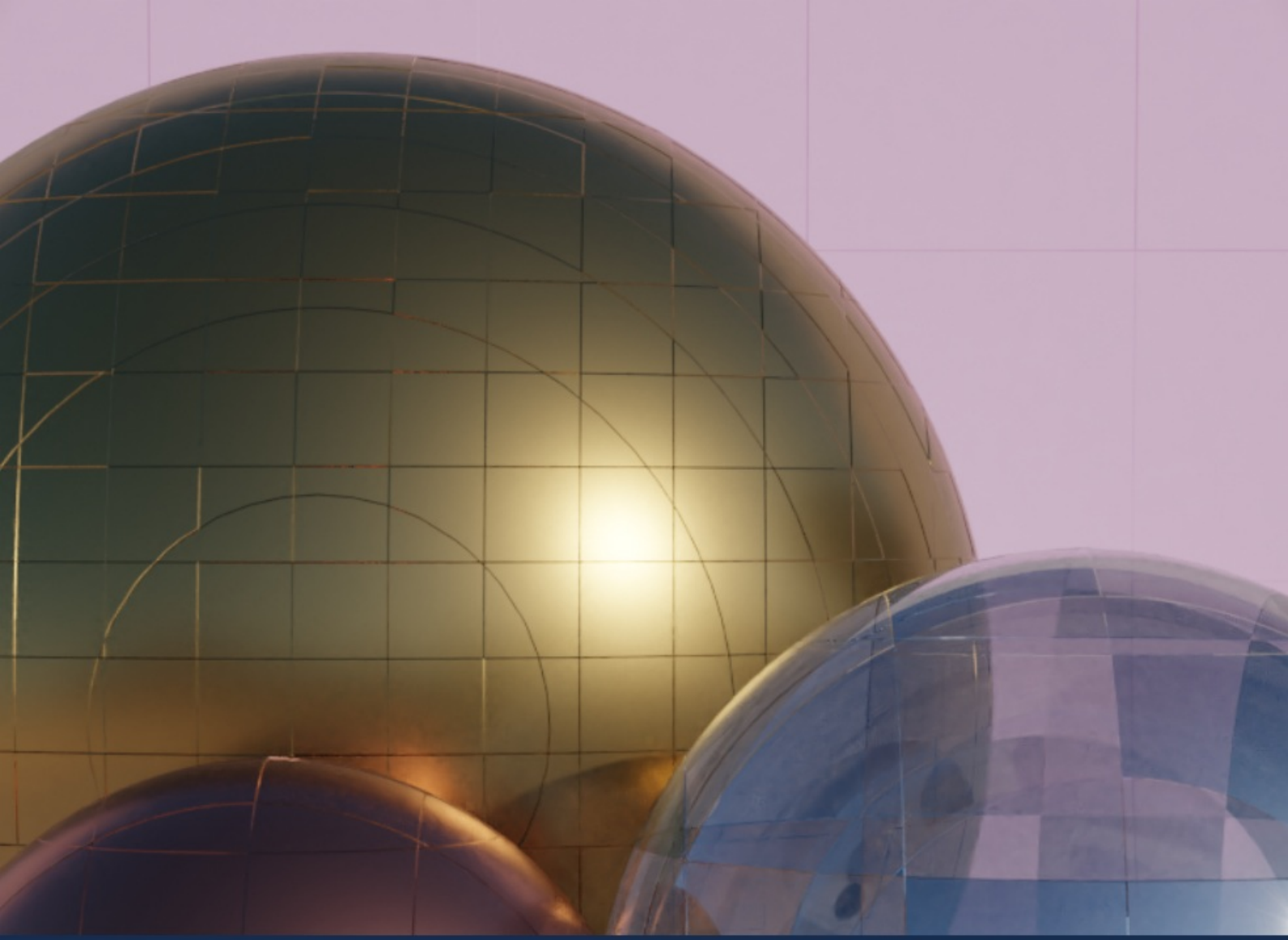


developer media

Entwickler-Almanach 2021

Fachliches. Unterhaltsames. Hilfreiches.

Der Entwickler-Almanach 2021

Alexander Dürrschnabel, Lucas Fernandes da Costa, Florian Mathieu

Tilman Börner

EBNER MEDIA GROUP GmbH & Co. KG

Der Entwickler-Almanach 2021

Zum Jahresausklang: Der Entwickler-Almanach 2021

Wir sind wieder da. Nach einem Jahr Zwangspause aufgrund der pandemischen Notlage gibt es dieses Jahr wieder einen Entwickler-Almanach.

Wie? Was? Sie wissen gar nicht mehr, was der Entwickler-Almanach ist? Nun, es handelt sich um ein kostenloses E-Book, das die Entwickler-Division developer media der Ebner Media Group um die Weihnachtszeit verschenkt. Bei developer media versorgen wir Sie mit Fachinformation für professionelle Softwareentwickler. Wir bieten Expertenwissen auf Konferenzen wie der Developer Week, in Trainings mit David Tielke, Christian Giesswein, Alexander Herz und vielen anderen mehr und in unseren beiden Magazinen, der dotnetpro und der web & mobile developer.

Unter dem Motto **Fachliches. Unterhaltsames. Hilfreiches.** wollen wir Ihnen für die stillen Tage zwischen den Jahren eine kleine Lektüre an die Hand geben, die zum Schmökern einlädt.

Zum Beispiel die Geschichte von dem Mann, der sich plötzlich in der Rente mit dem Problem konfrontiert sah, dass er sein Gehirn weiter trainieren musste. Also baute er kurzerhand einen Computer. Aber nicht irgendeinen Computer, sondern die legendäre PDP-8 von digital. Die konnte sogar schon Virtualisierung. Da der Mann in unserer Geschichte aber eine Aversion gegen den intensiven Kontakt mit dem Lötkolben hat, entstand der Computer mithilfe von

Field Programmable Gate Arrays. Im Artikel erklärt er, auf welche Probleme er stieß und wie er sie gelöst hat.

Oder: Kennen Sie Monte Carlo? Nein, nicht das Spielcasino, sondern das Verfahren aus der Wahrscheinlichkeitstheorie. Aber keine Angst: Wenig verlangtes Mathematikverständnis trifft in diesem Artikel auf großen Gewinn. Der Autor Lucas F. Costas zeigt, wie man diese Methode anwenden kann, um in der Softwareentwicklung nicht irgendwelche Pi-mal-Daumen-Schätzungen für die Dauer eines Softwareprojektes aufsetzen zu müssen. Vielmehr lässt sich die Dauer ziemlich genau errechnen.

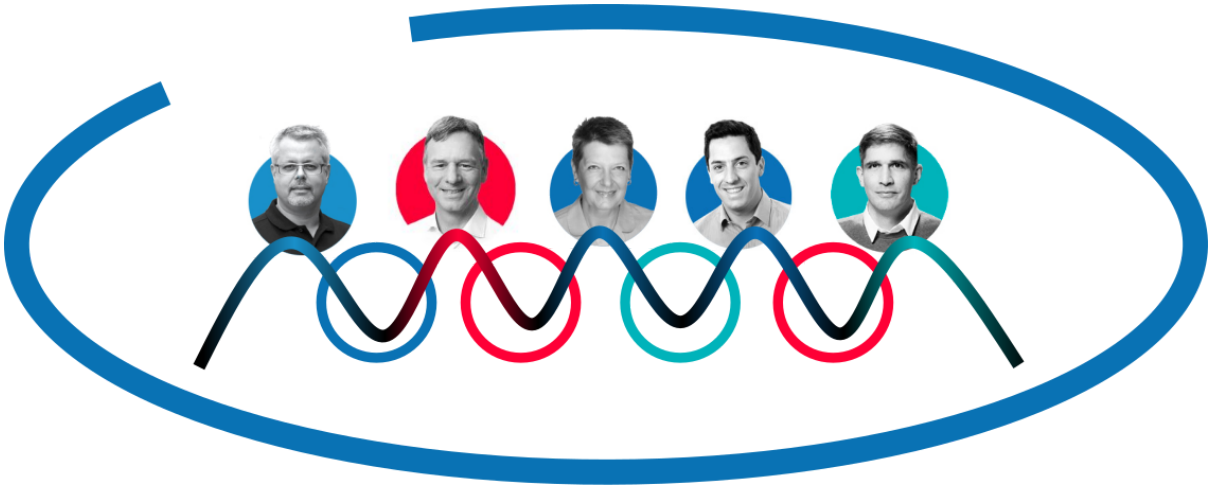
Auch Sie können etwas für die Umwelt tun. Schließlich belastet auch Ihr Code die Umwelt, weil er auf einem Rechner läuft, der wiederum Strom benötigt. Mit dem passenden Code kann man aber Strom sparen.

Genug des Wissens? Sie wollen spielen? Klar, dann bekommen Sie auch das. Wir haben fünf Spieleklassiker gesammelt, die Sie nicht einmal installieren müssen, sondern sie sofort im Browser spielen können. Mit dabei: Prince of Persia und das Ballerspiel Descent.

Auch große Dienste wie Youtube sind nicht davor gefeit, Opfer eines Hacking-Angriffs zu werden. Florian Mathieu ist durch Zufall auf ein Problem gestoßen, das den Dienst ziemlich belasten könnte, würde es nicht gefixed.

Gewürzt wird der Inhalt wie immer mit Einsprengseln von Witz und Wissen.

Gespannt? Na, dann los - stürzen Sie sich in den Entwickler-Almanach 2021. Wir wünschen Ihnen viel Spaß und kurzweilige Erkenntnisse.



willkommen

Das Team von developer media wünscht Ihnen ein gesundes und erfolgreiches Jahr 2022.
Fernando, Tilman, Susanne, Jonas, Philipp

developer media jobs
Der Stellenmarkt für Entwickler

LANGE SUCHEN
ODER
GLEICH FINDEN.
WAS DARF'S SEIN?

JETZT
ANZEIGE
SCHALTEN!

jobs.developer-media.de
Reinhold Fritsch, Tel.: +49 731 88005-8285 | Sabine Vockrodt, Tel.: +49 731 88005-8222
oder per E-Mail an: jobs@developer-media.de

I've been at this for 30 years, and I am absolutely overwhelmed by the amount of technical information I have packed into my brain.

It's because I was given the space to fail and I was not chased out of tech early that I have these abilities.

Be welcoming and patient.

@shanselman

Trend Programmiersprachen

Liegen Sie noch im Trend? Setzen Sie noch auf die richtige Programmiersprache? Sollten Sie eine andere lernen? Wer seine Chancen im Arbeitsmarkt verbessern will, sollte auch die passenden Sprachen beherrschen. Dafür sind Trendbarometer, die eine Beliebtheit der Programmiersprachen abbilden, hilfreich.

Sammelt man diese Rankings aus dem Jahr 2021, so ergibt sich ein ganz klares Bild: nämlich keines. Es gibt keine Sprache, die meilenweit vor allen anderen liegt. Denn je nachdem, welche Datenbasis, Befragungs- oder Auswertungsmethode verwendet wird, ergeben sich immer ein andere Bilder.

In der folgenden Tabelle sind die ersten zehn Plätze aus dem Ranking von vier Institutionen aufgelistet.

	Redmonk	IEEE Spectrum	GitHub Octoverse	Tiobe
1	JavaScript	Python	JavaScript	Python
2	Python	Java	Python	C

	Redmonk	IEEE Spectrum	GitHub Octoverse	Tiobe
3	Java	C	Java	Java
4	PHP	C++	TypeScript	C++
5	C#	JavaScript	C#	C#
6	C++	C#	PHP	Visual Basic
7	CSS	R	C++	JavaScript
8	TypeScript	Go	Shell	Assembly language
9	Ruby	Html	C	SQL
10	C	Swift	Ruby	PHP

Ein klares Bild zeigt das nicht. Trotzdem kann man doch gewisse Favoriten herauslesen.

JavaScript und Python: Diese beiden Sprachen tauchen an Position eins oder zwei oder zumindest weit oben auf. Klar: JavaScript ist durch Node.js und Websites immer noch aktuell. Interessanterweise taucht der Aufsatz TypeScript gar nicht so weit oben auf, obwohl Angular beispielsweise darauf basiert. Und Python ist zum einen wegen der guten Datenverarbeitung damit sicher weit oben. Zum anderen aber haben einige Open-Source-Projekte wie Blender eine Möglichkeit, die Programme mit Python zu erweitern.

C# steht bei allen Rankings irgendwo um den fünften Platz. Und in der Gegend steht die Sprache für .NET schon eine ganze Weile.

Das ist gut für .NET-Entwickler, denn das bedeutet, dass die Investition darin, sie zu erlernen, sich auszuzahlen scheint.

Die systemnahe Sprache C taucht auch in allen Rankings auf. Aber an unterschiedlichen Stellen. Heißt: Sie hat immer noch eine Wichtigkeit. Die Bewertung hängt aber sehr von der befragten Zielgruppe beziehungweise von der Methode ab.

Veranstalter: developer media Präsentiert von: dotnetpro web & mobile DEVELOPER

PDP-8: Kleiner Rechner nachgebaut

Gefühlt ist es gar nicht so lange her, als mich PDP-8 zum ersten Mal faszinierte. Freilich war der ein oder andere Leser damals noch gar nicht geboren. Denn wir sprechen hier von den sechziger Jahren des letzten Jahrhunderts.

Gerne hätte ich damals eine PDP-8 besessen. Aber sie war unerreichbar teuer. Jetzt im Ruhestand hab ich mir diesen Traum erfüllt. Allerdings nicht mit der Original-Hardware, sondern ich habe den Rechner mithilfe von FPGAs (Field Programmable Gate Array) nachgebaut.

Das war die PDP-8

Die PDP-8 war ein sogenannter Minicomputer - als Abkürzung für Minimal Computer. Er war der kleinste Computer, mit dem man etwas Sinnvolles anfangen konnte. Gleichzeitig hat die PDP-8 eine Eleganz in puncto Prozessor, die man heute nicht mehr findet. Mit acht verschiedenen Befehlen (einer davon ist für I/O und einer führt Operationen auf dem Accumulator aus) und 4096 Speicherplätzen

für Programme/Daten war sie in der Grundstruktur sehr einfach gehalten. Trotzdem war alles möglich, was auch mit den modernen CPUs möglich ist.

Multi-User-Betrieb und Virtualisierung gab es damals schon. Da die PDP-8 so einfach aufgebaut war, kann man mit etwas Aufwand den Schaltplan des Rechners und, falls vorhanden, den Quellcode jedes Betriebssystems und jedes Compilers verstehen. Von DEC wurden einige Handbücher zum Thema Rechnerarchitektur geschrieben. Diese finden Sie heute noch bei bitsavers. In ihnen steht, wie und warum die PDP-8 so entworfen und implementiert wurde.

Die PDP-8 von digital (Quelle: Alkivar, englische Wikipedia) (Bild 1)

Die PDP-8 war ein ziemlich verbreiteter Rechner, der seit 1965 angebogten wurde. Sein fast kompatibler Vorläufer PDP-5 kam 1963 auf den Markt. Zu beiden gibt es im Internet ziemlich viele Informationen und Handbücher - auch bei bitsavers und etlichen privaten Websites. Einige Programme/Compiler (zum Beispiel der größte Teil der Sammlung von DECUS) sind zwar verlorengegangen, doch findet man bei einer Suche im Internet sofort LISP, ALGOL, FOCAL, BASIC, FORTRAN, ASSEMBLER und viele Compiler und Programme.

Ein Ausgangspunkt, um sich mit dem Computer zu beschäftigen, ist der Eintrag in der englischen Wikipedia zur PDP-8. Etliche Betriebssysteme sind ebenfalls erhältlich wie zum Beispiel DMS (Disk Management System), OS/8 (fühlt sich an wie CP/M), TSS-8 (macht Timesharing und virtualisiert die PDP-8) und Multos8, das mehrere OS/8 gleichzeitig laufen lässt. Die Community der PDP-8 ist noch immer sehr groß, auch wenn inzwischen einige wichtige Personen wie Charles Lasner gestorben sind.

