bis schließlich das Ende erreicht ist:

1 bottle of beer on the wall, 1 bottle of beer.

Take one down and pass it around, no more bottles of beer on the wall.

No more bottles of beer on the wall, no more bottles of beer.

Go to the store and buy some more, 99 bottles of beer on the wall.

Es gibt dazu im Internet auch reichlich Musterlösungen. Eine solche findest Du hinter dem folgenden Link: 99 bottle of beer in Java.

Zahlen raten

Beliebt ist auch das Raten von Zahlen zwischen 1 und *n*. Es wird per Zufall eine Zahl in der genannten Spanne generiert. Der Benutzer wird aufgefordert diese Zahl zu erraten. Das

Programm wertet die Antwort aus. Bei der Verarbeitung kannst Du mehrere Versionen realisieren. Manche sagen nur richtig/ falsch. Andere Implementierungen geben Hinweise ob die gesuchte Zahl größer oder kleiner ist (gerade bei großen Zahlenbereichen). Noch ausgefeilter: Erst ab der 10. falschen Antwort Hinweise ausgeben.

Umrechner

Sehr beliebt sind Programme zur Umrechnung verschiedener Werte. Alles was einem bestimmten Algorithmus folgt oder mit einem definierten Faktor zu berechnen ist, kannst Du leicht umsetzen. Nur einige Beispiele sind die Umrechnung von Dollar nach Euro und umgekehrt, PSI nach BAR (Druckangabe) oder Celsius nach Fahrenheit.

Das kann auch für fortgeschrittene Übungen dienen. So kannst Du zum Beispiel aktuelle Umrechnungskurse für den Euro von der EZB als XML-Dokument herunterladen und mit dem jetzigen Kurs weiterarbeiten.

So bleibst Du am Ball

Nachhaltig wirst Du nur Java lernen, wenn Du die Motivation aufrecht halten kannst. Das klingt mehr als logisch, ist in der Praxis aber nicht immer ganz leicht. Es ist ein wenig wie verliebt zu sein. Die ersten Schmetterlinge im Bauch und das tolle Gefühl, geliebt zu werden. Es fühlt sich alles genau richtig an. Scheinbar nichts kann diese Verbindung erschüttern. Aber diese erste Phase geht irgendwann vorbei. Und dann trennt sich Spreu vom Weizen. Du wirst Dich auch

mal durch vermeintlich weniger spannende Themen arbeiten müssen, wenn Du Erfolg haben möchtest. Hier einige Tipps, um die Motivation aufrecht zu erhalten.

Mit einem Partner lernen

Das beste was Dir passieren kann: Schließ Dich mit anderen Entwicklern zusammen. Vielleicht findest Du eine Usergroup oder einen Stammtisch in Deiner Umgebung? Oder einfach ein Studienkollege, ein Freund oder sonst ein Bekannter? Dieser Austausch ist von unschätzbaren Wert. Du kannst Dich mit jemandem über die Inhalte austauschen, Ihr könnt euch gegenseitig helfen und neue Erkenntnisse teilen. Es ist ein nicht zu unterschätzender Faktor beim Lernen. Und noch spannender: Ihr stachelt euch automatisch gegenseitig an. Schon tauchst Du in neue Themen ein und präsentierst stolz Deine neuste Entwicklung. Gleichermaßen wirst Du von den Experimenten Deiner Kollegen angestachelt. Einfach ein geniales Gefühl und eine Win-Win Situation.

Definierte Projektideen umsetzen

Besonders wichtig sind abgeschlossene Projekte. Du solltest Dir für jedes neue Thema ein eigenes Projekt ausdenken. Setz es konsequent bis zum Ende um. Wirklich. Verpflichte Dich dazu jedes angefangene Projekt fertig zu stellen. Das hilft unglaublich bei der Motivation. Wenn Du einfach ein großes Projekt hast, in dem Du wie auf einer Spielwiese experimentierst, kommst Du nie an einem Ziel an. Die gesamte Reise durch die Java Welt erscheint viel länger. Irgendwann bekommst Du das Gefühl auf der Stelle zu treten. Ich bin selbst Marathon Läufer. Eine beliebte Strategie: Du läufst in

Gedanken niemals 42,195km am Stück. Es sind viel mehr viele kleine Ziele, die unterwegs erreicht werden. Das können erstmal 10km sein. Dann ist fast der Halbmarathon greifbar. Schließlich werden die Abstände kürzer. Gegen Ende ist einfach jeder Kilometer eine Errungenschaft. So kannst Du Dich immer wieder belohnen und brauchst viel weniger "langen Atem".

