»Die Zukunft ist schon hier – sie ist nur nicht besonders gleichmäßig verteilt.«

– William Gibson

Vorwort des Herausgebers der englischen Ausgabe

Mein Enkel lernt jetzt, zu programmieren.

Ja, Sie haben richtig gelesen. Mein 18-jähriger Enkel lernt, Computer zu programmieren. Wer unterrichtet ihn? Seine Tante, meine jüngste Tochter, die 1986 geboren wurde und sich vor 16 Monaten entschlossen hat, doch nicht Chemikerin, sondern Programmiererin zu werden. Und für wen arbeiten die beiden? Mein ältester Sohn ist gerade im Begriff, zusammen mit meinem jüngsten Sohn ein zweites Software-Beratungsunternehmen zu gründen.

Ja, in unsere Familie spielt Software eine große Rolle. Und ja, ich programmiere schon sehr, sehr lange.

Wie dem auch sei, meine Tochter hat mich gebeten, dass ich eine Stunde mit meinem Enkel verbringen möge und ihm die Grundlagen und Anfänge der Programmierung von Computern erläutere. Wir setzten uns also zusammen, und ich erklärte ihm, was Computer eigentlich sind, wie alles begann, wie die ersten Computer aussahen und ... na ja, Sie wissen schon.

Als die Lektion sich dem Ende näherte, programmierte ich einen Algorithmus zum Multiplizieren zweier binärer Ganzzahlen in der PDP-8-Assembler-Sprache. Falls Ihnen das nichts sagt: Bei der PDP-8 gab es keine Multiplikationsanweisung. Man musste selbst einen Algorithmus schreiben, um Zahlen miteinander zu multiplizieren. Tatsächlich gab es noch nicht einmal eine Subtraktionsanweisung. Man musste das Zweierkomplement verwenden und eine scheinbar negative Zahl addieren.

Nachdem ich das Codebeispiel fertiggestellt hatte, wurde mir klar, dass ich meinen Enkel wohl zu Tode erschreckt hatte. Als ich 18 Jahre alt war, haben mich solche abgefahrenen Details fasziniert, aber für einen 18-Jährigen, dessen Tante versucht, ihm beizubringen, wie man einfache Clojure-Programme schreibt (Clojure ist ein moderner Lisp-Dialekt), ist das vielleicht nicht so interessant.

Jedenfalls musste ich wieder daran denken, wie schwierig es tatsächlich ist, zu programmieren. Und es ist schwierig, wirklich schwierig. Vielleicht ist es das Schwierigste, was Menschen jemals versucht haben.

Ich will damit nicht sagen, dass es schwierig ist, den Code zu schreiben, um ein paar Primzahlen oder eine Fibonacci-Folge zu berechnen oder ein einfaches Bubble-Sort zu programmieren. Das ist nicht allzu schwierig. Aber ein System für die Luftverkehrskontrolle? Ein System zur Reisegepäckverfolgung? Eine Warenbestandsliste? Angry Birds? Das ist schwierig. Wirklich richtig schwierig.

Ich kenne Mark Seemann nun schon seit einigen Jahren. Ich kann mich aber nicht daran erinnern, ihn jemals getroffen zu haben. Gut möglich, dass wir niemals im selben Raum gewesen sind. Aber wir beide stehen durchaus durch Newsgroups und soziale Medien in regelmäßigem Kontakt. Er gehört zu den Leuten, denen ich am liebsten widerspreche.

Wir beide sind uns bei den unterschiedlichsten Dingen uneinig. Wir haben zur statischen bzw. dynamischen Typisierung, zu Betriebssystemen und Plattformen und zu Programmiersprachen verschiedene Meinungen. Wir sind uns bei vielen intellektuell herausfordernden Dingen nicht einig. Aber wenn man Mark widerspricht, muss man sehr vorsichtig sein, denn die Logik seiner Argumente ist bestechend.

