KAPITEL 1

Einführung (es geht immer um Komplexität)

Das Schreiben von Computersoftware gehört zu den uneingeschränkt kreativen Aktivitäten in der Geschichte der Menschheit. Programmierer und Programmiererinnen sind nicht an Beschränkungen wie die Gesetze der Physik gebunden – wir können spannende virtuelle Welten erschaffen, die so in der Realität nie möglich wären. Zum Programmieren benötigt man keine großen physischen Voraussetzungen und auch keine gute Koordination wie beim Ballett oder beim Basketball. Sie brauchen nur einen kreativen Kopf und die Fähigkeit, Ihre Gedanken zu ordnen. Können Sie sich ein System vorstellen, können Sie es vermutlich auch in einem Computerprogramm implementieren.

Die größte Einschränkung beim Schreiben von Software ist also unsere Fähigkeit, das zu erschaffende System zu verstehen. Wenn sich ein Programm weiterentwickelt und mehr Features notwendig sind, wird es komplizierter, und es gibt subtile Abhängigkeiten zwischen den Komponenten. Mit der Zeit summiert sich die Komplexität auf, und es wird beim Programmieren immer schwieriger, alle relevanten Faktoren beim Anpassen des Systems im Kopf zu behalten. Das verlangsamt das Entwickeln und führt zu Fehlern, was die Entwicklung noch mehr verlangsamt und noch teurer macht. Die Komplexität steigt mit zunehmender Lebenszeit jedes Programms ganz unvermeidlich. Je umfangreicher es ist und je mehr Personen daran arbeiten, desto schwieriger wird es, sie im Griff zu behalten.

Gute Entwicklungswerkzeuge können uns dabei helfen, mit der Komplexität umzugehen, und in den letzten Jahrzehnten entstanden viele großartige Tools. Aber es gibt eine Grenze bei dem, was wir allein mit Tools erreichen können. Wollen wir das Schreiben von Software leichter machen, damit wir mit weniger Aufwand leistungsfähigere Systeme bauen können, müssen wir Wege finden, die Software einfacher zu gestalten. Die Komplexität wird trotz all unserer Bemühungen mit der Zeit wachsen, aber ein einfacheres Design erlaubt uns, größere und leistungsfähigere Systeme zu bauen, bevor uns die Komplexität erschlägt.

Es gibt im Allgemeinen zwei Ansätze, Komplexität zu bekämpfen, und auf beide werden wir in diesem Buch eingehen. Der erste ist, die Komplexität zu reduzieren, indem wir den Code einfacher und offensichtlicher machen. So können wir beispielsweise Spezialfälle ausmerzen oder Namen konsistent einsetzen.

Der zweite Ansatz besteht darin, die Komplexität zu kapseln, sodass man in einem System programmieren kann, ohne mit der gesamten Komplexität auf einmal konfrontiert zu werden. Dieser Ansatz nennt sich *modulares Design*. Dabei wird ein Softwaresystem in *Module* aufgeteilt, wie zum Beispiel in einer objektorientierten Sprache in Klassen. Die Module sind so entworfen, dass sie voneinander relativ unabhängig sind, sodass man beim Programmieren mit einem Modul arbeiten kann, ohne die Details anderer Module verstehen zu müssen.

Weil Software so formbar ist, handelt es sich beim Softwaredesign um einen fortlaufenden Prozess, der den gesamten Lebenszyklus eines Softwaresystems begleitet – das unterscheidet das Softwaredesign vom Entwurf physischer Systeme, etwa von Gebäuden, Schiffen oder Brücken. Aber Softwaredesign wurde nicht immer so betrachtet. In der Anfangszeit der Programmierung konzentrierte sich das Design zunächst im Wesentlichen auf den Beginn eines Projekts – so wie in anderen Ingenieurdisziplinen. Der extremste Ansatz nennt sich *Wasserfallmodell*, bei dem ein Projekt in getrennte Phasen unterteilt wird, wie zum Beispiel Anforderungsdefinition, Design, Programmieren, Testen und Wartung. Im Wasserfallmodell wird jede Phase erst abgeschlossen, bevor die nächste beginnen kann – in vielen Fällen sind auch unterschiedliche Personen für die verschiedenen Phasen verantwortlich. Das gesamte System wird auf einmal in der initialen Designphase entworfen. Dann wird das Design am Ende der Phase eingefroren, und die folgenden Phasen haben die Aufgaben, dieses Design mit Leben zu füllen und zu implementieren.

