KAPITEL 3

Modularität

Bevor wir anfangen, wollen wir einige (manchmal zu) häufig verwendete Begriffe klären, die in Diskussionen über Architektur im Zusammenhang mit Modularität verwendet werden und Definitionen für ihre weitere Verwendung in diesem Buch festlegen.

> 95 % der Wörter [über Softwarearchitektur] werden dafür benutzt, die Vorteile der »Modularität« zu loben. Der Rest wird – wenn überhaupt – verwendet, um darüber zu sprechen, wie dieses Ziel erreicht werden soll.
>
> *– Glenford J. Myers (1978)*

Viele Plattformen haben unterschiedliche Mechanismen zur Code-Wiederverwendung. Aber alle unterstützen irgendeine Möglichkeit, verwandten Code zu *Modulen* zusammenzufassen. Und obwohl dieses Konzept ein selbstverständlicher Teil der Softwarearchitektur ist, lässt es sich nur schwer definieren. Eine einfache Internetsuche ergibt Dutzende von Definitionen ohne Konsistenz (dafür mit einigen Widersprüchen). Dabei ist das Zitat von Myers kein neues Problem. Da es keine allgemein akzeptierte Definition gibt, müssen wir uns der Situation stellen und unsere eigenen Definitionen festlegen, um zumindest in diesem Buch für Konsistenz zu sorgen.

Für Architekten ist es absolut notwendig, Modularität in ihren vielen Erscheinungsformen für die Entwicklungsplattform ihrer Wahl zu verstehen. Viele Werkzeuge, die wir für die Analyse der Architektur einsetzen (Metriken, Fitnessfunktionen, Visualisierungen und so weiter), basieren auf diesen Modularitätskonzepten. Modularität ist ein Organisationsprinzip. Erstellt ein Architekt ein System, ohne darauf zu achten, wie die Einzelteile zusammenpassen, führt das zu einer Unzahl von Schwierigkeiten. Um eine Analogie aus der Physik zu verwenden: Softwaresysteme modellieren komplexe Systeme, die zur Entropie (oder Unordnung) neigen. Um die Ordnung aufrechtzuerhalten, muss einem physischen System Energie zugeführt werden. Das gilt auch für Softwaresysteme: Architekten müssen ständig Energie aufwenden, um eine gute strukturelle Stabilität sicherzustellen. So etwas passiert aber nicht zufällig.

Die Erhaltung guter Modularität ist ein passendes Beispiel für unsere Definition einer *impliziten* Architektureigenschaft: Kaum ein Projekt fordert von einem Archi-

tekten, eine gute Abgrenzung und Kommunikation zwischen Modulen sicherzustellen. Eine nachhaltige Codebasis braucht jedoch Ordnung und Konsistenz.

Definition

Das Wörterbuch definiert *Module* als »alle Teile eines Satzes standardisierter Bausteine oder unabhängiger Einheiten, die zur Konstruktion einer komplexeren Struktur verwendet werden können.« Wir verwenden den Begriff *Modularirät*, um eine logische Gruppierung verwandten Codes zu beschreiben, der eine Gruppe von Klassen in einer objektorientierten Sprache oder Funktionen in einer strukturierten oder funktionalen Sprache sein kann. Die meisten Sprachen enthalten Mechanismen zum Erreichen von Modularität (`package` in Java, `namespace` in .NET und so weiter). Entwickler verwenden Module typischerweise, um verwandten Code zu gruppieren. So könnte das Java-Package `com.mycompany.customer` Code enthalten, der für kundenbezogene Aufgaben zuständig ist.

Sprachen besitzen heutzutage eine Vielzahl an Packaging-Mechanismen, was den Entwicklern die Auswahl erschwert. So können Entwickler in vielen modernen Sprachen Verhalten in Funktionen/Methoden, Klassen oder Packages/Namensräumen definieren, jeweils mit unterschiedlicher Sichtbarkeit und unterschiedlichen Geltungsbereichen. Andere Sprachen machen die Sache noch komplizierter, indem sie Konstrukte wie das Metaobjekt-Protokoll (*https://oreil.ly/9Zw-J*) verwenden, um Entwicklern noch mehr Erweiterungsmöglichkeiten zu bieten.

Architekten müssen darauf achten, wie Entwickler ihren Code in Packages strukturieren, weil dies große Auswirkungen auf die Architektur hat. Sind beispielsweise mehrere Packages sehr eng aneinandergekoppelt, erhöht das die Schwierigkeiten, wenn sie für ähnliche Projekte wiederverwendet werden sollen.

Modulare Wiederverwendung vor dem Aufkommen der Klassen

Entwickler, die mit Sprachen ohne Objektorientierung groß geworden sind, fragen sich möglicherweise, warum es überhaupt so viele Möglichkeiten der Codetrennung gibt. Ein großer Teil der Antwort auf diese Frage hat mit Rückwärtskompatibilität zu tun – nicht des Codes wegen, sondern wegen der Denkweise der Entwickler. Im März 1968 veröffentlichte Edsger Dijkstra einen Brief in *Communications of the ACM* mit dem Titel »Go To Statement Considered Harmful.« (»Go-To-Anweisung als schädlich betrachtet«). Er kritisierte die übliche Verwendung der `GOTO`-Anweisung, die in damaligen Sprachen Standard war. Sie erlaubte es, im Code umherzuspringen, was seine Analyse und das Debugging erschwerte.

Das Papier half dabei, ein Zeitalter *strukturierter* Programmiersprachen einzuläuten mit Beispielen wie Pascal und C, die zu einem tieferen Nachdenken darüber anregten, wie die Einzelteile zusammenpassten. Die Entwickler merkten schnell,

dass den meisten Sprachen eine Möglichkeit fehlte, Dinge logisch zu gruppieren. So begann das Zeitalter der *modularen* Sprachen wie Modula (die nächste Sprache des Pascal-Schöpfers Niklaus Wirth) und Ada. Diese Sprachen verwendeten das Programmierkonstrukt des Moduls auf ähnliche Weise wie wir heutzutage Packages oder Namensräume einsetzen (allerdings ohne die Klassen).

Das Zeitalter der modularen Programmierung war ziemlich kurzlebig. Objektorientierte Sprachen wurden populär, weil sie neue Möglichkeiten boten, Code zu verkapseln und wiederzuverwenden. Und trotzdem bemerkten die Sprachschöpfer den Nutzen von Modulen und behielten sie in Form von Packages, Namensräumen und so weiter bei. Um diese unterschiedlichen Paradigmen zu unterstützen, besitzen Sprachen eine Vielzahl seltsamer Kompatibilitätsfeatures. Java unterstützt beispielsweise modulare (in Form von Packages, Initialisierung auf Package-Ebene in Form statischer Initializer), objektorientierte und funktionale Paradigmen. Dabei besitzt jeder Programmierstil eigene Regeln für Geltungsbereiche und andere Eigenarten.