Diese Menschen hinterlassen Lücken, die nicht mehr zu füllen sind. Ein Beispiel dafür ist ein von Charles geschriebenes Betriebssystem, das zwar kompatibel, aber besser als OS/8 ist. Er wollte das veröffentlichen, kam aber wegen der Corona-Pandemie nicht mehr dazu.

Zur Programmerstellung auf dem PC gibt es zwei Crossassembler. Diese reichen zur Erstellung von Firmware, welche auf dem FPGA läuft.

Daneben gibt es auch etliche Emulatoren für die PDP-8. Diese helfen vor allem bei der Fortbildung: Wenn ich etwas in den Handbüchern nicht verstanden habe, dann fand ich die Erklärung immer in einem der Emulatoren. Im Internet findet man zusätzlich viele vollständige und größere Implementationen auf einem FPGA, verglichen mit dem Projekt, das ich unten beschreibe.

An mangelndem Wissen scheitert es also nicht, wenn man sich mit der PDP-8 beschäftigen will.

Lernen als Motivation

Die Motivation für das Projekt war deshalb nicht, eine PDP-8 zum Laufen zu bekommen. Das wäre mit einem Emulator oder einer bestehenden FPGA-Implementation viel schneller möglich. Die Motivation war der Wissensvorsprung und das Lernen.

Wenn man mit dem Arbeiten aufhört, muss man sein Gehirn am Laufen halten. Das heißt, man muss sich eines oder mehrere Projekte vornehmen, die so groß sind, dass sie nicht trivial sind und so klein, dass man sie selbst in einer überschaubaren Zeit realisieren kann. Weiter sollte es ein Projekt aus einem Gebiet sein, das man im Berufsleben nicht beackert hat und das einem trotzdem gefällt.

Nachdem ich überlegt hatte, was mich interessiert, was man heute im Studium lernt, was ich noch nicht weiß und was ich schon immer wissen wollte, kam ich zum Thema FPGA und PDP-8.

1974 konnte ich mit 16 Jahren an einer Olivetti P603 das Programmieren erlernen. Dieser Rechner war für kaufmännische Programme entworfen. In anderen Schulen sah ich, dass PDP-8

von DEC und Nova von Data General verwendet wurden. So einen Computer hätte ich auch gerne gehabt. Doch erstens war er zu teuer und zweitens, da ich mir alles selbst beigebracht hatte, zu komplex für mich. Später gab es dann von Intersil beziehungsweise Harris einen Microprozessor, der mit der PDP-8 kompatibel war. Allerdings implementiert dieser nicht die Befehlserweiterungen, die zur Virtualisierung des Prozessors nötig sind. Das war also wieder nichts.

Doch dann kam der Ruhestand und jetzt konnte ich loslegen.

Zeit und Geld

Einen großen finanziellen Aufwand habe ich dafür nicht treiben müssen. Man benötigt ein FPGA-Evaluationsboard. Ich nutze momentan das BASYS 3 Board von Digilent, welches momentan um die 150 Euro kostet. Die nötige Software auf dem PC ist Vivado Webpack von Xilinx, das nichts kostet. Einschränkungen gibt es nur bei sehr großen FPGAs. Die auf der PDP-8 laufenden Programme und Programmiersprachen sind für Hobbyzwecke ebenfalls kostenlos.

Noch ein Hinweis zum verwendeten Board: In Foren bekommt man den Ratschlag, als Anfänger mit einem kleinen Board anzufangen, da das für erste Versuche ausreichen würde. Inzwischen habe ich einige Boards verwendet, die immer solange ausreichend waren, bis ich die nächste Funktion implementiert habe. Somit wäre es aus der Rückschau schlauer gewesen, gleich ein großes Board zu verwenden. Andererseits ist es ärgerlich, wenn man ein teures Board schrottet. Deshalb würde ich dem Rat aus den Foren folgen und neue Sachen zuerst mit einem billigen Board ausprobieren.

Mit der Community habe ich sehr gute Erfahrungen gemacht. Gute Beispiele dafür sind Ian Schofield (siehe weiter unten) oder David Gesswein, der nach Input von mir das verloren geglaubte Edu25-Basic wiederherstellen konnte.

Es gibt in der Community auch weiterhin interessante Projekte. Mir fällt zuerst PiDP-8 ein. Das baut eine emulierte PDP-8 mit einem Raspberry Pi (RPi) und dem Emulator simh nach. Kaufen kann man dazu ein wunderschönes Gehäuse aus Holz, mit bedruckter Frontscheibe, vielen Schaltern und LEDs. Ich habe mir eines davon besorgt, werde dort aber keinen RPi einbauen, sondern später die PDP-8 auf dem FPGA. Irgendwie finde ich es abwegig, einen Computer in einem anderen Computer zu simulieren, wenn man auch eine Implementation in Hardware machen kann.

Als ich das erste Mal bewusst VHDL und FPGAs gesehen habe - das muss so um das Jahr 1995 gewesen sein - habe ich mir gedacht: VHDL ist als Programmiersprache ähnlich wie Pascal, da ist keine größere Einarbeitung nötig, das kannst du gleich.

VHDL: Deklarativ statt imperativ

So arg geirrt habe ich mich selten. Bis ich verstanden habe, dass VHDL keine Programmiersprache ist, sondern Hardware beschreibt und alles irgendwie parallel läuft, verging etliche Zeit, in der nichts funktioniert hat.

Wenn man eine imperative Programmiersprache wie Pascal, C oder Visual Basic gelernt hat, muss man alles darüber vergessen und ganz von vorne anfangen. Anders wahrscheinlich, wenn man

eine deklarative Programmiersprache wie Prolog oder eine funktionale wie Haskell gelernt hat: Dann tut man sich leichter.

Irgendwann habe ich dann entdeckt, dass man in Xilinx Vivado anschauen kann, welche Hardware generiert wird. Danach lief es besser.

Einen zweiten großen Fehler habe ich gemacht, indem ich zu wenig Testbenches geschrieben habe. Als Programmierer denkt man sich, das Programm läuft sowieso, und wenn nicht, dann zeige ich auf der Konsole alles an, was ich zur Fehlersuche brauche, oder benutze den Debugger.

Mit dieser Herangehensweise kann man kein korrektes VHDL erstellen. Ob ein Signal einen Takt zu früh oder zu spät kommt, sieht man auf diese Weise nicht, und deshalb erleidet man damit Schiffbruch. Es ist natürlich lustiger und befriedigender, eine LED auf einem FPGA-Board blinken zu lassen, als das nur in einem Testbench festzustellen.

Damals habe ich gemeint, dass ich auf die Ratschläge in verschiedenen Foren, die ich als Lurker besucht habe, nicht hören muss. Es war ein ziemlicher Irrweg, und darauf bin ich nicht stolz.

Heute lernt man VHDL, Entwurf synchroner digitaler Logik und State Machines im Studium. Bei mir war das eine Zeit mit Versuch und Irrtum, da ich damals zwar alles über TT-Logik, KV-Diagramme und Flop-Flops gelernt hatte, aber nichts über pogrammierbare Logik.

Auch geht das Lernen nicht mehr so leicht, wenn man älter ist. Angefangen habe ich deshalb mit einem auf einem Xilinx-Board

nachgebauten Wecker. Dem folgte der kleinste Prozessor, den ich kenne, der MCPU von Tim Böschke mit 72 Zeilen VHDL. Ein Prozessor in VHDL kann also aus wenigen Zeilen bestehen. Nach einigen Tagen war ich der Meinung, dass ich den Prozessor verstanden habe - war aber nicht so, weil ich, wie oben beschrieben, mit dem falschen Ansatz an das Problem herangegangen bin.

Vor vielen Jahren hatte ich mir das Buch The Art of Digital Design von Franklin P. Prosser und David E. Winkel gekauft, das man sich inzwischen kostenlos herunterladen kann.

In dem Buch steht alles zum synchronen Entwurf von Digitalschaltungen mit State-Machines, und wenn man alles in dem Buch verstanden hat, ist man in diesem Thema fit. Mehr Literatur braucht man nicht. Da das Buch zum Unterricht gedacht war und die Studenten zum Abschluss eine PDP-8, auf der FOCAL läuft, aufbauen mussten, gibt es dazu genaue Erläuterungen.

Damals (1986) wurde die PDP-8 auf einem Wire-Wrap-Board aus ca. 120 TTL-ICs aufgebaut. Es gibt dazu zwei Realisierungsweisen: einmal von Hand verdrahtet, das andere mal einen durch Microcode realisierten Prozessor. 120 TTL-ICs zu verdrahten war mir zu viel Arbeit, und beim Löten verbrenne ich mir regelmäßig die Finger.

FPGA statt Wire-Wrap-Board

Ich habe dann mit FPGAs versucht, den Prozessor nachzubauen und dazu das Buch als Vorlage genommen. Das Buch ist didaktisch so aufgebaut, dass eine korrekte und verständliche und nicht eine

möglichst effektive Implementation beschrieben wird. Zum Beispiel wird die State Machine aus mehr Zuständen als nötig realisiert. Ich habe dann versucht, die Implementation zu vereinfachen (weniger States) und zu erweitern (zum Beispiel Verwaltung von mehr als 4K Worte).

Auch sollte man nie etwas optimieren, von dem man keine Ahnung hat. Irgendwann war das so verbastelt, dass ich nach einem neuen Anfang gesucht habe. Wenn man das als Hobby macht, dann spielt Zeit keine Rolle, und man kann alles solange wegwerfen, bis man eine optimale Implementation hat. Das sehe ich als wesentlichen Vorteil gegenüber dem Berufsleben, bei dem ein Problem innerhalb einer bestimmten Zeit realisiert werden muss.

Auf OpenCores gibt es eine minimale Implementation, die pd8l von Ian Schofield. Die habe ich dann als Startpunkt genommen. Herr Schofield hat mir dabei sehr geholfen und meine Fragen zu dem Projekt schnell und präzise beantwortet. Das war unglaublich hilfreich. Andererseits habe ich in dem Projekt dann einen Fehler gefunden.

Schließlich habe das Projekt dann mal schneller und mal langsamer erweitert.

1. Als erstes habe ich auf dem FPGA-Board (BASYS 3) eine Anzeige für die internen Register des Rechners und die Steuerung des Testablaufs realisiert, damit man einfacher debuggen kann. Normalerweise nimmt man zu Testen einen in VHDL geschriebenen Testbench. Das ist aber in diesem Fall unpraktisch, da eine riesige Anzahl von Testdaten anfällt, wenn zum Beispiel in den Videospeicher falsche Buchstaben

geschrieben werden. Jeder Befehl der PDP-8 wird durch mehrere States (drei bei IOT, bis zu zehn bei ISZ/DCA/JMS indirect mit Autoincrement) der Implementation realisiert, und da ich weitere PDP-8 zu Gerätesteuerung verwende, wird das noch komplexer. Die Korrektheit einzelner Befehle habe ich natürlich jeweils mit einem VDHL-Testbench überprüft. Mit dem momentan eingebauten Debugger kann man die Maschine stoppen und starten, die Register AC, L und PC anschauen sowie einzelne Befehle im Single-Step ausführen.

2. Die PDP-8 kann als 12-Bit-Maschine nur 4096 Worte adressieren. Damit lässt sich einiges realisieren wie zum Beispiel FOCAL oder Basic. Beide Programmiersprachen erlauben weder lange Programme noch eine komplexe Syntax. Für größere Programme oder mehr Daten gibt es eine Speichererweiterung, mit der 8 Adressräume angesprochen werden. Standardmäßig werden diese durch ein I/O-Register adressiert, und es gibt ein 4K-Datenfeld und ein 4K-Adressfeld, damit beispielsweise Daten zwischen zwei Feldern kopiert werden können. Damit sind dann 32768 Worte adressierbar. Der größere Speicher ist dann im BRAM, dazu muss der FPGA-Baustein genügend groß sein.
3. Danach habe ich einen I/O-Bus implementiert. Normalerweise würde ich das über einen Bus machen, auf dem die Geräte, welche über I/O-Adressen gerade angesprochen werden, ihre Daten über ein Tri-State-Register auf den Bus legen. Bei einem FPGA geht das nicht, da dieser intern keine Tri-State-Gatter kennt. Deshalb wird das über Multiplexer realisiert, so dass man entweder eine Point-zu-Point Verbindung oder hintereinander geschaltete Multiplexer hat, was dann die maximale Taktrate

bremsen kann. Die Geräte bei einer PDP-8 können neben Programmed-I/O zwei Arten von einfachem Direct Memory Access(DMA). Beim Ausgangsprojekt (pdp-8l, siehe oben) war nur ein Gerät implementiert, und der Zugriff über DMA wurde deshalb in der CPU so implementiert, dass nach der Ausführung eines Befehls ein Datentransfer für DMA gemacht werden kann. Ich habe das mit einer getrennten Einheit für DMA erledigt. Die BRAMs im Speicher des FPGAs erlauben zwei gleichzeitige Speicherzugriffe, so dass dies wirklich die CPU-Befehle gleichzeitig ausführt, während parallel dazu eine Datenübertragung mit DMA erfolgt. Eine Änderung, die ich vorgenommen habe, fehlt noch. Wenn man mehrere Geräte hat, braucht man einen Arbiter, der auswählt, welches Gerät DMA machen kann. Ich habe dazu eine Round-Robin-Funktion zum sequentiellen Zugriff auf DMA implementiert; eine prioritätsgesteuerte Zuteilung könnte auch implementiert werden. Was fehlt, wäre, das die DMA-Zugriffe atomar auf Bytegröße und nicht atomar auf Blockgröße erfolgen und dass eine der zwei DMA-Arten noch implementiert wird. Aber etwas will ich ja im Ruhestand noch zu tun haben …

4. Im ursprünglichen Projekt wurde eine RSR-232-Schnittstelle zur Datenübertragung verwendet. Das funktioniert über USB prima, das verwendete Board hat einen Wandler von RS-232 nach USB drauf, und mit Tera Term kann man den ansprechen. Eine historische PDP-8 hat eigentlich immer einen ASR-33-Fernschreiber zur Kommunikation gehabt. Ich habe deshalb ein VT-52-Terminal implementiert, das an einer internen seriellen Schnittstelle hängt, die wie ein ASR-33 angesprochen wird. Das Terminal wird wiederum von einer PDP-8 als Gerätecontroller

gesteuert, auf der geeignete Firmware läuft zur Emulation des VT-52-Terminals. In deren Speicher sind die anzuzeigenden Daten in den Speicher eingeblendet. Blättern oder das Löschen des Bildschirms werden damit für die CPU, welche das Terminal implementiert, ganz normale Speicherzugriffe. Da die BRAMs, wie oben erwähnt, zwei gleichzeitige Speicherzugriffe erlauben, läuft die Anzeige auf einem Bildschirm gleichzeitig mit dem Ablauf der CPU. Die Bildschirmausgabe geht vom Bildschirmspeicher zu einem Font-ROM, und dort kommen die einzelnen Pixel heraus. Die gehen dann zusammen mit den horizontalen und vertikalen Synchronisationssignalen (640x480) über den VGA-Anschluss zu einem VGA-TFT-Display mit 1280x1024. Das Signal ist deshalb etwas matschig. Damals wollte ich mit dem Kauf eines Displays mit 640x480 Punkten (gab es bei Pollin) noch warten, bis das VT-52-Terminal läuft. Zusätzlich habe ich eine Eingabe mit der PS/2-Tastaturschnittstelle dazugebaut und einen Video-Multiplexer, damit man mehrere Terminals gleichzeitig verwenden kann, welche sich mit einer Taste auf der Tastatur umschalten lassen. Da ein VT-52-Terminal nur 24 Zeilen hat, stelle ich auf den letzten Zeilen verschiedene Statusinformationen dar, und das oben beschriebene Debug Interface mit der 7-Segment-Anzeige passt in erweiterter Form (PC, L, AC, MQ, IR und der aktuelle Befehl zurückübersetzt in die Assembler-Form) auf den Bildschirm.