Als ich dieses Buch in Händen hielt, stellte ich mir vor, wie viel Vergnügen ich dabei haben würde, es durchzulesen und anderer Ansicht zu sein. Und genauso war es auch. Ich war bei einigen Punkten anderer Ansicht und hatte meinen Spaß dabei, einen Weg zu finden, dass meine Logik die seine aussticht. Ich denke, in ein oder zwei Fällen ist mir das vielleicht sogar gelungen.

Aber das ist nicht der Punkt. Das Entscheidende ist, dass Softwareentwicklung schwierig ist und in den letzten sieben Jahrzehnten viel Zeit mit dem Versuch verbracht wurde, sie etwas einfacher zu machen. In diesem Buch hat Mark die besten Ideen aus diesen sieben Jahrzehnten zusammengetragen.

Darüber hinaus hat er sie in Form von Heuristiken und Verfahren organisiert und sie in der Reihenfolge sortiert, in der man sie anwenden würde. Die Heuristiken und Verfahren bauen aufeinander auf und helfen Ihnen so, bei der Entwicklung eines Softwareprojekts schrittweise voranzukommen.

Tatsächlich entwickelt Mark im Verlauf des Buchs ein Softwareprojekt und erklärt bei jeder Stufe, wie die Heuristiken und Verfahren funktionieren, von der Sie in dieser Stufe profitieren.

Mark verwendet C# (hier hätte ich mich anders entschieden), aber das ist nicht relevant. Der Code ist einfach, und die Heuristiken und Verfahren lassen sich auch auf irgendeine andere Sprache anwenden, die Sie vielleicht verwenden.

Er behandelt Themen wie Checklisten, TDD, Trennung von Befehlen und Anfragen, Git, zyklomatische Komplexität, referenzielle Transparenz, Vertical Slicing, die Überarbeitung von veraltetem Code und Outside-in-Entwicklung, um nur einige zu nennen.

Zudem finden sich buchstäblich überall im Buch verstreut kleine Schätze. Sie lesen so vor sich hin, und plötzlich schreibt er etwas wie »Wenn Sie sich vorstellen, dass Sie den Code um 90 Grad drehen, sollte der Umfang in etwa dem Umfang des dazugehörigen *Act*-Abschnitts entsprechen« oder »Das Ziel ist nicht, Code möglichst schnell zu schreiben. Das Ziel ist nachhaltige Software« oder »Übergeben Sie das Datenbankschema dem Git-Repository«.

Einige dieser kleinen Schätze sind tiefgründig, einige nur beiläufige Erwähnungen und wiederum andere sind Spekulationen, aber alle sind Beispiele für die weitreichenden Erkenntnisse, die Mark im Laufe der Jahre gewonnen hat.

Lesen Sie dieses Buch. Lesen Sie es sorgfältig. Durchdenken Sie Marks bestechende Logik. Verinnerlichen Sie die Heuristiken und Verfahren. Halten Sie inne, und machen Sie sich über die aufschlussreichen kleinen Schätze Gedanken, wenn sie auftauchen. Und wenn die Zeit dafür gekommen ist, dass Sie Ihren Enkeln etwas beibringen möchten, dann werden Sie sie vielleicht nicht zu Tode erschrecken.

— Robert C. Martin

Einleitung

In der zweiten Hälfte der 2000er-Jahre habe ich angefangen, fachliche Gutachten für einen Verlag zu erstellen. Nachdem ich eine Handvoll Bücher beurteilt hatte, kontaktierte mich der Herausgeber wegen eines Buchs über Dependency Injection.

Das Angebot war ein wenig merkwürdig. Wenn der Verlag mich wegen eines Buchs kontaktierte, hatte es für gewöhnlich schon einen Autor und ein Inhaltsverzeichnis. Dieses Mal war das nicht der Fall. Der Herausgeber bat mich lediglich um einen Anruf, um zu besprechen, ob das Thema des Buchs tauglich sei.

Ich dachte ein paar Tage darüber nach und fand das Thema anregend. Andererseits konnte ich nicht erkennen, warum dafür ein ganzes Buch erforderlich sein sollte. Schließlich war das Wissen schon vorhanden: in Blogbeiträgen, in der Dokumentation von Bibliotheken, in Fachmagazinartikeln, und es gab sogar einige Bücher, die das Thema streiften.

