BE SOFTWARE ARCHITECTURE WISE · KAPITEL 3

Designprinzipien und Grundlagen

Jede Architektur baut auf denselben wenigen Ideen über Struktur auf: wie man ein System in Teile zerlegt, wie viel diese Teile voneinander wissen sollten, in welche Richtung ihre Abhängigkeiten zeigen sollten und wie man das Ganze so einfach hält, wie das Problem es erlaubt. Dieses Kapitel vermittelt diese Ideen auf jeder Ebene, von einer Python-Klasse über eine Komponente bis zu einer ganzen Flotte von Services: Modularität, Kopplung und Kohäsion, Information Hiding, Abhängigkeitsmanagement, SOLID, Connascence, das Gesetz von Conway, Einfachheit und die Metriken, mit denen man Struktur misst. Es eignet sich für Entwicklerinnen und Entwickler, die präzise über Design nachdenken wollen, und für Architekturfachleute, die ein gemeinsames Vokabular für Reviews brauchen.

bewiseapps.com/de/software-architecture-wise/design-principles/ · Version 2026-11-03. Nur Bildungsinformation, keine professionelle Beratung.

AUF EINEN BLICK

Das lernen Sie

- Das Kriterium von Parnas für die Zerlegung eines Systems in Module erklären und damit zwei mögliche Zerlegungen vergleichen.
- Abhängigkeiten anhand der klassischen Stufen von Kopplung und Kohäsion einordnen und zeitliche, Deployment- und semantische Kopplung zwischen Services erkennen.
- Information Hiding von Kapselung unterscheiden, eine Schnittstelle als Vertrag spezifizieren und stabile APIs entwerfen, die Leaky Abstractions und Hyrums Gesetz berücksichtigen.
- Instabilität, Abstraktheit und den Abstand von der Hauptreihe berechnen und sie zusammen mit dem Acyclic Dependencies Principle, dem Stable Dependencies Principle und dem Stable Abstractions Principle nutzen, um Abhängigkeitsstrukturen zu verbessern.
- Die SOLID-Prinzipien und die Kohäsionsprinzipien für Komponenten auf Klassen-, Komponenten- und Service-Ebene anwenden und erkennen, wann sie dogmatisch angewandt werden.
- Mit Connascence Stärke, Lokalität und Grad der Kopplung beschreiben und Refactorings wählen, die sie abschwächen.
- Das Gesetz von Conway, die Belege dafür und das Inverse Conway Maneuver erklären und die Kommunikationspfade eines Teams berechnen.
- Essenzielle von akzidenteller Komplexität trennen, die zyklomatische Komplexität berechnen und strukturelle Regeln in Fitness Functions umsetzen, ohne in die Falle von Goodharts Gesetz zu tappen.

Bevor Sie beginnen

Kapitel 0 bis 2, insbesondere Abschnitt 0.3 (Code organisieren), Abschnitt 1.7 (technische Schulden) und Abschnitt 2.6 (Änderbarkeit). Es hilft, kurze Python-Beispiele lesen zu können.

Dauer

Etwa 3 Stunden Lesezeit; 45 Minuten für den Test.

Schlüsselbegriffe

Modularität · Information Hiding · Kopplung · Kohäsion · afferente und efferente Kopplung · Abstraktion · Design by Contract · Hyrums Gesetz · Acyclic Dependencies Principle · Instabilität und Abstraktheit · SOLID · Connascence · Gesetz von Conway · akzidentelle Komplexität

Warum Prinzipien, und warum auf jeder Ebene

Kapitel 2 hat die Qualitäten beschrieben, die eine Architektur liefern muss. Die meisten davon, allen voran die Änderbarkeit, hängen weniger von einer bestimmten Technologie ab als von der Struktur: wie ein System in Teile zerlegt ist, was jeder Teil über die anderen weiß und in welche Richtung die Abhängigkeiten zwischen ihnen verlaufen. Ein Team kann das angesagteste Framework und die beste Cloud-Plattform wählen und trotzdem bei einem System landen, in

dem jede Änderung zehn Stellen berührt, weil die Teile an den falschen Stellen geschnitten wurden. Die Prinzipien in diesem Kapitel sind das gesammelte Wissen aus mehr als fünfzig Jahren darüber, wie man sie gut schneidet.

Es sind alte Ideen, und das ist eine Stärke. Information Hiding stammt aus dem Jahr 1972, Kopplung und Kohäsion aus der Arbeit am Structured Design Ende der 1960er-und in den 1970er-Jahren, das Gesetz von Conway aus dem Jahr 1968. Sie haben überdauert, weil sie etwas beschreiben, das sich nicht ändert: Die Kosten einer Änderung hängen davon ab, wie weit sich die Änderung ausbreiten muss. Geändert hat sich die Ebene, auf der Architekturfachleute sie anwenden. Dasselbe Prinzip, das Ihnen rät, die Felder einer Klasse privat zu halten, rät Ihnen, die Datenbank eines Service privat zu halten; dieselbe Überlegung, die einen Abhängigkeitszyklus zwischen zwei Paketen verbietet, verbietet zwei Services, die sich nur gemeinsam releasen lassen. Jeder Abschnitt zeigt daher jede Idee zweimal: in einem kleinen Stück Code und auf der Ebene von Komponenten und Services.

Wie das Kapitel aufgebaut ist

Die ersten vier Abschnitte legen die Grundlagen. Abschnitt 3.1 fragt, was ein Modul ist und wo seine Grenzen verlaufen sollten. Er zeigt an David Parnas' KWIC-Beispiel, warum eine Zerlegung nach Verarbeitungsschritten scheitert und eine Zerlegung nach wahrscheinlichen Änderungen gelingt, und unterscheidet logische Module von den physischen Einheiten, die gebaut und deployt werden. Abschnitt 3.2 liefert das Vokabular, um Grenzen zu beurteilen: die klassischen Stufen von Kopplung und Kohäsion, afferente und efferente Kopplung sowie die Arten von Kopplung, die nur zwischen Services auftreten. Abschnitt 3.3 betrachtet die Schnittstelle selbst: Information Hiding im Vergleich zur Kapselung, Verträge, Leaky Abstractions, tiefe Module und Hyrums Gesetz. Abschnitt 3.4 wendet sich der Richtung zu: Robert C. Martins Prinzipien für Abhängigkeiten zwischen Komponenten, die Metriken Instabilität und Abstraktheit, wie man Zyklen auflöst, Schichtenbildung und wie man all das automatisch prüft.

Die zweiten vier Abschnitte bauen darauf auf. Abschnitt 3.5 behandelt die fünf SOLID-Prinzipien und die Kohäsionsprinzipien für Komponenten, jeweils im Code und zwischen Services gezeigt, zusammen mit der Kritik, die sie auf sich ziehen, wenn man sie mechanisch anwendet. Abschnitt 3.6 führt Connascence ein, ein feineres Maß für Kopplung mit klaren Regeln, wie man sie schwächer und lokaler macht. Abschnitt 3.7 weitet den Blick auf die Organisation: das Gesetz von Conway, die Belege dafür, die Arithmetik der Kommunikationspfade und das Inverse Conway Maneuver. Abschnitt 3.8 schließt mit Einfachheit und Messung: essenzielle und akzidentelle Komplexität, KISS, YAGNI und DRY richtig verstanden, der Preis dafür, Optionen offenzuhalten, Strukturmetriken wie die zyklomatische Komplexität und die Grenzen aller Metriken.

Eine Idee unter allen

Zusammen gelesen ergeben die acht Abschnitte ein einziges Argument. Änderungen sind der dominierende Kostenfaktor von Software, also ist eine gute Struktur eine, in der wahrscheinliche Änderungen lokal bleiben. Jedes Prinzip in diesem Kapitel ist ein Weg, das zu erreichen. Parnas sagt: Bringen Sie jede wahrscheinliche Änderung in einem Modul unter. Kopplung und Kohäsion sagen: Halten Sie zusammen, was sich gemeinsam ändert, und trennen Sie, was sich getrennt ändert. Information Hiding sagt: Sorgen Sie dafür, dass das, was sich ändern kann, von außen unsichtbar ist. Die Abhängigkeitsprinzipien sagen: Lassen Sie Abhängigkeiten auf das zeigen, was sich am wenigsten ändert. Connascence sagt: Halten Sie jede starke Übereinkunft zwischen zwei Codestücken so nah beieinander wie möglich. Das Gesetz von Conway ergänzt, dass Menschen Teil der Struktur sind: Eine Änderung, für die sich zwei Teams einigen müssen, ist nicht lokal, wie auch immer der Code angeordnet ist.

Dieses Argument erklärt auch, warum die Prinzipien manchmal in Konflikt geraten und warum keines absolut gilt. Das Entfernen von Duplikation kann die Kopplung erhöhen; Schnittstellen, die eine Abhängigkeit umkehren, können ein Modul flacher machen; das Aufteilen einer Komponente zugunsten der Kohäsion kann einen Netzwerksprung erzeugen. Die professionelle Fähigkeit besteht nicht darin, jedes Prinzip überall anzuwenden, sondern für eine bestimmte Änderung, die in einem bestimmten System wahrscheinlich ist, zu fragen, wie weit sie sich ausbreiten müsste und welches Prinzip diese Strecke zu den geringsten Kosten verkürzt.

Wie Sie dieses Kapitel nutzen

Wer neu im Design ist, kann die Abschnitte der Reihe nach lesen; jeder beginnt mit einer kurzen Erklärung aus dem Alltag und verweist auf Abschnitt 0.3, wo die Grundlagen zuerst eingeführt wurden. Erfahrene Fachleute möchten vielleicht mit den ausgearbeiteten Beispielen beginnen: Die Zerlegung des Onlineshops in Abschnitt 3.1, das Refactoring von Bestellung und Abrechnung in Abschnitt 3.2, die Zahlungsschnittstelle in Abschnitt 3.3 und die Metrikberechnung in Abschnitt 3.4 sind dafür gedacht, dass man darüber streitet. Die Prinzipien hier werden im ganzen restlichen Kurs verwendet: Die Architekturstile der Kapitel 5 und 6 sind weitgehend unterschiedliche Antworten auf die Fragen, wo Grenzen verlaufen und in welche Richtung Abhängigkeiten zeigen, und Kapitel 11 macht aus den hier eingeführten automatisierten Prüfungen eine Governance-Praxis mithilfe von Fitness Functions.

Modularität: ein System in Teile zerlegen

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.3 Code organisieren
- 2.6 Änderbarkeit und Deploybarkeit

Begriffe von früher, die er verwendet: Abhängigkeit, Bibliothek, Monolith, Architekturerosion, Fitness Function

DIE IDEE IN KÜRZE

Denken Sie an ein großes Krankenhaus. Es ist kein riesiger Raum, in dem alle alles tun; es ist in Abteilungen gegliedert: Radiologie, Apotheke, die Geburtsstation. Jede hat ihr eigenes Personal und ihre eigene Arbeitsweise, und andere haben über einen Empfang mit klaren Anforderungsformularen mit ihr zu tun. Wenn die Apotheke ihr Lagersystem ändert, merken die Stationen nichts davon, weil sie weiterhin dasselbe Formular ausfüllen. Software wird auf die gleiche Weise in Module gegliedert. Die Kunst besteht darin, die Wände so zu setzen, dass die erwarteten Änderungen innerhalb einer Abteilung bleiben.

WAS SIE WISSEN MÜSSEN

- Ein Modul ist eine Zerlegungseinheit mit einer Verantwortlichkeit und einer Schnittstelle: eine Funktion, Klasse, ein Paket, eine Komponente oder ein Service, je nachdem, auf welcher Ebene Sie arbeiten.
- Modularität zahlt sich aus durch Verständlichkeit, parallele Arbeit von Teams, Isolierung von Änderungen, Wiederverwendung und Tests, aber nur, wenn die Grenzen an den richtigen Stellen gezogen werden.
- David Parnas (1972) zeigte am KWIC-Beispiel, dass eine Zerlegung nach Verarbeitungsschritten jede Designentscheidung über viele Module verteilt, und schlug stattdessen vor, dass jedes Modul eine Entscheidung verbirgt, die schwierig ist oder sich wahrscheinlich ändert.
- Logische Module (Verantwortungseinheiten im Design) unterscheiden sich von physischen Modulen (Einheiten für Build und Deployment); dasselbe logische Design kann als eine Einheit oder als viele Services deployt werden.