Für unsere Diskussionen rund um Softwarearchitektur verwenden wir Modularität als allgemeinen Begriff, der eine Gruppierung verwandten Codes beschreibt: Klassen, Funktionen oder andere Formen. Das bedeutet nicht zwingend eine physische, sondern eher eine logische Trennung. Dieser Unterschied kann gelegentlich wichtig sein. Aus Gründen der Bequemlichkeit kann es sinnvoll erscheinen, eine große Zahl von Klassen in einer monolithischen Applikation zusammenzupacken. Irgendwann ist es jedoch an der Zeit, die Architektur neu zu strukturieren. Besitzt der Code durch die lose Trennung eine starke Kopplung, wird dies zu einem Hindernis, wenn der Monolith aufgetrennt werden muss. Daher ist es sinnvoll, das Konzept der Modularität unabhängig von einer physischen Trennung zu betrachten, die durch eine bestimmte Plattform impliziert oder erzwungen wird.

Ebenso lohnt es sich, das allgemeine Konzept des *Namensraums* bei der .NET-Plattform unabhängig von der technischen Implementierung zu betrachten. Oft brauchen Entwickler genaue, qualifizierte Namen für Softwarebestandteile (Komponenten, Klassen und so weiter), um diese voneinander trennen zu können. Das offensichtlichste aktuelle Beispiel hierfür ist das Internet: einmalige globale Identifier, die an IP-Adressen gebunden sind. Um verschiedene Bestandteile zu organisieren, verwenden die meisten Sprachen einen Modularitätsmechanismus, der gleichzeitig als Namensraum dient: Variablen, Funktionen und/oder Methoden. Manchmal wird die Modulstruktur auch physisch abgebildet. So ist es in Java erforderlich, dass die Packagestruktur der Verzeichnisstruktur der physischen Klassendateien entspricht.

Java 1.0: Eine Sprache ohne Namenskonflikte

Die ursprünglichen Schöpfer von Java besaßen umfassende Erfahrung im Umgang mit Namenskonflikten und -widersprüchen mit den verschiedenen damals üblichen Programmierplattformen. Das Java-Originaldesign verwendete einen cleveren

Hack, um Verwechslungen zwischen zwei gleichnamigen Klassen zu vermeiden. Angenommen, es gibt eine Bestellmöglichkeit (mit dem Namen *order*) und eine Installationsreihenfolge (ebenfalls mit dem Namen *order*). Beide verwenden den gleichen Namen (*order*), besitzen aber sehr unterschiedliche Funktionalität (und Klassen). Die Lösung in Java bestand in der Schaffung des Namensraum-Mechanismus des `package` in Kombination mit der Anforderung, dass die physische Verzeichnisstruktur dem Packagenamen entsprechen musste. Weil Dateisysteme es nicht gestatten, dass sich zwei Dateien gleichen Namens im selben Verzeichnis befinden, konnten diese Eigenschaften des Betriebssystems genutzt werden, um Verwechslungen zu vermeiden. Daher enthielt der `classpath` in Java ursprünglich nur Verzeichnisse, wodurch Namenskonflikte vermieden wurden.

Allerdings haben die Sprachschöpfer bemerkt, dass die zwingende Verwendung einer vollständigen Verzeichnisstruktur besonders bei großen Projekten ziemlich mühsam werden konnte. Zudem war die Erstellung wiederverwendbarer Bestandteile schwierig: Frameworks und Bibliotheken mussten »zerlegt« werden, um sie auf eine Verzeichnisstruktur abzubilden. Daher wurde das zweite große Java-Release (1.2, als Java 2 bezeichnet) um den `jar`-Mechanismus erweitert, wodurch eine Archivdatei als Verzeichnisstruktur im Klassenpfad verwendet werden konnte. Im folgenden Jahrzehnt kämpften Java-Entwickler damit, den `classpath` als Kombination aus Verzeichnissen und JAR-Dateien korrekt einzurichten. Und natürlich war die ursprüngliche Absicht damit hinfällig: Jetzt konnten zwei JAR-Dateien im Klassenpfad für Namenskonflikte sorgen, was zu unzähligen Kriegsberichten über das Debugging von Class Loadern führte.

Modularität messen

Weil Modularität für Architekten so wichtig ist, brauchen sie Werkzeuge, um sie zu verstehen. Glücklicherweise haben Forscher eine Reihe sprachunabhängiger Metriken entwickelt, die Architekten beim Verständnis der Modularität helfen. Wir konzentrieren uns auf drei Schlüsselkonzepte: *Kohäsion* (»cohesion«), *Kopplung* (»coupling«) und *Konnaszenz* (»connascence«).

Kohäsion

Die *Kohäsion* gibt an, wie weit die Teile eines Moduls im gleichen Modul enthalten sein sollten. Man misst also, wie sich verwandte Teile zueinander verhalten. Idealerweise befinden sich alle Teile eines kohäsiven Moduls im gleichen Package. Würde man ein solches Modul in kleinere Einheiten aufteilen, müssten diese anhand von Aufrufen zwischen den Modulen aneinandergekoppelt werden, um nützliche Ergebnisse zu erhalten.

> Der Versuch, ein kohäsives Modul aufzuteilen, hätte nur eine gesteigerte Kopplung und eine verringerte Lesbarkeit zur Folge.
>
> – *Larry Constantine*

Informatiker haben verschiedene Arten der Kohäsion definiert, die hier von gut nach schlecht aufgelistet sind:

Funktionale Kohäsion
: Alle Teile des Moduls haben eine Beziehung zueinander, und das Modul enthält alles Nötige, um zu funktionieren.

Sequenzielle Kohäsion
: Zwei Module interagieren miteinander, wobei eines Daten ausgibt, die als Eingabe für das andere dienen.

Kommunikatorische Kohäsion
: Zwei Module bilden eine Kommunikationskette, wobei jedes Modul bestimmte Informationen verarbeitet und/oder zu einer Ausgabe beiträgt. Ein Beispiel wäre das Hinzufügen eines Datenbankeintrags und das Erstellen einer E-Mail basierend auf diesen Informationen.

Prozedurale Kohäsion
: Zwei Module müssen Code in einer bestimmten Reihenfolge ausführen.