Als ich darüber nachdachte, fiel mir auf, dass die Informationen zwar vorhanden waren, sie waren aber weit verstreut, und es wurde eine uneinheitliche und teilweise widersprüchliche Terminologie verwendet. Es wäre durchaus sinnvoll, dieses Wissen zusammenzutragen und in einer einheitlichen Sprache zu präsentieren.

Zwei Jahre später war ich der stolze Autor eines veröffentlichten Buchs.

Nachdem ein paar Jahre vergangen waren, begann ich darüber nachzudenken, ein weiteres Buch zu verfassen. Nicht dieses, sondern ein Buch über ein anderes Thema. Dann hatte ich eine dritte Idee und eine vierte, aber nicht die Idee zu diesem Buch.

Ein Jahrzehnt verging, und mir wurde klar, dass ich, wenn ich Teams anleitete, besseren Code zu schreiben, Verfahren vorschlug, die ich von anderen gelernt hatte, die klüger waren als ich. Und wieder fiel mir auf, dass der größte Teil dieses Wissens schon vorhanden, aber weit verstreut war, und nur wenige hatten einen Zusammenhang hergestellt, um eine einheitliche Beschreibung zu geben, wie man Software entwickelt.

Aufgrund meiner Erfahrung mit dem ersten Buch wusste ich, dass es durchaus sinnvoll ist, die verschiedenen Informationen zusammenzutragen und sie auf einheitliche Weise darzustellen. Dieses Buch ist mein Versuch, genau das zu tun.

Wer dieses Buch lesen sollte

Dieses Buch richtet sich an Programmierer, die bereits einige Jahre Berufserfahrung haben. Ich gehe davon aus, dass Leser einige schlechte Softwareentwicklungsprozesse erleben mussten und Erfahrung mit nicht wartbarem Code haben. Zudem erwarte ich, dass die Leser das verbessern wollen.

Zur Zielgruppe gehören vor allem Entwickler, die in Unternehmen tätig sind, insbesondere Backend-Entwickler. Ich war in meiner Laufbahn meistens in diesem Bereich tätig, daher spiegelt das nur meine eigene Kompetenz wider. Wenn Sie Frontend-Entwickler, Spieleprogrammierer oder Programmierer von Entwicklungstools oder etwas ganz anderes sind, gehe ich davon aus, dass die Lektüre dieses Buchs Ihnen dennoch eine Menge bringen wird.

Sie sollten damit vertraut sein, den Code einer kompilierten objektorientierten Sprache aus der C-Familie zu lesen. Ich war in meiner Laufbahn zwar die meiste Zeit C#-Programmierer, habe aber auch eine Menge aus Büchern mit Codebeispielen in C++ oder Java gelernt.[1] Bei diesem Buch sind die Rollen vertauscht. Die Codebeispiele sind zwar in C# geschrieben, aber ich hoffe, dass Entwickler, die Java, TypeScript oder C++ verwenden, sie ebenfalls nützlich finden.

Voraussetzungen

Dieses Buch ist nicht für Anfänger gedacht. Es wird zwar erläutert, wie man Quellcode organisiert und strukturiert, aber die meisten Grundlagen kommen nicht zur Sprache. Ich gehe davon aus, dass Sie bereits wissen, warum Einrückungen nützlich und lange Funktionen problematisch sind, dass globale Variablen schlecht sind und so weiter.

Ein Hinweis für Softwarearchitekten

Der Begriff »Architekt« hat für verschiedene Menschen unterschiedliche Bedeutung, sogar wenn der Kontext auf Softwareentwicklung eingeschränkt ist. Manche Architekten konzentrieren sich auf das Gesamtbild; sie helfen einer ganzen Organisation dabei, dass ihre Anstrengungen erfolgreich sind. Andere Architekten befassen sich intensiv mit dem Code und sind vor allem an der Nachhaltigkeit einer bestimmten Codebasis interessiert.