- Granularität ist eine Abwägung: Zu große Module vermischen voneinander unabhängige Änderungen, zu kleine Module vervielfachen Schnittstellen, Koordination und über ein Netzwerk hinweg auch Latenz und Ausfallarten.
- Grenzen, die nicht durchgesetzt werden, erodieren; deshalb nutzen Fachleute die Sichtbarkeitsregeln der Sprache, Modulsysteme wie das Java Platform Module System, Module im Build-Werkzeug und automatisierte Abhängigkeitsprüfungen.

Was ein Modul ist

Ein Modul ist ein Teil eines Systems mit einer klaren Verantwortlichkeit und einer festgelegten Art, wie andere Teile es nutzen. Diese Definition ist bewusst unabhängig von der Größe. In Abschnitt 0.3 war ein Modul eine Python-Datei; in diesem Kapitel umfasst das Wort jede Einheit der Zerlegung, also des Aufteilens eines Systems in Teile: eine Funktion, eine Klasse, ein Paket, eine Bibliothek, eine Komponente innerhalb einer Anwendung oder einen separat deployten Service. Auf jeder Ebene stellen sich dieselben Fragen: Wofür ist dieser Teil verantwortlich, was bietet er anderen an, was behält er für sich, und wovon hängt er ab?

Jedes Modul hat zwei Gesichter. Seine Schnittstelle ist das, was andere Module wissen müssen, um es zu nutzen: Funktionsnamen und Parameter, eine HTTP-API, die Ereignisse, die es veröffentlicht, und auch die informellen Zusagen, etwa wie schnell es antwortet und welche Fehler es zurückgeben kann. Seine Implementierung ist alles andere: der Code, die Datenstrukturen, die Datenbanktabellen, die Drittanbieter-Bibliotheken, die es zufällig verwendet. Gute Modularität hält die Schnittstelle klein und stabil und lässt die Implementierung sich frei ändern.

David Parnas, dessen Artikel von 1972 die Grundlage dieses Abschnitts ist, definierte ein Modul als *Zuweisung einer Verantwortlichkeit* statt als Stück Programmtext. Für Architekturfachleute ist das die nützlichste Sichtweise. Ein Modul ist ein Versprechen, dass ein Teil des Systems, und meist ein Team, für eine bestimmte Menge von Entscheidungen verantwortlich ist. Ob der Code am Ende in einer Datei oder in vierzig liegt, in einem Prozess oder in zehn, ist eine separate Frage, die weiter unten unter logischen und physischen Modulen behandelt wird.

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist

Stellen Sie sich eine Essensliefer-App vor. Irgendwo in ihrem Code gibt es Logik, um Speisekarten anzuzeigen, Zahlungen entgegenzunehmen, Kuriere zu verfolgen und Benachrichtigungen zu senden. Modularität bedeutet einfach, dass jede dieser Aufgaben in einem eigenen, klar abgegrenzten Teil lebt, mit einer kleinen, vereinbarten Menge von Zugängen. Der Rest dieses Abschnitts handelt davon, wo man die Wände setzt und wie man verhindert, dass jemand darüber klettert.

Warum überhaupt modularisieren

Ein System aus ein paar hundert Zeilen lässt sich als Ganzes verstehen. Ein System aus ein paar Millionen Zeilen nicht, und auch die Organisation, die es baut, kann nicht als eine ungeteilte Gruppe arbeiten. Modularität ist die Eigenschaft, in Module gegliedert zu sein, die sich weitgehend für sich allein verstehen, ändern und durchdenken lassen. Sie ist eines der fünf Teilmerkmale der Wartbarkeit in ISO/IEC 25010 (Abschnitt 2.6) und unterstützt mindestens fünf verschiedene Vorteile.

Verständlichkeit. Eine Entwicklerin oder ein Entwickler kann ein Modul und seine Schnittstellen studieren, ohne den Rest des Systems zu lesen. **Parallele Arbeit.** Teams können gleichzeitig an verschiedenen Modulen arbeiten und müssen sich kaum abstimmen, weil sie die Schnittstellen vorab vereinbart haben. **Isolierung von Änderungen.** Eine Änderung an einer Designentscheidung, etwa welche Steuerregeln gelten, erfolgt in einem Modul und breitet sich nicht nach außen aus. **Wiederverwendung.** Ein gut abgegrenztes Modul, etwa eine Komponente zur Adressvalidierung, kann von mehreren Produkten genutzt werden. **Tests.** Ein Modul mit einer klaren Schnittstelle lässt sich für sich allein testen, wobei seine Abhängigkeiten durch Test Doubles ersetzt werden.

Parnas nannte 1972 drei davon: den organisatorischen Vorteil (kürzere Entwicklungszeit, weil getrennte Gruppen mit wenig Kommunikation an getrennten Modulen arbeiten), Flexibilität (drastische Änderungen an einem Modul, ohne andere zu ändern) und Verständlichkeit (das System Modul für Modul studieren). Ein halbes Jahrhundert später sind das immer noch die Gründe. Beachten Sie aber, dass keiner davon automatisch daraus folgt, dass man ein System in Stücke schneidet. Ein System kann in fünfzig Module gegliedert sein und trotzdem keinen dieser Vorteile liefern, wenn jede Änderung zehn davon berührt. Die Vorteile hängen ganz davon ab, *wo* die Grenzen gezogen werden.

Trennung der Anliegen (Separation of Concerns)

Edsger Dijkstra gab dieser Idee 1974 in einem kurzen Essay ihren Namen: „On the role of scientific thought“ (EWD 447). Er beschrieb eine Denkweise, bevor er eine Bauweise beschrieb: Sie untersuchen einen Aspekt eines Problems gründlich, etwa ob ein Programm korrekt ist, und wissen dabei, dass es andere gibt, etwa ob es effizient ist oder ob es überhaupt jemand will. Er nannte das, „zugleich ein- und mehrgleisig zu denken“, und räumte ein, dass die Trennung nie perfekt ist. Ein *Anliegen* (Concern) ist schlicht etwas, worüber Sie nachdenken müssen.

Im Softwareentwurf bedeutet **Separation of Concerns** (Trennung der Anliegen), ein System so anzuordnen, dass jeder Teil ein Anliegen behandelt und verstanden und geändert werden kann, ohne die anderen im Kopf zu behalten. Auf der Essenslieferplattform sind die Preisberechnung einer Bestellung, die Schätzung der Lieferzeit, die Zahlungsabwicklung und das Senden von Benachrichtigungen getrennte Anliegen. Sind sie in einer langen Funktion verheddert, bedeutet eine Änderung am Wortlaut der Benachrichtigung, die Preisregeln zu lesen, und ein Preistest muss eine Zahlung vorbereiten. Getrennt lässt sich jedes für sich durchdenken, testen und ersetzen. Dasselbe Denken steckt hinter vielen vertrauten Strukturen: HTML für den Inhalt und CSS für die Darstellung, Model und View, eine Geschäftsregel getrennt von der Datenbank, die ihr Ergebnis speichert.

Es ist das breiteste der Prinzipien in diesem Kapitel, und die anderen schärfen es. Modularität liefert die Einheiten. Information Hiding (Abschnitt 3.3) sagt, wo die Linie zu ziehen ist: um Entscheidungen, die sich wahrscheinlich ändern. Das Single Responsibility Principle (Abschnitt 3.5) sagt, wessen Bedürfnissen ein Teil dienen sollte.

Zwei Vorbehalte gelten. Erstens lassen sich manche Anliegen nicht an einem Ort festhalten. Logging, Autorisierung und Wiederholungen berühren fast alles und heißen daher **Querschnittsanliegen (Cross-Cutting Concerns)**; sie werden meist von gemeinsamer Middleware, einem Gateway oder einem Service Mesh übernommen (Abschnitt 8.5), damit der Geschäftscode sie nicht wiederholt. Zweitens kann die falsche Trennung alles verschlimmern. Eine strikte Aufteilung in technische Schichten trennt die Benutzeroberfläche von den Daten, verstreut aber ein fachliches Anliegen wie das Trinkgeld für Kuriere über jede Schicht, sodass ein Feature sie alle berührt. Deshalb organisiert Abschnitt 5.3 Module nach fachlichen Fähigkeiten. Fragen Sie, welche Anliegen sich unabhängig voneinander ändern, und trennen Sie diese.

Parnas und das KWIC-Beispiel

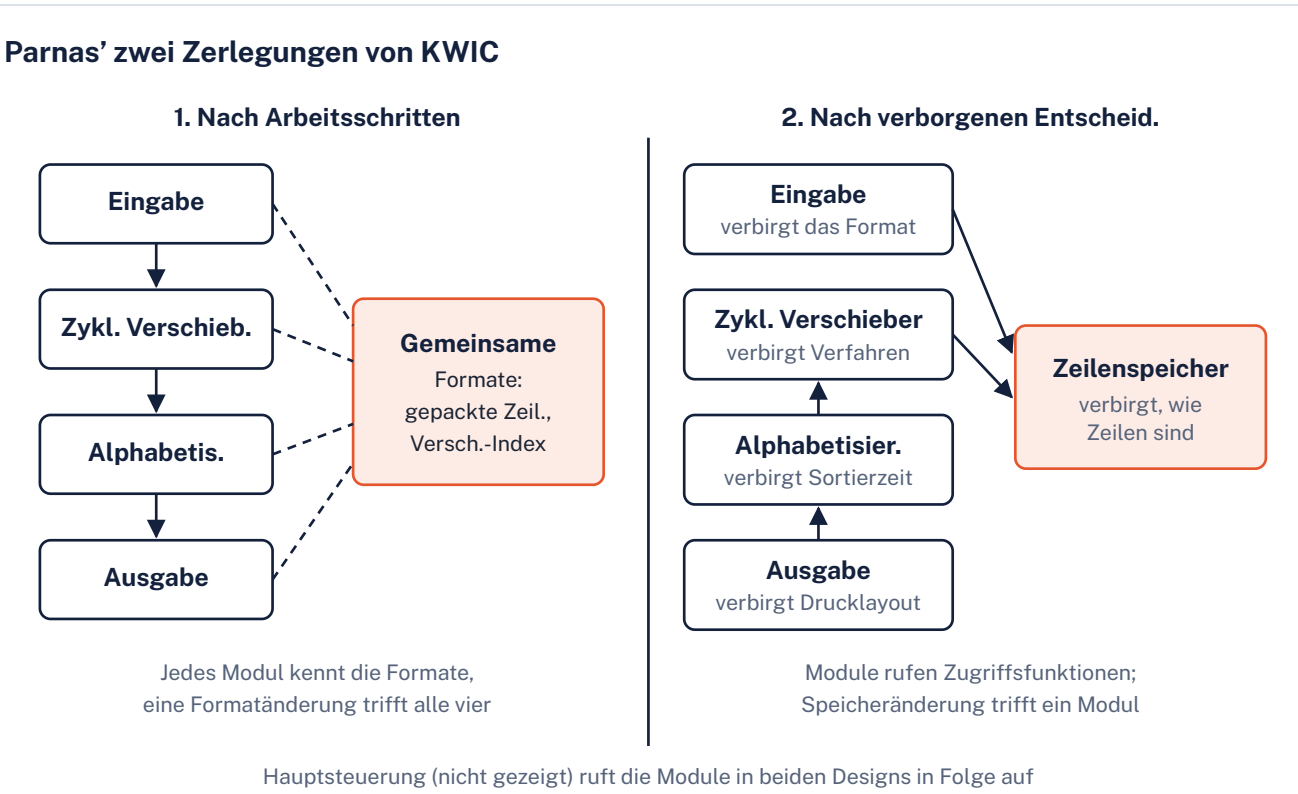
Parnas veranschaulichte das an einem kleinen Programm namens KWIC (Key Word in Context), das einen Index von Titeln erzeugt. Es liest Textzeilen ein, erzeugt jede zyklische Verschiebung jeder Zeile (indem es wiederholt das erste Wort ans Ende stellt), sortiert alle Verschiebungen

alphabetisch und gibt sie aus. Die Zeile `Design of Software` ergibt `Design of Software`, `of Software Design` und `Software Design of`, sodass eine Leserin oder ein Leser den Titel unter jedem seiner Wörter finden kann.