Leider funktioniert das Wasserfallmodell für Software nur sehr selten. Softwaresysteme sind intrinsisch viel komplexer als physische Systeme – es ist nicht möglich, das Design eines großen Softwaresystems so gut darzustellen, dass alle seine Implikationen verstanden werden, bevor man etwas baut. Im Ergebnis wird das initiale Design viele Probleme haben. Diese werden erst sichtbar, wenn die Implementierung voranschreitet. Aber das Wasserfallmodell ist nicht dazu gedacht, an diesem Punkt noch große Designveränderungen zu ermöglichen (zum Beispiel können sich die Designteams schon einem anderen Projekt zugewandt haben). Daher versucht die Entwicklung, die Probleme zu umgehen, ohne das Gesamtdesign zu verändern. Das führt zu einer Komplexitätsexplosion.

Aus diesem Grund verfolgen die meisten aktuellen Softwareprojekte einen inkrementellen Ansatz, zum Beispiel die *agile Entwicklung*, bei der sich das initiale Design auf eine kleine Untermenge der Gesamtfunktionalität konzentriert. Diese Untermenge wird entworfen, implementiert und dann evaluiert. Probleme mit dem ursprünglichen Design werden erkannt und behoben, dann werden ein paar weitere Features entworfen, implementiert und evaluiert. Jede Iteration deckt Probleme des bestehenden Designs auf, die man behebt, bevor die nächsten Features entworfen werden. Indem wir das Design so immer weiter ausbauen, können wir Probleme mit dem initialen Design beheben, während das System noch klein ist – später hinzukommende Features profitieren von der Erfahrung, die während der Implementierung früherer Features gewonnen wurde, sodass weniger Probleme entstehen werden.

Der inkrementelle Ansatz funktioniert für Software, weil diese so flexibel ist, dass auch während der Implementierung noch signifikante Designänderungen vorgenommen werden können. Für physische Systeme sind größere Designänderungen hingegen viel schwieriger – so wäre es beispielsweise nicht praktikabel, die Anzahl der Pfeiler, die eine Brücke stützen, während der Bauphase anzupassen.

Inkrementelle Entwicklung bedeutet, dass Softwaredesign niemals fertig ist. Sie findet über den gesamten Lebenszyklus eines Systems hinweg statt, beim Entwickeln sollte man daher stets über Designthemen nachdenken. Inkrementelle Entwicklung heißt auch kontinuierliches Redesign. Das erste Design eines Systems oder einer Komponente ist so gut wie nie auch schon das beste – Erfahrung führt unvermeidlich zu besseren Wegen beim Umsetzen von Dingen. Als Softwareentwicklerin oder -entwickler sollten Sie immer nach Gelegenheiten Ausschau halten, das Design des Systems, an dem Sie arbeiten, zu verbessern, und Sie sollten einen Teil Ihrer Zeit für Verbesserungen am Design einplanen.

Wenn man während des Entwickelns von Software fortlaufend über Designthemen nachdenken soll und das Reduzieren der Komplexität das wichtigste Element des Softwaredesigns ist, sollte man beim Entwickeln immer auch über Komplexität nachdenken. Dieses Buch dreht sich darum, wie Sie das Thema Komplexität nutzen können, um sich beim Design von Software über deren gesamten Lebenszyklus leiten zu lassen.

Dieses Buch hat zwei größere Ziele. Das erste ist, die Natur der Komplexität von Software zu beschreiben: Was bedeutet »Komplexität«, warum ist sie wichtig, und wie können Sie erkennen, ob ein Programm unnötig komplex ist? Das zweite Ziel des Buchs ist herausfordernder: Ich präsentiere Techniken, die Sie während des Softwareentwicklungsprozesses nutzen können, um die Komplexität zu minimieren. Leider gibt es kein einfaches Rezept, das immer ein tolles Softwaredesign garantiert. Stattdessen werde ich eine Reihe von High-Level-Konzepten vorstellen, die sich philosophischen Aussagen bereits stark annähern, wie zum Beispiel: »Klassen sollten tief sein.« oder: »Definieren Sie die Existenz von Fehlern weg.« Diese Konzepte werden nicht sofort das beste Design aufzeigen, aber Sie können sie einsetzen, um Designalternativen zu vergleichen und um hierbei Ihren Gestaltungsspielraum zu erkunden.

