Vorwort

Seit mehr als 80 Jahren schreiben Menschen Programme für elektronische Computer, aber es gibt erstaunlich wenige Diskussionen darüber, wie diese Programme zu designen sind oder wie gute Programme aussehen sollten. Es wurde viel über die Prozesse bei der Softwareentwicklung gesprochen – zum Beispiel über agile Entwicklung oder Entwicklungswerkzeuge wie Debugger, Systeme zur Versionsverwaltung und Tools zur Testabdeckung. Auch gab es umfassende Analysen zu Programmiertechniken wie zum Beispiel objektorientierter oder funktionaler Programmierung, aber auch zu Design Patterns und Algorithmen. Alle diese Diskussionen sind sehr wertvoll, aber das zentrale Problem des Softwaredesigns wird größtenteils nicht angegangen. Der Grundsatzartikel »On the Criteria to be used in Decomposing Systems into Modules« von David Parnas erschien 1971, aber der Stand der Technik hat sich im Softwaredesign in den folgenden 45 Jahren nicht sonderlich weiterentwickelt.

Das grundlegendste Problem in der Informatik ist die *Problemzerlegung*: Wie teilen Sie ein komplexes Problem so in Teile auf, dass diese unabhängig voneinander gelöst werden können? Die Problemzerlegung ist die zentrale Designaufgabe, der man sich beim Programmieren tagtäglich gegenübersieht – und trotzdem habe ich abgesehen von den hier beschriebenen Arbeiten noch keinen einzigen Kurs an einer Universität finden können, in dem die Problemzerlegung ein zentrales Thema wäre. Wir unterrichten `for`-Schleifen und objektorientierte Programmierung, aber kein Softwaredesign.

Zudem schwanken Qualität und Produktivität derjenigen, die programmieren, sehr stark, wir haben uns aber bisher keine große Mühe damit gegeben, herauszufinden, was die Besten so viel besser macht, oder diese Fähigkeiten in unseren Kursen zu unterrichten. Ich habe mit vielen Leuten gesprochen, deren Programmierkünste ich hoch einschätze, aber die meisten hatten Probleme damit, die spezifischen Techniken zu benennen, die ihnen ihren Vorteil verschaffen. Viele gehen davon aus, dass die Fähigkeit zu gutem Softwaredesign »angeboren« sei und nicht gelehrt werden könne. Aber es gibt eine recht gute wissenschaftliche Evidenz dafür, dass außergewöhnliche Leistungen in vielen Bereichen mehr mit hochwertiger Praxis denn mit angeborenen Fähigkeiten zu tun haben (siehe zum Beispiel das Buch *Talent wird überschätzt* von Geoff Colvin).

Viele Jahre haben mich diese Themen verwirrt und frustriert. Ich fragte mich, ob Softwaredesign gelehrt werden kann, und stellte die Hypothese auf, dass die Fähigkeit zum Design gerade den Unterschied zwischen großartigem und normalem Programmieren ausmacht. Schließlich wurde mir klar, dass ich diese Fragen nur beantworten kann, indem ich Softwaredesign unterrichte. Das Ergebnis ist CS 190 an der Stanford University. In diesem Kurs präsentiere ich die Prinzipien des Softwaredesigns. Die Studierenden nutzen dann eine Reihe von Projekten, um sich mit diesen Prinzipien vertraut zu machen und sie einzuüben. Der Kurs ist ähnlich aufgebaut wie ein klassischer Universitätskurs zum Schreiben von Texten. Dabei folgen die Studierenden einem iterativen Prozess, in dem sie einen Entwurf erstellen, Feedback erhalten und den Text dann neu schreiben, um besser zu werden. In CS 190 entwickeln die Studierenden eine realistische Software von Grund auf. Dann werden umfangreiche Code Reviews durchgeführt, um Designprobleme aufzudecken, und die Studierenden überarbeiten ihre Projekte, um diese Probleme zu beheben. So können sie sehen, wie sich ihr Code durch das Anwenden von Designprinzipien verbessern lässt.

Ich habe den Kurs in Softwaredesign nun drei Mal gehalten, und dieses Buch basiert auf den Designprinzipien, die sich daraus entwickelt haben. Sie sind ziemlich allgemein gehalten und stoßen bereits an die Grenzen zur Philosophie (»Definieren Sie die Existenz von Fehlern weg«), dadurch ist es für Studierende schwierig, die Ideen in dieser Abstraktheit zu verstehen. Sie lernen am besten, indem sie Code schreiben, Fehler machen und dann sehen, wie ihre Fehler und deren notwendige Fixes mit den Prinzipien in Zusammenhang stehen.

Jetzt fragen Sie sich vielleicht, wieso ich der Meinung bin, beim Softwaredesign alle Antworten zu kennen. Nun, ehrlich gesagt: Ich kenne sie nicht. Als ich das Programmieren erlernte, gab es keine Kurse zum Softwaredesign, und es gab niemanden, der mir die Designprinzipien erklärt hat. Auch Code Reviews waren damals quasi nicht existent. Meine Ideen zum Softwaredesign entstammen meiner persönlichen Erfahrung beim Lesen und Schreiben von Code. Im Laufe meiner Karriere schrieb ich etwa 250.000 Zeilen Code in diversen Programmiersprachen. Ich arbeitete in Teams, die drei Betriebssysteme von Grund auf erstellt haben. Es entstanden eine Reihe von Datei- und Storage-Systemen, Infrastrukturtools wie Debugger, Build-Systeme und GUI-Toolkits, eine Skriptsprache und interaktive Editoren für Text, Grafiken, Präsentationen und integrierte Schaltkreise. In dieser Zeit erfuhr ich die Probleme großer Systeme aus erster Hand und experimentierte mit diversen Designtechniken. Zudem las ich ziemlich viel Code von anderen Programmiererinnen und Programmierern, sodass ich viele unterschiedliche Ansätze kennenlernen konnte – sowohl gute wie auch schlechte.