Die richtigen Gewohnheiten

Wie ich vorab schon erwähnt habe, sind Gewohnheiten ein extrem mächtiges Erfolgsrezept. Lass das Java lernen zur Regel werden. Je knapper die freie Zeit, desto wichtiger sind feste und regelmäßige Zeiten. Das ist beim Sport die gleiche Idee. Wenn Du konsequent Montag, Mittwoch und Freitag etwas Zeit blockierst, wird das schnell zu einer Gewohnheit. Das hat zwei Vorteile: Du selbst fragst Dich gar nicht mehr ob Du nun gerade Lust und Zeit hast. Es wird einfach der Regelfall. Ein weiterer Nebeneffekt wirkt sich auf Dein Umfeld aus. Das kann mit einer festen Planung ebenfalls viel besser planen.

So geht es weiter

Nun habe ich eine ganze Reihe Ideen und Möglichkeiten in den Raum geworfen. Bleibt noch konkrete Empfehlungen für den tieferen Einstieg in die Materie zu liefern. Hier schlage ich Dir folgende Ressourcen vor:

Videotraining von codingtutor.de

Ich stehe mit voller Leidenschaft für das Konzept der Videotrainings von codingtutor.de ein. Alle sind wohl überlegt strukturiert und didaktisch sinnvoll aufbereitet. Das Training

Programmieren lernen mit Java kann ein wunderbarer Einstieg in diese Thematik sein.

Videotrainings von Rheinwerk

Gleichermaßen empfehle ich Dir auch uneingeschränkt die Programmiertrainings vom Rheinwerk Verlag (ehemals Galileo Press, die mit der Java Insel). Ich weiß durch meine Arbeit dort wie akribisch hinter den Kulissen gearbeitet wird. Auch für diese Produkte lege ich daher meine Hand ins Feuer.

Buchempfehlungen zu Java

Außerdem gibt es natürlich auch sehr gute Literatur zu Java. Da sind im wesentlich drei Bücher zu nennen. Das Werk *Programmieren lernen mit Java* sowie *Java ist auch eine Insel*, und die *Java Insel 2* vom Rheinwerk Verlag (ehemals Galileo Press). Speziell die letzten beiden Bücher richten sich auch an fortgeschrittene Entwickler.

Und jetzt Du

Egal wohin Deine Reise gehen soll oder welchen Weg Du wählst: Beginne jetzt! There are seven days in a week. Someday isn't one of them.

In einem Jahr wirst Du Dir wünschen, Du hättest heute begonnen. Also gleich loslegen. Vorher aber noch Deinen Freunden bei Twitter und Facebook von diesem wirklich umfassenden Ratgeber erzählen. :-)

Die folgenden Gewohnheiten findest Du bei erfolgreichen Programmierern. Es lohnt sich, das ein oder andere davon zu abzuschauen. Doch was bedeutet das eigentlich – Erfolg? Vor allen Dingen in diesem Kontext? Jeder hat andere Ziele und Wertvorstellungen. Erfolg im Leben wird subjektiv vor allem daran gemessen. Kann ich überhaupt von “erfolgreichen Entwicklern” sprechen? Ja – unbedingt!

So kannst Du Dir erfolgreiche Entwickler vorstellen

Ich verstehe unter einem erfolgreichen Entwickler eine Person, die sich in der Wirtschaft langfristig behauptet. Eine Festanstellung behältst Du nicht über Jahre, wenn keine Leistung erfolgt. Gleichzeitig ist persönlicher Erfolg aber nicht nur an rein logischen Fakten messbar: renommiertes Unternehmen, finanziell attraktive Entlohnung oder lukrative Benefits. Noch wichtiger finde ich die Zufriedenheit mit den Rahmenbedingungen, den Respekt anderer Entwickler und eine Integration im Team. Du musst Dich jeden Morgen in Deiner Haut wohl fühlen, um es salopp auszudrücken.