Temporale Kohäsion
: Module können, basierend auf zeitbezogenen Abhängigkeiten, miteinander in Beziehung stehen. Viele Systeme enthalten beispielsweise Bestandteile ohne Bezug zueinander, die beim Start des Systems initialisiert werden müssen. Diese verschiedenen Aufgaben sind temporal kohäsiv.

Logische Kohäsion
: Die Daten innerhalb von Modulen stehen in einer logischen, aber nicht in einer funktionalen Beziehung zueinander. Nehmen wir als Beispiel ein Modul, das Informationen aus Text, Objekten oder Streams konvertiert. Ein häufiges Beispiel für diese Art der Kohäsion gibt es in so gut wie jedem Java-Projekt in Form des `StringUtils`-Packages, einer Gruppe statischer Methoden, die an einem `String` arbeiten, aber ansonsten nichts miteinander zu tun haben.

Zufällige Kohäsion
: Elemente eines Moduls, die nichts miteinander gemeinsam haben, als dass sie sich in der gleichen Quellcodedatei befinden. Dies ist die schlechteste Form der Kohäsion.

Obwohl wir hier sieben Varianten aufgezählt haben, ist die *Kohäsion* eine weniger präzise Messmethode als die *Kopplung*. Oftmals liegt das Maß der Kohäsion eines bestimmten Moduls im Ermessen des Architekten. Nehmen Sie beispielsweise folgende Moduldefinition:

`Customer Maintenance` (Kundendaten verwalten)

- `add customer` (Kunden hinzufügen)
- `update customer` (Kunden aktualisieren)
- `get customer` (Kundendaten auslesen)
- `notify customer` (Kunden informieren)

- `get customer orders` (Kundenbestellungen auslesen)
- `cancel customer orders` (Kundenbestellung abbrechen)

Sollten sich die letzten beiden Punkte in diesem Modul befinden, oder sollte der Entwickler zwei separate Module erstellen, wie unten gezeigt?

`Customer Maintenance`

- `add customer`
- `update customer`
- `get customer`
- `notify customer`

`Order Maintenance` (Bestellungen verwalten)

- `get customer orders`
- `cancel customer orders`

Welche von beiden ist die korrekte Struktur? Wie immer: Es kommt darauf an.

- Sind die zwei zuletzt genannten die einzigen Operationen für `Order Mainteneance`? Falls ja, könnte es sinnvoll sein, diese Operationen wieder in `Customer Mainteneance` zu integrieren.
- Wird erwartet, dass `Customer Mainteneance` stark wächst? Das könnte dazu führen, dass Entwickler versuchen, Verhalten auszulagern.
- Benötigt `Order Mainteneance` so viel Wissen aus `Customer Mainteneance`, dass für eine Trennung der beiden Module ein hoher Grad an Kopplung notwendig wäre, um sie funktional zu machen? Dann wären wir wieder bei dem Zitat von Larry Constantine.

Diese Fragen zeigen die Art von Kompromissanalyse, die das Zentrum der Aufgaben eines Softwarearchitekten darstellt.

Trotz der Subjektivität der Kohäsion haben Informatiker überraschenderweise eine gute strukturelle Metrik zur Ermittlung der Kohäsion entwickelt (genauer gesagt, ihres Fehlens). So wurde die Chidamber and Kemerer objektorientierte Metrik-Suite (*https://oreil.ly/-1lMh*) von den gleichnamigen Autoren entwickelt, um die einzelnen Aspekte objektorientierter Softwaresysteme messen zu können. Die Suite enthält eine Reihe von Code-Metriken, wie die zyklomatische Komplexität (*https://de.wikipedia.org/wiki/McCabe-Metrik*) (siehe »Zyklomatische Komplexität« auf Seite 81) und andere wichtige Kopplungs-Metriken, die wir in »Kopplung« auf Seite 44 besprechen.

Die »Chidamber and Kemerer Lack of Cohesion in Methods«-(LCOM-)Metrik misst die strukturelle Kohäsion eines Moduls, typischerweise einer Komponente. Die ursprüngliche Version sehen Sie in Formel 3-1.

Formel 3-1: LCOM, Version 1

$$LCOM = \begin{cases} |P| - |Q|, & \text{if } |P| > |Q| \\ 0, & \text{andernfalls} \end{cases}$$

P wächst um eins für jede Methode, die nicht auf ein bestimmtes gemeinsam genutztes Feld zugreift. *Q* verringert sich um eins für Methoden, die ein bestimmtes gemeinsam genutztes Feld nutzen. Die Autoren haben Sympathie mit den Lesern, die diese Formel nicht verstehen. Noch schlimmer: Die Formel ist im Laufe der Zeit noch komplexer geworden. Die zweite Variante wurde 1996 eingeführt (daher der Name *LCOM96B*), siehe Formel 3-2.

Formel 3-2: LCOM 96b

$$LCOM96b = \frac{1}{a} \sum_{j=1}^{a} \frac{m - \mu(Aj)}{m}$$

Wir werden uns nicht weiter damit beschäftigen, die Variablen und Operatoren in Formel 3-2 zu entwirren, weil die folgende schriftliche Erklärung besser verständlich ist. Grundsätzlich lässt sich über die LCOM-Metrik die zufällige Kopplung zwischen Klassen ermitteln. Hier eine bessere Definition von LCOM:

LCOM
: Die Summe der nicht gemeinsam genutzten Methodensätze, die nicht auf geteilte Felder zugreifen.

Angenommen, Sie haben eine Klasse mit den privaten Feldern `a` und `b`. Viele der Methoden greifen nur auf `a` zu und viele andere Methoden nur auf `b`. Die Summe der Methodensätze, die nicht über geteilte Felder (`a` und `b`) gemeinsam genutzt werden, ist hoch. Das ergibt für diese Klasse einen hohen LCOM-Wert, was eine geringe Kohäsion in den Methoden ausdrückt. Nehmen Sie beispielsweise die drei in Abbildung 3-1 gezeigten Klassen:

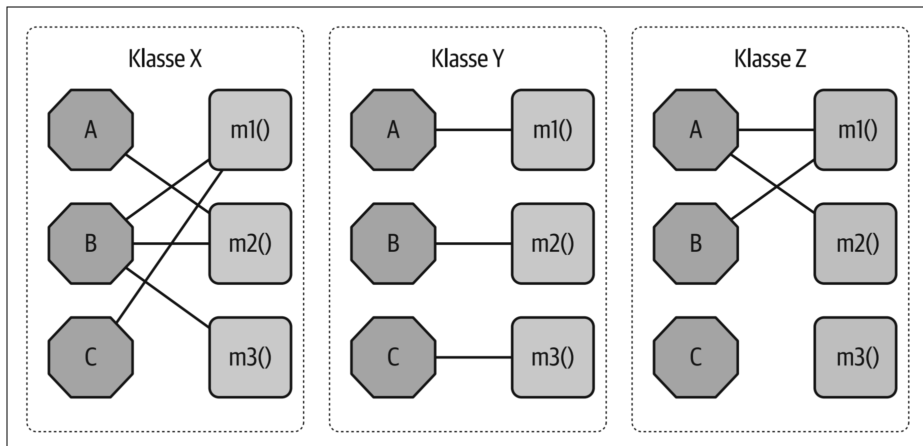

Abbildung 3-1: Illustration der LCOM-Metrik. Felder sind als Achtecke, Methoden als Quadrate dargestellt.