Mit dieser Erfahrung versuchte ich, aus häufig anzutreffenden Gedankengängen vermeidbare Fehler, aber auch sinnvolle Techniken zu extrahieren und abzuleiten. Dieses Buch spiegelt meine Erfahrungen wider: Jedes hier beschriebene Problem ist eines, dem ich mich persönlich gegenübersah, und jede vorgeschlagene Technik ist eine, die ich in meinem eigenen Code erfolgreich eingesetzt habe.

Ich erwarte nicht, dass dieses Buch das letzte Wort zum Thema Softwaredesign sein wird – ich bin sicher, dass es noch andere lohnenswerte Techniken gibt, die ich nicht kenne, und manche meiner Vorschläge werden sich vielleicht langfristig als nicht so gute Ideen herausstellen. Aber ich hoffe, mit diesem Buch eine Diskussion über Softwaredesign anzustoßen. Vergleichen Sie die Ideen in diesem Buch mit Ihren eigenen Erfahrungen und entscheiden Sie selbst, ob die hier beschriebenen Vorgehensweisen die Komplexität der Software tatsächlich reduzieren. Dieses Buch ist eine Meinungsäußerung, daher werden manche Leserinnen und Leser ein paar meiner Vorschläge nicht gut finden. Wenn Sie anderer Meinung sind, versuchen Sie, herauszufinden, woran das liegt. Ich bin daran interessiert, von Ihnen zu hören, was für Sie funktioniert, was nicht funktioniert und welche Ideen Sie ansonsten zum Thema Softwaredesign haben. Ich hoffe, dass die sich daraus ergebenden Gespräche unser gemeinsames Verständnis von Softwaredesign verbessern werden. In zukünftigen Auflagen dieses Buchs werde ich mit aufnehmen, was ich daraus gelernt habe.

Am besten kommunizieren Sie mit mir über das Buch, indem Sie eine E-Mail an diese Adresse schicken:

```
software-design-book@googlegroups.com
```

Interessiert bin ich an Ihrem spezifischen Feedback zum Buch – wie zum Beispiel an Verbesserungsvorschlägen oder eventuellen Fehlern –, aber auch an allgemeinen Gedanken und Erfahrungen rund um Softwaredesign. Besonders freue ich mich auf spannende Beispiele, die ich in zukünftigen Auflagen verwenden kann. Die besten Beispiele illustrieren ein wichtiges Designprinzip, während sie gleichzeitig so einfach sind, dass sie sich in ein oder zwei Absätzen erläutern lassen. Möchten Sie wissen, was andere Menschen an diese E-Mail-Adresse schicken, oder möchten Sie sich an Diskussionen beteiligen, können Sie der Google Group *software-design-book* beitreten.

Sollte diese Gruppe aus irgendwelchen Gründen in Zukunft nicht mehr existieren, suchen Sie im Web nach meiner Homepage – sie wird aktuelle Informationen zur Kommunikation zu diesem Buch enthalten. Schicken Sie bitte keine E-Mails rund um das Buch an meine private E-Mail-Adresse.

Ich empfehle Ihnen, die Vorschläge in diesem Buch mit einer gewissen Vorsicht zu genießen. Das Gesamtziel ist eine Reduktion der Komplexität – das ist wichtiger als alle anderen spezifischen Prinzipien oder Ideen, von denen Sie hier lesen werden. Wenn Sie eine Idee aus diesem Buch ausprobieren und feststellen, dass sie die Komplexität gar nicht verringert, sollten Sie sich nicht dazu genötigt fühlen, sie trotzdem einzusetzen (aber lassen Sie mich an Ihrer Erfahrung teilhaben – ich möchte mitbekommen, was funktioniert und was nicht).

Viele Menschen haben Kritik geäußert oder Vorschläge gemacht, wie man die Qualität dieses Buchs verbessern könnte. Von folgenden erhielt ich hilfreiche Kommentare zu den diversen Entwürfen: Abutalib Aghayev, Jeff Dean, Will Duquette, Sanjay Ghemawat, John Hartman, Brian Kernighan, James Koppel, Amy Ouster-

hout, Kay Ousterhout, Rob Pike, Partha Ranganathan, Daniel Rey, Keith Schwartz und Alex Snaps. Christos Kozyrakis hat die Begriffe »deep« (»tief«) und »shallow« (»flach«) für Klassen und Schnittstellen vorgeschlagen, die anstelle von »thick« und »thin« traten, die nicht so eindeutig waren. Ich bedanke mich auch bei den Studierenden in CS 190 – das Lesen ihres Codes und die Diskussionen mit ihnen haben mir dabei geholfen, meine Gedanken zum Design besser ordnen zu können.