Aktueller Projektstatus

Was läuft, und wie soll es weitergehen? Momentan habe ich auf dem FPGA verschiedene Versionen der PDP-8 lauffähig.

1. Das erste Projekt ist eine PDP-8 im Basisausbau mit 4096 Worte Speicher und einer seriellen Schnittstelle. Darauf läuft FOCAL und man kann die PDP-8 wie einen Taschenrechner verwenden, auf den man über die serielle Schnittstelle zugreifen kann.
2. Als zweites Projekt habe ich eine emulierte PDP-8, auf der das Edu20-System läuft. Die Maschine arbeitet mit zwei (im FPGA) seriell angeschlossenen VT-52 Terminals und 8192 Worte Speicher. Das Edu20-System ist ein Multi-User Basic (ja, kein Fehler). Mit zwei Terminals (mehr sind möglich) können gleichzeitig zwei Programme unabhängig voneinander laufen. Man sieht, wie viel mit ganz wenig Ressourcen gemacht werden kann. Da das System zum Einstieg für eine Schule gedacht war, gibt es keinen Massenspeicher und das Laden/Speichern erfolgte über einen am ASR-33 eingebauten Lochstreifenleser/-stanzer.
3. Es gibt dann noch ein drittes Projekt. Dabei handelt es sich um eine PDP-8, die 4096 Worte Speicher hat und an der ein Festplattenspeicher DF32 von 32768 Worten Speicher angeschlossen ist. Diese Konfiguration entspricht der, die von OpenCores stammt, läuft aber auf der erweiterten PDP-8. Der Speicher des DF32 wird beim Einschalten aus dem Konfigurationsspeicher des FPGA in das RAM geladen. Die Änderungen sind nach dem Ausschalten natürlich weg. Als Betriebssystem wird DMS verwendet, das aus dem DF32 geladen wird, der Speicher von 4K reicht dafür aus, und man kann Fortran-2-Progamme editieren, übersetzen und laufen lassen, Assembler geht natürlich genauso. Der simulierte Festplattenspeicher besitzt momentan keine Intelligenz. Es ist

keine PDP-8 in ihn eingebaut, es geht alles über eine State-Machine.

Geplante Änderungen

1. Als nächstes habe ich vor, den Festplattenspeicher so umzubauen, dass die Daten nach dem Ausschalten nicht verloren gehen. Dazu möchte ich zuerst ein FRAM (das ist nichtflüchtiger Speicher, auf den aber genauso schnell zugegriffen werden kann wie auf ein RAM) und danach eine SD-Karte verwenden. Passend dazu werde ich weitere externe Speicher (Festplatte, da gab es verschiedene Ausführungen und Floppy) implementieren.
2. Zur der oben beschriebenen Speicherverwaltung gibt es eine Erweiterung, die ich nach der Änderung des Festplattenspeichers einbauen möchte. Diese Erweiterung erlaubt einen System-Mode, in dem alles erlaubt ist und einen User-Mode, in dem bei eine I/O-Operation oder einem HLT-Befehl ein Interrupt ausgelöst wird. Mit dieser Erweiterung kann man ein richtiges Betriebssystem verwenden, mit dem die physikalische Maschine in viele virtuelle Maschinen aufgeteilt wird. Virtualisierung gibt es daher schon lange vor VMware oder Hyper-V und ist keine neue Erfindung, sondern eine Idee, die immer wieder aufkommt seit CP67/CMS. Hier ist, im Unterschied zu einem heutigen Virtualisierer, das Betriebssystem TSS-8 erstens im Sourcecode vorhanden und zweitens so klein, dass man das selbst verstehen kann.

Was habe ich dabei gelernt?

Ganz wichtig ist für mich der Einsatz von State Machines. Der Prozessor, die Hardware der Peripheriegeräte und die Firmware sind alle als State Machines implementiert. Das ist Wissen, das man auf die Softwareentwicklung übertragen kann. Jeder kennt das Problem, dass man noch ein paar Erweiterungen in sein Programm einbauen muss. Dazu gibt es hier ein Schalterchen und dort ein Schalterchen und dann funktioniert das manchmal und manchmal nicht. Mit State Machines löst man solche Probleme endgültig. Im Internet bei Quantum Leaps findet man dazu viele Informationen und sie haben ein interessantes Buch dazu herausgebracht. State Machines werde ich zukünftig auch in Software mehr einsetzen. Das Umdenken, nachdem man VHDL gelernt hat, nützt einem auch beim Programmieren.

Was habe ich weiterhin gelernt? Man ist nie zu alt, um etwas Neues zu lernen. Mit über 60 geht das zwar nicht mehr so leicht, wie mit 20. Man braucht dazu mehr Zeit und Notizen - bei mir ist alles voll mit kleinen Notizzetteln. Zum Thema FPGA und VHDL habe ich nicht nur ein Buch gekauft und durchgearbeitet, sondern mehrere, weil ich das zum Verstehen gebraucht habe und manchmal ist ein Buch ein Fehlkauf, nicht weil der Inhalt schlecht ist, sondern weil einem die Art, wie der Autor das erklärt, nicht liegt.

Auch als Einzelkämpfer kann man ein nicht triviales Projekt realisieren. Vor dem Internet musste das gehen, ohne dass man das mit Stackoverflow gelöst hat oder in einem Forum nachgefragt hat. Zum Thema PDP-8 gibt es dort leider wenig Sekundärliteratur.

Als Anfänger bekommt man oft den Hinweis, das Buch “The Designer’s Guide to VHDL”, Peter J. Ashenden, ISBN 0120887851

zu kaufen und damit VHDL lernen. Ich habe in dem Buch genau gar nichts verstanden. Das ist so, als würde man einem Programmieranfänger das Buch “The Art of Computer Programming”, Donald E. Knuth, ISBN 0321751043 empfehlen.

Was ich hingegen empfehlen kann, wenn man sich nur ein Buch kaufen will, ist “FPGA Prototyping by VHDL Examples”, Pong P. Chu, ISBN 0470185317. Das Buch gibt es in zwei Ausgaben, eine für Xilinx und eine für Altera/Intel. Wenn jemand Verilog verwenden will, gibt es die Bücher auch dafür. Die Fachleute im FPGA-Forum bei Mikrocontroller.net halten zwar nicht soviel von dem Buch (Thema asynchrone Resets), aber ich habe mit einem älteren Buch von Prof. Chu vieles gelernt. Wenn es ein deutsches Buch sein soll, würde ich “VHDL-Synthese”, Jürgen Reichardt, Bernd Schwarz, ISBN 9783110375053 nehmen. Das Buch ist gewiss nicht schlecht, aber mir gefiel der Schreibstil der Autoren nicht. Das Thema “Modellierung digitaler Filter” wird ausführlich besprochen, das hat mich nicht interessiert, anderseits wird ein RISC-Prozessor gebaut, das war ziemlich interessant.

Eine weitere Sache, die ich dabei gelernt habe, ist “Technik kommt immer wieder” und deshalb lohnt es sich, die alte Technik zu verstehen. Das was uns oft als Neuheit angepriesen wird, gab es vor 20 Jahren, vor 40 Jahren, … immer schon. Wie ich weiter oben geschrieben habe, ist das mir beim Thema Virtualisierung aufgefallen.

Wenn man sieht, dass auf der PDP-8 eine funktionale Programmiersprache wie LISP läuft, weiß man, dass funktionale Programmierung nichts neues ist.

Auch interessant ist, dass ein Thema, welches wieder neu hochkommt, immer komplexer wird. Man sieht das schön bei der PDP-8. Dort gibt es einen Interrupt, der nur einen Vektor hat und keine Prioritäten. Man muss dann alle Geräte durchklappern, bis man das findet, welches den Interrupt ausgelöst hat. Beim Ur-PC war das schon komplizierter mit einem Interrupt-Controller und bei einem heutigen PC ist das mit dem APIC so kompliziert, dass man das fast nicht mehr verstehen kann.

Ähnlich ist es bei der DMA oder der Speicherverwaltung, damals mit einem Adressfeld und heute mit virtuellem Speicher und Paging (TSS-8 kann das auch, dort können die virtuellen Maschinen mehr Speicher belegen, als physischer Speicher eingebaut ist).

Fazit

Dieses Projekt hat unheimlich Spaß gemacht und tut es immer noch. Nichts stellt einen so zufrieden wie ein Projekt, das am Ende (naja, bei Zwischenschritten, so ein Projekt ist nie fertig) so läuft, wie man sich das vorgestellt hat. Wenn man das in weiteren Schritten noch erweitern kann, bekommt man gleich neue Motivation.

Alexander Dürrschnabel war über 30 Jahre hinweg treuer Leser der basicpro und der dotnetpro. Im Ruhestand liest er zwar die dotnetpro nicht mehr, doch hat er nun Zeit, sich den Technologien zu widmen, die ihn schon lange begeistern.

We are looking for a

Pull Stack Developer

(m/w/d)

You are able to pull a card from a stack and develop a serious hysterical crying when you loose.

Contact: Cesars Palace, Las Vegas

We are looking for a

Fond End Developer

(m/w/d)

After a long love story there is only one thing left: the happy end. In this team you develop ends that are cozy, happy and fond.

Contact: TriState Area Love Stories, Los Angeles

Wir sind eine mittelständische Firma, die ihre Wurzeln in den Geschäftsfeldern Objekt- und Personenschutz sowie im Handel mit seltenen Gütern hat.

Für unser Buchhaltungsteam suchen wir zum nächstmöglichen Zeitpunkt einen

Senior Software Engineer

der das selbstgestaltete Buchhaltungssystem weiterentwickelt.

Bewerbungen richten Sie bitte an:
Cosa Nostra, Palermo, Sizilien

Sie

haben schon in BASIC programmiert?
haben Notepad bedient?
haben schon einmal HTML geschrieben?

Dann bewerben Sie sich

`$[a-z](.*)?w+$`

Mit uns finden Sie Ihren Traumjob als
* Systemprogrammierer OS/2,
* Junior Software Engineer Coltin,
* KOBOL-Entwickler

Head A. Hunter, Personalvermittlung

developer media

Git-Masterclass

developer media T Trainings

Ihr Trainer: Michael Kaufmann

Git hat sich als Quellcodeverwaltung durchgesetzt und ist der De-facto-Standard in der Softwareentwicklung geworden. Und nicht nur die Entwicklung – auch Administration, Sicherheit und Dokumentation finden zunehmend auf Git statt. Dieses Training bietet einen praxisorientierten und vollständigen Einstieg in das komplexe Thema.

Was wird behandelt

- Hinzufügen und Ändern von Dateien
- Add, Commit, Branch, Tag und Merge
- Zusammenarbeit im Team
- Git Workflows
- Mono- vs. Multi-Repo
- Arbeiten mit Pull Requests und Forks
- Produktiver werden

3 Tage

developer media

Continuous Delivery & Integration

developer media T Trainings

Ihr Trainer:
Stephan M. Rossbach

In diesem Training zeigt Stephan Rossbach anhand zahlreicher Praxisbeispiele, wie Unternehmen mit Continuous Delivery und Continuous Integration wirkungsvolle Mechanismen etablieren können, um die Agilität der Softwareentwicklung in alle nachgelagerten Abläufe zu übertragen. Die Teilnehmer sind nach dem Training in der Lage, eine eigene Deployment-Pipeline für ihr Projekt zu planen und zu etablieren.

Was wird behandelt

- Deployment-Pipeline
- Testautomatisierung
- Integrationstests
- Lasttests
- Deployment-Strategien
- Performancetests
- Capacityests
- Codemetriken
- Tooling

2 Tage, remote

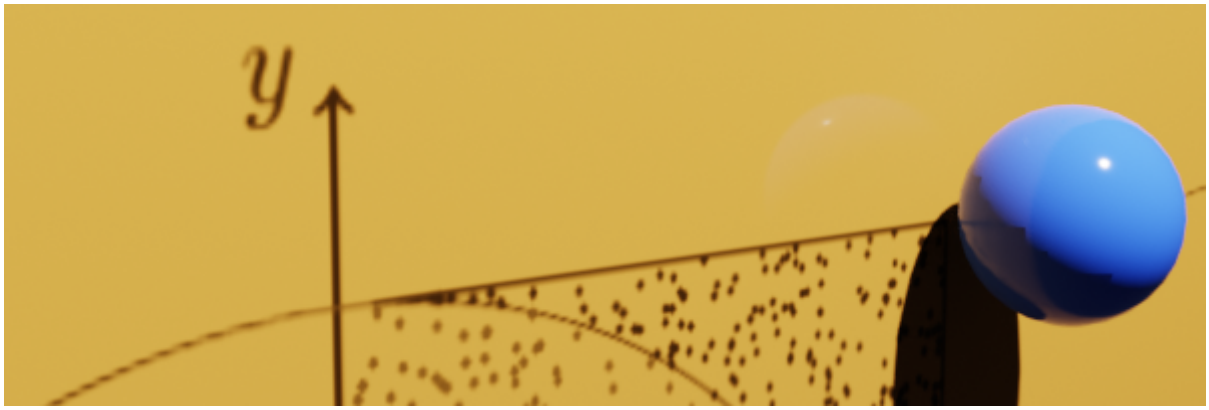
Weitere Informationen unter www.developer-media.de/trainings ••• Termine nach Absprache

Ihre Ansprechpartnerin: Susanne Herl • +49 (0)731 88005-8835 • susanne.herl@developer-media.de

Wie man Schätzungen und Vermutungen durch eine Monte-Carlo-Simulation ersetzt

Es gibt viele Möglichkeiten, die Dauer eines Softwareprojekts abzuschätzen. Sie alle sind Zeitverschwendung.

Manche ziehen es vor, tagelang ihre Änderungen zu analysieren und zu planen, um eine genauere Schätzung vornehmen zu können - andere multiplizieren ihre Schätzungen einfach mit N. Ich würde keinen dieser Ansätze empfehlen.



Das Problem mit dem ersten Ansatz ist, dass er die Softwareentwicklung als deterministischen Prozess betrachtet, obwohl sie in Wirklichkeit stochastisch ist. Mit anderen Worten: Sie können nicht genau bestimmen, wie lange es dauern wird, ein bestimmtes Stück Code zu schreiben, wenn Sie es nicht bereits geschrieben haben.

Das Problem bei der Multiplikation von Schätzungen mit N ist, dass man einfach die Zykluszeit gegen die Vorhersagbarkeit eintauscht.

Anstatt sich wirklich um Genauigkeit zu bemühen, fügen Sie Ihren Schätzungen lediglich einen Puffer hinzu. Dieser Puffer wäre kein Problem, wenn die Ingenieure ihn nicht dadurch ermitteln würden, dass sie sich die Fingerspitze lecken und sie in die Luft halten.

Leider kann man beim Schätzspiel nur gewinnen, wenn man es nicht spielt.

> “Schätzung ist Zeitverschwendung. Tun Sie es nicht.”
>
> *VACANTI, Daniel*

Anstatt “informierte” Vermutungen anzustellen oder Schätzungen mit N zu multiplizieren, können wir die Zufälligkeit und Variabilität, die mit dem Schreiben von Software verbunden sind, akzeptieren und geeignetere statische Methoden, in diesem Fall stochastische Modellierungstechniken, verwenden, um bessere Prognosen zu erstellen. Eine dieser Techniken ist die Monte-Carlo-Methode, die ich im weiteren Verlauf dieses Beitrags zur Erstellung von Prognosen verwenden werde.

