Modernes Software Engineering

Bessere Software schneller und effektiver entwickeln

DAS VORWORT

Vorwort

Ich habe an der Universität Informatik studiert und natürlich mehrere Module absolviert, die »Software Engineering« oder Variationen davon im Titel führten.

Als ich mein Studium begann, war das Programmieren nicht neu für mich und ich hatte bereits ein voll funktionsfähiges Inventarsystem für die Job- und Karrierebliothek meiner High School implementiert. Ich erinnere mich, dass mich der Begriff »Software Engineering« hochgradig verwirrte. Das alles schien, als ob es nur dazu diente, dem eigentlichen Schreiben von Code und der Auslieferung einer Anwendung in die Quere zu kommen.

Als ich in den ersten Jahren dieses Jahrhunderts meinen Abschluss machte, arbeitete ich in der IT-Abteilung eines großen Automobilkonzerns. Wie nicht anders zu erwarten, stand dort die Software-Entwicklung ganz weit oben auf der Liste. Hier habe ich mein erstes (aber sicher nicht mein letztes!) Gantt-Diagramm gesehen und die Wasserfall-Entwicklung miterlebt. Das heißt, ich sah, wie Software-Teams viel Zeit und Mühe in die Anforderungserhebung und die Entwurfsphase steckten, und viel weniger Zeit in die Implementierung (das Programmieren), was natürlich auch für die Testzeit galt, und für das Testen ... nun ja, dafür war nicht mehr viel Zeit.

Es schien, als ob das, was uns als »Software Engineering« verkauft wurde, der Entwicklung von qualitativ hochwertigen Anwendungen, die für unsere Kunden nützlich sind, im Weg stand.

Wie viele Entwickler dachte ich, dass es einen besseren Weg geben muss.

Ich habe etwas über Extreme Programming und Scrum gelesen. Ich wollte in einem agilen Team arbeiten und habe ein paarmal den Job gewechselt, um eines zu finden. Viele sagten, sie seien agil, aber oft lief es darauf hinaus, dass man Anforderungen oder Aufgaben auf Karteikarten schrieb, sie an die Wand klebte, eine Woche als *Sprint* bezeichnete und dann vom Entwicklungsteam verlangte, in jedem Sprint »x« Karten zu liefern, um eine willkürliche Frist einzuhalten. Die Abschaffung des traditionellen »Software Engineering«-Ansatzes schien auch nicht zu funktionieren.

Zehn Jahre nach Beginn meiner Karriere als Entwickler bewarb ich mich bei einer Finanzhandelsplattform in London. Der Leiter der Software-Abteilung erzählte mir, dass sie Extreme Programming anwendeten, einschließlich TDD und Pair

Programming. Er sagte, dass sie etwas praktizierten, das sie *Continuous Delivery* nannten, was etwa der Continuous Integration entsprach, sich aber bis in die Produktion erstreckte.

Ich hatte für große Investmentbanken gearbeitet, bei denen die Bereitstellung mindestens drei Stunden dauerte und mithilfe eines 12-seitigen Dokuments mit manuellen Schritten und Befehlen, die man eingeben musste, »automatisiert« wurde. Continuous Delivery schien eine schöne Idee zu sein, war aber sicher nicht möglich.

Der Leiter der Software-Abteilung war Dave Farley, und er war gerade dabei, sein Buch über *Continuous Delivery* zu schreiben, als ich in das Unternehmen kam.

Ich habe dort vier Jahre lang mit ihm zusammengearbeitet, Jahre, die mein Leben und meine Karriere verändert haben. Wir haben tatsächlich Pair Programming, TDD und Continuous Delivery angewendet. Außerdem lernte ich etwas über verhaltensgesteuerte Entwicklung (Behavior-driven Development), automatisierte Akzeptanztests, domänengetriebenes Design (Domain-driven Design), Trennung von Zuständigkeiten, Anti-Corruption-Layer, Mechanical Sympathy und Ebenen der Indirektion.

Ich lernte, wie man in Java hochperformante Anwendungen mit geringer Latenzzeit erstellt. Endlich verstand ich, was die O-Notation wirklich bedeutet und wie sie in der realen Welt der Programmierung eingesetzt werden kann. Kurz gesagt, all das, was ich an der Universität gelernt und in Büchern gelesen hatte, wurde tatsächlich angewendet.

Es wurde so eingesetzt, dass es Sinn hatte, funktionierte und eine extrem hochwertige, leistungsstarke Anwendung lieferte, die etwas bot, was es vorher nicht gab. Mehr noch, wir waren glücklich in unserem Job und zufrieden als Entwickler. Wir machten keine Überstunden, wir hatten keine Engpässe kurz vor der Veröffentlichung, der Code wurde in diesen Jahren nicht unübersichtlicher und unwartbarer und wir lieferten durchgehend und regelmäßig neue Funktionen und »Geschäftswert«.

Wie haben wir das erreicht? Indem wir den Verfahren folgten, die Dave in diesem Buch beschreibt. Es war nicht so formalisiert, und Dave hat eindeutig seine Erfahrungen aus vielen anderen Unternehmen eingebracht, um die spezifischen Konzepte herauszuarbeiten, die für ein breiteres Spektrum von Teams und Geschäftsbereichen anwendbar sind.