Die erste Zerlegung, die er beschreibt, ist die, die die meisten Programmierenden beim Zeichnen eines Flussdiagramms erzeugen würden: ein Eingabemodul, das die Zeilen liest und in einem gepackten Format im Speicher ablegt, ein Modul für die zyklische Verschiebung, das einen Index der Startpositionen jeder Verschiebung aufbaut, ein Alphabetisierungsmodul, das diesen Index sortiert, ein Ausgabemodul, das das Ergebnis ausgibt, und ein Hauptsteuermodul, das sie der Reihe nach aufruft. Das ist ein völlig vernünftiges Design. Jedes Modul ist klein, und die Schnittstellen zwischen ihnen sind die Datenformate im Speicher.

Die zweite Zerlegung wirkt seltsamer. Sie hat ein Zeilenspeichermodul, das Funktionen wie *liefere das Zeichen in Zeile r, Wort w, Position c* anbietet und verbirgt, wie die Zeilen tatsächlich gespeichert sind; ein Eingabemodul, das Zeilen liest und an den Zeilenspeicher übergibt; einen zyklischen Verschieber, der die Verschiebungen über ähnliche Zugriffsfunktionen so präsentiert, *als ob* sie gespeichert wären; einen Alphabetisierer, der *gib mir die i-te Verschiebung in der Reihenfolge* anbietet, ohne zu sagen, wann oder wie er sortiert; ein Ausgabemodul und eine Hauptsteuerung. Die Module entsprechen nicht mehr den Schritten der Verarbeitung.

Parnas listete dann Entscheidungen auf, die fragwürdig waren und sich wahrscheinlich ändern würden: das Eingabeformat; alle Zeilen im Speicher zu halten, was bei sehr großen Eingaben unpraktikabel ist; mehrere Zeichen in jedes Maschinenwort zu packen; einen Index der Verschiebungen statt der Verschiebungen selbst zu speichern; und alles einmal zu sortieren, statt bei Bedarf zu suchen. In der ersten Zerlegung betreffen die meisten dieser Änderungen mehrere Module, weil jedes Modul die gemeinsamen Datenformate kennt. In der zweiten bleibt jede Änderung auf das eine Modul beschränkt, das diese Entscheidung verbirgt. Das zweite Design ließ sich auch leichter parallel entwickeln, weil sich die Teams nur auf Funktionsnamen und Parameter einigen mussten, nicht auf detaillierte Tabellenlayouts, und es ließ sich leichter Modul für Modul verstehen.



In der Flussdiagramm-Zerlegung teilen alle Module die Datenformate; in der Information-Hiding-Zerlegung verbirgt jedes Modul eine Entscheidung hinter Zugriffsfunktionen.

Das Kriterium: nach Entscheidungen zerlegen, die sich wahrscheinlich ändern

Parnas kam zu dem Schluss, dass es „fast immer falsch“ ist, die Zerlegung eines Systems auf der Grundlage eines Flussdiagramms zu beginnen. Stattdessen schlug er vor, mit einer Liste schwieriger Designentscheidungen oder solcher, die sich wahrscheinlich ändern, zu beginnen und jedes Modul so zu entwerfen, dass es eine solche Entscheidung vor den anderen verbirgt. Er nannte das Prinzip Information Hiding; Abschnitt 3.3 behandelt es ausführlich.

Die Idee klingt einfach, kehrt aber den Instinkt der meisten von uns um. Ein Flussdiagramm beschreibt *Zeit*: Erst passiert dies, dann das. Designentscheidungen liegen quer zur Zeit. Die Entscheidung, wie eine Bestellung dargestellt wird, spielt eine Rolle, wenn die Bestellung eingeht, bepreist, bezahlt, gespeichert und bestätigt wird. Folgen die Module den Schritten, ist diese eine Entscheidung über alle verteilt, und wer sie ändert, muss alle ändern. Besitzt ein Modul die Entscheidung, fragen alle Schritte dieses Modul, und die Entscheidung lässt sich an einer Stelle ändern.

In der Praxis kommt es auf zwei Präzisierungen an. Erstens geht es beim Kriterium um *wahrscheinliche* Änderungen; Architekturfachleute müssen also eine fundierte Vermutung über die Zukunft anstellen: Was hat sich in ähnlichen Systemen geändert, was plant das Unternehmen, welche Lieferanten und Vorschriften sind instabil? Die vier Fragen zur

Änderbarkeit aus Abschnitt 2.6 (was ändert sich, wie wahrscheinlich, wer ändert es, zu welchen Kosten) liefern genau die Eingaben, die dafür nötig sind. Zweitens unterschied Parnas eine gute Zerlegung von einer Hierarchie. Ein System kann sauber geschichtet und trotzdem schlecht zerlegt sein, und umgekehrt; Abschnitt 3.4 behandelt die Richtung von Abhängigkeiten gesondert.

WOHER DAS KOMMT

Woher das kommt

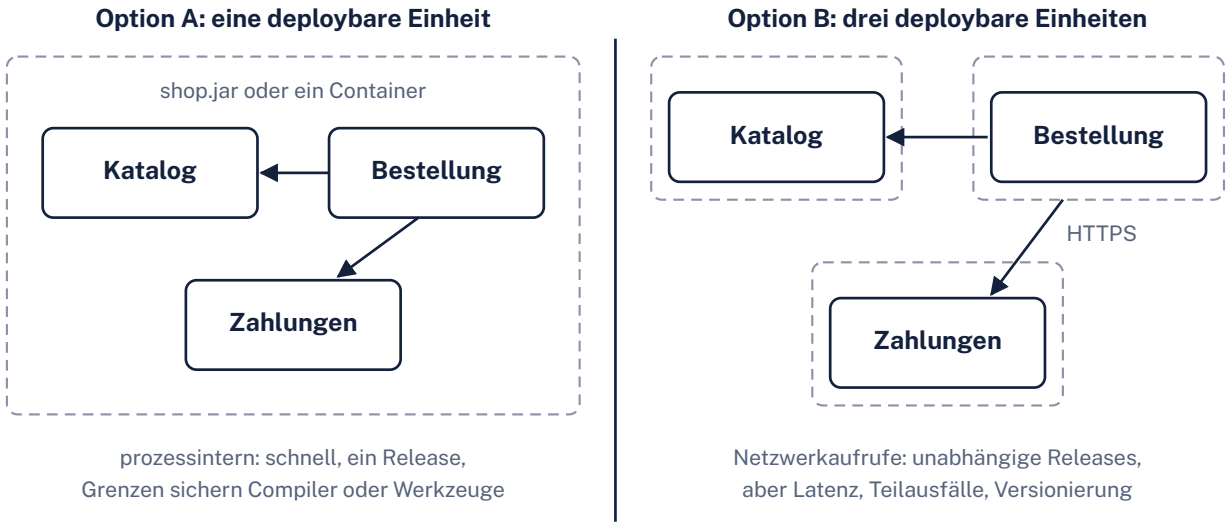
Parnas' Artikel *On the Criteria To Be Used in Decomposing Systems into Modules* erschien im Dezember 1972 in den *Communications of the ACM* (Band 15, Heft 12). Damals wurde modulare Programmierung allgemein empfohlen, aber es gab kaum Hinweise, wie man die Module auswählt. Der Artikel ist nur sechs Seiten lang, sehr gut lesbar und bis heute einer der meistzitierten in der Softwaretechnik. Parnas wies sogar auf einen Fehler in seinem eigenen zweiten Design hin: Die Schnittstelle des zyklischen Verschiebers legte die Reihenfolge der Verschiebungen fest und gab damit mehr preis als nötig.

Logische und physische Module

Ein logisches Modul ist eine Verantwortungseinheit im Design: Katalog, Bestellungen, Zahlungen. Ein physisches Modul ist eine Einheit, die gebaut, versioniert oder deployt wird: eine Java-JAR-Datei, ein Python-Wheel, ein npm-Paket, ein Container-Image, eine Serverless-Funktion. Abschnitt 1.1 nannte diese Modulstrukturen und Zuordnungsstrukturen. Beide hängen zusammen, sind aber nicht dasselbe, und wer sie verwechselt, begeht einige der teuersten Fehler moderner Architektur.

Dasselbe logische Design lässt sich auf sehr unterschiedliche Weise auf physische Module abbilden. Alle logischen Module können in einer deploybaren Einheit leben und sich prozessintern gegenseitig aufrufen: ein modularer Monolith. Oder jedes kann als eigener Service deployt werden und die anderen über das Netzwerk aufrufen: Microservices. Die logischen Grenzen und die Entscheidungen, die jedes Modul verbirgt, können in beiden Fällen identisch sein. Was sich ändert, sind die Kosten, eine Grenze zu überqueren (ein Funktionsaufruf dauert Nanosekunden, ein Netzwerkaufruf Millisekunden und kann fehlschlagen), die Möglichkeit, Teile unabhängig zu releasen und zu skalieren, und der Betriebsaufwand.

Ein logisches Design, zwei physische Designs



Gleiche logische Module und Abhängigkeiten; andere physische (Deployment-)Module

Logische Module beschreiben Verantwortlichkeiten; physische Module beschreiben, was gebaut und deployt wird, und dasselbe logische Design lässt sich auf beide Arten verpacken.

Ein häufiger und kostspieliger Fehler ist es, physische Trennung so zu behandeln, als würde sie logische Modularität schaffen. Wer ein schlecht zerlegtes System in Services aufteilt, verbirgt damit keine einzige Entscheidung; er macht aus prozessinternen Aufrufen zwischen verflochtenen Modulen Netzwerkaufrufe zwischen verflochtenen Services, was manchmal als verteilter Monolith bezeichnet wird. Auch der umgekehrte Fehler kommt vor: Ein Team behält eine deploybare Einheit, lässt aber ihre internen Module ineinanderwachsen, sodass es sie später nie mehr trennen kann. Der praktische Rat vieler Fachleute lautet, zuerst die logischen Grenzen richtig zu ziehen, sie durchzusetzen und erst dann zu entscheiden, welche davon es verdienen, zu physischen Grenzen zu werden.

Granularität und ihre Kosten

Wie groß sollte ein Modul sein? Granularität ist die Größe von Modulen im Verhältnis zum System, und es gibt keine allgemeingültige richtige Antwort, weil die Kosten in entgegengesetzte Richtungen ziehen. Größere Module haben weniger Schnittstellen, die man entwerfen und pflegen muss, aber jedes enthält mehr voneinander unabhängige Änderungsgründe, mehr Personen, die gleichzeitig daran arbeiten, und mehr, was man verstehen muss, bevor man etwas ändert. Kleinere Module sind jeweils leichter zu verstehen, aber das System hat mehr Schnittstellen, mehr Abhängigkeiten, die verwaltet werden müssen, und mehr Stellen, an denen ein einzelnes Feature zusammengeflickt werden muss.

Die Kosten einer Schnittstelle hängen auch von der Art der Grenze ab. Zwischen zwei Klassen in einem Programm kostet eine zusätzliche Schnittstelle ein wenig Lesezeit. Zwischen zwei Services kostet sie einen Netzwerksprung, Serialisierung, eine Versionierungsrichtlinie, Monitoring, eine Ausfallart und möglicherweise die Zustimmung eines zweiten Teams zu jeder Änderung. Deshalb ist die richtige Granularität für Services meist viel gröber als die richtige Granularität für Klassen.