In diesem Beitrag erfahren Sie, wie Sie Schätzungen und Vermutungen durch eine Monte-Carlo-Simulation ersetzen können.

Zunächst werde ich erklären, was die Monte-Carlo-Methode ist und wie sie funktioniert. Dann zeige ich Ihnen, wie Sie damit vorhersagen können, wann ein Projekt wahrscheinlich abgeschlossen sein wird.

Im dritten Abschnitt dieses Blogbeitrags zeige ich Ihnen, wie Sie Ihre Prognosen mit einem Wahrscheinlichkeitswert versehen und sie so in tatsächliche Prognosen [1] umwandeln können. In diesem

Abschnitt erfahren Sie, wie Sie je nach Risikobereitschaft mehr oder weniger optimistische Prognosen erstellen können.

Nach diesen ersten drei Abschnitten erläutere ich einige Vorbehalte und Faktoren, die Sie bei der Durchführung Ihrer Monte-Carlo-Simulationen zur Erstellung von Prognosen beachten sollten.

Zum Schluss erkläre ich, wie es in den nächsten Blogbeiträgen weitergeht, und empfehle weiterführende Literatur.

Was ist die Monte-Carlo-Methode?

Bei der Monte-Carlo-Methode werden wiederholt zufällige Einzelereignisse simuliert, um die Wahrscheinlichkeit der möglichen Ergebnisse zu ermitteln.

Stellen Sie sich zum Beispiel vor, Sie möchten die Wahrscheinlichkeit berechnen, dass Sie beim Würfeln mit zwei Würfeln eine Summe von sieben erhalten.

Eine Möglichkeit, dies zu tun, besteht darin, alle 36 möglichen Ergebniskombinationen aufzuzählen und zu zählen, wie viele dieser Ergebnisse eine Summe von sieben ergeben. Wenn man davon ausgeht, dass sechs dieser 36 Ergebnisse eine Summe von sieben ergeben, liegt die Wahrscheinlichkeit, eine Sieben zu würfeln, bei 16,67 Prozent.

Alternativ könnten Sie in den nächsten Monaten zwei Würfel werfen, alle Ergebnisse notieren und überprüfen, wie oft Sie eine Sieben gewürfelt haben. Je öfter Sie gewürfelt haben, desto präziser wird Ihre endgültige Wahrscheinlichkeitsschätzung sein.

Nun ist tagelanges Würfeln nicht sehr spannend (und dauert zu lange). Stattdessen könnten wir den Computer die Würfel für uns rollen lassen. Dank der Maschinen können wir zwei Würfel ziemlich schnell eine Million Mal würfeln.

Schreiben wir nun ein einfaches Rust-Programm, das zwei Würfel eine Million Mal würfelt und uns sagt, wie oft es eine Sieben würfelt.

```
extern crate rand;

use rand::distributions::{Distribution,
  Uniform};
use rand::thread_rng;

const TOTAL_ROLLS: i32 = 1_000_000;

fn main() {
  let mut rng = thread_rng();
  let die = Uniform::from(1..7);

  let mut sevens = 0;
  for _ in 0..TOTAL_ROLLS {
    let first_die = die.sample(&mut rng);
    let second_die = die.sample(&mut rng);
    if first_die + second_die == 7 {
        sevens += 1
    }
  }

  let p: f64 = f64::from(sevens) /
    f64::from(TOTAL_ROLLS) * f64::from(100);

  println!("Total Rolls: {}", TOTAL_ROLLS);
  println!("Total Sevens: {}", sevens);
  println!("Probability of rolling a seven:
    {:.2}%", p);
}
```

Wenn dieses Programm ausgeführt wird, simuliert es eine Million Würfelwürfe, teilt die Anzahl der geworfenen Sieben durch die Gesamtzahl der Simulationen und gibt die empirische Wahrscheinlichkeit aus, eine Sieben zu würfeln. Diese liegt nahe bei 16,67 Prozent, also der tatsächlichen Wahrscheinlichkeit, eine Sieben zu würfeln.

```
    Total Rolls: 1000000
    Total Sevens: 166446
    Probability of rolling a seven: 16.64%
```

Herzlichen Glückwunsch, Sie haben gerade Ihre erste Monte-Carlo-Simulation durchgeführt.

Simulieren, wann ein Projekt abgeschlossen sein wird

Nachdem Sie nun eine Monte-Carlo-Simulation durchgeführt haben, zeige ich Ihnen, wie Sie die gleichen Prinzipien auf ein Softwareprojekt anwenden können, um abzuschätzen, wann Sie es abschließen werden.

Nehmen wir an, Sie haben einen Blogbeitrag von Tim Ferriss gelesen, in dem es heißt, dass die optimale Anzahl von Blogbeiträgen pro Jahr 60 oder mehr ist. Da es sich um Tim Ferriss handelt, sind Sie davon überzeugt, dass er Recht hat, und Sie möchten seinen Rat befolgen.

Bevor Sie sich jedoch auf dieses zeitaufwändige Unterfangen einlassen, möchten Sie wissen, wie wahrscheinlich es ist, dass Sie in 365 Tagen 60 oder mehr Beiträge schreiben können, und, falls

nicht, welche Änderungen Sie vornehmen könnten, um diese Marke zu erreichen. Wie können wir das erreichen?

Ein naiver Ansatz: gleichmäßig verteilte Zykluszeiten

Zunächst wählen wir einen naiven Ansatz: Wir gehen davon aus, dass Sie wissen, wie lange es gedauert hat, die einzelnen Blogbeiträge zu schreiben, die Sie in den letzten drei Monaten veröffentlicht haben. Für dieses Beispiel nehmen wir an, dass der kürzeste Beitrag zwei Tage und der längste Beitrag zehn Tage gedauert hat.

Dann nehmen wir an, dass die Wahrscheinlichkeit, dass Sie Beiträge in einer beliebigen Anzahl von Tagen zwischen zwei und zehn fertigstellen, gleich groß ist. Anders ausgedrückt: **Wir nehmen an, dass die Verteilung der Bearbeitungszeit Ihrer Beiträge gleichmäßig** ist (wie bei einem Würfel).

Wenn diese Annahmen zutreffen, könnten Sie ein Programm schreiben, das wiederholt simuliert, wie lange es dauern würde, 60 Blogbeiträge zu schreiben.

Bei dieser Simulation würde Ihr Programm für jeden Blogbeitrag zufällig eine Zahl zwischen zwei und zehn wählen, als ob dies die Anzahl der Tage wäre, die Sie zum Schreiben des Beitrags benötigen.

Nachdem Sie simuliert haben, wie lange Sie für jeden Beitrag gebraucht haben, summiert Ihr Programm die Bearbeitungszeit jedes Beitrags und prüft, ob alle 60 Beiträge weniger als 365 Tage gebraucht haben.

Schließlich würde Ihr Programm, nachdem es alle Simulationen durchgeführt hat, die Erfolgswahrscheinlichkeit berechnen, indem es die Anzahl der “erfolgreichen” Simulationen durch die Gesamtzahl der Simulationen teilt.

```
extern crate rand;

use rand::distributions::{Distribution,
  Uniform};
use rand::thread_rng;

const TOTAL_RUNS: i32 = 1_000_000;
const TOTAL_BLOG_POSTS: i32 = 60;

fn main() {
  let mut rng = thread_rng();
  let time_to_completion =
    Uniform::from(2..11);

  let mut successes = 0;

  for _ in 0..TOTAL_RUNS {
    let mut current_duration = 0;

    for _ in 0..TOTAL_BLOG_POSTS {
      current_duration +=
        time_to_completion.sample(&mut rng);
    }

    if current_duration <= 365 {
      successes += 1
    }
  }

  let p = f64::from(successes) /
    f64::from(TOTAL_RUNS) * f64::from(100);
```

```
  println!("Total Simulations: {}",
    TOTAL_RUNS);
  println!("Successes: {}", successes);
  println!("Probability of succeeding: {:.2}%",
    p);
}
```

Nach dem Lauf des Programms sollten sich ein Ergebnis wie das folgende zeigen:

```
    Total Simulations: 1000000
    Successes: 608671
    Probability of succeeding: 60.87%
```

Das ist ein großartiger erster Versuch, aber sagen wir mal, er fühlt sich für Sie nicht ganz richtig an. Was könnten Sie tun, um Ihre Simulation genauer zu machen?

Ein guter Ansatz: gewichtete Wahrscheinlichkeiten

Nehmen wir an, dass die kürzeste Geschichte zwar zwei Tage und die längste zehn Tage gedauert hat, Sie sich aber gemerkt haben, dass die Zeit für die meisten Beiträge in Richtung der Zehn-Tage-Marke tendiert.

Mit anderen Worten, Sie glauben nicht, dass die Zykluszeiten der Beiträge gleichmäßig verteilt sind: Die meisten Beiträge dauern lange, und nur wenige brauchen zwei oder drei Tage.

Um das Modell genauer zu machen und eine realistischere Prognose zu erhalten, erfassen Sie das Anfangs- und Enddatum jedes Beitrags und berechnen die Zykluszeit jedes Beitrags.

N	Titel	Anfangsdatum	Enddatum	Benötigte Zeit
1	Why Your Software Never Works Out the Way You Plan	12/01/2022	19/01/2022	8
2	Ways Your Mother Lied to You About Software	21/01/2022	29/01/2022	9
3	A Software Success Story You'll Never Believe	02/02/2022	03/02/2022	2
4	Darth Vader's Guide to Software	18/02/2022	23/02/2022	6
5	How to Win Big in the Software Industry	24/02/2022	29/02/2022	6
6	Why Do People Think Software is a Good Idea?	01/03/2022	08/03/2022	8
7	Shocking Ways Software Will Make You a Better Dancer	09/03/2022	12/03/2022	3
8	What The Beatles Could Learn from Software	14/03/2022	23/03/2022	10
9	What Your Parents Never Told You About Software	23/03/2022	28/03/2022	6
10	How To Create The Worst Blog Post Titles	28/03/2022	31/03/2022	4

Mit diesen Daten können Sie die Wahrscheinlichkeit für die tatsächliche Dauer berechnen und eine Tabelle füllen wie die folgende:

Dauer	Häufigkeit	Wahrscheinlichkeit

Dauer	Häufigkeit	Wahrscheinlichkeit
10	1	10%
9	1	10%
8	2	20%
6	3	30%
4	1	10%
3	1	10%
2	1	10%

Wenn Sie wissen, wie wahrscheinlich jede Dauer ist, wird Ihre Simulation viel genauer.

Bei dem naiven Ansatz, den Sie gerade gesehen haben, dauerte ein Problem mit gleicher Wahrscheinlichkeit zwei, sechs oder acht Tage, was nicht der Realität entspricht.

Anhand der obigen Tabelle können Sie leicht erkennen, dass die Wahrscheinlichkeit, dass eine Aufgabe sechs Tage dauert, dreimal so hoch ist wie die von zwei Tagen. Tatsächlich dauern nur 30 Prozent der Geschichten vier Tage oder weniger.

Aktualisieren wir nun unsere Simulation so, dass sie die Wahrscheinlichkeit des Eintretens jeder Dauer beachtet.

```
extern crate rand;

use rand::distributions::{Distribution,
  Uniform};
use rand::thread_rng;

const TOTAL_RUNS: i32 = 1_000_000;
```

```
const TOTAL_BLOG_POSTS: i32 = 60;

const DURATIONS: [i32; 10] =
  [2, 3, 4, 6, 6, 6, 8, 8, 9, 10];

fn main() {
  let mut rng = thread_rng();
  let time_to_completion =
    Uniform::from(0..DURATIONS.len());

  let mut successes = 0;

  for _ in 0..TOTAL_RUNS {
    let mut current_duration = 0;

    for _ in 0..TOTAL_BLOG_POSTS {
      let random_index =
        time_to_completion.sample(&mut rng);
      current_duration +=
        DURATIONS[random_index];
    }

    if current_duration <= 365 {
      successes += 1
    }
  }

  let p = f64::from(successes) /
    f64::from(TOTAL_RUNS) * f64::from(100);

  println!("Total Simulations: {}",
    TOTAL_RUNS);
  println!("Successes: {}", successes);
  println!("Probability of succeeding: {:.2}%",
    p);
}
```

Nach der Durchführung dieser Simulation, die viel genauer ist, werden Sie feststellen, dass Ihre Erfolgswahrscheinlichkeit bei 36,69 % liegt. Das ist fast die Hälfte der vorherigen Schätzung!

```
    Total Simulations: 1000000
    Successes: 366944
    Probability of succeeding: 36.69%
```

Diese Schätzung ist wahrscheinlich gut genug für eine einzelne Person, aber sie wird nicht für ein Team funktionieren. Zumindest nicht ohne Anpassungen [2].

Das Problem bei dieser Simulation ist, dass sie davon ausgeht, dass die Blogbeiträge in Serie und nicht parallel geschrieben werden.

Würden zwei oder mehr Personen parallel arbeiten, müsste man ihre individuellen Zykluszeiten aufzeichnen, simulieren, wie viele Blogeinträge jede Person in 365 Tagen schreiben könnte, und den Output jeder Person summieren.

Das ist nicht allzu schwierig, aber es gibt einen einfacheren Weg.

Stichprobenartiger Durchsatz

Eine einfachere Methode, die Leistung eines Teams zu simulieren, ohne die Zykluszeiten der einzelnen Personen zu verwenden, ist die Stichprobe des Durchsatzes des Teams für jeden Tag der Simulation.

Um eine solche Simulation durchzuführen, müssen Sie die tägliche Anzahl der fertiggestellten Artikel über einen bestimmten Zeitraum hinweg aufzeichnen. Mit diesen Daten haben Sie dann die

Wahrscheinlichkeit, dass an einem bestimmten Tag ein Durchsatz von 0, 1, 2, 3 oder mehr Artikeln erreicht wird, die Sie bei der Stichprobenziehung verwenden.

Wenn Sie ein Programm schreiben würden, das genau das tut, hätten Sie einen Code wie den folgenden:

```
extern crate rand;

use rand::distributions::{Distribution,
  Uniform};
use rand::thread_rng;

const TOTAL_RUNS: i32 = 1_000_000;
const TOTAL_STORIES: i32 = 500;

const TEN_DAY_THROUGHPUTS: [i32; 10] =
  [1, 2, 0, 1, 1, 2, 3, 1, 2, 1];

fn main() {
  let mut rng = thread_rng();
  let throughput = Uniform::from
    (0..TEN_DAY_THROUGHPUTS.len());

  let mut successes = 0;

  for _ in 0..TOTAL_RUNS {
    let mut stories_completed = 0;

    for _ in 0..366 {
      let random_index =
        throughput.sample(&mut rng);
      stories_completed +=
        TEN_DAY_THROUGHPUTS[random_index];
    }

    if stories_completed > TOTAL_STORIES {
```

```
      successes += 1
    }
  }

  let p = f64::from(successes) /
    f64::from(TOTAL_RUNS) * f64::from(100);

  println!("Total Simulations: {}",
    TOTAL_RUNS);
  println!("Successes: {}", successes);
  println!("Probability of succeeding: {:.2}%",
    p);
}
```

Sie könnten diesen Ansatz verwenden, um zu ermitteln, ob ein Softwareentwicklungsteam eine bestimmte Anzahl von Aufgaben innerhalb eines bestimmten Zeitraums abschließen kann.

Dinge, die es bei der Erstellung von Monte-Carlo-Simulationen zu beachten gilt

Bevor ich behaupte, dass die obige Simulation zuverlässig ist, muss ich auf einige Dinge hinweisen, die Sie beachten müssen, wenn Sie eine Monte-Carlo-Simulation zur Erstellung von Prognosen verwenden:

- Die Qualität der Inputs, von denen Ihre Simulationen abhängen.
- Die Konsistenz und Vorhersagbarkeit Ihres Teams.
- Die Notwendigkeit, neue Prognosen zu erstellen.
- Die Größe Ihrer Arbeitspakete.