Wie Sie dieses Buch einsetzen

Viele der hier beschriebenen Designprinzipien sind mehr oder weniger abstrakt, daher werden sie sich nur schwer würdigen lassen, wenn man sich keinen echten Code dazu anschaut. Es war schwierig, Beispiele zu finden, die klein genug waren, um sie in dieses Buch aufnehmen zu können, aber gleichzeitig groß genug, um Probleme an realen Systemen zu demonstrieren (wenn Sie gute Beispiele kennen, schicken Sie sie mir bitte zu). Daher kann es sein, dass dieses Buch allein nicht ausreicht, um zu lernen, wie Sie die Prinzipien anwenden.

Am besten nutzen Sie das Buch zusammen mit Code Reviews. Lesen Sie den Code anderer und denken Sie darüber nach, ob er den hier vorgestellten Konzepten entspricht und wie das mit der Komplexität des Codes zusammenhängt. Es ist einfacher, Designprobleme im Code von anderen Entwicklerinnen und Entwicklern zu finden als in eigenem Code. Sie können die hier beschriebenen Warnzeichen nutzen, um Probleme zu erkennen und Ideen für Verbesserungen zu entwickeln. Durch das Begutachten von Code werden Sie zudem neue Designansätze und Programmiertechniken kennenlernen.

Einer der besten Wege zum Verbessern Ihrer Designfähigkeiten ist, ein gutes Gespür für *Warnzeichen* zu entwickeln – Anzeichen, dass ein Stück Code vermutlich komplizierter ist, als es sein müsste. Im Verlauf des Buchs werde ich Warnzeichen (durch das Skorpion-Icon hervorgehoben) vorstellen, die auf Probleme hinweisen, die mit den jeweiligen Designaspekten verbunden sind – die wichtigsten habe ich noch einmal am Ende des Buchs zusammengefasst. Sie können sie beim Programmieren nutzen: Sehen Sie ein Warnzeichen, nehmen Sie sich etwas Zeit und suchen Sie nach einem alternativen Design, das das Problem behebt. Verfolgen Sie diesen Ansatz das erste Mal, werden Sie eventuell eine Reihe von Designalternativen prüfen müssen, bevor Sie die finden, die das Warnzeichen verschwinden lässt. Aber geben Sie nicht zu schnell auf: Je mehr Alternativen Sie ausprobieren, ehe Sie das Problem beheben, desto mehr werden Sie lernen. Mit der Zeit werden Sie feststellen, dass Ihr Code immer weniger Warnzeichen enthält und Ihre Designs immer klarer werden. Mit zunehmender Erfahrung werden Sie auch auf andere Warnzeichen aufmerksam, die Ihnen dabei helfen werden, Designprobleme zu erkennen (und ich würde mich freuen, davon zu hören).

Wenn Sie die Ideen aus diesem Buch anwenden, ist es wichtig, mit Augenmaß und besonnen vorzugehen. Jede Regel hat ihre Ausnahmen, und jedes Prinzip hat seine Grenzen. Treiben Sie eine Designidee ins Extreme, wird Sie das nicht dahin bringen, wohin Sie möchten. Schönes Design spiegelt eine Balance zwischen konkurrierenden Ideen und Ansätzen wider. Einige Kapitel enthalten Abschnitte, die »Wann Sie zu weit gehen« heißen und in denen beschrieben ist, wie Sie erkennen, dass Sie es mit einer guten Sache zu gut meinen.

Annähernd alle Beispiele in diesem Buch nutzen Java oder C++, und ein Großteil der Diskussion dreht sich um das Design von Klassen in einer objektorientierten Sprache. Aber die Ideen passen auch in anderen Domänen. Fast alle Ideen beziehen sich auf Methoden, die auch auf Funktionen in einer Sprache ohne Objektorientierung angewendet werden können, wie zum Beispiel in C. Die Designideen passen auch zu Modulen, die gerade keine Klassen sind, wie zum Beispiel Subsysteme oder Netzwerkservices.

Mit diesen Gedanken im Hinterkopf wollen wir uns nun genauer anschauen, was Komplexität verursacht und wie man Softwaresysteme einfacher gestaltet.

KAPITEL 2

Die Natur der Komplexität

In diesem Buch geht es darum, wie man Softwaresysteme so entwirft, dass ihre Komplexität minimiert wird. Der erste Schritt ist, den Feind zu kennen. Was genau ist »Komplexität«? Wie können Sie erkennen, ob ein System unnötig komplex ist? Was sorgt dafür, dass Systeme komplex werden? Dieses Kapitel wird diese Fragen im Überblick beleuchten – folgende Kapitel werden Ihnen dann zeigen, wie Sie Komplexität auf niedrigerer Ebene anhand spezifischer struktureller Merkmale erkennen.