Sofern man mich als Softwarearchitekt bezeichnen kann, gehöre ich zu Letzteren. Meine Kompetenz besteht darin, den Quellcode so zu organisieren, dass langfristige geschäftliche Ziele erreicht werden. Ich schreibe über das, was ich weiß. Sofern dieses Buch für Architekten nützlich ist, dann für Architekten der zweiten Gruppe.

1 Sehen Sie sich die Bibliografie an, wenn Sie neugierig sind, welche Bücher ich meine.

Zu Themen wie *Architecture Tradeoff Analysis Method* (ATAM), *Failure Mode and Effects Analysis* (FMEA), Diensterkennung und so weiter werden Sie nichts lesen, denn sie gehen über den Rahmen dieses Buchs hinaus.

Aufbau des Buchs

In diesem Buch geht es um Methodologien, deshalb orientiert es sich an einem Codebeispiel, das im Buch durchgängig verwendet wird. Ich habe mich dafür entschieden, damit die Lektüre überzeugender ist als das Lesen eines typischen Musterkatalogs. Diese Entscheidung hat zur Folge, dass ich Verfahren und Heuristiken dann vorstelle, wenn sie zu den geschilderten Themen passen. In dieser Reihenfolge stelle ich die Verfahren normalerweise auch vor, wenn ich Teams berate.

Die Schilderung beruht auf einer Beispiel-Codebasis, die ein Reservierungssystem für ein Restaurant implementiert. Der Quellcode für dieses Beispiel ist unter `www.mitp.de/0514` verfügbar.

Wenn Sie das Buch als Nachschlagewerk verwenden möchten, finden Sie im Anhang eine Liste aller Verfahren mit der Angabe, wo Sie im Buch weitere Informationen finden.

Über den Codestil

Der Beispielcode ist in C# geschrieben, einer Sprache, die sich in den letzten Jahren rasant weiterentwickelt hat. Die Sprache übernimmt immer mehr Syntaxelemente aus der funktionalen Programmierung. Beispielsweise wurden *unveränderliche Record-Datentypen* veröffentlicht, während ich dieses Buch geschrieben habe. Ich habe mich dafür entschieden, einige der neuen Sprachfeatures zu ignorieren.

Früher einmal hatte Java-Code große Ähnlichkeit mit C#-Code. Moderner C#-Code hat kaum noch Ähnlichkeit mit Java.

Ich möchte, dass der Code für so viele Leser wie möglich verständlich ist. Ebenso wie ich viel aus Büchern mit Java-Beispielen gelernt habe, möchte ich, dass die Leser in der Lage sind, das Buch zu verwenden, ohne die neueste C#-Syntax zu kennen. Deshalb versuche ich, nur eine Teilmenge von C# zu verwenden, die auch für Programmierer anderer Sprachen verständlich sein sollte.

Das ändert nichts an den im Buch vorgestellten Konzepten. Ja, in manchen Fällen sind kompaktere C#-spezifische Alternativen möglich, aber das bedeutet nur, dass zusätzliche Verbesserungen möglich sind.

var verwenden oder var nicht verwenden

Das Schlüsselwort `var` wurde 2007 in C# eingeführt. Es ermöglicht, eine Variable zu deklarieren, ohne ausdrücklich ihren Typ anzugeben. Stattdessen ermittelt der Compiler den Typ anhand des Kontexts. Der Deutlichkeit halber: Mit `var` deklarierte Variablen sind ebenso statisch typisiert wie Variablen, die explizit mit einem Typ deklariert werden.

Die Verwendung dieses Schlüsselworts war lange umstritten, aber die meisten verwenden es *inzwischen*. Ich auch, aber gelegentlich stoße ich dabei auf Widerstand. Beruflich verwende ich zwar `var`, aber den Code für ein Buch zu schreiben, ist ein etwas anderer Kontext. Unter normalen Umständen steht eine IDE zur Verfügung. Eine moderne Entwicklungsumgebung kann Ihnen schnell den Typ einer implizit typisierten Variable anzeigen, aber ein Buch kann das nicht.