Symptome falscher Granularität

Zu grob	Zu fein
Voneinander unabhängige Features ändern dasselbe Modul; viele Personen bearbeiten dieselben Dateien und geraten in der Versionskontrolle in Konflikt.	Ein einzelnes kleines Feature erfordert abgestimmte Änderungen an vielen Modulen oder Services.
Lange Build- und Testzeiten bei jeder Änderung; niemand versteht das ganze Modul.	Viele triviale durchreichende Schnittstellen, die keine Abstraktion hinzufügen; Aufrufketten über mehrere Module hinweg.
Teams stehen für ein Release Schlange, weil alles zusammen ausgeliefert wird.	Über ein Netzwerk: hohe Latenz durch gesprächige Aufrufe, Teilausfälle und verteilte Transaktionen.
Der Name des Moduls ist vage, etwa `core`, `common` oder `manager`.	Jedes Modul ist flach: Seine Schnittstelle ist fast so komplex wie seine Implementierung (siehe Abschnitt 3.3).

Modulgrenzen im Code durchsetzen

Eine Grenze, die nur in einem Diagramm existiert, übersteht keinen Termindruck. Früher oder später importiert jemand eine interne Klasse, weil es schneller geht, und sobald es eine Abkürzung gibt, folgen weitere; Abschnitt 1.7 nannte das Erosion. Sprachen und Build-Werkzeuge bieten mehrere Möglichkeiten, die beabsichtigte Grenze zum Weg des geringsten Widerstands zu machen oder dafür zu sorgen, dass ihr Überschreiten den Build scheitern lässt.

Python setzt auf Konventionen: Namen mit führendem Unterstrich gelten als intern (PEP 8), und die `__init__.py` eines Pakets kann nur seine öffentlichen Funktionen re-exportieren. Nichts verhindert einen Import von `shop.payments._stripe_client`, daher setzen Teams, denen das wichtig ist, ein Prüfwerkzeug wie import-linter ein, das den Build scheitern lässt, sobald ein verbotener Import auftaucht. Java bietet Package-private-Sichtbarkeit (der

Standard, wenn kein Modifizierer angegeben ist) und seit Java 9 im Jahr 2017 das Java Platform Module System: Eine Datei `module-info.java` deklariert, welche Module dieses Modul benötigt (`requires`) und welche seiner Pakete es exportiert (`exports`); nicht exportierte Pakete lassen sich von außen nicht nutzen, weder zur Kompilierzeit noch zur Laufzeit. Go behandelt jedes Paket unterhalb eines Verzeichnisses namens `internal` so, dass es nur von Code importiert werden kann, der im übergeordneten Verzeichnis dieses Verzeichnisses wurzelt. C# hat `internal` , sichtbar nur innerhalb einer Assembly; Rust hat `pub(crate)` .

```
# shop/payments/__init__.py  (the public face of the package)
from shop.payments.api import charge, refund, PaymentResult

__all__ = ["charge", "refund", "PaymentResult"]

# shop/payments/_provider_client.py   internal: leading underscore
# shop/payments/_retry.py             internal
# shop/payments/api.py                the only module callers should use

# Elsewhere in the shop:
from shop.payments import charge          # fine
from shop.payments._retry import backoff  # works, but breaks the boundary
```

Ein Python-Paket, das eine kleine Schnittstelle veröffentlicht und seine Implementierungsmodule per Konvention intern hält.

```
// module-info.java for the payments module
module shop.payments {
    requires shop.orders.api;        // what this module depends on
    exports shop.payments.api;       // the only package others can use
    // shop.payments.provider and shop.payments.retry are not exported,
    // so code in other modules cannot compile against them
}
```

Dieselbe Grenze in Java, durchgesetzt von Compiler und Laufzeitumgebung über einen Moduldeskriptor.

Möglichkeiten, eine Grenze durchzusetzen

Mechanismus	Beispiele	Stärke
Namenskonvention	Führender Unterstrich in Python; `internal` im Namen eines Namensraums	Schwach: setzt auf Disziplin und Code-Review
Sichtbarkeit in der Sprache	Package-private in Java, `internal` in C#, `pub(crate)` in Rust, `internal`-Verzeichnisse in Go	Stark innerhalb eines Builds; vom Compiler geprüft
Modulsystem der Sprache	Java Platform Module System (`requires`, `exports`); das Feld `exports` in einer Node.js-`package.json`	Stark: nicht deklarierter Zugriff scheitert beim Kompilieren oder Laden
Module im Build-Werkzeug	Multi-Modul-Builds mit Maven oder Gradle; Sichtbarkeitsregeln in Bazel; separate Pakete in einem Monorepo	Stark: Ein Modul kann nur nutzen, was es als Abhängigkeit deklariert
Automatisierte Architekturprüfungen	import-linter, ArchUnit, dependency-cruiser (Abschnitt 3.4)	Stark, wenn sie in der Build-Pipeline als Fitness Function laufen
Prozess- und Netzwerkgrenze	Separate Services mit eigenen Datenspeichern, nur über eine API erreichbar	Am stärksten, aber am teuersten zu überqueren und zu ändern

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: einen Onlineshop nach Flussdiagramm und nach wahrscheinlichen Änderungen zerlegen

Der Checkout eines Onlineshops wird zuerst als Flussdiagramm entworfen: Bestellung entgegennehmen, Warenkorb validieren, Preis berechnen (Aktionen und Mehrwertsteuer), Zahlung entgegennehmen, Bestellung speichern, Bestätigung senden. Alle sechs Schritte lesen und schreiben einen gemeinsamen Bestelldatensatz, ein Dictionary, dessen Schlüssel jeder Schritt kennt. Das alternative Design hat sechs Module, die jeweils eine Entscheidung verbergen: Bestellung (wie eine Bestellung dargestellt und gespeichert wird), Preisgestaltung (Aktionsregeln), Steuer (Mehrwertsteuerregeln), Zahlungsgateway (welcher Anbieter und wie er aufgerufen wird), Benachrichtigungen (Kanäle und Vorlagen) und ein Checkout-Workflow, der nur die Reihenfolge der Aufrufe festlegt, wie Parnas' Hauptsteuerung. Fünf Änderungen, die das Unternehmen im nächsten Jahr erwartet, werden für beide Designs durchgezählt.

Änderung 1: einen zweiten Zahlungsanbieter hinzufügen	Flussdiagramm: Zahlung entgegennehmen sowie Bestellung speichern, das das Format der Transaktionsreferenz des Anbieters speichert = 2 Module. Nach Änderungen: Zahlungsgateway = 1.
Änderung 2: Die Mehrwertsteuerregeln für eine Produktkategorie ändern sich	Flussdiagramm: Preis berechnen, Bestellung speichern (Mehrwertsteuerspalten) und Bestätigung senden (Aufschlüsselung der Mehrwertsteuer) = 3. Nach Änderungen: Steuer = 1.
Änderung 3: die Lieferadresse in strukturierte Felder aufteilen	Flussdiagramm: Jeder Schritt liest den gemeinsamen Bestelldatensatz = 6. Nach Änderungen: Bestellung = 1 (die anderen rufen ihre Methoden auf).
Änderung 4: SMS-Bestätigungen hinzufügen	Flussdiagramm: Bestätigung senden = 1. Nach Änderungen: Benachrichtigungen = 1.
Änderung 5: neues Feature, eine Grußbotschaft zu Bestellungen	Flussdiagramm: Bestellung entgegennehmen, Bestellung speichern, Bestätigung senden = 3. Nach Änderungen: Checkout-Workflow, Bestellung, Benachrichtigungen = 3.
Insgesamt berührte Module	Flussdiagramm: 2 + 3 + 6 + 1 + 3 = 15. Nach Änderungen: 1 + 1 + 1 + 1 + 3 = 7.

Änderungen an bestehenden Entscheidungen (Anbieter, Steuerregeln, Darstellung) bleiben im änderungsbasierten Design innerhalb eines Moduls; genau das hat Parnas vorhergesagt.

Manche Änderungen kosten in beiden Designs gleich viel (SMS), und ein wirklich neues Feature (Grußbotschaft) überquert in beiden Modulgrenzen. Information Hiding schützt Entscheidungen, die Sie vorhergesehen haben, nicht jedes mögliche Feature.

Das Zählen ist eine schnelle, wiederholbare Methode, um Zerlegungskandidaten in einem Design-Review zu vergleichen: Listen Sie zuerst die wahrscheinlichen Änderungen auf und zählen Sie dann die Module, die jede davon berührt.

Häufige Fehler

Nur nach technischen Schichten zerlegen. Ordner namens `controllers`, `services` und `repositories` sind gewissermaßen Module, aber eine Änderung an der Mehrwertsteuerberechnung berührt alle drei. Schichten sind nützlich (Abschnitt 3.4), doch eine Zerlegung, die nur aus Schichten besteht, ist ein verkapptes Flussdiagramm. Die meisten ausgereiften Systeme kombinieren eine Aufteilung nach Geschäftsfähigkeiten mit Schichtenbildung innerhalb jedes Teils.

Alles gemeinsam nutzen. Ein Modul namens `common` oder `utils`, das jedes andere Modul importiert, wird zu dem Ort, an dem sich voneinander unabhängige Entscheidungen ansammeln, und jede Änderung daran breitet sich überallhin aus. **Grenzen ohne Durchsetzung.** Diagramme zeigen saubere Module, während der Code Hunderte von Querimporten enthält. **Module, die durch das Organigramm einer früheren Umstrukturierung definiert sind** statt durch die Entscheidungen, die das System verbergen muss; Abschnitt 3.7 erklärt, warum Organisationen und Modulgrenzen aneinander ziehen. **Verfrühte Verteilung.** Wer jedes logische Modul zu einem Service macht, bevor sich die Grenzen als stabil erwiesen haben, macht spätere Grenzkorrekturen teuer.

Checkliste: eine vorgeschlagene Zerlegung prüfen

- [ ] **Benennen Sie die Entscheidung, die jedes Modul verbirgt**
  Wenn Sie nicht in einem Satz sagen können, was ein Modul geheim hält, ist es wahrscheinlich ein Verarbeitungsschritt oder ein Sammelsurium.
- [ ] **Listen Sie die wahrscheinlichen Änderungen auf**
  Schreiben Sie fünf bis zehn Änderungen auf, die das Geschäft oder die Technologie wahrscheinlich mit sich bringen, und zählen Sie die Module, die jede davon berühren würde.
- [ ] **Prüfen Sie die Größe der Schnittstelle**
  Die Schnittstelle sollte viel kleiner sein als die Implementierung. Große Schnittstellen mit wenig Verhalten dahinter deuten auf den falschen Schnitt hin.
- [ ] **Trennen Sie Logisches von Physischem**
  Einigen Sie sich zuerst auf die logischen Module; entscheiden Sie separat, welche davon eine eigene deploybare Einheit verdienen und warum.
- [ ] **Legen Sie fest, wie die Grenze durchgesetzt wird**
  Wählen Sie einen Sichtbarkeitsmechanismus oder eine automatisierte Prüfung und fügen Sie sie dem Build hinzu, bevor die erste Abkürzung auftaucht.
- [ ] **Richten Sie Grenzen an Zuständigkeiten aus**
  Jedes Modul sollte ein verantwortliches Team haben; Module, die mehrere Teams für den Großteil ihrer Arbeit gemeinsam nutzen, sind ein Warnsignal.

Wie Fachleute Modularität anwenden

Erfahrene Architekturfachleute behandeln die Modulstruktur als das wichtigste Designartefakt für die Wartbarkeit, und sie überprüfen sie immer wieder. Früh im Lebenszyklus eines Produkts, wenn die Domäne noch wenig verstanden ist, bevorzugen sie weniger, größere Module mit starken internen Grenzen, weil das Zusammenführen und Aufteilen von Modulen innerhalb einer Codebasis billig ist. Wenn sich die Domäne stabilisiert und wahrscheinliche Änderungen klar werden, verfeinern sie die Grenzen und überlegen erst dann, welche davon separat deployt werden sollten. Techniken aus dem Domain-Driven Design, etwa Bounded Contexts, helfen, Grenzen zu finden, die der Sprache des Fachbereichs folgen.