Die Fähigkeit zum Erkennen von Komplexität ist eine entscheidende Designfähigkeit. Sie erlaubt Ihnen, Probleme aufzudecken, bevor Sie viel Aufwand in ihre Lösung investieren, und bei der Auswahl von Alternativen eine gute Wahl zu treffen. Es ist leichter, ein einfaches Design zu erkennen, als eines zu erstellen. Haben Sie aber einmal erkannt, dass ein System zu kompliziert ist, können Sie diese Fähigkeit nutzen, um Ihre Designphilosophie hin zu mehr Einfachheit auszurichten. Erscheint ein Design kompliziert, versuchen Sie es mit einem anderen Ansatz und schauen, ob dieser zu mehr Einfachheit führt. Mit der Zeit werden Sie feststellen, dass bestimmte Techniken eher zu einfacheren Designs führen, während andere mit Komplexität in Zusammenhang stehen. Das erlaubt Ihnen, schneller für ein einfacheres Design zu sorgen.

Dieses Kapitel stellt zudem ein paar grundlegende Annahmen vor, die im Rest des Buchs zur Anwendung kommen. Spätere Kapitel werden auf das Material aus diesem Kapitel zurückkommen und eine Reihe von Verfeinerungen und Schlussfolgerungen liefern.

Definition der Komplexität

Für dieses Buch definiere ich »Komplexität« eher praktisch: **Komplexität ist all das, was mit der Struktur eines Softwaresystems in Zusammenhang steht und dieses schwer verständlich und schwer anpassbar macht.** Komplexität kann viele Formen annehmen. So kann es beispielsweise schwierig sein, zu verstehen, wie ein be-

stimmter Codeabschnitt funktioniert; es kann viel Aufwand erfordern, eine kleine Verbesserung umzusetzen, oder es ist nicht klar, welche Teile des Systems für die Verbesserung angepasst werden müssen; es kann schwer sein, einen Fehler zu beheben, ohne gleich einen neuen Fehler einzubauen. Lässt sich ein Softwaresystem nur schwer verstehen und anpassen, ist es kompliziert – lässt es sich leicht verstehen und anpassen, ist es einfach.

Sie können die Komplexität auch unter Kosten- und Nutzenaspekten betrachten. In einem komplexen System ist viel Arbeit erforderlich, um selbst kleine Verbesserungen umzusetzen. In einem einfachen System können größere Verbesserungen mit weniger Aufwand erreicht werden.

Komplexität sieht man sich beim Entwickeln zu einem bestimmten Zeitpunkt gegenüber, wenn man versucht, ein bestimmtes Ziel zu erreichen. Sie hängt nicht zwingend mit der Gesamtgröße oder der Gesamtfunktionalität des Systems zusammen. Häufig wird das Wort »komplex« genutzt, um große Systeme mit ausgefeilten Features zu beschreiben, aber wenn es einfach ist, an solch einem System zu arbeiten, ist es – im Rahmen dieses Buchs – nicht komplex. Natürlich arbeitet es sich mit so gut wie allen großen und ausgefeilten Softwaresystemen schwer, daher erfüllen sie auch meine Definition von Komplexität, aber das muss nicht zwingend der Fall sein. Es ist auch für ein kleines und schlichtes System möglich, ziemlich komplex zu sein.

Komplexität wird durch die am häufigsten vorkommenden Aktivitäten bestimmt. Besitzt ein System ein paar Teile, die zwar sehr kompliziert sind, aber so gut wie nie angefasst werden, haben diese keinen allzu großen Einfluss auf die Gesamtkomplexität des Systems. Wir stellen das mathematisch einmal sehr rustikal dar:

$$C = \sum_p c_p t_p$$

Die Gesamtkomplexität eines Systems (C) wird bestimmt durch die Komplexität jedes Teils p (c_p), gewichtet mit dem Anteil der Zeit, die Entwickler an diesem Teil arbeiten (t_p). Wenn Sie die Komplexität an einem Ort isolieren, an dem sie niemand zu sehen bekommt, ist das fast so gut, als würden Sie die Komplexität komplett ausmerzen.