Was hilft das große Gehalt, wenn dafür der Rest nicht stimmt? Reichtum kann auf Dauer nicht aufwiegen, dass Du schon Sonntags Abends keine Lust auf die kommende Woche hast. Du tauscht mit 40 Stunden Woche einen erheblichen Teil Deines Lebens gegen Geld. Da solltest Du versuchen, die Zeit gleichzeitig auch für Deine persönlichen Ziele zu nutzen. Dich selbst und Deine Wertvorstellungen verwirklichen.

Auch wenn das persönliche Interesse am Aufgabengebiet nicht vorhanden ist, bist Du nicht erfolgreich. Vielleicht vergeht irgendwann ganz die Lust? Zur Arbeit und in die Firma musst Du Dich zwingen? Auch das kann aus meiner Sicht Geld auf keinen Fall ausgleichen.

Nachdem ich zumindest schematisch mein Bild eines "erfolgreichen Entwicklers" gezeichnet habe, hier nun einige Tipps für Deine erfolgreiche Karriere als Programmierer.

Entwickle mit der Kundenbrille

Software wird in Unternehmen nach Vorgaben erstellt. Vielleicht sind es Dienstleistungen für Auftraggeber. Hier entscheidet der Kunde und steckt Ziele fest. Bei internen Projekten für ein Unternehmen ist bspw. ein Projektleiter oder eine Abteilung der "interne Kunde". Auch hier gilt es Ziele zu erreichen. Das können verbesserter Umsatz im Webshop sein, eine bessere Conversionrate auf einer Landingpage oder ein neues Feature in einer App. Schaust Du durch die Kundenbrille auf ein Softwareprojekt, stehen diese Ziele im Fokus. Mitdenken erwünscht, speziell wenn damit die Ziele noch besser erreicht werden können!

Ein Auge für technische Details und Raffinesse ist nicht falsch. Speziell wenn es mit etwas Weitblick verbunden ist, mögliche Erweiterungen berücksichtigt werden oder der Aufwand für Pflege und damit verbundene Betriebskosten verringert werden. Im Hinterkopf sollte jedoch das KISS-Prinzip bleiben: Keep It Stupid Simple. Anders gesagt: So

einfach wie möglich, so komplex wie nötig.

Mach Dich mit Versionsverwaltung vertraut

Versionsverwaltung ist ein MUSS. Professionelle Entwicklung kann gar nicht ohne eine Versionskontrolle erfolgen. Das ist Fakt. Alle Alternativen Ansätzen führen früher oder später zu Chaos, redundantem Code und einem Mehraufwand bei der Pflege. Noch schlimmer: Änderungen lassen sich später gar nicht mehr im Detail nachvollziehen. Auch ein bestimmter Stand vom Zeitpunkt “x” kann ohne eine Softwarelösung wie Git nicht wiederhergestellt werden. Und das sind nur Probleme die sich ergeben, wenn keine weiteren Entwickler beteiligt sind.

Wenn Du im Team an der gleichen Software arbeitest, entwickeln sich Probleme anderer Qualität. Die Fehlerquellen steigen. Wie stellst Du sicher, dass nicht einer von fünf Entwicklern gerade an der gleichen Datei arbeitet? Noch schlimmer: Was wenn dies aus Versehen passiert? Was wenn dadurch gegenseitig die Änderungen in der gleichen Datei überschrieben werden?

Solche Zustände solltest Du aktiv vermeiden und immer mit einer Versionsverwaltung wie Git arbeiten.

Mach Dich selbst entbehrlich

Ein Software-Entwickler sollte sich in einem Team entbehrlich machen. Dabei ist unmittelbar der Programmierer gemeint – der technische Aspekt Deiner Arbeit. Eine Rolle als

Projektleiter sei hier ausgeklammert.