In Abbildung 3-1 sind die Felder mit einzelnen Buchstaben gekennzeichnet, die Methoden werden als Quadrate dargestellt. In `Klasse X` ist der LCOM-Wert nied-

rig, was auf eine gute strukturelle Kohäsion hinweist. In `Klasse Y` ist die Kohäsion dagegen geringer. Jedes Paar aus Feld und Methode in `Klasse Y` könnte in einer eigenen Klasse stehen, ohne dadurch das Verhalten zu verändern. In `Klasse Z` ist die Kohäsion nicht einheitlich. Hier könnten Entwickler das letzte Paar aus Feld und Methode in eine eigene Klasse refaktorieren.

Die LCOM-Metrik ist hilfreich, wenn Architekten Codebasen analysieren, um von einem Architekturstil zu einem anderen zu migrieren. Ein häufiges Problem bei der Migration sind gemeinsam genutzte Hilfsklassen. Die LCOM-Metrik kann Architekten helfen, zufällig gekoppelte Klassen zu finden, die eigentlich nie als einzelne Klasse hätten definiert werden sollen.

Viele Software-Metriken haben ernste Mängel, und auch LCOM ist dagegen nicht immun. Diese Metrik ist nur in der Lage, einen *strukturellen* Kohäsionsmangel festzustellen. Sie kann nicht logisch ermitteln, ob bestimmte Teile zusammenpassen. Das ist in unserem zweiten Gesetz der Softwarearchitektur ausgedrückt: Das *Warum* sollte Vorrang vor dem *Wie* haben.

Kopplung

Zum Glück gibt es bessere Werkzeuge, um die Kopplung in Codebasen zu analysieren, die teilweise auf der Graphentheorie basieren: Da Methodenaufrufe und -rückgaben einen Aufrufgraphen bilden, ist eine mathematisch basierte Analyse möglich. Im Jahr 1979 veröffentlichten Edward Yourdon und Larry Constantine das Buch *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Prentice-Hall). Es definiert eine Reihe von Metriken, darunter die *afferente* und *efferente* Kopplung. Die *afferente* Kopplung misst die Zahl der *eingehenden* Verbindungen für ein Codeartefakt (Komponente, Klasse, Funktion und so weiter). Die *efferente* Kopplung misst die *ausgehenden* Verbindungen zu anderen Codeartefakten. Es gibt für so gut wie jede Plattform Werkzeuge, mit denen Architekten die Kopplungseigenschaften von Code messen können, um ihnen bei der Restrukturierung, Migrierung oder dem Verständnis von Code zu helfen.

Warum haben diese Kopplungsmetriken so ähnliche Namen?

Warum haben zwei für die Architektur so wichtige Metriken, die für gegensätzliche Konzepte stehen, fast identische Namen, die sich nur durch einen Vokal am Anfang unterscheiden? Diese Begriffe stammen aus Yourdons und Constantines Buch *Structured Design*. Die Begriffe der afferenten und efferenten Kopplung sind der Mathematik entlehnt. Eigentlich sollten sie besser als eingehende und ausgehende Kopplung bezeichnet werden. Die Autoren räumten mathematischer Symmetrie einen höheren Stellenwert ein als der Klarheit. Entwickler behelfen sich mit einer Reihe von Eselsbrücken: *a* steht im Alphabet vor *e*, entsprechend den englischen Begriffen *incoming* (eingehend) und *outgoing* (ausgehend) oder der Beobachtung, dass der Anfangsbuchstabe *e* von *efferent* dem von *exit* (Ausgang) entspricht.

Abstraktheit, Instabilität und Entfernung von der Hauptsequenz

Zwar hat der Rohwert für die Komponentenkopplung einen gewissen Nutzen für Architekten, andere abgeleitete Metriken ermöglichen jedoch einen tieferen Einblick. Diese wurden ursprünglich von Robert Martin für ein Buch über C++ entwickelt, sie lassen sich aber auch auf viele andere objektorientierte Sprachen übertragen.

Die *Abstraktheit* ist das Verhältnis zwischen abstrakten (abstrakte Klassen, Interface, und so weiter) und konkreten Artefakten (Implementierung). Sie dient als Maß der Abstraktheit im Vergleich zur Implementierung. Nehmen Sie beispielsweise eine Codebasis ohne jegliche Abstraktion, die nur eine riesige Funktion (etwa eine einzelne `main()`-Methode) enthält. Der Nachteil einer Codebasis mit geringer Abstraktion liegt darin, dass Entwickler nur schwer die Zusammenhänge erkennen können (es ist beispielsweise nicht leicht, auf den ersten Blick zu sehen, was `AbstractSingletonProxyFactoryBean` tut).

Die Formel für Abstraktheit sehen Sie in Formel 3-3.

Formel 3-3: Abstraktheit

$$A = \frac{\sum m^a}{\sum m^c}$$

In der Gleichung steht m^a für die *abstrakten* Elemente (Interfaces oder abstrakte Klassen) des Moduls und m^c für die *konkreten* Elemente (nichtabstrakte Klassen). Diese Metrik überprüft die gleichen Kriterien. Stellen Sie sich beispielsweise eine Applikation mit 5.000 Codezeilen vor, die alle in einer `main()`-Methode stehen. Der Zähler für die Abstraktheit ist hier 1, während der Nenner den Wert 5.000 hat. Die Abstrakheit liegt also fast bei 0. Diese Metrik misst demnach das Verhältnis der Abstraktionen in Ihrem Code.

Architekten berechnen die *Abstraktheit*, indem sie das Verhältnis zwischen der Summe der abstrakten und der Summe der konkreten Artefakte ermitteln.

Eine weitere abgeleitete Metrik ist die *Instabilität*. Sie ist definiert als das Verhältnis zwischen efferenter Kopplung und der Summe aus efferenter und afferenter Kopplung, wie in Formel 3-4 gezeigt.