Komplexität ist eher beim Lesen als beim Schreiben zu erkennen. Wenn Sie Code schreiben, der Ihnen einfach erscheint, aber andere der Meinung sind, er sei komplex, dann ist er auch komplex. Finden Sie sich selbst in solchen Situationen wieder, lohnt es sich, mit den anderen Entwicklern und Entwicklerinnen zu sprechen, um herauszufinden, warum sie den Code komplex finden – vermutlich erhalten Sie dadurch einige interessante Lektionen, aus denen Sie lernen können, warum es einen Unterschied zwischen deren und Ihrer Meinung gibt. Beim Entwickeln ist es Ihre Aufgabe, nicht nur Code zu schaffen, mit dem Sie leicht arbeiten können, sondern auch solchen, mit dem andere leicht arbeiten können.

Symptome der Komplexität

Komplexität manifestiert sich ganz allgemein in drei verschiedenen Ausprägungen, die im Folgenden beschrieben werden. Jede dieser Manifestationen erschwert es, Entwicklungsaufgaben zu erledigen.

Ausweiten von Änderungen: Das erste Symptom der Komplexität ist, wenn für eine scheinbar einfache Änderung Code an vielen verschiedenen Stellen angepasst werden muss. Stellen Sie sich beispielsweise eine Website mit vielen Seiten vor, die jeweils ein Banner mit einer Hintergrundfarbe anzeigen. In vielen frühen Websites war die Farbe explizit auf jeder Seite definiert (siehe Abbildung 2-1 (a)). Um den Hintergrund für solch eine Website anzupassen, müsste jede bestehende Seite per Hand geändert werden – für eine große Site mit Tausenden von Seiten nahezu unmöglich. Zum Glück nutzen moderne Websites einen Ansatz wie den in Abbildung 2-1 (b), bei dem die Bannerfarbe einmal an zentraler Stelle definiert wird und alle einzelnen Seiten auf diesen gemeinsamen Wert zugreifen. So kann die Bannerfarbe der gesamten Website durch eine einzelne Änderung angepasst werden. Eines der Ziele guten Designs ist, die Menge an Code zu verringern, die durch jede Designentscheidung betroffen ist, sodass für Designänderungen nicht so viele Codeanpassungen erforderlich sind.

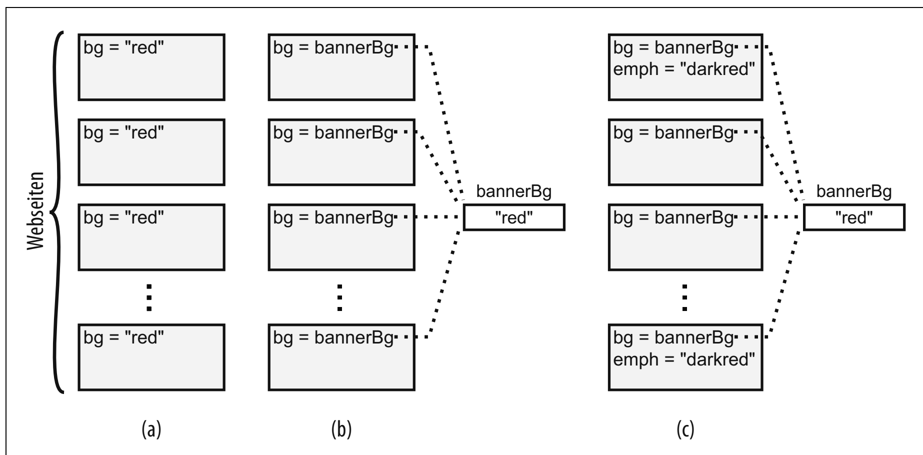

Abbildung 2-1: Jede Seite einer Website zeigt ein farbiges Banner an. In (a) ist die Hintergrundfarbe jedes Banners explizit auf jeder Seite festgelegt. In (b) steht die Hintergrundfarbe in einer gemeinsamen Variablen, und jede Seite bezieht sich auf diese Variable. In (c) nutzen manche Seiten eine zusätzliche Farbe zum Hervorheben, bei der es sich um eine dunklere Variante der Hintergrundfarbe handelt – ändert sich die Hintergrundfarbe, muss sich auch die Farbe der Hervorhebung ändern.