Sie messen auch. Wer die Module zählt, die von jüngsten Änderungen berührt wurden, was mit der Historie der Versionskontrolle leicht geht, sieht, ob die Grenzen funktionieren; Abschnitt 3.8 führt Change Coupling und Hotspots ein. Der nächste Abschnitt, Abschnitt 3.2, liefert das

Vokabular, um zu beschreiben, wie gute und schlechte Grenzen aussehen: Kopplung zwischen Modulen und Kohäsion innerhalb von Modulen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum nannte Parnas ein Modul eine „Zuweisung einer Verantwortlichkeit“ und nicht ein Unterprogramm?

Weil es darauf ankommt, für welche Designentscheidungen ein Modul verantwortlich ist und welche es verbirgt, nicht darauf, wie sein Code verpackt ist; eine Verantwortlichkeit kann mehrere Routinen umfassen, und die Routinen müssen nicht den Verarbeitungsschritten entsprechen.

▼ Warum betraf im KWIC-Beispiel eine Änderung an der Speicherung der Zeilen in der ersten Zerlegung mehrere Module?

Alle Module nutzten die Datenformate im Speicher direkt gemeinsam, daher war die Speicherentscheidung jedem Modul bekannt, statt in einem verborgen zu sein.

▼ Was ist der Unterschied zwischen einem logischen und einem physischen Modul?

Ein logisches Modul ist eine Verantwortungseinheit im Design; ein physisches Modul ist eine Einheit, die gebaut oder deployt wird, etwa eine JAR-Datei, ein Paket, ein Container oder ein Service. Ein logisches Design lässt sich physisch auf viele Arten verpacken.

▼ Nennen Sie zwei Anzeichen dafür, dass Services zu feingranular sind.

Kleine Features erfordern abgestimmte Änderungen über mehrere Services hinweg, und Anfragen durchlaufen gesprächige Ketten von Netzwerkaufrufen mit hoher Latenz und Teilausfällen.

▼ **Probieren Sie es aus: Ein Ticket-Bezahlvorgang ist in die Schritte Auswählen, Reservieren, Bepreisen, Bezahlen und Ausstellen aufgeteilt, die sich alle einen Buchungsdatensatz teilen, dessen Felder jeder Schritt kennt. Aktionen und eine neue Art, Reservierungen zu speichern, stehen an. Kritisieren Sie das, und schlagen Sie dann eine bessere Aufteilung vor.**

Das ist eine Zerlegung nach Flussdiagramm: Jede Designentscheidung verteilt sich über alle Schritte, sodass eine Änderung am Buchungsdatensatz oder an der Reservierungsspeicherung alle fünf betrifft. Machen Sie nach Parnas jedes Modul zum Versteck für eine Entscheidung, die sich wahrscheinlich ändert: Booking verbirgt die Form des Datensatzes, Holds verbirgt die Speicherung der Reservierungen, Pricing verbirgt die Aktionsregeln, und Payments verbirgt den Anbieter. Jedes bietet eine schmale Schnittstelle, sodass Aktionen ein Modul ändern. Setzen Sie die Grenzen durch, sonst erodieren sie.

QUELLEN

- D. L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules (Communications of the ACM, 1972): https://doi.org/10.1145/361598.361623
- JEP 261: Module System (OpenJDK): https://openjdk.org/jeps/261
- Go 1.4 release notes: internal packages: https://go.dev/doc/go1.4
- Import Linter documentation: https://import-linter.readthedocs.io/en/stable/
- Dijkstra: On the role of scientific thought (EWD 447): https://www.cs.utexas.edu/~EWD/transcriptions/EWD04xx/EWD447.html

Kopplung und Kohäsion

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.3 Code organisieren
- 2.6 Änderbarkeit und Deploybarkeit
- 3.1 Modularität

Begriffe von früher, die er verwendet: Modul, Schnittstelle, Synchroner Aufruf, Asynchrones Messaging, Refactoring

DIE IDEE IN KÜRZE

Denken Sie an zwei Nachbarn, die sich einen Garten teilen. Wenn sie sich nur auf Nachfrage gelegentlich ein Werkzeug leihen, kann jeder seinen eigenen Garten ohne Aufhebens umgestalten: Sie sind lose gekoppelt. Wenn sie sich einen Schuppen, einen Schlauch und einen Schlüssel unter dem Blumentopf teilen, braucht jede Änderung ein Gespräch: Sie sind eng gekoppelt. Denken Sie nun an eine Küchenschublade. Enthält sie nur Besteck, wissen Sie, wo Sie suchen müssen: Sie ist kohäsiv. Enthält sie Besteck, Batterien, Schnur und alte Kassenbons, ist sie es nicht. Gute Software hat kohäsive Teile, die lose miteinander gekoppelt sind.

WAS SIE WISSEN MÜSSEN

- Kopplung ist der Grad, in dem ein Modul von einem anderen abhängt; Kohäsion ist der Grad, in dem die Elemente innerhalb eines Moduls zusammengehören. Streben Sie lose Kopplung und hohe Kohäsion an, die meist miteinander einhergehen.
- Die Skalen des Structured Design ordnen Kopplung von Inhalts-, globaler, externer, Kontroll- und Stempelkopplung bis zur Datenkopplung (von der engsten zur losesten) und Kohäsion von zufälliger, logischer, zeitlicher, prozeduraler, kommunikativer und sequenzieller bis zur funktionalen (von der schwächsten zur stärksten).
- Afferente Kopplung (Ca) zählt eingehende Abhängigkeiten und efferente Kopplung (Ce) ausgehende; ein Modul mit hohem Ca ist riskant zu ändern, und eines mit hohem Ce ist den Änderungen anderer ausgesetzt.
- Verteilte Systeme fügen zeitliche Kopplung (beide Seiten müssen gleichzeitig verfügbar sein), Deployment-Kopplung (sie müssen gemeinsam ausgeliefert werden) und semantische Kopplung (sie müssen eine verborgene Bedeutung teilen) hinzu, die eine

saubere API allein nicht beseitigt.

- Zu den Taktiken, die Kopplung verringern, gehören schmale Schnittstellen, Vermittler wie Adapter und Broker, Ereignisse statt Befehle und strikte Datenhoheit, jede mit ihren eigenen Kosten.
- Kopplung lässt sich nicht beseitigen, nur wählen: Ein Teil der Kopplung ist der Zweck des Systems, also besteht das Ziel darin, sie dort explizit, schwach und lokal zu machen, wo Änderungen wahrscheinlich sind.

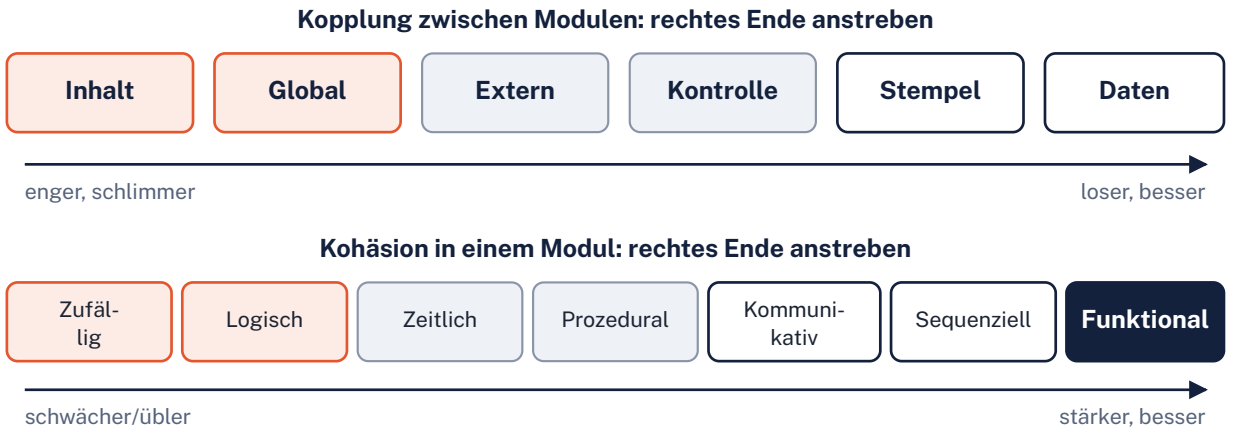
Zwei Maße für Struktur

Kopplung ist der Grad, in dem ein Modul von einem anderen abhängt, sodass eine Änderung am einen eine Änderung am anderen erzwingen kann. Kohäsion ist der Grad, in dem die Elemente innerhalb eines Moduls zusammengehören, weil sie einem einzigen Zweck dienen. Beide Ideen führte Larry Constantine Ende der 1960er-Jahre im Rahmen der Structured-Design-Bewegung ein; veröffentlicht wurden sie 1974 im Artikel *Structured Design* von Wayne Stevens, Glenford Myers und Constantine im *IBM Systems Journal* und dann ausführlicher in Edward Yourdons und Constantines Buch *Structured Design* (1979). Abschnitt 0.3 hat sie informell eingeführt; dieser Abschnitt macht sie präzise.

Beide hängen zusammen. Wenn die Dinge, die sich gemeinsam ändern, im selben Modul liegen (hohe Kohäsion), überqueren weniger Änderungen Modulgrenzen, und die Module müssen weniger voneinander wissen (geringe Kopplung). Yourdon und Constantine gingen so weit zu sagen, dass in der Praxis die Kohäsion die wichtigere der beiden sei. Die Änderbarkeitstaktiken in Abschnitt 2.6 (Kohäsion erhöhen, Kopplung verringern) beruhen genau auf diesem Paar.

Gleich zu Beginn eine Klarstellung: Kopplung ist kein Mangel. Ein System, dessen Module nie voneinander abhingen, würde nichts Nützliches tun. Martin Fowler warnte 2001 in einem Beitrag zu diesem Thema, dass der Versuch, Kopplung ganz zu beseitigen, alles nur in ein riesiges Modul schiebt, in dem die Kopplung verborgen statt verschwunden ist. Das Ziel ist, Kopplung bewusst zu wählen: wenige Abhängigkeiten, über schmale und stabile Schnittstellen, in sinnvolle Richtungen und so schwach, wie es die Aufgabe erlaubt.

Die klassischen Skalen



Die Skalen des Structured Design: Kopplung von Inhalt (am engsten) bis Daten (am losesten) und Kohäsion von zufällig (am schwächsten) bis funktional (am stärksten).

Die Stufen der Kopplung

Structured Design ordnete die Arten, wie zwei Module verbunden sein können, von der schädlichsten bis zur harmlosesten. Die unten gezeigte Form mit sechs Stufen wird meist auf Glenford Myers' Bücher zum Composite Design Mitte der 1970er-Jahre zurückgeführt und später von Meilir Page-Jones bekannt gemacht; Namen und Abgrenzungen unterscheiden sich zwischen den Quellen leicht, aber dies ist die Form, die man in den meisten Lehrbüchern findet. Definiert wurden die Stufen für Prozeduren innerhalb eines Programms, doch jede hat eine direkte Entsprechung zwischen Komponenten und Services, und dort begegnen sie Architekturfachleuten heute.

Kopplungsstufen, die engste zuerst

Niveau	Was sie bedeutet	Im Code	Zwischen Services
Inhalt	Ein Modul greift in die Interna eines anderen ein: ändert dessen private Daten oder springt in dessen Code.	`cart._items` von außen setzen; die Funktion eines anderen Moduls per Monkey-Patching ersetzen.	Die Datenbanktabellen eines anderen Service direkt lesen oder schreiben.
Global	Mehrere Module lesen und schreiben dieselben globalen Daten.	Ein globales `config`-Dictionary, das jede Funktion ändern darf.	Mehrere Services teilen sich ein Datenbankschema oder einen gemeinsamen Cache-Schlüsselraum.
Extern	Module teilen sich ein von außen vorgegebenes Format, Protokoll oder Gerät.	Zwei Module parsen beide dasselbe Dateiformat eines Herstellers.	Mehrere Services hängen vom Dateilayout eines Partners oder von einer bestimmten Version eines Nachrichtenprotokolls ab.
Kontrolle	Ein Modul übergibt ein Flag, das dem anderen sagt, was es tun soll; der Aufrufer kennt also die interne Logik des Aufgerufenen.	`process(order, mode="refund")`	Eine API mit einem `action`-Feld, das völlig unterschiedliche Verhalten auswählt.
Stempel	Ein Modul erhält eine ganze Datenstruktur, nutzt aber nur einen Teil davon.	Das vollständige `Customer`-Objekt an eine Funktion übergeben, die nur die E-Mail-Adresse braucht.	Den gesamten Kundendatensatz in jedem Ereignis senden, obwohl die Consumer zwei Felder brauchen.
Daten	Module kommunizieren nur über Parameter mit einfachen Daten, die alle benötigt werden.	`send_receipt(email, order_id, total)`	Eine schmale, dokumentierte API oder ein Ereignis mit nur den Feldern, die die Consumer brauchen.