Qualität der Inputs

Angenommen, Sie hatten letztes Jahr ein Team von zehn Ingenieuren, aber die Hälfte von ihnen hat vor drei Monaten das

Unternehmen verlassen. Wenn Sie simulieren wollen, wie lange dieses Team für die Fertigstellung eines Projekts benötigt, sollten Sie keine Durchsatzdaten verwenden, die länger als drei Monate zurückliegen.

Wenn Sie die Durchsatzdaten eines 10-Mann-Teams verwenden, um die Leistung eines 5-Mann-Teams zu prognostizieren, werden Sie zu optimistische Ergebnisse erhalten.

Ähnlich verhält es sich, wenn sich die Größe Ihres Teams verdoppelt. In diesem Fall sollten Sie für Ihre Simulation keine historischen Daten verwenden, die aus der Zeit stammen, als Sie nur halb so viele Mitarbeiter hatten. In diesem Fall können Sie auch nicht einfach davon ausgehen, dass Projekte nur die Hälfte der Zeit in Anspruch nehmen werden, denn es ist äußerst unwahrscheinlich, dass die Produktivität linear ansteigt, wenn Sie mehr Entwickler einstellen.

Außerdem können andere Faktoren wie Feiertage oder ein umfangreicher Rewrite Ihre Eingabedaten beeinflussen.

Der Durchsatz Ihres Teams zwischen Weihnachten und Silvester wird zum Beispiel wahrscheinlich nicht derselbe sein wie im Februar. In ähnlicher Weise wird eine umfangreiche Neufassung, die die Möglichkeiten zur Änderung und Erweiterung Ihrer Software drastisch erhöht, einen ebenso dramatischen Einfluss auf Ihre Zykluszeiten haben.

Wie Daniel Vacanti in When Will It Be Done? erklärt, *“geht die Monte-Carlo-Simulation von der Grundannahme aus, dass die*

Zukunft, die man vorherzusagen versucht, ungefähr so aussieht wie die Vergangenheit, für die man Daten hat”.

Konsistenz und Vorhersagbarkeit

Unsere Simulation wird unter den oben genannten Umständen gut funktionieren, weil ich vorhersehbarere Durchsatzwerte gewählt habe.

Die Bandbreite der in diesem Code verwendeten Durchsatzwerte ist gering, und an den meisten Tagen ist das Team sehr konsistent: Es erledigt ein oder zwei Aufgaben. Mit anderen Worten, die Standardabweichung für diese Zykluszeitverteilung ist klein.

In einem anderen Blogbeitrag werde ich erklären, was passiert wäre, wenn ich im vorherigen Abschnitt unregelmäßigere Durchsatzwerte verwendet hätte. Solange Sie ein Team mit konstantem Durchsatz haben, wird diese Simulation zuverlässig sein.

Die Quantifizierung von “Vorhersagbarkeit” und “Konsistenz” ist auch ein Thema für einen anderen Blogbeitrag.

Re-forecast!

Mit dem Fortschreiten Ihres Projekts werden Sie auf Verzögerungen stoßen, und manchmal werden Sie dem Zeitplan sogar voraus sein. Daher wird Ihre Prognose durch eine erneute Vorausschätzung im Laufe des Projekts genauer, da es weniger Posten und mehr Daten und somit weniger Raum für Fehler gibt.

> Je mehr Informationen und je weniger zu simulierende Elemente Sie haben, desto mehr wird sich Ihre

Prognose der Realität annähern.

Es ist viel schwieriger, den Sieger der Premier League zu erraten, bevor sie beginnt, als nach der Hälfte der Spielzeit zu erraten.

Wenn die Liga beginnt, haben Sie viele Spiele zu simulieren, so dass Sie mehr Spielraum für Fehler haben.

Außerdem liegen zu diesem Zeitpunkt noch keine Daten vor, die die aktuelle Leistung der Mannschaft widerspiegeln. Stellen Sie sich vor, Sie könnten die Leistung des FC Barcelona im Jahr 2021 anhand der letztjährigen Aufstellung vorhersagen. Ohne Messi wird sie sicher nicht dieselbe sein.

Eine erneute Vorhersage ist auch wichtig, um flexibel zu bleiben.

Nehmen wir an, Sie machen nur eine einzige Prognose zu Beginn Ihres Projekts und erstellen auf der Grundlage dieser Schätzungen ein Gantt-Diagramm. In diesem Fall führen Sie einfach einen Wasserfallprozess mit statistisch genauen Daten durch.

In einem agilen Kontext können Sie bei der erneuten Durchführung dieser Simulationen Ihren Plan und Ihre Prozesse auf der Grundlage der Ergebnisse anpassen, die immer genauer werden, je mehr sich die Realität den geschätzten Ergebnissen annähert.

Zusammenfassend lässt sich sagen: Je weniger Tage und Elemente Sie simulieren müssen, desto geringer ist die Fehlerspanne.

Größenüberlegungen

Zu Beginn habe ich behauptet, dass ein statistischer Ansatz alle Schätzungen überflüssig machen könnte.

Das ist zwar technisch richtig, da man keine Story Points mehr zuweisen muss (und auch keine fundierten Schätzungen mehr vornehmen kann), aber bei durchsatzbasierten Simulationen wird davon ausgegangen, dass die Arbeitspakete ungefähr gleich groß sind. Daher könnte man **behaupten, dass die annähernd gleiche Größe von Arbeitspaketen eine andere Form der Schätzung ist**.

Trotz dieser Behauptung liefern diese statistischen Ansätze immer noch viel bessere Ergebnisse, da Ausreißer Ihre Simulation nicht wesentlich beeinträchtigen.

Außerdem müssen Sie keine Zeit mit der Diskussion verschwenden, ob etwas zwei oder drei Story Points wert ist. Stattdessen können Sie immer versuchen, die Geschichten so klein wie möglich zu machen, damit Sie aufschlussreichere Daten erhalten.

Einsatz von Monte-Carlo-Simulationen in der Praxis: Konfidenzintervalle

In der realen Welt suchen wir selten nach der Wahrscheinlichkeit, dass ein Projekt innerhalb eines bestimmten Zeitraums erfolgreich ist. Stattdessen geht es darum, die möglichen Szenarien zu verstehen, in die wir geraten könnten, und wie wahrscheinlich es ist, dass diese Szenarien eintreten. Mit anderen Worten, **wir wollen wissen, wie die Ergebnisse unserer Simulation verteilt sind** und **unsere Prognosen mit “Vertrauenswerten” versehen**.

Für die Visualisierung einer Ergebnisverteilung ist ein Histogramm ein hervorragendes visuelles Werkzeug. Die dem Histogramm zugrundeliegenden Daten ermöglichen es uns außerdem, unseren Prognosen Vertrauenswerte zuzuordnen.

Um zu veranschaulichen, wie Sie Histogramme in der Praxis einsetzen können, stellen wir uns vor, dass Ihr Unternehmen ein Projekt mit etwa 50 Stories durchführen muss[3]. Damit das Projekt erfolgreich verläuft, möchte Ihr Unternehmen, dass Sie ein Enddatum vorhersagen, damit sich die Marketing- und Vertriebsteams auf den Start vorbereiten können. In Anbetracht der Tatsache, dass das Marketing für den Erfolg dieses wichtigen Produkts von entscheidender Bedeutung ist, möchte die Geschäftsleitung, dass Sie sich bei Ihrer Prognose ziemlich sicher sind.

Lassen Sie uns nun versuchen, dieses Problem zu lösen und eine genaue Prognose zu erstellen.

Da Sie auf Nummer sicher gehen wollen, definieren wir eine “präzise” Prognose als eine Prognose, bei der Sie sich zu mindestens 95 Prozent sicher sind.

Zunächst passen wir unseren Code an, um Histogramme zu erstellen, die die voraussichtlichen Termine für den Abschluss des Projekts anzeigen.

Aktualisieren Sie den Code, um zu speichern, wie oft jedes Datum in Ihren Simulationen vorkommt.

```
extern crate rand;
```

```
use rand::distributions::{Distribution,
  Uniform};
use rand::thread_rng;
use std::collections::HashMap;

const TOTAL_RUNS: i32 = 1_000_000;
const STORIES_TARGET: i32 = 50;
const TEN_DAY_THROUGHPUTS: [i32; 10] =
  [1, 2, 0, 1, 1, 2, 3, 1, 2, 1];

fn main() {
  let mut rng = thread_rng();
  let throughput = Uniform::from(
    0..TEN_DAY_THROUGHPUTS.len());

  let mut outcomes: HashMap<i64, i32> =
    HashMap::new();

  let one_day = Duration::days(1);

  let start_date: DateTime<Local> =
    Local::now();

  for _ in 0..TOTAL_RUNS {
    let mut current_date = start_date;
    let mut stories_completed = 0;

    while stories_completed < STORIES_TARGET {
      let random_index =
        throughput.sample(&mut rng);
      stories_completed +=
        TEN_DAY_THROUGHPUTS[random_index];
      current_date = current_date + one_day;
    }

    let count = outcomes.entry(
      current_date.timestamp()).or_insert(0);
    *count += 1;
```

```
    }

    println!("Total Simulations: {}", TOTAL_RUNS);
}
```

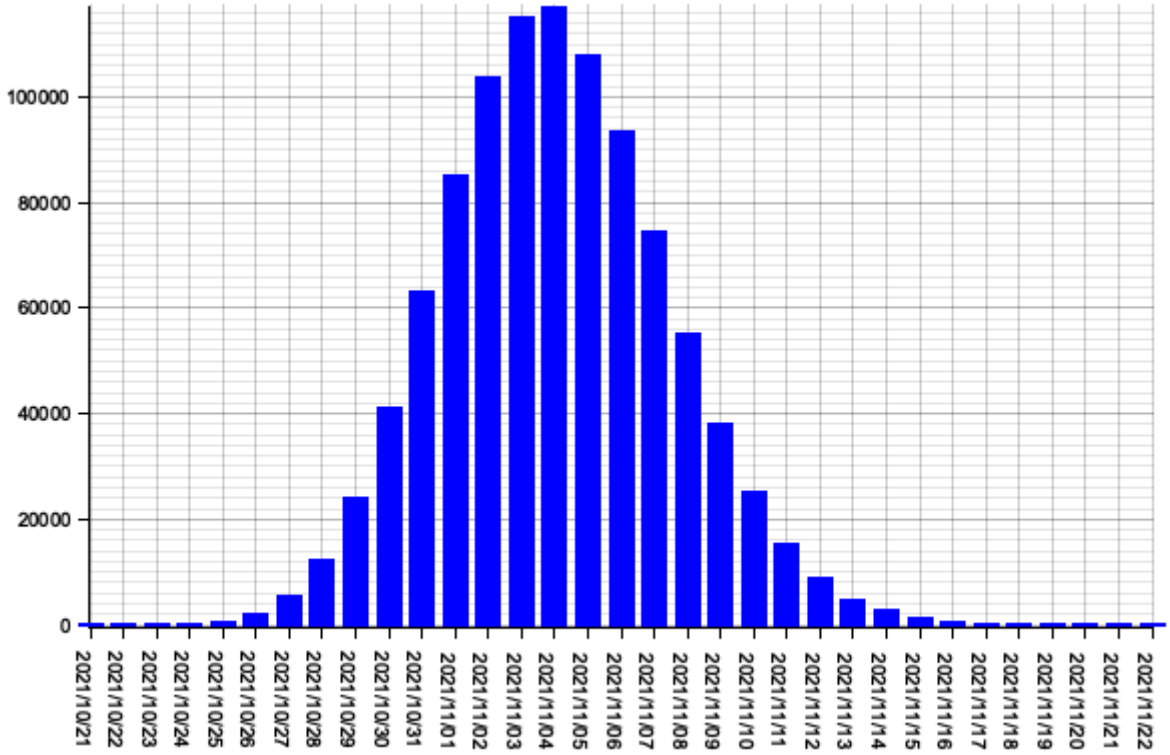
Wie Sie im obigen Code sehen können, zielen wir nicht mehr auf ein bestimmtes Ergebnis ab. Stattdessen verfolgen wir, wie oft das Projekt zu einem bestimmten Datum beendet wurde.

Denken Sie daran: Wir wollen verstehen, wie wahrscheinlich es ist, dass jedes Lieferdatum eintritt, damit wir eine Prognose erstellen können, auf die wir zu 95 Prozent vertrauen können.

Nach dieser Änderung werden wir eine Funktion schreiben, die the plotters crate verwendet, um ein Histogramm zu erzeugen und es auf der Festplatte zu speichern. Dieses Histogramm zeigt auf der X-Achse die möglichen Liefertermine an und wie oft jeder Liefertermin in den Simulationen auftrat.

> Die Plotting-Funktion ist ziemlich lang und für unsere Aufgabe nicht wirklich interessant, daher werde ich sie in diesem Beitrag weglassen. Wenn Sie sie trotzdem sehen wollen, werfen Sie einen Blick auf das zu diesem Beitrag gehörende GitHub-Repository.

Nach der Ausführung der Simulation sollte Ihr Programm ein Histogramm ähnlich dem folgenden liefern.



Ein Histogramm, das zeigt, dass das Projekt in den meisten Simulationen X Tage bis zur Fertigstellung benötigte
Die geschätzten Liefertermine unseres Projekts folgen einer Normalverteilung, wie das Histogramm zeigt.

Anhand dieses Histogramms können Sie deutlich erkennen, dass, wenn wir heute (29. September 2021) mit der Arbeit an den 50 Stories des Projekts beginnen, der wahrscheinlichste Termin für die Fertigstellung der 4. November 2021 wäre.

Wenn es sich nicht um ein wichtiges Projekt handeln würde, für das das Unternehmen eine genaue Vorhersage benötigt, könnten Sie den Führungskräften sagen, dass der 4. November 2021 der bestmögliche Termin für den Abschluss des Projekts ist.

Leider wurden Sie gebeten, eine präzise Prognose abzugeben, so dass Sie nicht einfach das wahrscheinlichste Enddatum nennen

können. Stattdessen müssen Sie eine Prognose abgeben, auf die Sie sich zu mindestens 95 Prozent verlassen können.

Um eine Prognose zu erstellen, bei der Sie mindestens 95 Prozent sicher sind, müssen Sie definieren, was “95 Prozent sicher” in Bezug auf unsere Monte-Carlo-Simulation bedeutet. In diesem Fall **könnte man sagen, dass man zu 95 Prozent sicher ist, dass ein bestimmtes Enddatum erreicht wird, wenn in 95 Prozent aller Simulationen das Projekt an oder vor diesem Datum abgeschlossen wird**.

Wenn Sie zum Beispiel ein bestimmtes Projekt hundertmal simulieren und in 95 dieser Simulationen das Projekt am oder vor dem 1. April abschließen, dann sind Sie zu 95 Prozent sicher, dass Sie das Projekt an oder vor diesem Datum abschließen können.

Für diejenigen, die sich besser mit Statistik auskennen, sind dies lediglich Perzentile. Ein Perzentil ist ein Wert, auf oder unter den ein bestimmter Prozentsatz der Verteilungswerte fällt (unter Berücksichtigung einer einschließenden Definition).