Was für zwei oder drei Teams an einer hochperformanten Finanzhandelsplattform funktioniert, wird nicht auf genau dieselbe Weise für ein großes Unternehmensprojekt in einem Produktionsbetrieb oder für ein schnelllebiges Start-up-Unternehmen gelten.

In meiner derzeitigen Position als Developer Advocate spreche ich mit Hunderten von Entwicklern aus den unterschiedlichsten Unternehmen und Geschäftsbereichen und höre mir ihre Probleme (von denen viele meinen eigenen Erfahrungen vor 20 Jahren nicht unähnlich sind) und ihre Erfolgsgeschichten an. Die Konzepte, die Dave in diesem Buch behandelt, sind allgemein genug, um in all diesen Umgebungen zu funktionieren, und spezifisch genug, um in der Anwendung hilfreich zu sein.

Komischerweise begann ich mich mit der Bezeichnung *Software Engineer* unwohl zu fühlen, nachdem ich Daves Team verlassen hatte. Ich dachte nicht, dass das, was wir als Entwickler tun, Engineering ist; ich dachte nicht, dass es das Engineering war, das das Team erfolgreich gemacht hatte. Ich dachte, dass Engineering eine zu strukturierte Disziplin für das ist, was wir tun, wenn wir komplexe Systeme entwickeln. Mir gefällt die Idee, dass es sich dabei um ein »Handwerk« handelt, da dies sowohl Kreativität als auch Produktivität beinhaltet, auch wenn es die Teamarbeit nicht ausreichend betont, die für die Arbeit an Software-Problemen in großem Maßstab erforderlich ist. Die Lektüre dieses Buchs hat meine Meinung geändert.

Dave erklärt deutlich, warum wir falsche Vorstellungen davon haben, was »echtes« Engineering ist. Er zeigt, dass Engineering eine wissenschaftsbasierte Disziplin ist, die aber nicht starr sein muss. Er erklärt Schritt für Schritt, wie wissenschaftliche Prinzipien und Engineering-Techniken auf die Software-Entwicklung angewendet werden können und warum die produktionsbezogenen Techniken, von denen wir dachten, sie seien ingenieurwissenschaftlich, für die Software-Entwicklung nicht geeignet sind.

Was ich an Daves Buch so toll finde, ist, dass er Konzepte nimmt, die abstrakt und schwierig auf echten Code anzuwenden scheinen, mit dem wir in unserem Job arbeiten müssen, und zeigt, wie wir sie als Tools einsetzen, um unsere konkreten Probleme anzugehen.

Das Buch begegnet der chaotischen Realität der Code-Entwicklung – oder sollte ich sagen, der Software-Entwicklung –, mit offenen Armen: Es gibt nicht die eine, einzig richtige Lösung. Die Dinge werden sich ändern. Was zu einem bestimmten Zeitpunkt richtig war, ist manchmal schon kurze Zeit später völlig falsch.

Die erste Hälfte des Buchs bietet praktische Lösungen, um in dieser Realität nicht nur zu überleben, sondern zu gedeihen. In der zweiten Hälfte werden Themen aufgegriffen, die von manchen als abstrakt oder akademisch angesehen werden könnten, und es wird gezeigt, wie man sie anwendet, um besseren (z.B. robusteren oder besser wartbaren oder in anderen Merkmalen »besseren«) Code zu designen.

Dabei bedeutet Design nicht unbedingt seitenlange Designdokumente oder UML-Diagramme, sondern kann so einfach sein wie »über den Code nachzudenken,

bevor oder während man ihn schreibt«. (Als ich mit Dave zusammen Pair Programming betrieben habe, ist mir unter anderem aufgefallen, wie wenig Zeit er mit dem eigentlichen Schreiben des Codes verbringt. Es stellte sich heraus, dass das Nachdenken über das, was wir schreiben, bevor wir es schreiben, uns eine Menge Zeit und Mühe ersparen kann).

Dave versucht nicht, Widersprüche bei der gemeinsamen Anwendung der Verfahren zu vermeiden oder die mögliche Verwirrung, die durch ein einzelnes Verfahren verursacht werden kann, aufzuklären. Weil er sich die Zeit nimmt, über die Kompromisse und die häufig auftretenden Unklarheiten zu sprechen, habe ich stattdessen zum ersten Mal verstanden, dass es genau die Balance und die Spannung zwischen diesen Dingen ist, die »bessere« Systeme schafft. Es geht darum zu verstehen, dass diese Dinge Richtlinien sind, ihre Vor- und Nachteile zu sehen und sie als Linsen zu verstehen, durch die man den Code/das Design/die Architektur betrachten kann, und gelegentlich als Stellschrauben, an denen man drehen kann, anstatt als binäre, schwarz-weiße, Richtig-oder-falsch-Regeln.

Durch die Lektüre dieses Buchs habe ich verstanden, warum wir in der Zeit, in der ich mit Dave gearbeitet habe, als »Software Engineers« so erfolgreich und zufrieden waren. Ich hoffe, dass Sie durch die Lektüre dieses Buchs von Daves Erfahrung und Ratschlägen profitieren können, ohne einen Dave Farley für Ihr Team einstellen zu müssen.

Viel Spaß beim Entwickeln!

– Trisha Gee, Developer Advocate und Java Champion