Aus diesem Grund verwende ich gelegentlich explizit typisierte Variablen. Der meiste Beispielcode verwendet das Schlüsselwort `var`, weil der Code dadurch kürzer wird und die Zeilenbreite in einem gedruckten Buch begrenzt ist. In einigen wenigen Fällen habe ich mich allerdings bewusst dafür entschieden, den Typ einer Variablen explizit zu deklarieren, in der Hoffnung, dass der Code besser verständlich ist, wenn man ihn in einem Buch liest.

Code-Listings

Die meisten Code-Listings sind derselben Codebasis entnommen. Dabei handelt es sich um ein Git-Repository und die Codebeispiele entstammen verschiedenen Stufen der Entwicklung. Bei den Code-Listings ist der relative Pfad zur entsprechenden Datei angegeben. Ein Teil davon ist eine Git-Commit-ID.

Bei Listing 2.1 ist beispielsweise der relative Pfad *Restaurant/f729ed9/Restaurant.RestApi/Program.cs* angegeben. Das bedeutet, dass das Beispiel der Commit-ID `f729ed9` entstammt, und die Datei ist `Restaurant.RestApi/Program.cs`. Mit anderen Worten: Wenn Sie diese Version der Datei anzeigen möchten, müssen Sie dieses Commit auschecken:

```
$ git checkout f729ed9
```

Wenn Sie das erledigt haben, können Sie die Datei `Restaurant.RestApi/Program.cs` in ihrem vollständigen ausführbaren Kontext anzeigen.

Hinweis zur Bibliografie

In der Bibliografie finden Sie verschiedene Ressourcen: Bücher, Blogbeiträge und Videoaufzeichnungen. Viele der Quellen sind online, daher habe ich natürlich

URLs angegeben. Ich habe mich bemüht, vor allem Ressourcen anzugeben, bei denen ich Grund zu der Annahme habe, dass sie im Internet dauerhaft verfügbar sind.

Aber die Dinge ändern sich. Wenn Sie dieses Buch in der Zukunft lesen und eine URL ungültig geworden ist, können Sie versuchen, auf einen Internetarchivdienst zurückzugreifen. Derzeit ist `https://archive.org` der beste Kandidat, aber diese Website könnte es in der Zukunft auch nicht mehr geben.

Eigene Zitate

Neben den anderen Ressourcen enthält die Bibliografie auch eine Liste meiner eigenen Arbeiten. Mir ist klar, dass eigene Zitate an sich keine stichhaltigen Argumente sind.

Es soll aber kein Taschenspielertrick sein, wenn ich auf meine eigene Arbeit verweise. Ich stelle diese Ressourcen vielmehr für Leser zur Verfügung, die möglicherweise an weiteren Details interessiert sind. Wenn ich mich selbst zitiere, mache ich das, weil Sie in den Ressourcen, auf die ich verweise, möglicherweise ausführlichere Argumente oder detailliertere Codebeispiele finden können.

Danksagung

Ich danke meiner Frau Cecilie für ihre Liebe und Unterstützung in all den Jahren, die wir zusammen sind, und meinen Kindern Linea und Jarl, dass sie uns keinen Kummer bereiten.

Neben meiner Familie gilt mein Dank meinem langjährigen guten Freund Karsten Strøbæk , der nicht nur seit 25 Jahren mein Dasein toleriert, sondern auch der erste Leser dieses Buchs war. Zudem hat er mit verschiedenen Tipps und Tricks für LaTeX geholfen und dem Stichwortverzeichnis der englischen Ausgabe viele Einträge hinzugefügt.

Ich möchte auch Adam Tornhill für sein Feedback zum Abschnitt über seine Arbeit danken.

Außerdem bin ich Dan North dafür zu Dank verpflichtet, dass er den Ausdruck *Code That Fits in your Head* in mein Unterbewusstsein »eingepflanzt« hat. Das war wohl schon 2011 [72].

Downloads zum Buch

Den Programmcode der im Buch verwendeten Codebeispiele finden Sie unter `www.mitp.de/0514` im Reiter »Downloads« unterhalb des Buchcovers.