Formel 3-4: Instabilität

$$I = \frac{C^e}{C^e + C^a}$$

In dieser Gleichung steht c^e für *efferente* (ausgehende) Kopplung und c^a für *afferente* (eingehende) Kopplung.

Die *Instabilitätsmetrik* ermittelt, wie volatil eine Codebasis ist. Eine Codebasis mit hoher Instabilität geht durch den hohen Kopplungsgrad bei Änderungen leichter in

die Brüche. Ruft eine Klasse beispielsweise viele andere Klassen auf, um Arbeit zu delegieren, kann es in der aufrufenden Klasse leichter zu Störungen kommen, wenn eine der aufgerufenen Methoden verändert wird.

Entfernung von der Hauptsequenz

Eine der wenigen ganzheitlichen Metriken, mit der Architekten die architektonische Struktur messen können, ist die *Entfernung von der Hauptsequenz (distance from the main sequence)*. Diese abgeleitete Metrik basiert auf *Instabilität* und *Abstraktheit*, wie in Formel 3-5 gezeigt.

Formel 3-5: Entfernung von der Hauptsequenz

$$D = |A + I - 1|$$

In der Gleichung steht `A` für die *Abstraktheit* und I für die *Instabilität*.

Beachten Sie, dass *Abstraktheit* und *Instabilität* Verhältnisse sind. Ihre Ergebnisse liegen also immer zwischen 0 und 1. Wenn wir die Beziehung grafisch darstellen, erhalten wir daher den in Abbildung 3-2 gezeigten Graphen.

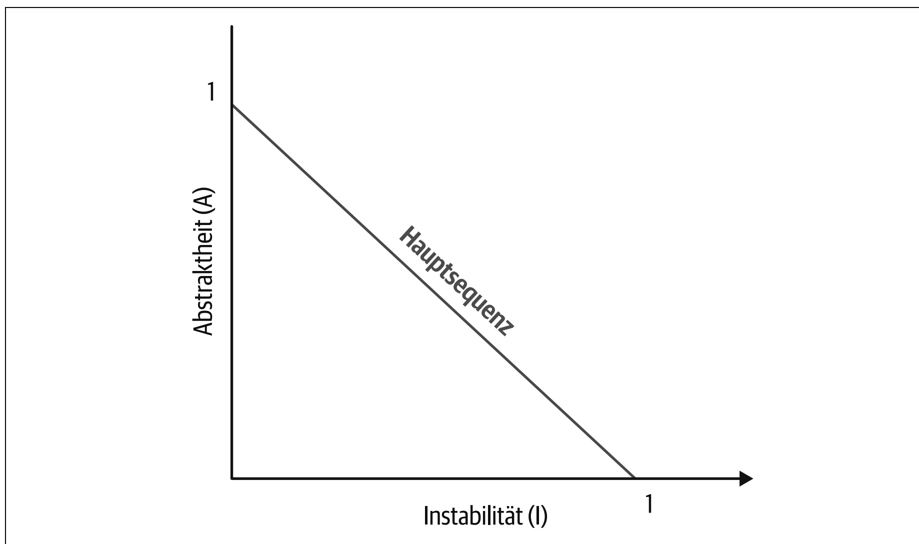

Abbildung 3-2: Die Hauptsequenz definiert die ideale Beziehung zwischen Abstraktheit und Instabilität.

Die *Entfernungs*-Metrik geht von einer idealen Beziehung zwischen Abstraktheit und Instabilität aus. Klassen, die sich nahe an dieser idealisierten Linie befinden, besitzen eine gesunde Mischung aus beiden Faktoren. Durch die grafische Darstellung einer bestimmten Klasse können Entwickler die Metrik der *Entfernung von der Hauptsequenz* ermitteln, wie in Abbildung 3-3 gezeigt.

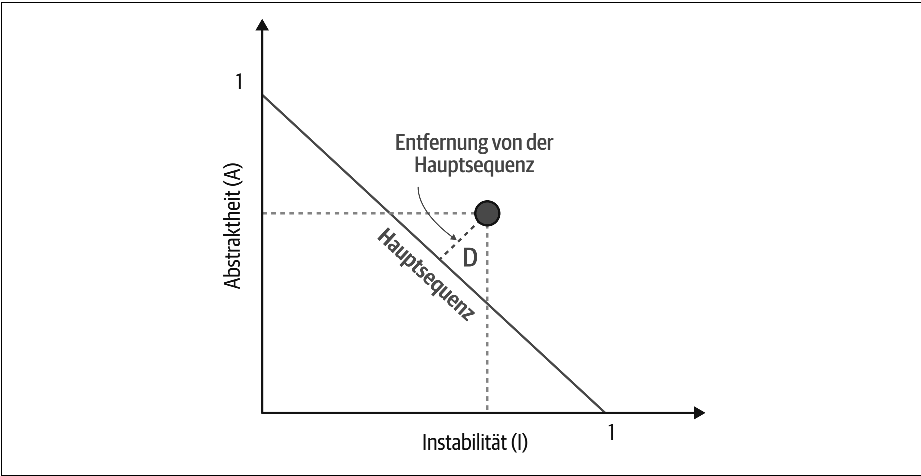

Abbildung 3-3: Normalisierte Entferunung von der Hauptsequenz für eine bestimmte Klasse

In Abbildung 3-3 erstellen Entwickler einen Graphen für die fragliche Klasse und messen dann die Entfernung von der idealisierten Linie. Je näher sich die Klasse an der Linie befindet, desto ausgewogener ist sie. Klassen, die sich zu nahe an der rechten oberen Ecke befinden, geraten in die sogenannte *Zone der Nutzlosigkeit*. Zu abstrakter Code ist schwer zu benutzen. Umgekehrt gerät Code, der sich der linken unteren Ecke nähert, in die *Zone des Schmerzes*: Code mit zu viel Implementierung und zu wenig Abstraktion wird spröde und schwer zu warten, wie in Abbildung 3-4 gezeigt.