Zwei Warnungen. Erstens ist die Skala ein Wegweiser für Risiken, kein Gesetz. Stempelkopplung ist oft eine vernünftige Abwägung: Ein ganzes Objekt zu übergeben ist einfacher, als sechs Felder auszupacken, und der Preis ist nur, dass der Empfänger jetzt brechen kann, wenn Felder geändert werden, die er gar nicht nutzt. Zweitens sagt die Skala nichts darüber, wie viele Abhängigkeiten es gibt oder wie oft sich die andere Seite ändert. Ein einzelner kontrollgekoppelter Aufruf an ein Modul, das sich nie ändert, kann harmlos sein;

Datenkopplung an ein Modul, das sich wöchentlich ändert, kann schmerzen. Später in diesem Kapitel bietet Connascence ein feineres Vokabular, und Vlad Khononovs *Balancing Coupling in Software Design* (2024) argumentiert, dass die Stärke der Kopplung immer zusammen mit der Distanz zwischen den Modulen und ihrer Volatilität beurteilt werden sollte.

```
# Control coupling: the caller chooses the callee's internal branch
def handle_payment(order, mode):
    if mode == "charge":
        ...  # call the provider, store the transaction
    elif mode == "refund":
        ...  # look up the transaction, reverse it
    elif mode == "dry_run":
        ...  # validate only

handle_payment(order, "refund")

# Data coupling: separate operations, each taking only what it needs
def charge(order_id: str, amount_pence: int) -> str: ...
def refund(transaction_id: str, amount_pence: int) -> None: ...

refund("tx_81f2", 1999)
```

Kontrollkopplung (oben) im Vergleich zu Datenkopplung (unten): Das Flag zwingt den Aufrufer zu wissen, wie der Aufgerufene intern organisiert ist.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: eine Quittung, vier Arten von Kopplung

Die schwierigste Idee dieses Abschnitts ist, warum manche Arten von Kopplung schlimmer sind als andere. Grob gesagt: Je mehr sich ein Modul auf Dinge verlässt, die andere Module kontrollieren oder ändern können, desto mehr ihrer Änderungen können es brechen. Verfolgen Sie eine kleine Aufgabe, das Senden einer Quittung, auf vier Arten geschrieben (vier der sechs Stufen der Tabelle, die engste zuerst).

- Der Quittungscode braucht die E-Mail-Adresse eines Kunden, die Bestellnummer und den Gesamtbetrag. In der ersten Version greift er in die privaten Daten des Kundenmoduls und liest das E-Mail-Feld direkt: Inhaltskopplung. Wird dieses Feld umbenannt, bricht die Quittung, obwohl niemand zugesagt hat, dass es so bleibt.
- In der zweiten Version legt der Checkout die E-Mail-Adresse in einem globalen Dictionary ab, das jede Funktion lesen oder ändern darf, und der Quittungscode liest sie von dort: globale Kopplung. Jetzt kann jeder Code, der in dieses Dictionary schreibt, die Quittung fehlleiten, und nichts im Quittungscode zeigt, welcher.
- In der dritten Version erhält der Quittungscode das ganze Customer-Objekt und nutzt nur die E-Mail-Adresse: Stempelkopplung. Er greift nicht mehr ins Innere, kann aber weiterhin brechen, wenn Felder geändert werden, die er nicht nutzt.
- In der vierten Version erhält er nur die drei einfachen Werte, die er braucht: Datenkopplung, die loseste Stufe. Er weiß nichts darüber, wie Kunden oder Bestellungen gespeichert sind, sodass sich das Kundenmodul frei umorganisieren kann.
- **Die Lehre:** Mit jedem Schritt zum Datenende der Skala verlässt sich der Quittungscode auf weniger, was andere Module kontrollieren, sodass weniger ihrer Änderungen ihn brechen können. Die Skala ist ein Wegweiser für Risiken, kein Gesetz: Das ganze Objekt zu übergeben ist oft eine vernünftige Abwägung.

Die Stufen der Kohäsion

Kohäsion fragt, warum die Elemente eines Moduls zusammen sind. Yourdon und Constantine beschrieben sieben Arten der Zusammengehörigkeit, wieder von der schlechtesten zur besten. Die unteren Stufen gruppieren Dinge aus Gründen, die nichts mit dem Problem zu tun haben (sie wurden zufällig zur selben Zeit geschrieben oder laufen im selben Moment); die oberen Stufen gruppieren Dinge, weil sie mit denselben Daten oder auf denselben einzelnen Zweck hin arbeiten. Spätere Lehrbuchzusammenfassungen sind sich in der groben Rangfolge einig: Zufällige und logische Kohäsion sind schlecht, kommunikative und sequenzielle sind gut und funktionale ist am besten, wobei die Abstände zwischen den Stufen nicht gleichmäßig sind.

Kohäsionsstufen, die schwächste zuerst

Niveau	Elemente sind gruppiert, weil …	Beispiel
Zufällig	… es überhaupt keine sinnvolle Beziehung gibt.	Ein `utils`-Modul mit Datumsformatierung, einem Retry-Decorator und einem Mehrwertsteuersatz.
Logisch	… sie die gleiche Art von Aufgabe erledigen, ausgewählt über ein Flag.	Ein `exports`-Modul mit einer Funktion, die je nach Parameter CSV, PDF oder XML schreibt.
Zeitlich	… sie zur selben Zeit laufen.	Eine `startup()`-Routine, die die Datenbank öffnet, Caches vorwärmt und einen Monitoring-Ping sendet.
Prozedural	… sie in einer Ablauffolge aufeinanderfolgen, allerdings mit unterschiedlichen Daten.	Eine Funktion, die ein Formular validiert, dann die IP-Adresse des Benutzers protokolliert und dann ein Menü aktualisiert.
Kommunikativ	… sie mit denselben Daten arbeiten.	Ein Modul, das mehrere Berichte aus derselben Bestellhistorie berechnet.
Sequenziell	… die Ausgabe des einen die Eingabe des nächsten ist.	Einen Kontoauszug parsen, dann seine Posten kategorisieren, dann jede Kategorie aufsummieren.
Funktional	… sie zusammen eine klar definierte Aufgabe erfüllen und sonst nichts.	Ein `Tax`-Modul, das die Mehrwertsteuer für einen Warenkorb berechnet, und nur das.

Auf Architekturebene ist Kohäsion die Frage, ob eine Komponente oder ein Service einen einzigen Daseinsgrund hat. Ein Service namens `CustomerService`, der Anmeldung, Treuepunkte und Marketingeinstellungen verwaltet, ist bestenfalls logisch kohäsiv: Drei Teams werden ihn aus drei voneinander unabhängigen Gründen ändern. Eine gute Faustregel ist, den Zweck eines Moduls in einem Satz zu beschreiben, ohne *und* zu verwenden; Abschnitt 3.5 entwickelt das zum Single Responsibility Principle weiter, das über die Personen definiert ist, die Änderungen anfordern.

Afferente und efferente Kopplung

Die obigen Stufen beschreiben die *Art* einer Abhängigkeit. Architekturfachleute müssen Abhängigkeiten außerdem zählen. Afferente Kopplung (Ca) ist die Anzahl der Module außerhalb einer Komponente, die von ihr abhängen: ihre eingehenden Pfeile. Efferente Kopplung (Ce) ist die Anzahl der Module außerhalb der Komponente, von denen sie abhängt: ihre ausgehenden Pfeile. Die Begriffe stammen aus der Anatomie, wo afferente Nerven Signale

hinein- und efferente Nerven sie hinausleiten. Robert C. Martin definierte sie 1994 für Pakete, indem er Klassen zählte; heutige Werkzeuge zählen Klassen, Module oder Pakete, prüfen Sie also immer, was ein Werkzeug zählt, bevor Sie Zahlen vergleichen.

Die beiden Zahlen erzählen unterschiedliche Geschichten. Eine Komponente mit hohem Ca, etwa eine gemeinsame `money`-Bibliothek, die von vierzig Modulen genutzt wird, ist *verantwortlich*: Jede Änderung an ihr kann viele andere brechen, also muss sie sich vorsichtig und selten ändern. Eine Komponente mit hohem Ce, etwa ein Checkout-Workflow, der Preisgestaltung, Lagerbestand, Zahlung, Lieferung und Benachrichtigungen aufruft, ist *exponiert*: Sie muss sich unter Umständen immer dann ändern, wenn sich eine davon ändert. Keins von beiden ist an sich falsch; das Problem entsteht, wenn eine Komponente stark von anderen genutzt wird und selbst von vielen volatilen Dingen abhängt. Abschnitt 3.4 macht aus Ca und Ce Martins Instabilitätsmetrik und die darauf aufbauenden Regeln.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Ca und Ce in einer Essensliefer-App zählen

Sieben Module (Checkout, Speisekarte, Preisgestaltung, Zahlungen, Benachrichtigungen, Kurierverfolgung und Admin) haben folgende Abhängigkeiten („A hängt von B ab“ wird als A zu B geschrieben): Checkout zu Speisekarte, Checkout zu Preisgestaltung, Checkout zu Zahlungen, Checkout zu Benachrichtigungen; Kurierverfolgung zu Benachrichtigungen; Preisgestaltung zu Speisekarte; Admin zu Speisekarte, Admin zu Preisgestaltung.

Speisekarte	Ca = 3 (Checkout, Preisgestaltung, Admin); Ce = 0
Preisgestaltung	Ca = 2 (Checkout, Admin); Ce = 1 (Speisekarte)
Benachrichtigungen	Ca = 2 (Checkout, Kurierverfolgung); Ce = 0
Checkout	Ca = 0; Ce = 4 (Speisekarte, Preisgestaltung, Zahlungen, Benachrichtigungen)
Lesen	Von Speisekarte hängen die meisten Module ab, also braucht ihre Schnittstelle die meiste Sorgfalt und die langsamsten Änderungen. Checkout hängt von den meisten ab, sollte also am leichtesten zu ändern sein und Änderungen anderer auffangen.

Das Zählen dauert mit einem Abhängigkeitswerkzeug Minuten und macht aus einem vagen Gefühl („Speisekarte ist zentral“) eine Zahl, die Sie im Zeitverlauf verfolgen können.

Kopplung in verteilten Systemen

Wenn Module zu Services werden, die über ein Netzwerk kommunizieren, gelten die klassischen Stufen weiter, aber drei weitere Arten von Kopplung treten in den Vordergrund. Keine davon ist in einer Funktionssignatur sichtbar, weshalb Teams, die ein System in Services aufteilen, oft überrascht sind, wie stark gekoppelt die Services noch sind.

Zeitliche Kopplung bedeutet, dass ein Teil nur funktioniert, wenn ein anderer im selben Moment verfügbar ist und antwortet. Ein synchroner HTTP-Aufruf von Checkout an Zahlungen koppelt ihre Verfügbarkeit: Fällt Zahlungen aus, fällt auch der Checkout aus, und die gemeinsame Verfügbarkeit ist das Produkt der beiden (Abschnitt 2.5). Asynchrone Nachrichten über eine Queue beseitigen das, um den Preis von letztlich statt sofort vorliegenden Ergebnissen. (Im Design auf Code-Ebene bedeutet derselbe Begriff manchmal, dass Aufrufe in einer festen Reihenfolge erfolgen müssen, etwa `open()` vor `read()`; der Kontext macht die Bedeutung meist klar.)