Kognitive Last: Das zweite Symptom der Komplexität ist die kognitive Last, die sich darauf bezieht, wie viel man beim Entwickeln wissen muss, um eine Aufgabe zu erfüllen. Eine höhere kognitive Last bedeutet, dass mehr Zeit mit dem Aufneh-

men und Erfassen der erforderlichen Informationen verbracht werden muss und dass es ein höheres Fehlerrisiko gibt, weil eventuell etwas Wichtiges vergessen wurde. Stellen Sie sich beispielsweise eine Funktion in C vor, die Speicher anfordert, einen Zeiger auf diesen Speicher zurückgibt und annimmt, dass der Aufrufende den Speicher freigeben wird. Das erhöht die kognitive Last beim Einsatz der Funktion – wird vergessen, den Speicher freizugeben, führt das zu einem Speicherleck. Kann das System so umstrukturiert werden, dass sich der Aufrufende nicht um das Freigeben des Speichers kümmern muss (dasselbe Modul, das den Speicher anfordert, sorgt auch dafür, dass er wieder freigegeben wird), reduziert das die kognitive Last. Solcherart kognitive Last entsteht auf vielerlei Weisen, wie zum Beispiel durch APIs mit vielen Methoden, globale Variablen, Inkonsistenzen und Abhängigkeiten zwischen Modulen.

Systemdesigner gehen manchmal davon aus, dass sich Komplexität anhand der Menge an Codezeilen messen lässt. Sie nehmen an, dass eine kürzere Implementierung auch einfacher sein muss – sind nur ein paar Zeilen Code für eine Änderung erforderlich, muss die Änderung einfach sein. Diese Sichtweise ignoriert aber die Kosten, die mit der kognitiven Last verbunden sind. Ich habe Frameworks gesehen, die es erlaubten, Anwendungen mit ein paar Zeilen Code zu schreiben. Es war aber außerordentlich schwierig, herauszufinden, wie diese Zeilen auszusehen hatten. **Manchmal ist ein Ansatz, der mehr Codezeilen erfordert, tatsächlich einfacher, weil er die kognitive Last verringert.**

Unbekannte Unbekannte (Unknown Unknowns): Das dritte Symptom der Komplexität ist, dass nicht offensichtlich ist, welcher Code angefasst werden muss, um eine Aufgabe abzuschließen, oder welche Informationen man haben muss, um die Aufgabe erfolgreich umzusetzen. In Abbildung 2-1 (c) kann man dieses Problem sehen. Die Website nutzt eine zentrale Variable, um die Hintergrundfarbe des Banners festzulegen, daher scheint eine Änderung einfach zu sein. Aber ein paar Webseiten verwenden eine dunklere Schattierung der Hintergrundfarbe für Hervorhebungen, und diese dunklere Farbe ist explizit in den einzelnen Seiten festgelegt. Ändert sich die Hintergrundfarbe, muss sich auch die Farbe für die Hervorhebungen ändern. Leider ist das beim Entwickeln oft nicht klar, daher wird eventuell nur die zentrale Variable `bannerBg` geändert, ohne die Farbe zum Hervorheben zu ändern. Und selbst wenn sich ein Entwickler dieses Problems bewusst ist, ist nicht offensichtlich, welche Seiten die Farbe zum Hervorheben verwenden – jede Seite der Website muss also durchsucht werden.

Von den drei Manifestationen der Komplexität sind die unbekannten Unbekannten (Unknown Unknowns) die schlimmsten. Denn das bedeutet, dass es etwas gibt, das Sie wissen müssen, es aber keine Möglichkeit gibt, herauszufinden, was das ist oder ob es überhaupt ein Problem ist. Erfahren werden Sie es erst, wenn nach einer Änderung Fehler auftauchen. Dass sich Änderungen ausweiten, ist nervig, aber solange klar ist, welcher Code angepasst werden muss, wird das System funktionieren, sobald die Änderung abgeschlossen ist. Genauso wird eine hohe kognitive Last die Kosten für eine Änderung erhöhen, aber wenn klar ist, welche In-

formationen gelesen werden müssen, wird die Änderung immer noch sehr wahrscheinlich richtig sein. Bei unbekannten Unbekannten ist unklar, was zu tun ist oder ob eine vorgeschlagene Lösung überhaupt funktionieren wird. Sicher kann man nur sein, wenn man jede Zeile Code im System liest, was für ausgewachsene Systeme ab einer bestimmten Größe völlig unmöglich ist. Und selbst das mag nicht ausreichen, weil eine Änderung von einer subtilen Designentscheidung abhängen kann, die nie dokumentiert wurde.