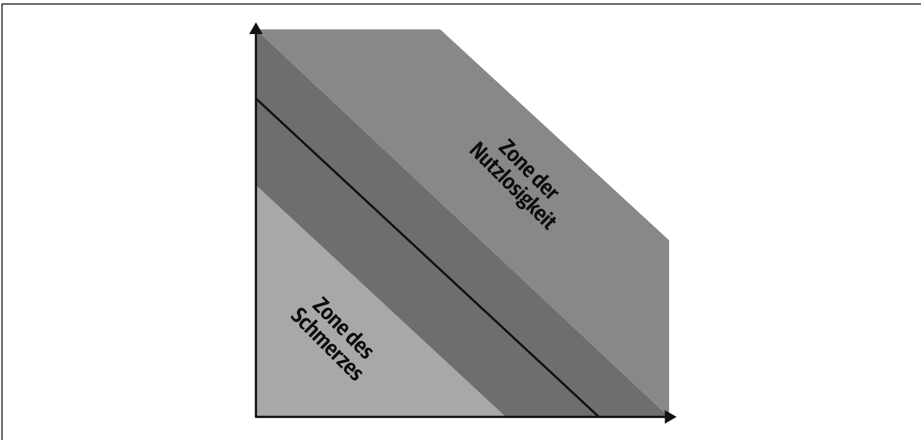

Abbildung 3-4: Die Zone der Nutzlosigkeit und die Zone des Schmerzes

Zur Emittlung dieser Metriken gibt es Werkzeuge für viele Plattformen, die Architekten bei der Analyse von Codebasen helfen, um sie aufgrund einer Migration oder einer Bewertung der technischen Schulden (technical debt) besser kennenzulernen.

Grenzen der Metriken

Zwar gibt es in unserer Branche einige Metriken auf Codeebene, mit denen sich wertvolle Erkenntnisse über Codebasen gewinnen lassen. Im Vergleich mit anderen Ingenieursdisziplinen sind unsere Werkzeuge jedoch erstaunlich stumpf. Selbst Kennzahlen, die direkt aus der Codestruktur gewonnen werden, müssen interpretiert werden. Die *zyklomatische Komplexität* (siehe »Zyklomatische Komplexität« auf Seite 81) misst beispielsweise die Komplexität von Codebasen, kann aber nicht zwischen *essenzieller* (das zugrunde liegende Problem ist komplex) und *zufälliger Komplexität* unterscheiden (der Code ist komplexer als nötig). So gut wie alle Metriken auf Codeebene bedürfen einer Interpretation. Dennoch helfen sie, die Grundlagen für wichtige Kennzahlen wie die zyklomatische Komplexität zu schaffen, anhand derer Architekten erkennen können, mit welcher Art von Code sie es zu tun haben. Das Einrichten solcher Tests besprechen wir in »Governance und Fitnessfunktionen« auf Seite 84.

Beachten Sie, dass das zuvor genannte Buch von Edward Yourdon und Larry Constantine (*Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*) aus der Zeit stammt, in der es noch keine objektorientierten Sprachen gab und sich daher auf funktionale Programmierkonstrukte wie Funktionen (nicht Methoden) konzentriert. Im Buch werden außerdem verschiedene Arten der Kopplung definiert, die wir hier nicht weiter behandeln, weil sie durch die *Konnaszenz* ersetzt wurden.

Konnaszenz

1996 veröffentlichte Meilir Page-Jones das Buch *What Every Programmer Should Know About Object-Oriented Design* (Dorset House), in dem die afferenten und efferenten Kopplungsmetriken verfeinert und für objektorientierte Sprachen angepasst wurden. Hierfür benutzte er ein Konzept namens *Konnaszenz*, das er folgendermaßen definiert:

> Zwei Komponenten sind konnaszent, wenn eine Änderung in einer Komponente eine Änderung der anderen erfordert, um die Gesamt-Korrektheit des Systems aufrechtzuerhalten.
>
> *– Meilir Page-Jones*

Er entwickelte zwei Konnaszenztypen: *statisch* und *dynamisch*.

Statische Konnaszenz

Statische Konnaszenz bezieht sich auf die Kopplung auf Quellcodeebene (im Gegensatz zur Kopplung zur Laufzeit, die in »Dynamische Konnaszenz« auf Seite 50 behandelt wird). Sie ist eine Verfeinerung der afferenten und efferenten Kopplung,

die in *Structured Design* definiert wird. Architekten betrachten die folgenden Typen statischer Konnaszenz als Grad afferenter oder efferenter Kopplung.

Namensbasierte Konnaszenz (Connascence of Name, CoN)
: Mehrere Komponenten müssen sich über den Namen einer Einheit (»entity«) einig sein.

  Methodennamen sind die häufigste und wünschenswerteste Art für die Kopplung in Codebasen. Das gilt besonders für moderne Refaktorierungswerkzeuge, mit denen systemweite Namensänderungen problemlos durchführbar sind.

Typbasierte Konnaszenz (Connascence of Type, CoT)
: Mehrere Komponenten müssen sich über den Typ einer Einheit einig sein.

  Dieser Konnaszenztyp bezieht sich auf die häufige Konvention vieler statisch typisierter Sprachen, Variablen und Parameter auf bestimmten Typen zu beschränken. Diese Fähigkeit ist aber kein reines Sprachmerkmal – einige dynamisch typisierte Sprachen verfügen ebenfalls über selektive Typisierung, namentlich Clojure (*https://clojure.org*) und Clojure Spec (*https://clojure.org/about/spec*).

Bedeutungsbasierte Konnaszenz (Connascence of Meaning, CoM) oder konventionsbasierte Konnaszenz (Connascence of Convention, CoC)
: Mehrere Komponenten müssen sich über die Bedeutung bestimmter Werte einig sein.

  Der häufigste Fall für diesen Konnaszenztyp in Codebasen sind hartcodierte Zahlen anstelle von Konstanten. So ist es in einigen Sprachen üblich, irgendwo zu definieren, dass `int TRUE = 1; int FALSE = 0`. Stellen Sie sich vor, was passiert, wenn jemand diese Werte vertauscht ...

Positionsbasierte Konnaszenz (Connascence of Position, CoP)
: Mehrere Komponenten müssen sich über die Reihenfolge von Werten einig sein.

  Dieses Problem tritt bei Parameterwerten für Methoden- und Funktionsaurufe auf, und zwar selbst in Sprachen mit statischer Typisierung. Angenommen, ein Entwickler erstellt die Methode `void updateSeat(String name, String seatLocation)` und ruft sie mit den Werten `updateSeat("14D", "Ford, N")` auf, so ist die Semantik selbst dann falsch, wenn die Typen korrekt sind.

Algorithmenbasierte Konnaszenz (Connascence of Algorithm, CoA)
: Mehrere Komponenten müssen sich über einen bestimmten Algorithmus einig sein.

  Ein häufiger Fall für diesen Konnaszenztyp tritt auf, wenn ein Entwickler einen Sicherheits-Hashingalgorithmus definiert. Der Algorithmus läuft sowohl auf dem Server als auch im Client und muss identische Ergebnisse erzeugen, um den Benutzer zu authentifizieren. Dies ist offensichtlich eine sehr starke Form der Kopplung – wird der Algorithmus auf einer Seite (Server/Client) verändert, funktioniert die gesamte Authentifizierung nicht mehr.