Erstellen wir nun eine Funktion, die das 95. Perzentil für unsere Simulation berechnet. Dieses Perzentil gibt das Datum an, an dem oder bevor 95 Prozent der Simulationen abgeschlossen sind. Wir können also sagen, dass wir “zu 95 Prozent sicher” sind, dass dies das Enddatum des Projekts sein wird.

```
fn calculate_percentile(data: &HashMap<i64,
  i32>, percentile: i32) {
    let total_sims: i32 = data.values().sum();

    let p_qtd: i32 = total_sims -
      (total_sims as f64 / (100_f64 /
```

```
    percentile as f64)).ceil() as i32;

  let mut hash_vec: Vec<(&i64, &i32)> =
    data.iter().collect();
  hash_vec.sort_by(|a, b| b.0.cmp(a.0));

  let mut sum = 0;
  let mut percentile_timestamp: i64 = 0;

  for (timestamp, freq) in hash_vec {
    sum += freq;

      if sum >= p_qtd {
        percentile_timestamp = *timestamp;
        break;
      }
  }

  println!(
    "Percentile {}: {} ({}/{})",
      percentile,
      Local
        .timestamp(percentile_timestamp, 0)
        .format("%Y/%m/%d")
        .to_string(),
      sum,
        total_sims
      );
  }
```

Sobald Sie diese Funktion erstellt haben, aktualisieren Sie Ihr Programm so, dass es `calculate_percentile` aufruft und dabei die Daten des Histogramms und `95` als das Perzentil, an dem Sie interessiert sind, übergibt. Wenn Sie diese Änderung vornehmen und das Programm erneut ausführen, erhalten Sie ähnliche Ergebnisse wie die folgenden:

```
    Total Simulations: 1000000
    Percentile 95: 2021/11/13 (60032/1000000)
```

Nachdem Sie die Ergebnisse des Programms gesehen haben, wissen Sie, dass 95 Prozent Ihrer Simulationen am oder vor dem 13. November abgeschlossen sind. Daher können Sie der Unternehmensleitung mitteilen, dass Sie zu 95 Prozent sicher sind, dass das Projekt an oder vor diesem Datum abgeschlossen wird.

Stellen Sie sich nun vor, die Führungskräfte Ihres Unternehmens teilten Ihnen mit, dass sie gerne wissen würden, wie hoch die Wahrscheinlichkeit ist, dass Ihr Projekt zu anderen Terminen abgeschlossen wird. Mit diesen Daten könnten sie entscheiden, ob sie mehr Marketingfachleute einstellen, um zu verhindern, dass sich das Produkt verzögert, falls Sie den Code früher fertigstellen.

Um Ihren Führungskräften diese Daten zur Verfügung zu stellen, können Sie einfach ein paar weitere Aufrufe zu `calculate_percentile` hinzufügen, um das 25., 50., 75. und 85. Percentil zu berechnen. Dies wären die Daten, auf die 25 Prozent, 50 Prozent, 75 Prozent bzw. 85 Prozent der Simulationen fallen oder davor. Dies sind also die endgültigen Daten, bei denen man sagen kann, dass man zu 25 Prozent, 50 Prozent, 75 Prozent und 85 Prozent sicher ist.

```
    Total Simulations: 1000000
    Percentile 25: 2021/11/05 (765267/1000000)
    Percentile 50: 2021/11/07 (546177/1000000)
    Percentile 75: 2021/11/09 (320170/1000000)
    Percentile 85: 2021/11/11 (152931/1000000)
    Percentile 95: 2021/11/13 (59527/1000000)
```

Als Übung würde ich dem Leser auch empfehlen, diese Perzentile in ihren Histogrammen darzustellen. Der Autor wäre an Ihrer Lösung interessiert, da er es nach einem mühsamen Kampf mit Plottern und dem Rust-Compiler aufgegeben hat, dies zu tun.

Über Prognosen auf der Grundlage Ihrer Risikobereitschaft

Wenn Sie wissen, wie wahrscheinlich es ist, dass ein Ereignis eintritt, können Sie Ihre Risikobereitschaft einschätzen und entscheiden, wie genau Ihre Prognose sein muss.

Stellen Sie sich vor, Alice, Ihre beste Freundin, möchte eine Wette abschließen. Sie hat zwei Würfel und ist bereit, 10 Dollar zu wetten, dass Sie bei einem einzigen Wurf keine Summe von sieben erhalten werden.

Außerdem ist sie bereit, Ihnen im Tausch gegen einen Teil ihres Wetteinsatzes weitere Würfe zu gestatten. Wenn Sie zweimal würfeln, setzt sie fünf Dollar statt zehn Dollar. Wenn Sie viermal würfeln wollen, setzt sie 2,50 Dollar.

Mit ein paar schnellen Statistiken können Sie herausfinden, dass Ihre Chance, bei einem Wurf eine Sieben zu würfeln, etwa 17 Prozent beträgt. Bei zwei Würfen steigt sie auf 34 Prozent. Bei vier Würfen haben Sie eine Gewinnchance von 68 Prozent.

Da Sie nun die möglichen Auszahlungen und die Wahrscheinlichkeit des gewünschten Ergebnisses kennen, können Sie abschätzen, wie viele Würfe Sie bereit sind, zu bezahlen.

Wenn Sie zum Beispiel dringend 2,50 Dollar zusätzlich brauchen, um den Bus nach Hause zu bekommen, werden Sie wahrscheinlich 2,50 Dollar setzen und viermal würfeln wollen, um eine höhere Gewinnwahrscheinlichkeit zu haben. Wenn Sie hingegen der Meinung sind, dass es schön wäre, 10 Dollar zu gewinnen und sich eine tolle Pizza zu gönnen, sollten Sie vielleicht nur einmal würfeln.

Wie geht es jetzt weiter?

Ich habe mein Bestes getan, um diesen Blogbeitrag einfach zu halten und umsetzbare Ratschläge zu geben. Ich habe absichtlich zu viel klassische Statistik vermieden, aber ich beabsichtige, in einem weiteren Blogbeitrag Fragen wie diese zu beantworten:

- Wie viele Simulationen braucht man, um “genaue” Ergebnisse zu erhalten?
- Wie definiert man, was “genau” bedeutet, oder anders gesagt, wie quantifiziert man die Genauigkeit?
- Wie lässt sich quantifizieren, wie vorhersehbar Ihr Team ist?

Dies sind alles Fragen, die mit einfachen Statistiken beantwortet werden können. Denjenigen, die das Thema überspringen möchten, empfehle ich diese MIT-Lektion über Konfidenzintervalle.

Außerdem plane ich, einen Blogbeitrag darüber zu schreiben, wie Markov-Chain-Monte-Carlo-Simulationen bei der Erstellung von Prognosen helfen können, insbesondere für unberechenbarere Teams.

Referenzen

[1] Ich verwende Daniel Vacanti 's Definition einer Prognose. In seinen Worten: *"Eine Prognose ist eine Berechnung der Zukunft, die sowohl eine Spanne als auch eine Wahrscheinlichkeit des Eintretens dieser Spanne umfasst."* Diese Definition ist wichtig, weil, wie Daniel selbst in seinem Buch *"Actionable Agile Metrics For Predictability"* erklärt, *"eine Vorhersage ohne eine zugehörige Wahrscheinlichkeit deterministisch ist, und wie Sie wissen, ist die Zukunft alles andere als deterministisch."*

[2] Alternativ könnte ein Team auch seriell arbeiten, um die Zykluszeiten zu verkürzen. In diesem Fall brauchen Sie die Simulation nicht zu ändern, sondern nur die Daten zu aktualisieren, anhand derer bestimmt wird, welche Zykluszeit abgetastet wird. Das Problem bei diesem Ansatz ist, dass er oft unmöglich ist: Ich kann mir nicht vorstellen, dass mehr als zehn Personen gleichzeitig an einem Blogbeitrag schreiben. Wenn überhaupt, würde es zehnmal so lange dauern.

[3] Normalerweise würde ich in dieser Phase eines Projekts nicht empfehlen, Stories zu verwenden, vor allem, wenn es sich um viele handelt. Das würde bedeuten, dass Sie einen großen Aufwand betreiben müssten, um alle Anforderungen im Voraus zu spezifizieren, was überhaupt nicht agil ist. Stattdessen würde ich den Umfang in Epics aufteilen und mich darauf konzentrieren, frühzeitig und häufig zu veröffentlichen und die Prognosen im Laufe des Projekts zu überarbeiten. Denken Sie daran: Ich sage nicht, dass Sie Monte-Carlo-Simulationen verwenden sollten, um genauere Termine für Wasserfallprojekte festzulegen.

Ähnliche empfohlene Inhalte

- GUIMARÃES CARVALHO, Gabriel. The art of solving problems with Monte Carlo simulations
- VACANTI, Daniel. Actionable Agile Metrics For Predictability: An Introduction. ActionableAgile Press.
- VACANTI, Daniel. When Will It Be Done?: Lean-Agile Forecasting to Answer Your Customers' Most Important Question. ActionableAgile Press.

Lucas Fernandes da Costa *ist ein brasilianischer Software-Ingenieur, der in London lebt. Er schreibt die meiste Zeit JavaScript und hat eine Leidenschaft für Open-Source. In den letzten Jahren hat er Chai.js und Sinon.js betreut. Er ist sehr aktiv auf GitHub und hat zu zahlreichen Projekten beigetragen, darunter Jest und NodeSchool.*

Als Autor hat er das Buch "Testing JavaScript Applications" geschrieben, das bei Manning Publications erschienen ist. Er mag Bücher mit eigener Meinung, schönen Code, ausgereifte Prosa, Kommandozeilenschnittstellen und vim. Tatsächlich mag er vim so sehr, dass er sich ein :w auf den Knöchel tätowieren ließ.

Inhalte, die er besitzt, sind immer zu haben. Universitäten und Online-Kurse verwenden einige dieser Beiträge als Referenzmaterial, und Sie können dasselbe tun, wenn Sie wollen. Seine Artikel sind in viele Sprachen übersetzt worden, darunter Russisch, Mandarin, Französisch, Portugiesisch und Spanisch. Wenn Sie einen dieser Beiträge übersetzen möchten, müssen Sie nicht um Erlaubnis bitten.

Er ist der Meinung: "Alles, was ich produziere, gehört dem Internet. Es hat mich zu dem gemacht, was ich bin, und deshalb verdient es

alles, was ich habe.”

Der Artikel beruht auf dem Blogpost How to replace estimations and guesses with a Monte Carlo simulation.

WEB DEVELOPER CONFERENCE '22

DIE KONFERENZ FÜR WEB ENTWICKLER
FRÜHJAHR 2022 | HAMBURG

SAVE THE DATE

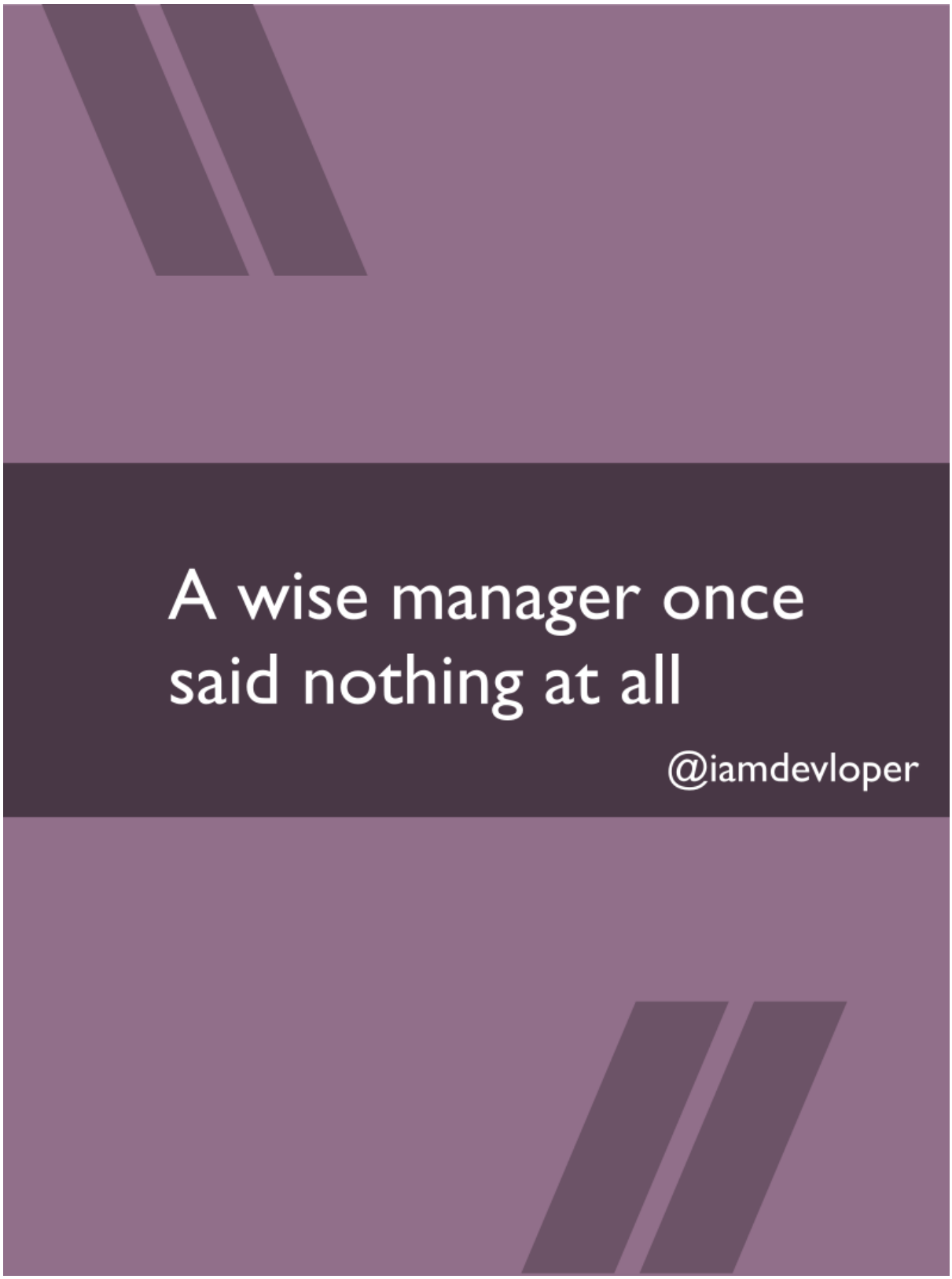
web-developer-conference.de | #wdc22 | Find us on

Präsentiert von:

developer media

dotnetpro

web & mobile DEVELOPER


A wise manager once said nothing at all
@iamdevloper

Fünf Spieleklassiker im Browser spielen

Sie haben alle Geschenke ausgepackt, die dritte Weihnachtsgans verspeist, die Familie mit Spazierengehen terrorisiert oder langweilen sich einfach? Dann spielen Sie doch einen der Klassiker wie beispielsweise den “Prince of Persia”.

Und zwar - ohne spezielle Spielekonsole - ohne Installation - Geld, einfach für lau.

Prince of Persia

Sie haben nur eine Stunde Zeit, um die Prinzessin zu retten, bevor sie den widerlichen Großwesir Jaffar heiraten muss. Sie schleichen durch das Verließ von Jaffar, immer auf der Hut vor den Wachen, heimtückischen Fallen oder zu tiefen Abgründen.

Prince of Persia

Prince of Persia 1990

Tetris

Was soll man da noch erklären: Sie sind Maurer und müssen Objekte in unterschiedlicher Form nahtlos aneinanderfügen. Nur bei vollständigen Zeilen verschwinden diese. Was nötig ist, denn von oben kommen immer neue Objekte. Langeweile ade.