Eines der wichtigsten Ziele guten Systemdesigns ist *Offensichtlichkeit*. Das ist das Gegenteil von hoher kognitiver Last und unbekannten Unbekannten. In einem offensichtlichen System kann man beim Entwickeln schnell verstehen, wie der bestehende Code funktioniert und was erforderlich ist, um eine Änderung vorzunehmen. Ein offensichtliches System ist eines, in dem ein Entwickler oder eine Entwicklerin schnell eine Vermutung dazu anstellen kann, was zu tun ist, ohne allzu sehr nachdenken zu müssen – und dabei trotzdem zuversichtlich sein kann, dass diese Vermutung korrekt ist. In Kapitel 18 werden Techniken vorgestellt, mit denen Code offensichtlicher wird.

Ursachen für Komplexität

Nachdem Sie nun die Symptome von Komplexität aus einer sehr allgemeinen Perspektive kennengelernt haben und wissen, warum sie die Softwareentwicklung erschwert, besteht der nächste Schritt darin, zu verstehen, was Komplexität verursacht, sodass wir Systeme so designen können, dass die Probleme vermieden werden. Komplexität wird durch zwei Dinge verursacht: *Abhängigkeiten* und *Unklarheit*. Dieser Abschnitt behandelt diese Faktoren auf sehr allgemeinem Niveau – folgende Kapitel werden darauf eingehen, in welchem Zusammenhang sie zu Designentscheidungen auf niedrigeren Ebenen stehen.

So wie es in diesem Buch verstanden wird, besteht dann eine Abhängigkeit, wenn ein gegebener Codeabschnitt nicht isoliert nachvollzogen und angepasst werden kann – der Code bezieht sich auf die eine oder andere Art und Weise auf anderen Code, und der andere Code muss berücksichtigt und/oder angepasst werden, wenn der gegebene Code geändert wird. Im Website-Beispiel aus Abbildung 2-1 (a) erzeugt die Hintergrundfarbe Abhängigkeiten zwischen allen Seiten. Alle Seiten müssen den gleichen Hintergrund haben – ändern Sie also den Hintergrund für eine Seite, muss er auch für alle anderen angepasst werden. Ein anderes Beispiel für Abhängigkeiten zeigt sich bei Netzwerkprotokollen. Typischerweise gibt es eigenen Code für den Sender und den Empfänger, aber sie müssen sich beide an das Protokoll halten – ändern Sie den Code für den Sender, müssen Sie so gut wie immer auch entsprechende Änderungen beim Empfänger vornehmen und umgekehrt. Die Signatur einer Methode erzeugt eine Abhängigkeit zwischen der Implementierung dieser Methode und dem Code, der sie aufruft – wird der Methode ein neuer Parameter hinzugefügt, müssen alle Aufrufe dieser Methode angepasst werden, um diesen Parameter anzugeben.

Abhängigkeiten sind ein fundamentales Element von Software, und sie können nicht vollständig verhindert werden. Tatsächlich fügen wir als Teil des Softwaredesignprozesses absichtlich Abhängigkeiten hinzu. Jedes Mal, wenn Sie eine neue Klasse schreiben, erzeugen Sie Abhängigkeiten rund um die API für diese Klasse. Aber eines der Ziele des Softwaredesigns ist, die Menge an Abhängigkeiten zu verringern und sie so einfach und offensichtlich wie möglich zu machen.

Denken Sie noch mal an das Beispiel der Website. Auf der alten Website mit dem Hintergrund, der auf jeder Seite getrennt spezifiziert ist, hingen alle Webseiten voneinander ab. Die neue Website hat dieses Problem behoben, indem sie die Hintergrundfarbe an einer zentralen Stelle definiert und eine API bereitstellt, über die die einzelnen Seiten diese Farbe abrufen können, wenn sie gerendert werden. Glücklicherweise ist die neue Abhängigkeit offensichtlicher: Es ist klar, dass jede einzelne Webseite von der Farbe `bannerBg` abhängt, und beim Entwickeln kann man leicht alle Stellen finden, an denen die Variable verwendet wird, indem man nach ihrem Namen sucht. Zudem helfen Compiler dabei, API-Abhängigkeiten zu verwalten: Ändert sich der Name der gemeinsam genutzten Variablen, werden Kompilierungsfehler in jedem Code auftauchen, der immer noch den alten Namen verwendet. Die neue Website ersetzt eine nicht offensichtliche und schwierig zu managende Abhängigkeit durch eine einfachere und offensichtlichere.