Andere Entwickler der gleichen Abteilung – ähnliche Erfahrung vorausgesetzt – sollten Deine Projekte übernehmen und verstehen können. Dazu gehört dokumentierter Quellcode, verständliche Implementierung und die Orientierung an sauberen Lösungen. Wenn die Programmiersprache oder ein Framework die Richtung vorgeben, schwimm nicht dagegen an. Schlag nicht durch "Workarounds" oder "Hacks" ein Sonderweg ein, den niemand erwartet. Sei immer Teil der Lösung, nicht Teil des Problems.

Kommunikation ist der Schlüssel

Im Team arbeiten bedeutet vor allen Dingen mit anderen Entwicklern zu kommunizieren: Aufteilen der Aufgaben, gegenseitige Hilfe bei Problemen, Absprachen mit Kunden und Kommunikation über Fortschritte. Auch die Rückmeldungen an den Projektleiter sind eine kommunikativer Prozess. Je besser Du mit Menschen umgehen kannst, desto eher kannst Du auch erfolgreich in einem Team arbeiten.

Sei auch offen und empfänglich für die Anliegen von anderen. Es ist eine Seite der Medaille wenn Du auf andere Leute zugehen kannst. Sei zeitgleich auch jemand den man gerne anspricht. Be easy to talk to.

Leidenschaft für Software-Entwicklung

Es gibt schlaue Leute die gerne das "lebenslange Lernen" erwähnen. Gerade im Bereich der IT ein Dauerbrenner.

Zugegeben entwickelt sich die Computerwelt auch extrem schnell. Die Sichtweise als Software-Entwickler ist jedoch eine ganz andere. Wenn Software-Entwicklung ohnehin Spaß macht, Deine Leidenschaft ist und Du dafür brennst, ist das Lernen eine Folge Deiner Neugier.

Ich kann mich noch an die Veröffentlichung von Swift erinnern, die neue Sprache von Apple. Die zu lernen war keine sachliche Abwägung von Argumenten. Ich wollte einfach herein schnuppern und habe mich dann regelrecht verliebt. So bin ich dabei geblieben. Steve Jobs hat die Begründung mit einem tollen Satz auf dem Punkt gebracht:

The only way to do great work is to love what you do.

Besser könnte man diesen Punkt nicht zusammenfassen.

Entwickle ein Entrepreneur-Mindset

Der durchschnittliche Angestellte betrachtet die Welt auch als solcher. Die geleistete Arbeit ist vergleichbar, aber der Fokus liegt auf dem eigenen Schicksal. Das ist bei erfolgreichen Unternehmern, oder Menschen die eben so denken, anders. Ist als Angestellter auch nicht wichtig? Falsch! Die folgenden Punkte stellen die beiden Denkweisen gegenüber:

Mentalität als Entrepreneur	Mentalität als Angestellter
Welchen Beitrag kann ich leisten?	Was steht mir zu?

Fokus auf das Resultat.	Fokus auf den Ertrag.
Konzentration auf das was gebraucht wird	Fokus auf das was gefordert wird
Sich selbst entbehrlich machen	Sich selbst unentbehrlich machen
Verantwortung übernehmen	Verantwortung abgeben
Lebensumstände sind Momentaufnahmen	Lebensumstände sind eher statisch

Quelle: Das wunderbare Buch *The Education of Millionaires* von Michael Ellsberg. Finde ich absolut treffend. Tolles Buch, klare Empfehlung! Mit dieser Einstellung erschaffst Du bessere Software, gerade als Dienstleister und angestellter Entwickler.

Erledige Aufgaben zuverlässig

Zuverlässigkeit ist überall wichtig. Was passiert wenn Dein Kollege ständig Aufgaben unzureichend löst? Oder was wenn seine Lösungen regelmäßig Probleme verursachen? Vermutlich arbeitest Du nur noch ungern mit ihm zusammen. Er gilt nicht mehr als zuverlässig. So möchtest Du sicher nicht wahrgenommen werden.