Tetris

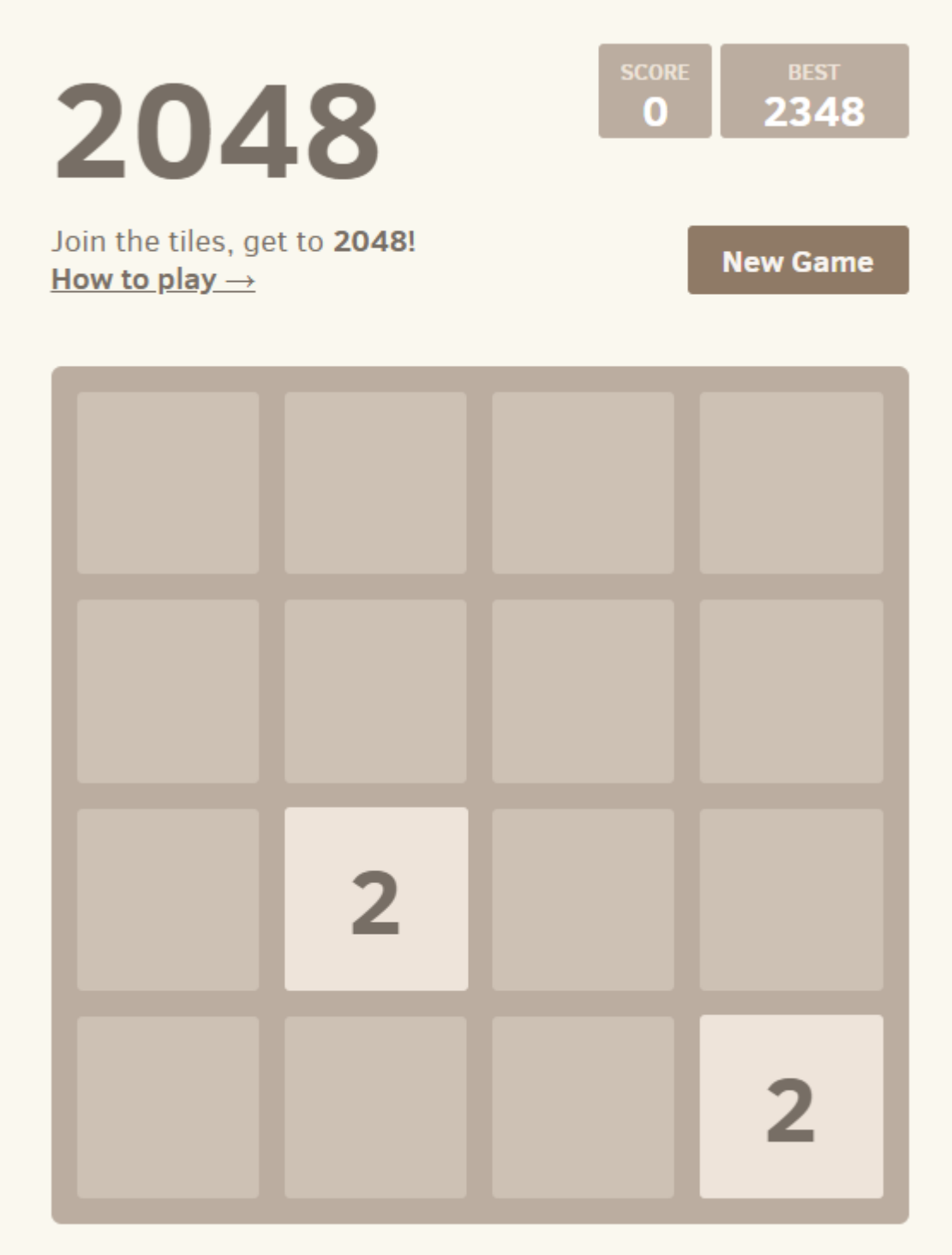
Freecell

Viele Karten, die von As zu König sortiert werden müssen. Auf vier Plätzen können Sie Karten zwischenparken. Los geht’s.

Freecell

2048

Sooo alt ist das Spiel ja noch gar nicht. Doch Achtung: Es macht süchtig. Mit den Cursortasten addiert man immer zwei nebeneinanderliegende Steine mit selbem Wert, bis zum Schluss der Stein mit 2048 erreicht ist oder sich kein Stein mehr bewegen lässt. In ersterem Fall hat man gewonnen, in letzterem Fall - nun ja, denken Sie sich Ihren Teil.



2048

2048

Descent

Die Musik passt zwar zur pixeligen Grafik, auf der es eher schwer ist, die feindlichen Roboter auszumachen. Trotzdem sollte es das erste sein, was man ausschaltet - der geistigen Gesundheit wegen. Dann aber taucht man ab in die Gänge einer fernen Raumstation, die es von feindlichen Robotern zu säubern gilt. Ein Joystick plus ein Keyboard sind für die Steuerung dringend angeraten. Verschiedene Waffen stehen zur Auswahl. Im Manual finden Sie alle Tastenbelegungen und Hilfen für die Einstellungen.

Descent

Jetzt kostenlos testen!

Das Fachmagazin für .NET-Entwickler

Testen Sie jetzt 2 kostenlose Ausgaben und erhalten Sie unseren exklusiven Newsletter gratis dazu.

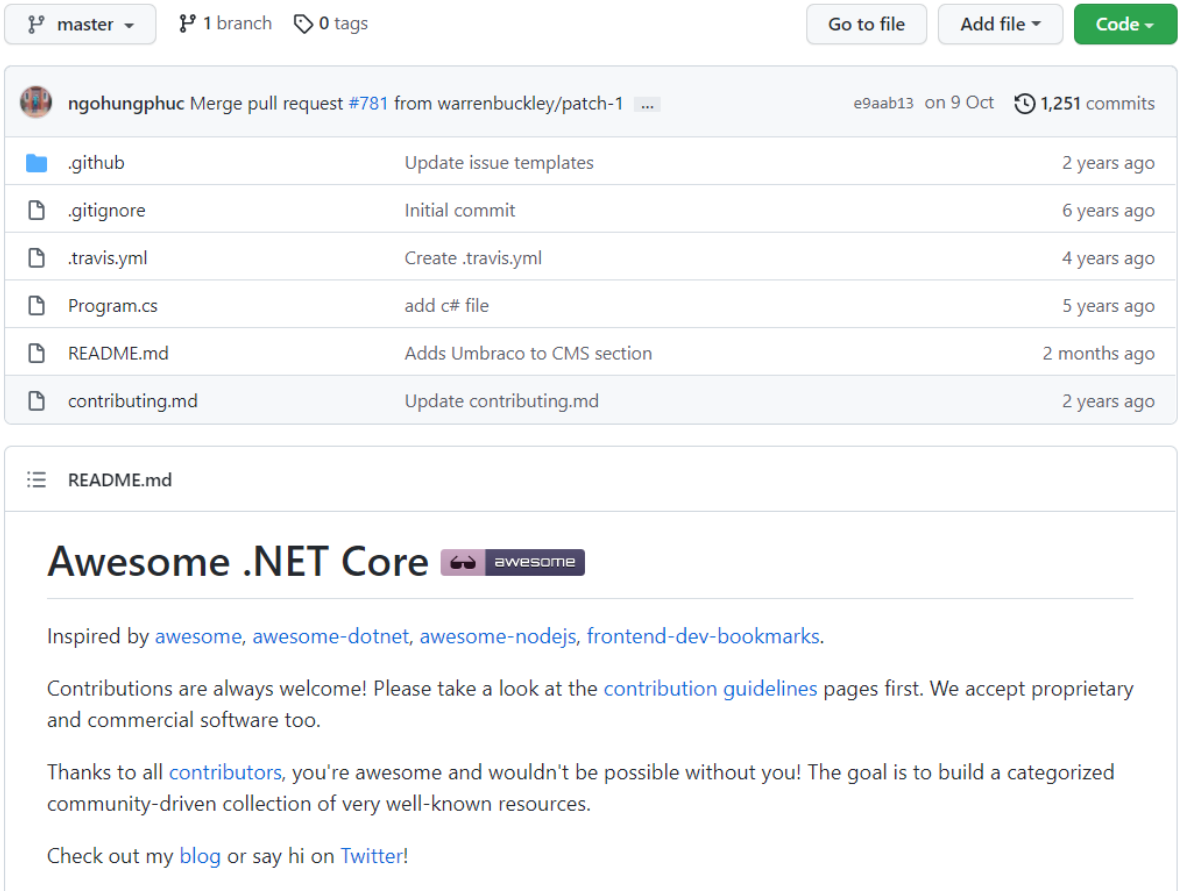
www.dotnetpro.de/dnpabo

Awesome .NET Core

Es dauerte lang, bis klar war, dass das englische Wort *awful* zwar einen gemeinsamen Wortstamm, aber eine diametral andere Bedeutung hat. Während awful soviel bedeutet wie *schrecklich*, steht *awesome* für *toll*, *stark* oder *nice*.

Awesome .NET Core

Und genauso heißt eine Sammlung von - hm, ja: Wofon eigentlich? - Bibliotheken, Tools, Source Code und so weiter, die einem in vielen Fällen weiterhelfen kann.



Haben Sie zum Beispiel keine Lust auf das Entity Framework von Microsoft, hält **Awesome .NET Core** noch andere OR-Mapper für Sie bereit.

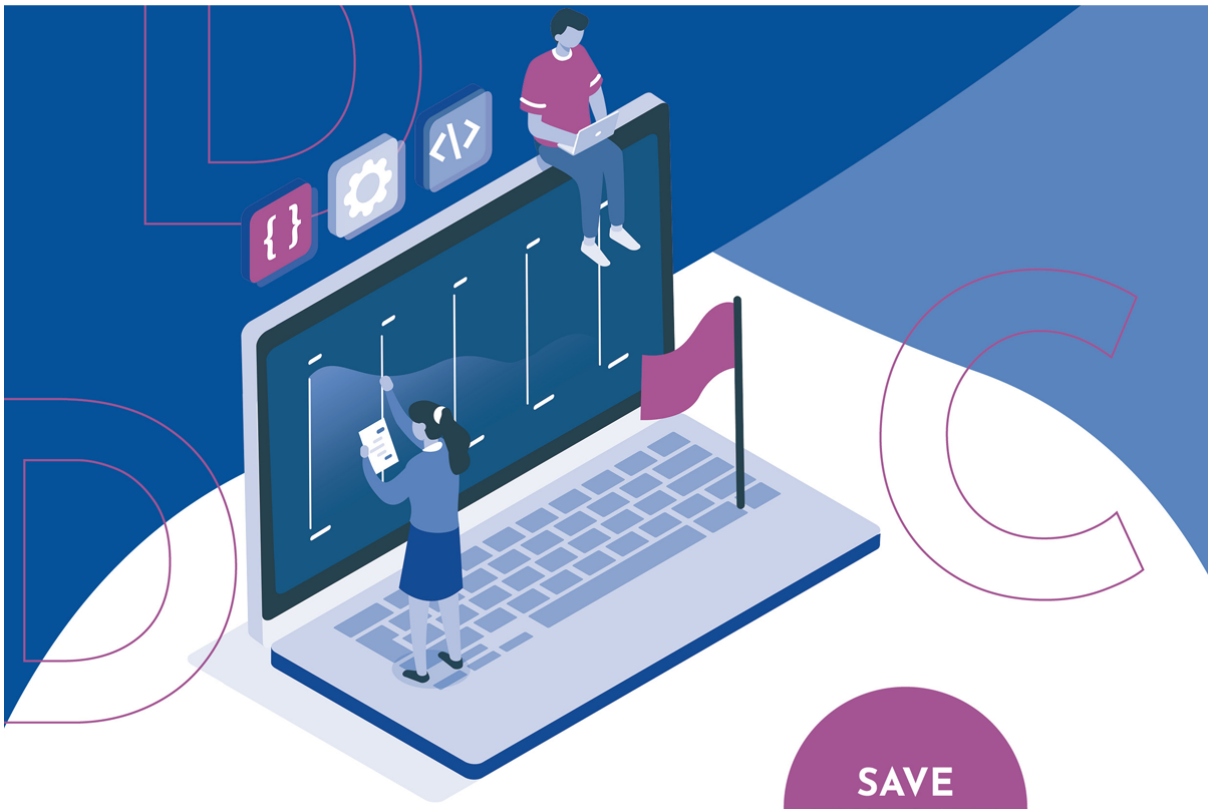
Wissen Sie nicht, welche Tools für den Build-Vorgang unter .NET zur Verfügung stehen, können Sie in der Liste von **Awesome .NET Core** viele Tools finden, die Ihnen dabei weiterhelfen.

Die Sammlung geht sogar bis hin zu vollständigen Content Management Systemen, die auf .NET basieren.

Code Analyse, Kryptografie oder Datenbankzugriff gewünscht? Dann werfen Sie einen Blick hinein. Es lohnt sich. Darin gibt es so viel Material, das Ihnen im Entwickleralltag weiterhelfen kann.

Awesome .NET Core finden Sie auf GitHub.


due to inflation k8s
is now k9s
@jeanqasaur



SAVE
THE DATE

.NET DEVELOPER CONFERENCE '22

DIE KONFERENZ FÜR .NET ENTWICKLER
WINTER 2022 | KÖLN

dotnet-developer-conference.de | #ddc22 | Find us on

Präsentiert von:

developer media dotnetpro web & mobile DEVELOPER

Greencoding: Jede Zeile Code zählt

Rechner heruntertakten oder Bildschirm dunkler regeln: All das hilft, Strom zu sparen und damit die CO2-Emissionen zu reduzieren. Als Softwareentwickler haben Sie aber noch andere Möglichkeiten einzugreifen. Und dieser Hebel kann durchaus gigantische Ausmaße einnehmen.

Beim CO2-Ausstoß denkt man sicher zunächst an die Kohleverstromung, den Flugverkehr, Autos und Lastwagen und vielleicht noch an die eigene Heizung. Die Digitalisierung hingegen sehen viele eher als Heilsbringer, da sich mit ihr beispielsweise Transporte koordinieren und damit CO2-Emissionen einsparen lassen. Aber das ist nur die eine Seite. Denn auch der Betrieb von Laptops, Routern, Servern, der Cloud und Netzen bedarf immer mehr Strom, und damit belastet auch die IT immer mehr die Umwelt durch CO2-Ausstoß. Von Mining-Farmen für Blockchain-Währungen ganz zu schweigen.

Scope 1, 2, 3

Eingedenk, dass ein Baum je nach Größe und Standort pro Jahr rund 10 Kilogramm CO2 bindet, sind Zahlen wie der summierte Stromverbrauch eines Laptops einigermaßen erschreckend: Acht Bäume braucht es im Jahr, um das CO2 zu kompensieren, das nur durch den Stromverbrauch des Computers erzeugt wird. Die Betonung liegt hier auf “nur den Stromverbrauch”. Denn so ein Laptop muss gebaut, transportiert und schließlich auch entsorgt werden.

Man teilt hier den CO2-Fußabdruck einer Firma in drei Bereiche, die Scopes genannt werden [1]. Zu Scope 1 gehören die direkten Emissionen aus den betrieblichen Einrichtungen wie Heizung oder Klimatisierung von Bürogebäuden und die Emissionen der Firmenfahrzeuge. Scope 2 ist der gekaufte Strom beziehungsweise die gekaufte Wärme etwa bei Fernwärme. Und zu Scope 3 zählen Dinge wie der Arbeitsweg, aber besonders die Emissionen, die durch Produktion, Transport und Entsorgung für die Ausgangsstoffe erzeugt werden, die für die Produktion der eigenen Produkte benötigt werden (vorgelagert), sowie Transport, Betrieb und Entsorgung für die eigenen Produkte (nachgelagert).

Während bei Software der vorgelagerte Scope 3 verschwindend gering ist, schlägt sie beim nachgelagerten Scope 3 voll zu. Software, die eine Firma für einen Kunden geschrieben hat, wird danach verwendet und verbraucht dabei Strom, benötigt Server und so weiter. Microsoft geht davon aus, dass sie im Jahr 2020 über alle Scopes hinweg 16 Millionen Tonnen CO2 emittiert haben [2]. Zwölf Millionen Tonnen, also 75 Prozent, davon entfallen auf Scope 3, also in erster Linie auf den Betrieb der Software, die Microsoft verkauft. Es ist also für Consultingunternehmen und

Softwareentwickler elementar, sich Gedanken über ihre Software zu machen, denn schlecht geschriebene Software kann für große CO2-Emissionen sorgen, wohingegen gut geschriebene Software unnötigen Stromverbrauch vermeiden hilft.