Deployment-Kopplung bedeutet, dass zwei Teile gemeinsam oder in einer festen Reihenfolge ausgeliefert werden müssen, weil eine Änderung am einen den anderen bricht. Wenn jedes Release des Bestellmoduls ein gleichzeitiges Release der Abrechnung braucht, sind die beiden in der Praxis eine deploybare Einheit, was auch immer das Diagramm sagt. Abwärtskompatible Änderungen, versionierte APIs und das Expand-and-Contract-Muster sind die üblichen Heilmittel.

Semantische Kopplung bedeutet, dass sich zwei Teile über eine Bedeutung einig sein müssen, die keine Schnittstelle ausdrückt: dass eine Bestellung mit `status = 4` versandt ist, dass Preise in Pence angegeben sind und die Mehrwertsteuer enthalten, dass ein „aktiver Kunde“ in den letzten 90 Tagen bestellt hat. Solche Übereinkünfte brechen lautlos, wenn eine Seite ihre Auslegung ändert. Das Heilmittel ist, die Bedeutung im Vertrag explizit zu machen (benannte Zustände, Einheiten in Feldnamen, veröffentlichte Schemas) oder die Regel an einer Stelle zu halten, die andere fragen.

Taktiken zur Verringerung von Kopplung

Schmale Schnittstellen. Legen Sie nur die wenigsten Operationen und Felder offen, die Clients brauchen. Jedes zusätzliche Feld in einer API-Antwort ist etwas, von dem ein Client abhängig werden kann (Abschnitt 3.3 zu Hyrums Gesetz). **Vermittler.** Setzen Sie etwas zwischen die beiden Parteien, damit keine die Details der anderen kennt: einen Adapter oder Anti-Corruption Layer, der das Modell eines Lieferanten übersetzt, ein API-Gateway oder einen Message Broker, der Sender und Empfänger zeitlich und in ihrer Identität entkoppelt. **Ereignisse statt Befehle.** Ein Befehl (*Abrechnung, erstelle eine Rechnung*) zwingt den Sender, den Empfänger und dessen Aufgabe zu kennen; ein Ereignis (*eine Bestellung wurde aufgegeben*) lässt den

Sender eine Tatsache feststellen und beliebig viele Empfänger reagieren, sodass sich neue Consumer hinzufügen lassen, ohne den Producer zu ändern. **Datenhoheit.** Jedes Datum hat einen Eigentümer, der es schreibt; andere Module fragen den Eigentümer über seine Schnittstelle oder halten eine eigene, über Ereignisse gespeiste Kopie und lesen nie die Tabellen des Eigentümers (Abschnitt 7.1).

Jede Taktik hat ihren Preis. Vermittler fügen einen Sprung hinzu und etwas, das zusätzlich betrieben werden muss. Ereignisse geben die sofortige Konsistenz auf und machen den Ablauf eines Geschäftsprozesses schwerer an einer Stelle zu überblicken. Kopien von Daten können veraltet sein. Datenhoheit bedeutet, APIs zu bauen und zu versionieren, die eine gemeinsame Tabelle umsonst geliefert hat. Die Abwägung lohnt sich dort, wo Änderungen häufig oder Teams getrennt sind, und oft nicht zwischen zwei Klassen, die eine einzige Person pflegt.

Das Gesetz von Demeter: Sprechen Sie nur mit Ihren direkten Freunden

Das **Gesetz von Demeter** entstand im Demeter-Projekt an der Northeastern University, wo Ian Holland die Regel 1987 erkannte; Karl Lieberherr, Holland und A. Riel veröffentlichten sie 1988 auf der OOPSLA-Konferenz. In seiner allgemeinen Form besagt es, dass jede Einheit nur begrenztes Wissen über andere Einheiten haben sollte, und zwar über jene, die ihr nahestehen: „Sprich nur mit deinen direkten Freunden.“ Objektorientiert ausgedrückt darf eine Methode Nachrichten an ihr eigenes Objekt, ihre Argumente, von ihr erzeugte Objekte und die direkten Bestandteile des Objekts senden, nicht aber an Fremde, die sie über diese erreicht.

Nehmen Sie eine Routine zur Kurierzuweisung auf der Essenslieferplattform, die `order.getCustomer().getAddress().getPostcode()` enthält. Sie braucht nur eines, eine Postleitzahl, weiß nun aber, dass eine Bestellung einen Kunden hat, dass ein Kunde eine Adresse hat und dass eine Adresse eine Postleitzahl hat. Jede dieser Tatsachen ist eine Abhängigkeit. Können Kunden künftig mehrere Adressen speichern, bricht Code, der mit Kunden nichts zu tun hat. Als `order.deliveryPostcode()` geschrieben, hängt derselbe Aufruf von einer Operation ab, und die Struktur dahinter kann sich frei ändern. Verwandt damit ist das Prinzip „Tell, Don’t Ask“: Statt Daten aus einem Objekt zu ziehen, um damit zu arbeiten, beauftragen Sie das Objekt mit der Arbeit. In den Begriffen von Abschnitt 3.6 verbreitet die lange Kette Wissen über die Struktur weit entfernt von dem Ort, an den es gehört.

Die Idee skaliert. Ein Client, der einer Aufrufkette folgt (die Bestellung holen, dann ihr Restaurant, dann das Auszahlungskonto dieses Restaurants), hängt davon ab, wie drei Services zusammenhängen. Bekommt er eine einzige Operation für das, was er braucht, vielleicht über ein Backend for Frontend, entfällt dieses Wissen, so wie es die schmalen Schnittstellen oben empfehlen.

Der Name verspricht zu viel. Martin Fowler sagte, ihm wäre lieber, es hieße *Vorschlag* von Demeter, und er warnt davor, „Tell, Don't Ask" als Grund zu nehmen, jede Abfragemethode zu entfernen. Punkte zu zählen ist ein schlechter Test: Eine Kette über einfache Daten oder ein Fluent Builder, der stets dasselbe Objekt zurückgibt, verrät nichts über versteckte Mitwirkende. Und blind angewendet füllt die Regel Klassen mit dünnen Weiterleitungsmethoden, was ihre Schnittstellen breiter macht. Nutzen Sie sie als Anstoß zu der Frage, warum dieser Code so viel wissen muss.

Komposition statt Vererbung

1994 machten Gamma, Helm, Johnson und Vlissides dies zum zweiten Prinzip von *Design Patterns*: „Bevorzugen Sie Objektkomposition gegenüber Klassenvererbung." Ihre Begründung betraf die Sichtbarkeit. Vererbung ist *White-Box*-Wiederverwendung: Die Unterklasse sieht Details ihrer Elternklasse und hängt von ihnen ab. Komposition ist *Black-Box*-Wiederverwendung: Ein Objekt nutzt ein anderes nur über dessen Schnittstelle. In den Begriffen dieses Abschnitts ist Vererbung eine der stärksten Kopplungen, die eine Sprache bietet, und sie wird beim Schreiben des Codes festgelegt, sodass sie sich während des Programmlaufs nicht ändern lässt. Eine Änderung an einer Elternklasse kann Unterklassen brechen, die nie angefasst wurden, ein Problem, das meist **Fragile Base Class** (fragile Basisklasse) heißt.

Die Essenslieferplattform zeigt, wie das schiefgeht. Ein Team modelliert `ExpressOrder` und `ScheduledOrder` als Unterklassen von `Order`. Dann kommen Gruppenbestellungen hinzu, und jemand braucht eine geplante Gruppenbestellung mit Expresslieferung. Einfachvererbung erzwingt die Wahl einer Elternklasse, und Klassen für jede Kombination zu ergänzen, wächst schnell an. Mit Komposition gilt: Ein `Order` *hat* eine `FulfilmentPolicy` und eine `PricingRule`. Jede davon ist eine kleine Schnittstelle mit mehreren Implementierungen, Kombinationen brauchen keine neue Klasse, und eine Policy lässt sich zur Laufzeit austauschen oder im Test durch ein Fake ersetzen. Das ist die Form der Zahlungsschnittstelle aus Abschnitt 3.3.

Das Prinzip hat sich bis in den Sprachentwurf verbreitet. Go kennt gar keine Typhierarchie: Typen erfüllen Schnittstellen automatisch, und Embedding bietet etwas, das der Vererbung ähnelt, aber nicht gleicht. Auf Architekturebene zeigt sich dieselbe Vorliebe in Plug-ins hinter einer stabilen Schnittstelle (Abschnitt 5.5) statt in einem Framework, das jedes Team erweitern muss. Eine gemeinsame Basisbibliothek, von der jeder Service erben muss, kann die Fragile Base Class in einer ganzen Organisation neu erschaffen.

Das Wort lautet *bevorzugen*, nicht *immer*. Vererbung ist sinnvoll für eine echte Subtyp-Beziehung, die die Liskov Substitution erfüllt (Abschnitt 3.5), besonders innerhalb eines Moduls, dessen Verantwortliche beide Seiten kontrollieren. Komposition hat eigene Kosten:

Weiterleitungscode, mehr kleine Objekte und Verhalten, das sich schwerer nachvollziehen lässt, wenn man nur eine Klasse liest. Fragen Sie, ob der neue Typ wirklich in jeder Hinsicht der alte *ist* oder nur einen Teil seines Codes wiederverwenden will. Trifft das Zweite zu, setzen Sie auf Komposition.

Ausgearbeitetes Beispiel: ein eng gekoppeltes Paar aus Bestellung und Abrechnung refaktorisieren

Das Modul Bestellungen eines Onlineshops ruft bei jeder aufgegebenen Bestellung direkt das Modul Abrechnung auf. Die Abrechnung liest dann die Bestelltabelle, um die Details zu ermitteln, interpretiert die numerischen Codes der Statusspalte und berechnet die Mehrwertsteuer aus den Positionspreisen neu. Der folgende Code ist eine vereinfachte Fassung; er zeigt drei der Probleme auf einmal.

```
# orders.py
def place_order(basket, customer_id):
    order_id = db.insert("orders", customer_id=customer_id, status=1,
                         lines=basket.lines)
    billing.process(order_id, mode="invoice")   # Orders waits for Billing
    return order_id

# billing.py
def process(order_id, mode):
    row = db.query_one("SELECT * FROM orders WHERE id = %s", order_id)
    if row["status"] ≠ 1:              # 1 means 'placed', by convention
        return
    net = sum(l["price"] * l["qty"] for l in row["lines"])
    vat = round(net * 0.2)               # VAT rule duplicated here
    if mode == "invoice":
        db.insert("invoices", order_id=order_id, net=net, vat=vat)
    elif mode == "credit_note":
        db.insert("credit_notes", order_id=order_id, net=-net, vat=-vat)
```

Vorher: Kontrollkopplung (das Modus-Flag), globale Kopplung (beide Module nutzen die Bestelltabelle) und semantische Kopplung (Statuscodes und Mehrwertsteuerregeln sind beiden bekannt).

Jede Änderung am Aufbau der Bestelltabelle, an den Statuscodes oder an den Mehrwertsteuerregeln muss nun in beiden Modulen gleichzeitig erfolgen. Bestellungen kann keine Bestellung annehmen, wenn die Abrechnung langsam ist oder ausfällt. Und die beiden Verhalten der Abrechnung werden über ein Flag ausgewählt, das Bestellungen kennen muss. Das Refactoring verläuft in vier Schritten: Bestellungen ruft die Abrechnung nicht mehr auf, sondern veröffentlicht ein `OrderPlaced`-Ereignis, das genau die Fakten enthält, die die

Abrechnung braucht, einschließlich der Summen, die Bestellungen bereits berechnet hat; die Abrechnung abonniert es und erfasst Rechnungen in einem eigenen Speicher; die beiden Verhalten werden zu zwei Operationen; und die Bestelltabelle wird privat für Bestellungen.