Die zweite Ursache für Komplexität ist Unklarheit. Unklarheit tritt dann auf, wenn wichtige Informationen nicht offensichtlich sind. Ein einfaches Beispiel ist ein Variablenname, der so generisch ist, dass er nur wenige nützliche Informationen enthält (zum Beispiel `time`). Oder die Dokumentation für eine Variable gibt nicht ihre Einheiten an, sodass Sie diese nur herausfinden können, indem Sie im Code nach Stellen suchen, an denen die Variable zum Einsatz kommt. Unklarheit entsteht oft in Verbindung mit Abhängigkeiten, wenn nicht offensichtlich ist, dass eine Abhängigkeit existiert. Wird beispielsweise ein System durch einen neuen Fehlerstatus ergänzt, kann es notwendig sein, eine Tabelle, die beschreibende Textstrings für jeden Status enthält, um einen Eintrag zu erweitern. Aber die Existenz dieser Message-Tabelle wird beim Programmieren nicht offensichtlich sein, wenn man sich nur die Statusdeklaration anschaut. Inkonsistenz trägt ebenfalls gern zu Unklarheit bei: Wird derselbe Variablenname für zwei verschiedene Zwecke eingesetzt, ist nicht offensichtlich, welchem Zweck eine bestimmte Variable dient.

In vielen Fällen entsteht Unklarheit durch nicht adäquate Dokumentation – Kapitel 13 behandelt dieses Thema. Aber Unklarheit schafft auch ein Designproblem. Hat ein System ein klares und offensichtliches Design, wird weniger Dokumentation benötigt. Der Bedarf nach umfassender Dokumentation ist oft ein Warnsignal dafür, dass das Design noch nicht perfekt ist. Der beste Weg, Unklarheit zu verringern, ist, das Systemdesign zu vereinfachen.

Zusammen sind Abhängigkeiten und Unklarheit für die drei im Abschnitt »Symptome der Komplexität« auf Seite 21 beschriebenen Manifestationen der Komplexität verantwortlich. Abhängigkeiten führen dazu, dass sich Änderungen stärker auswei-

ten, und sie führen zu einer hohen kognitiven Last. Unklarheit sorgt für unbekannte Unbekannte und trägt ebenfalls zu kognitiver Last bei. Finden wir Designtechniken, die Abhängigkeiten und Unklarheit verringern, können wir die Komplexität der Software reduzieren.

Komplexität ist inkrementell

Komplexität wird nicht durch einen einzelnen katastrophalen Fehler ausgelöst – sie summiert sich in kleinen Häppchen auf. Eine einzelne Abhängigkeit oder Unklarheit wird für sich allein die Wartbarkeit eines Softwaresystems eher wenig beeinflussen. Komplexität entsteht aus Hunderten oder Tausenden kleinen Abhängigkeiten und Unklarheiten, die mit der Zeit entstehen. Schließlich gibt es so viele dieser kleinen Problemchen, dass jede mögliche Änderung am System durch diverse dieser Probleme beeinträchtigt wird.

Die inkrementelle Natur der Komplexität macht es schwer, sie im Griff zu behalten. Sie können sich leicht einreden, dass das kleine bisschen Komplexität, das Sie durch Ihre aktuelle Änderung in das System bringen, kein Problem ist. Aber wenn jeder Entwickler und jede Entwicklerin diesen Ansatz bei jeder Änderung verfolgt, summiert sich die Komplexität schnell auf. Und ist das geschehen, kann man sie nur schwer wieder loswerden, da das Beheben einer einzelnen Abhängigkeit oder Unklarheit im Gesamtkontext keinen großen Unterschied machen wird. Um das Wachsen der Komplexität zu verringern, müssen Sie eine »Null-Toleranz-Philosophie« verfolgen, wie wir sie in Kapitel 3 besprechen.

Zusammenfassung

Komplexität entsteht aus einer Ansammlung von Abhängigkeiten und Unklarheiten. Mit zunehmender Komplexität nehmen Änderungen größere Ausmaße an, Sie erhalten eine hohe kognitive Last und viele unbekannte Unbekannte (Unknown Unknowns). Im Ergebnis sind mehr Codeänderungen für das Implementieren neuer Features notwendig. Und beim Entwickeln verbringt man mehr Zeit mit dem Zusammentragen von Informationen, um die Änderung sicher zu machen – im schlimmsten Fall lassen sich gar nicht alle erforderlichen Informationen finden. Das Fazit: Komplexität macht es schwierig und riskant, eine bestehende Codebasis zu verändern.