Wieviel Emissionen entfallen auf die IT?

Derzeit beträgt der Anteil an den Emissionen für die IT laut GFT drei Prozent im Vergleich zur gesamten Industrie mit 24 Prozent. Aber mit der fortschreitenden Digitalisierung wird dieser Anteil ansteigen. Wie andere Branchen auch, kann die IT gegen das Anwachsen allerdings einiges tun. Grüner Strom ist sicher ein probates Mittel, das langfristig aber nicht ausreichen wird. Vielmehr wird schon bei der Entwicklung von Software darauf geachtet werden müssen, dass stromzehrende Prozesse durch stromsparende ersetzt werden.

Inwiefern der Dark Mode eines Bildschirms hier etwas bringt, hängt vom Bildschirm ab. Aber vielfache Anfragen an einen Server, die durch eine einzige ersetzt werden, können Strom sparen. Wo bei der Softwareentwicklung gilt: “Eine nicht geschriebene Zeile Code enthält auch keine Bugs”, gilt für den Energiebedarf: “Jede Anfrage, die nicht gestellt werden muss, verbraucht auch keinen Strom.”

Die GFT stellt sich dieser Herausforderung und sammelt in einem internen Arbeitskreis Möglichkeiten, bei der Softwareentwicklung Strom zu sparen. Als Beispiel für Einsparpotentiale nennt Tim Schade, Softwarearchitekt bei GFT, die App-Entwicklung. Ist ein Entwicklungsteam in der Lage, den Startvorgang der App von drei auf zwei Sekunden reduzieren, so dass der Anwender eine Sekunde früher mit seiner Arbeit beginnen kann, so spart das bei

einer Million Nutzer der App in einem Jahr 132 Kilogramm CO2. Das entspricht der Menge an Treibhausgas, die rund 13 Bäume pro Jahr binden. Oder, anders ausgedrückt, den Emissionen einer Autofahrt von Berlin nach Wien.

Das Beispiel zeigt, dass es durchaus massive Einsparschätze gibt, die Entwickler mal einfacher, mal aufwändiger heben können. Recht einfach lässt sich beispielsweise die Datenmenge reduzieren, die bei einer Anfrage an einen Webserver auftritt. Das Stichwort heißt hier Komprimierung. Fragt der Browser beim Server die Inhalte einer Url an, so gibt der Server über den http-Header dem Server bekannt, welche Formate er versteht [3]. Der Server kann dementsprechend die Dateien in diesem Format schicken. Auf diese Weise lassen sich HTML-Seiten auch gezipt, also komprimiert übertragen. Der Browser gibt über den Eintrag

```
Accept-Encoding: gzip, compress, deflate
```

im Header der Anfrage bekannt, dass er sich auch auf komprimierte Datenpakete versteht. Der Server antwortet [4] darauf im Header beispielsweise mit

```
Content-Encoding: gzip
```

was so viel heißt wie: “Ich kann dir die Daten in einem gzip-Archiv schicken.” Stimmen Anfrage und Angebot überein, kann der Browser die Datei in diesem Format laden.

Bei HTML-Dateien reduziert das komprimierte Übertragen die Datenmenge um rund 90 Prozent. Die GFT rechnet vor, dass das bei HTML-Daten von 200 Kilobyte komprimiert auf 20 Kilobyte pro Jahr 10 Kilogramm CO2 einsparen kann, wenn die Seite rund eine

Million Mal pro Jahr übertragen wird. Das gilt allerdings nur für Textdateien und nicht für Bilder, da diese meist sowieso schon komprimiert sind.

Push und Cache

Der letzte Abschnitt zeigt schon, dass die Übertragung von Daten, oder besser die Nicht-Übertragung, der große Hebel sein kann. Mit einem Cache müssen viele Abfragen gar nicht erst stattfinden, da sie in der Vergangenheit schon einmal durchgeführt wurden und die Ergebnisse schon im Cache liegen.

Aber auch Techniken wie Push statt Pull bringen Einsparungen: Statt beispielsweise alle zehn Sekunden beim Server anzufragen, ob sich Daten geändert haben, registriert sich der Client beim Server. Ändern sich jetzt die Daten auf dem Server, teilt dieser das dem Client mit. Es muss also nur dann etwas übertragen werden, wenn sich auch tatsächlich etwas geändert hat.

Fazit

Zugegeben: Die Denke, dass man mit dem eigenen Code Strom und damit CO2-Emissionen einsparen kann, ist erst einmal befremdlich – aber wie die Beispiele zeigen, durchaus nachvollziehbar. Es hängt also auch an Ihnen, etwas für einen geringeren Stromverbrauch zu tun.

Tilman Börner *ist Chefredakteur der dotnetpro.*

Jetzt kostenlos testen!

2 x gratis!

Praxiswissen für Entwickler!

Testen Sie jetzt 2 kostenlose Ausgaben und erhalten Sie exklusiven Zugang zu unserem Archiv.

webundmobile.de/probelesen

Optimist: Das Glas ist halb voll
Pessimist: Das Glas ist halb leer
Programmier: Braucht kein Glas

unbekannt

Hacking YouTube With MP4

Während einer nächtlichen Debugging-Sitzung an einem MP4-Muxer entdeckte ich zufällig eine potenzielle Sicherheitslücke in YouTube. Dies ist eine Geschichte darüber, wie ein einfacher Fehler in meinem eigenen Code mich dazu brachte, die Sicherheitsauswirkungen einer Videotranskodierungspipeline zu überdenken.

Muxer verstehen

Wenn Sie nicht wissen, was ein “Muxer” ist, ist das in Ordnung. Ich hatte auch keinen blassen Schimmer, bis ich tatsächlich einen reparieren musste.

> Muxen ist eine Abkürzung für Multiplexen. Muxing ist der Prozess, bei dem mehrere kodierte Streams - Audio, Video und Untertitel (falls vorhanden) - in ein Containerformat wie AVI, Ogg oder Matroska gekapselt werden. (Zitiert von VideoLAN)

Ein Muxer ist nur ein Begriff für eine Software, die Multiplexing durchführt.

Mein Muxer hatte einen Fehler, einen schwerwiegenden Fehler. Ich hatte ihn so eingestellt, dass er zehn Sekunden Videomaterial aufnimmt, doch er gab eine beschleunigte Version von acht Sekunden aus. Da der Muxer Open Source ist und in Golang geschrieben wurde, dachte ich, dass dieses Problem relativ leicht zu beheben sein sollte. Einfach ein paar Zeitstempel ändern, richtig?

Nun, es stellte sich heraus, dass es mehrere Arten von Zeitstempeln im MP4-Format gibt:

- Decode Timestamps (DTS): Wann werden die Frames decodiert?
- Präsentationszeitstempel (PTS): Wann werden die Bilder auf dem Bildschirm angezeigt?
- Zeitstempel für die Zusammenstellung (CTS): Wann wird ein Bild zusammengesetzt?

Jeder dieser Zeitstempel dient einem bestimmten Zweck, aber mein Fehler lag bei den Präsentationszeitstempeln, meine Videobilder wurden nicht rechtzeitig angezeigt.

Zeitstempel im MP4-Format sind nicht das typische Format (Unix-Zeitstempel, …). Stattdessen handelt es sich eher um eine Zeitdifferenz zwischen dem Beginn und dem Ende eines Videos.

Angenommen, Sie möchten ein Bild bei der fünften Sekunde anzeigen, dann würden Sie es wie folgt umwandeln:

```
ts = 5 * timeScale
```

Wobei *timeScale* ein beliebiger Wert ist, den man wählen kann und der den Decodern im Wesentlichen sagt, dass:

```
1 Sekunde = timeScale
```

Entdeckung des Fehlers

Einer der Fehler in meinem Muxer bestand darin, dass ich in meinem MP4-Header eine timeScale von 12.000 angegeben hatte, in Wirklichkeit aber bei der Berechnung der Dauer eine timeScale von 10.000 verwendet wurde. Dies führte zu einem interessanten Nebeneffekt: Der Decoder dachte, dass ein Präsentationszeitstempel von 20.000 1,6 Sekunden statt 2 Sekunden sei.

Ich spielte mit den Werten für den Präsentationszeitstempel und produzierte plötzlich dieses Video mit einer Größe von **4 MB, das jedoch angeblich 15 Stunden Filmmaterial enthielt**.

Was würden Sie tun, wenn Sie auf eine so seltsame Datei stoßen? Nun, ich habe sie auf YouTube hochgeladen und zu meiner Überraschung wurde sie nicht abgelehnt, weil sie länger als 12 Stunden war. Der Thumbnail-Dienst versuchte sogar, Thumbnails für das Video zu erstellen. Es vergingen einige Tage, bis schließlich eine Fehlermeldung erschien

> Verarbeitung abgebrochen Video ist zu lang.

Bewertung der Auswirkungen

Ich beschloss, einige Tests durchzuführen und Videodateien zu generieren, um zu sehen, wie sie sich in der Videopipeline verhalten.

So erstellte ich eine dreistündige Videoversion, die ebenfalls 4 MB groß war, aber nachdem sie von YouTube verarbeitet worden war, zu einem Monster von 825 MB geworden war.

Das gab einen Hinweis darauf, was hinter den Kulissen passiert: YouTube transkodiert Videos immer in verschiedene Formate (4k, 1080p, 720p usw.), sorgt dabei aber auch für eine **konstante Bildrate (CFR)**, was bedeutet, dass jede Sekunde des Videomaterials genau die gleiche Anzahl von Bildern enthält. Die Alternative ist **Variable Frame Rate (VFR)**, die interessant sein kann, wenn man die Speicherung optimieren will.

Angesichts der Tatsache, dass das 3-stündige Video durch diese Konfiguration um 20.600 % aufgebläht wurde, hatte ich beschlossen, die Rate zu erhöhen. Ich hatte mich für ein Video mit einer Länge von 750 Tagen entschieden, was nach meinen Berechnungen eine Videodatei von mehreren Terabyte ergeben hätte.

Meldung an YouTube

Daraufhin beschloss ich, dies YouTube als potenzielles Sicherheitsproblem zu melden, und nicht allzu lange danach erhielt ich eine E-Mail:

> Hallo,
>
> Danke, dass Sie diesen Fehler gemeldet haben. Wir haben das Team über dieses Problem informiert; sie werden deinen Bericht überprüfen und entscheiden, ob sie eine Änderung vornehmen wollen oder nicht.

> Im Rahmen unseres Schwachstellen-Belohnungsprogramms haben wir entschieden, dass dies nicht die Voraussetzungen für eine finanzielle Belohnung erfüllt, aber wir möchten Ihren Beitrag zur Google-Sicherheit in unserer Hall of Fame würdigen:
>
> https://bughunter.withgoogle.com/rank/hm
>
> Wenn Sie auf die Seite der lobenden Erwähnungen aufgenommen werden möchten, erstellen Sie bitte hier ein Profil: https://bughunter.withgoogle.com/new_profile
>
> Ihr Ranking basiert auf der Anzahl der gültigen Meldungen.
>
> Grüße, Google Security Bot

Soweit ich weiß, war die Auswirkung eher gering, da die Transcoder so eingestellt sind, dass sie die Datei schließlich aufgeben, wenn sie zu viele Ressourcen beansprucht. Sie gaben nicht bekannt, was genau hinter den Kulissen geschah, daher kann ich nur raten. Ich glaube, dass der Dienst, der für die Erstellung von Miniaturansichten zuständig ist und die Datei dekodieren muss, irgendwann aufgibt und das Video in der Phase “in Bearbeitung” stecken bleibt.

Bonus: Naive Dauerschätzung ist eine Sünde

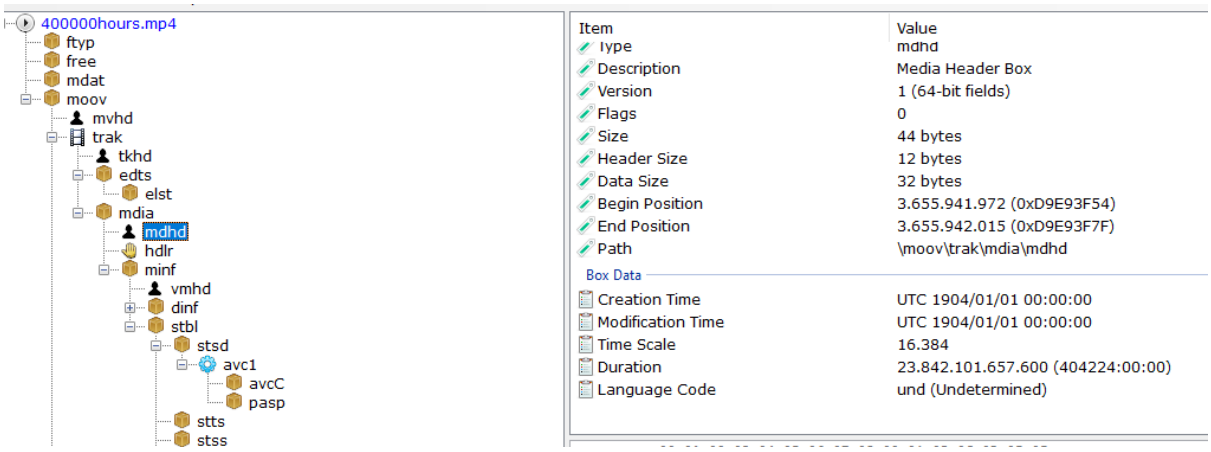
Eine MP4-Datei kann bei der Dauer in ihrem Header lügen und ein viel längeres Video enthalten als ursprünglich angegeben.

So kann zum Beispiel im Metadatenfeld eine Dauer von 15 Minuten angegeben sein, das Video ist aber 24 Stunden lang. Bei der Transkodierung wird die endgültige Dauer in der Regel neu berechnet und ist nicht von den Metadaten abhängig.

Es ist wichtig, dass Sie bei der Überprüfung der Dauer eines Videos diese aus den echten Videobildern und nicht aus den Metadaten ableiten.

Die Metadaten finden Sie unter:

> MOOV -> TRAK (Video) -> MDIA -> MDHD



Erkenntnisse?

- Die Größe einer Videodatei ist kein geeigneter Indikator für die Länge der Datei.
- CFR-Transkodierung kann zu DoS-Problemen führen, wenn Sie die Dauer nicht richtig validieren.
- Validieren Sie die Dauer eines Videos richtig, verlassen Sie sich nicht auf die Metadaten.

Florian Mathieu aka KeyboardWarrier

Der Artikel basiert auf dem Blogpost Hacking YouTube With MP4.

Bleiben Sie erfolgreich am Puls der Zeit – mit Trainings von developer media

Remote oder Inhouse

Hands-on Workshops zu Webthemen

ASP.NET Blazor – SPA-Anwendungen mit C# und .NET (3 Tage)

Advanced JavaScript (4 Tage)

Next Level TypeScript (2 Tage)

Clean-Code-Entwicklung für JavaScript- und TypeScript-Entwickler (3 Tage)

Angular im Enterprise-Projekt meistern: Redux & NGRX (2 Tage)

Moderne React Patterns (3 Tage)

Einführung in Vue.js (2 Tage)

Progressive Web App Bootcamp (3 Tage)

Einführung in GraphQL (2 Tage)

3D im Browser: Einstieg in WebGL (2 Tage)

Weitere Informationen auf developer-media.de/trainings
Preise auf Anfrage!

Ihre Ansprechpartnerin: Susanne Herl • +49 (731)880058-835 • susanne.herl@developer-media.de