Wenn etwas nicht fertig wird: kommuniziere das rechtzeitig. Sei es vor dem Wochenende oder dem Urlaub. Wenn Deine Jobs bei Kollegen landen, müssen sie sonst die Suppe auslöffeln. Auch wenn regelmäßig Deine Umsetzungen Probleme verursachen, wirst Du früher oder später als Fehlerquelle oder Problemquelle abgehakt. Vielleicht äußert das niemand öffentlich. Ein innerer Dialog findet bestimmt

statt. Das Ergebnis: Vielleicht bekommen auf Dauer andere die spannenden Aufgaben und Herausforderungen.

Zusammengefasst: Übernimm einfach für Deine Aufgaben die persönliche Verantwortung, arbeite gewissenhaft und sorg für ausreichende Kommunikation: mit den Kollegen, Projektleitern und Kunden.

Give something back

Als erfolgreicher Entwickler arbeiten ist nur eine Seite der Medaille. Wenn Du genug Erfahrung hast um anderen zu helfen solltest Du das tun! Das kann auf kleiner Ebene vor-Ort anfangen. Gibt es Auszubildende oder Praktikanten? Die freuen sich immer über Feedback und Unterstützung. Vielleicht kannst Du Dein Wissen auch in Blogs veröffentlichen? Oder schreibst Artikel für ein Fachmagazin? Eventuell kannst Du Dein Wissen über Youtube weitergeben? Oder sogar ein eBook schreiben? Es ist gar nicht falsch, wenn Du dabei auch profitieren kannst. Die besten Lösungen sind WIN-WIN Situationen. Außerdem hilft ein monetärer Ausgleich auch weitere Angebote zu erstellen. Wichtig nur: Du musst den Fokus auf das Resultat legen (siehe Punkt #6!). Schaffe echten Mehrwert für andere. Das Geld ist eher ein Nebenprodukt.

Positives Denkweise

Alle Entwickler die ich kenne sind positive Menschen. Und das ist nicht nur so dahingesagt. Das ist eine geistige Grundhaltung. Die wird nicht anboten, die kannst Du lernen.

Positives Denken macht Dich als Mensch viel interessanter. Wer möchte schon mit jemanden arbeiten der immer nur meckert? Manchmal sind es die Lebensumständen, manchmal der Montag, der “doofe Kunde” oder das Wetter, wenn sonst keine Zielscheibe greifbar ist. Das muss nicht sein. Bejahe das Leben! Umarme Herausforderungen.

Auge auf Qualität

Ein Auge für die Qualität einer Lösung ist wichtig. Nicht nur der Nutzen und der Mehrwert stehen im Fokus. Die Punkte kannst Du manchmal bei Kundenaufträgen gar nicht beeinflussen oder überhaupt einschätzen. Was aber immer geht: Qualität abliefern. Sicher ist nicht immer das Budget für automatisierte Tests vorhanden. Vielleicht bleibt nicht immer Zeit für die perfekte Ausgestaltung aller Details. Das ist aber nie eine Entschuldigung für unsaubere Arbeit.

Der Punkt grenzt sich von Punkt #7 ab, da hier wirklich die Qualität der Implementierung gemeint ist. Kopiere nicht einfach Quellcode. Refactoring und Vereinfachungen können helfen. Auch ein Code-Review mit Kollegen kann Wunder wirken. Vier Augen sehen mehr als zwei.

Vielleicht fragst Du Dich wie es nun weitergeht? Ich empfehle Dir unbedingt loszulegen. Am besten noch heute! Mach Dir Gedanken zu der Richtung in die es gehen soll. Dann wählst Du eine Sprache und setzt erste Projekte um.

Natürlich empfehle ich Dir unbedingt auf codingtutor.de vorbeizuschauen. Da findest Du allerlei Wege um Dich weiterzubilden und tiefer einzusteigen. In den Blogartikeln, Videotrainings, Webinaren, wöchentlichen Screencasts und eBooks findest Du sicher auch spannende Inhalte für Dich.

Der einzige Fehler wäre zu viel zu überlegen und am Ende gar nichts anzufangen, aus Angst einen Fehler zu machen. In diesem Sinne möchte ich das Buch mit zwei Zitaten beenden, um das zu unterstreichen:

Selbst die längste Reise beginnt mit dem ersten Schritt. - Laotse

-

In einem Jahr wirst Du Dir wünschen, Du hättest heute begonnen. - Unknown