Dynamische Konnaszenz

Der andere von Page-Jones definierte Konnaszenztyp ist die *dynamische Konnaszenz*, die Aufrufe zur Laufzeit analysiert. Unten sehen Sie eine Beschreibung der verschiedenen dynamischen Konnaszenztypen:

Ausführungsbasierte Konnaszenz (Connascence of Execution, CoE)
: Die Ausführungsreihenfolge mehrerer Komponenten ist wichtig.

  Nehmen Sie zum Beispiel folgenden Code:

```
email = new Email();
email.setRecipient("foo@example.com");
email.setSender("me@me.com");
email.send();
email.setSubject("whoops");
```

  Dieser Code funktioniert nicht, weil bestimmte Eigenschaften in der richtigen Reihenfolge definiert werden müssen.

Zeitbasierte Konnaszenz (Connascence of Timing, CoT)
: Das Timing mehrerer Komponenten ist wichtig.

  Der übliche Fall für diesen Konnaszenztyp ist eine Race Condition, die durch zwei gleichzeitig ausgeführte Threads verursacht wird, die das Ergebnis einer gemeinsamen Operation beeinflussen.

Wertbasierte Konnaszenz (Connascence of Values, CoV)
: Tritt auf, wenn verschiedene Werte einander beeinflussen und gemeinsam verändert werden müssen.

  Angenommen, ein Entwickler definiert ein Rechteck anhand von vier Punkten, die für die Ecken stehen. Um die Integrität der Datenstruktur sicherzustellen, kann der Entwickler nicht willkürlich einen Punkt ändern, ohne die Auswirkungen auf die übrigen Punkte zu berücksichtigen.

  Häufiger und problematischer sind jedoch Fälle, bei denen es um Transaktionen geht, besonders in verteilten Systemen. Wenn ein Architekt ein System mit mehreren Datenbanken erstellt und ein Wert in allen Datenbanken aktualisiert werden muss, so müssen alle diese Werte zusammen verändert werden oder gar nicht.

Identitätsbasierte Konnaszenz (Connascence of Identity, CoI)
: Tritt auf, weil mehrere Werte zusammengehören und gemeinsam verändert werden müssen.

  Das übliche Beispiel für diesen Konnaszenztyp verwendet zwei unabhängige Komponenten, die eine gemeinsame Datenstruktur verwenden und aktualisieren müssen, wie beispielsweise eine verteilte Queue (»distributed queue«).

Die dynamische Konnaszenz ist für Architekten schwerer zu ermitteln, weil uns die Werkzeuge fehlen, um Laufzeitaufrufe ebenso effektiv zu analysieren wie den Aufrufgraphen.

Eigenschaften der Konnaszenz

Die Konnaszenz ist ein Analysewerkzeug für Architekten und Entwickler. Dabei helfen einige Eigenschaften der Konnaszenz Entwicklern dabei, sie weise zu verwenden. Unten sehen Sie eine Beschreibung der einzelnen Eigenschaften der Konnaszenz:

Stärke

Architekten analysieren die Stärke der Konnaszenz, indem sie ermitteln, wie einfach Entwickler einen bestimmten Kopplungstyp refaktorieren können. Wie in Abbildung 3-5 zu sehen, sind verschiedene Arten der Konnaszenz wünschenswerter als andere. Architekten und Entwickler können die Kopplungseigenschaften ihrer Codebasis verbessern, indem sie auf bessere Konnaszenztypen hin refaktorieren.

Architekten sollten die statische der dynamischen Konnaszenz vorziehen, weil Entwickler diese durch einfache Quellcodeanalyse ermitteln können. Moderne Werkzeuge machen eine Verbesserung der statischen Konnaszenz zu einem Kinderspiel. Nehmen wir zum Beispiel den Fall der *bedeutungsbasierten Konnaszenz*. Diese können Entwickler verbessern, indem sie auf eine *namensbasierte Konnaszenz* hin refaktorieren, zum Beispiel durch die Verwendung einer benannten Konstante anstelle einer »magischen« Variablen.

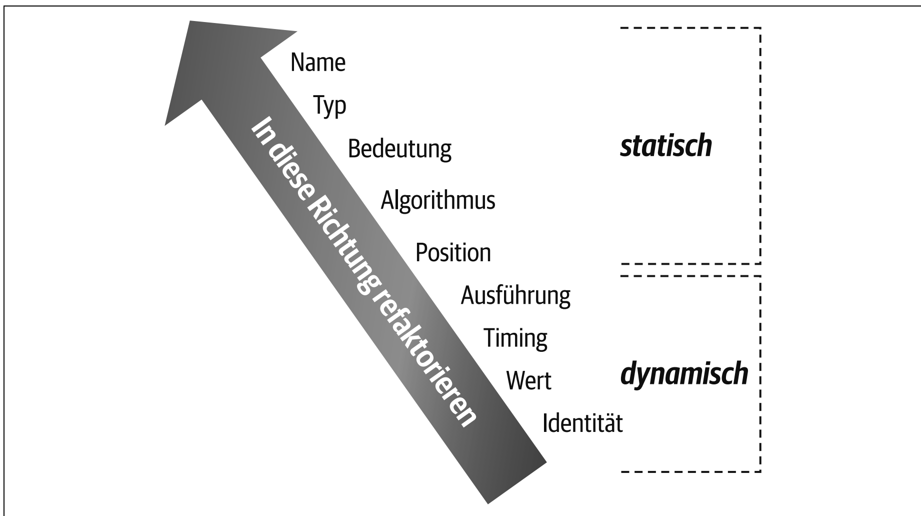

Abbildung 3-5: Die Stärke der Konnaszenz kann als Leitlinie zur Refaktorierung dienen.

Lokalität

Die *Lokalität* der Konnaszenz misst, wie nah sich die Module in der Codebasis sind. Naher Code (im gleichen Modul) besitzt typischerweise mehr und höhere Formen der Konnaszenz als »weiter entfernter« (in separaten Modulen oder

Codebasen befindlicher) Code. Anders gesagt: Formen der Konnaszenz, die bei großer Entfernung auf eine geringe Kopplung hinweisen, können gut näher beieinander sein. Haben zwei Klassen in der gleichen Komponente beispielsweise eine bedeutungsbasierte Konnaszenz, ist das weniger schädlich für die Codebasis, als wenn zwei Komponenten die gleiche Form der Konnaszenz hätten.

Entwickler müssen Stärke und Lokalität gemeinsam betrachten. Stärkere Formen der Konnaszenz im gleichen Modul sorgen für weniger schlecht riechenden Code als die Konnaszenz mit mehr Entfernung.

Grad

Der *Grad* der Konnaszenz bezieht sich auf die Größe ihrer Auswirkungen. Beeinflusst sie wenige Klassen oder viele? Ein geringerer Grad der Konnaszenz schadet einer Codebasis weniger. Anders gesagt: Eine hohe dynamische Konnaszenz ist nicht so schlimm, wenn nur ein paar Module verwendet werden. Codebasen haben jedoch die Tendenz, zu wachsen, wodurch kleine Probleme entsprechend größer werden können.

Page-Jones bietet drei Leitlinien für die Verwendung der Konnaszenz zur Verbesserung der Modularität von Systemen:

1. Minimiere die Gesamtkonnaszenz durch Aufteilung des Systems in verkapselte Elemente.
2. Minimiere die verbleibende Konnaszenz, die Verkapselungsgrenzen überschreitet.
3. Maximiere die Konnaszenz innerhalb der Verkapselungsgrenzen.

Der legendäre Innovator der Softwarearchitektur Jim Weirich hat das Konzept der Konnaszenz wieder populär gemacht und bietet zwei Ratschläge an:

> Gradregel: Wandle starke Formen der Konnaszenz in schwache Formen um.
>
> Lokalitätsregel: Je größer die Entfernung zwischen Softwareelementen, desto schwächere Formen der Konnaszenz sollten verwendet werden.

Kopplungs- und Konnaszenzmetriken vereinheitlichen

Bis jetzt haben wir Kopplung und Konnaszenz besprochen, Messgrößen aus verschiedenen Zeiten und mit unterschiedlichen Zielen. Aus Sicht eines Architekten überschneiden sich diese beiden Sichtweisen. Was Page-Jones als statische Konnaszenz bezeichnet steht für verschieden hohe Grade an eingehender und ausgehender Kopplung. Bei der strukturierten Programmierung kommt es nur auf das rein und raus an, während es bei der Konnaszenz darum geht, wie die Dinge miteinander verkoppelt sind. Abbildung 3-6 soll diese Überschneidung der Konzepte verdeutlichen.

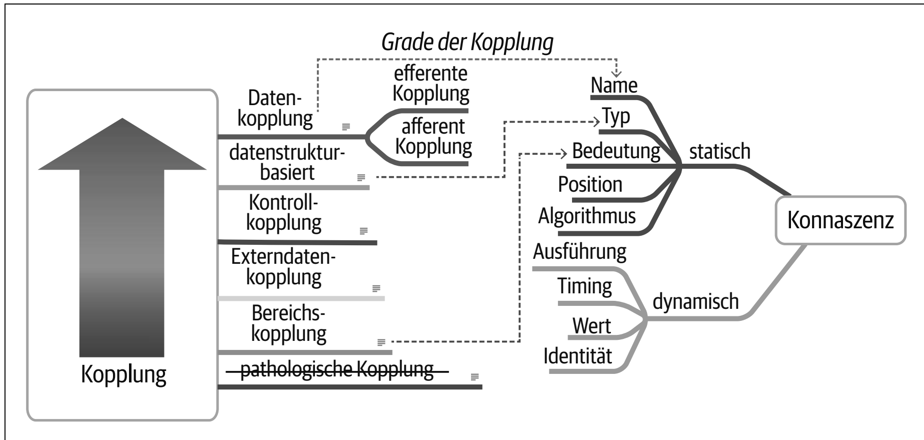

Abbildung 3-6: Einheitsdiagramm zu Kopplung und Konnaszenz

In Abbildung 3-6 stehen die Kopplungskonzepte der strukturierten Programmierung auf der linken Seite, während die Konnaszenzeigenschaften auf der rechten Seite zu sehen sind. Für das, was in der strukturierten Programmierung als Datenkopplung (Methodenaufrufe) bezeichnet wird, bietet die Konnaszenz Anhaltspunkte dazu, wie diese Kopplung umgesetzt werden sollte. Die Bereiche der dynamischen Konnaszenz werden von der strukturierten Programmierung nicht berührt. Wir werden dieses Konzept in Kürze in »Architektonische Quanten und Granularität« auf Seite 94 verkapseln.

Die Probleme mit der Konnaszenz der 1990er-Jahre

Architekten haben mit der Verwendung dieser sinnvollen Metriken zur Analyse und Erstellung von Systemen mehrere Probleme. Zum einen betrachten diese Messmethoden Details auf einer sehr niedrigen Codeebene und konzentrieren sich dabei eher auf Codequalität und Hygiene als auf eine architektonische Struktur. Architekten geht es eher darum, *wie* Module miteinandergekoppelt sind, als *wie sehr* sie gekoppelt sind. Einem Architekten ist es beispielsweise wichtiger, ob eine Kommunikation synchron oder asynchron abläuft, und weniger, wie diese implementiert ist.

Das zweite Problem mit der Konnaszenz ist die Tatsache, dass sie auf eine Grundentscheidung, die viele moderne Architekten treffen müssen, nicht eingeht: Soll in verteilten Architekturen wie Microservices synchrone oder asynchrone Kommunikation verwendet werden? Das erste Gesetz der Softwarearchitektur besagt, dass alles ein Kompromiss ist. Nachdem wir in Kapitel 7 besprochen haben, was alles zu den architektonischen Eigenschaften gehört, stellen wir Ihnen neue Wege vor, über moderne Konnaszenz nachzudenken.

Von Modulen zu Komponenten

Im Verlaufe dieses Buchs verwenden wir das Wort Modul als allgemeinen Begriff für die Zusammenfassung von verwandtem Code. Gleichzeitig unterstützen die meisten Plattformen eine Form der *Komponente*, einen der wichtigsten Grundbausteine für Softwarearchitekten. Das Konzept und die dazugehörige Analyse der logischen oder physischen Trennung existiert bereits seit den Anfangstagen der Informatik. Und trotz vielem Schreiben und Nachdenken über Komponenten und Trennung haben Entwickler und Architekten immer noch Schwierigkeiten, gute Ergebnisse zu erzielen.

Die Ableitung von Komponenten aus Problembereichen (»problem domains«) werden wir in Kapitel 8 besprechen. Zuvor müssen wir aber noch auf einen weiteren grundsätzlichen Aspekt der Softwarearchitektur eingehen: architektonische Eigenschaften und ihr Anwendungsbereich.