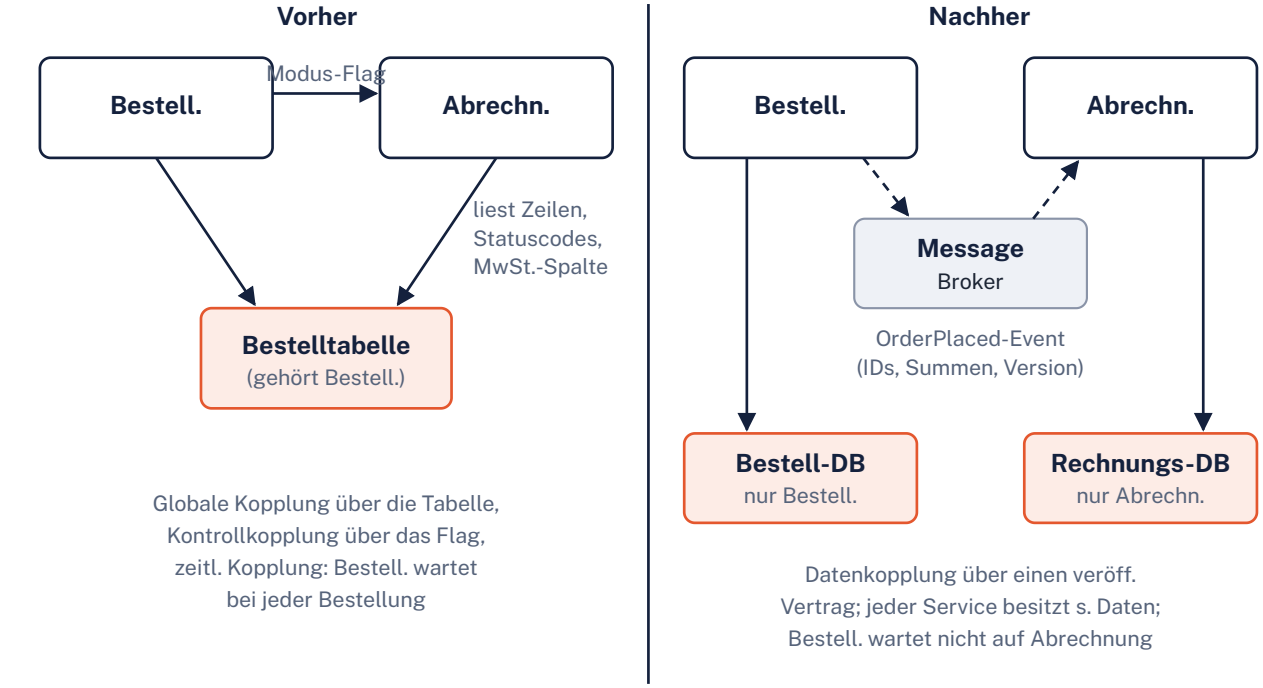
```
# orders.py
def place_order(basket, customer_id):
    order = Order.create(customer_id, basket)      # computes net and VAT once
    repository.save(order)
    events.publish("OrderPlaced", {
        "schema_version": 1, "order_id": order.id,
        "customer_id": customer_id,
        "net_pence": order.net_pence, "vat_pence": order.vat_pence})
    return order.id

# billing.py
@events.subscribe("OrderPlaced")
def on_order_placed(event):
    create_invoice(event["order_id"], event["net_pence"], event["vat_pence"])

def create_invoice(order_id, net_pence, vat_pence): ...
def create_credit_note(order_id, net_pence, vat_pence): ...
```

Nachher: Bestellungen stellt eine Tatsache über ein versioniertes Ereignis fest; die Abrechnung besitzt ihre eigenen Daten und kennt weder die Tabellen noch die Codes oder Steuerregeln von Bestellungen.

Das Paar aus Bestellung und Abrechnung vorher und nachher



Das Refactoring ersetzt einen direkten Aufruf mit Flag und eine gemeinsame Tabelle durch ein veröffentlichtes Ereignis und getrennte Datenspeicher.

AUSGEARBEITETES BEISPIEL

Was sich geändert hat, gemessen

Ein Vergleich der beiden Versionen anhand des Kopplungsvokabulars aus diesem Abschnitt.

Art der Kopplung	Vorher: inhaltsähnliche und globale Kopplung (gemeinsame Tabelle), Kontrollkopplung (Modus-Flag), semantische Kopplung (Status 1, Mehrwertsteuerregel). Nachher: Datenkopplung über ein versioniertes Ereignis mit fünf Feldern.
Efferente Kopplung von Bestellungen	Vorher: Ce umfasst die Abrechnung. Nachher: Bestellungen hängt nur vom Ereignisvertrag und vom Broker ab; es weiß nicht, dass die Abrechnung existiert.
Zeitliche Kopplung	Vorher: Eine Bestellung scheitert, wenn die Abrechnung ausgefallen ist. Nachher: Das Ereignis wartet im Broker, bis die Abrechnung wieder läuft.
Deployment-Kopplung	Vorher: Eine Änderung an der Bestelltabelle erfordert ein gemeinsames Release. Nachher: Bestellungen kann seine Tabelle frei ändern; nur Änderungen am Ereignis erfordern Sorgfalt, und die Versionierung fängt sie ab.
Neue Kosten	Ein Broker, der betrieben werden muss; Rechnungen erscheinen Sekunden später statt sofort; das Ereignisschema ist jetzt ein öffentlicher Vertrag, der versioniert werden muss; der Prozess ist ohne gute Observability schwerer nachzuverfolgen.

Das Ereignis enthält die Summen, die Bestellungen berechnet hat, sodass die Mehrwertsteuer an einer Stelle berechnet wird. Die Abrechnung muss weder die Steuerregeln noch den Aufbau der Bestellpositionen kennen.

Das Refactoring hat eine Wahl getroffen, nicht alle Kopplung beseitigt: Die Abrechnung hängt weiterhin von der Bedeutung von `OrderPlaced` ab. Diese Abhängigkeit ist jetzt explizit, schmal und versioniert, und genau das bedeutet lose Kopplung in der Praxis.

ACHTUNG

Verbreitete Irrtümer

- Microservices sind nicht automatisch lose gekoppelt. Services, die sich eine Datenbank teilen oder gemeinsam ausgeliefert werden müssen, sind eng gekoppelt, wie auch immer sie deployt werden.
- Das Entfernen von Duplikation kann die Kopplung erhöhen. Zwei Services, die sich eine Bibliothek für eine Geschäftsregel teilen, ändern sich nun gemeinsam; manchmal ist ein wenig Duplikation das loser gekoppelte Design (siehe Abschnitt 3.8 zu DRY).
- Asynchrones Messaging beseitigt zeitliche Kopplung, nicht semantische Kopplung. Ein Consumer, der ein Ereignisfeld falsch liest, ist genauso kaputt, nur später.
- Geringe Kopplung ist nicht dasselbe wie wenige Linien zwischen Kästen in einem Diagramm. Eine einzige Abhängigkeit von einer großen, instabilen, schlecht spezifizierten Schnittstelle kann schlimmer sein als fünf von kleinen, stabilen.

Wie Fachleute diese Ideen nutzen

Erfahrene Designer nennen die Namen der Stufen in Reviews selten, wenden sie aber ständig an: Sie hinterfragen jeden Flag-Parameter, suchen nach gemeinsamen Tabellen und globalem Zustand, fragen nach dem einen Zweck jedes Moduls und beobachten, welche Module sich in der Versionskontrolle gemeinsam ändern. An den Grenzen zwischen Teams und Services sind sie streng, weil Kopplung dort teuer aufzulösen ist; innerhalb eines kleinen Moduls, das einer Person gehört, sind sie gelassen, weil sie dort billig ist.

Außerdem machen sie Kopplung sichtbar. Ein aus dem Code erzeugter Abhängigkeitsgraph, eine Zählung von Ca und Ce pro Komponente und eine Liste der Datenbanktabellen, auf die mehr als ein Service zugreift, gehören zu den aufschlussreichsten Sichten auf eine Architektur, und alle lassen sich automatisch erzeugen und im Build verfolgen. Der nächste Abschnitt wendet sich dem Prinzip zu, das Kopplung von vornherein schwach hält: Entscheidungen hinter Schnittstellen verbergen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum ist eine Funktion mit einem Modus-Parameter, der völlig unterschiedliche Verhalten auswählt, ein Beispiel für Kontrollkopplung?

Weil der Aufrufer wissen muss, wie der Aufgerufene intern organisiert ist, und dessen internen Zweig auswählt; eine Änderung der Logik des Aufgerufenen kann Änderungen bei den Aufrufern erzwingen.

▼ **Ein Modul hat Ca = 12 und Ce = 1. Was deutet das an?**

Viele Module hängen von ihm ab, und es selbst hängt von wenig ab; Änderungen daran sind also riskant und sollten selten und vorsichtig erfolgen, und seine Schnittstelle verdient besondere Stabilität.

▼ **Zwei Services kommunizieren nur über Ereignisse, müssen aber immer gemeinsam ausgeliefert werden. Welche Art von Kopplung bleibt bestehen?**

Deployment-Kopplung, meist verursacht durch nicht abwärtskompatible Änderungen am gemeinsamen Ereignis oder durch verborgene semantische Übereinkünfte.

▼ **Nennen Sie einen Preis dafür, einen direkten Aufruf durch ein Ereignis zu ersetzen.**

Ergebnisse werden letztlich statt sofort konsistent; außerdem bekommen Sie einen Broker, der betrieben werden muss, ein Ereignisschema, das versioniert werden muss, und einen Ablauf, der schwerer nachzuverfolgen ist.

▼ **Probieren Sie es aus: Booking hängt von Seats, Payments und Email ab. Refunds hängt von Payments und Email ab. Reports hängt von Seats und Payments ab. Berechnen Sie Ca und Ce für Payments und Booking. Was ist riskanter zu ändern, und was ist stärker exponiert?**

Payments: Ca = 3 (Booking, Refunds, Reports), Ce = 0. Booking: Ca = 0, Ce = 3 (Seats, Payments, Email). Payments ist riskanter zu ändern, weil drei Module jede Änderung seiner Schnittstelle spüren; halten Sie diese also schmal und stabil. Booking ist exponierter, da eine Änderung an einer seiner drei Abhängigkeiten es brechen kann, aber Booking selbst zu ändern ist sicher, weil nichts davon abhängt. Diese Risikoverteilung ist gesund.

QUELLEN

- Martin Fowler, Reducing Coupling (IEEE Software, 2001): https://martinfowler.com/ieeeSoftware/coupling.pdf
- W. P. Stevens, G. J. Myers and L. L. Constantine, Structured Design (IBM Systems Journal, 1974): https://doi.org/10.1147/sj.132.0115
- Microservices: database per service pattern (microservices.io): https://microservices.io/patterns/data/database-per-service.html
- Martin Fowler, What do you mean by Event-Driven?: https://martinfowler.com/articles/201701-event-driven.html
- Law of Demeter: general formulation (Northeastern University): https://www2.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html

- Law of Demeter: object formulation (Northeastern University): https://www2.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/object-formulation.html
- Object-oriented programming: an objective sense of style (record): https://scilit.com/publications/8b7cd3e2eaafe26d184e99c8c9e37165
- Law of Demeter (Wikipedia, for authors and project): https://en.wikipedia.org/wiki/Law_of_Demeter
- Fowler: Mocks Aren't Stubs: https://martinfowler.com/articles/mocksArentStubs.html
- Fowler: TellDontAsk: https://martinfowler.com/bliki/TellDontAsk.html
- When did people favor composition over inheritance?: https://sicpers.info/2025/11/when-did-people-favor-composition-over-inheritance
- The Go FAQ: https://go.dev/doc/faq

Information Hiding, Abstraktion und Schnittstellen

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.3 Code organisieren
- 0.6 APIs und Integration
- 3.1 Modularität

Begriffe von früher, die er verwendet: Modul, API (Programmierschnittstelle), Klasse und Objekt, Breaking Change

DIE IDEE IN KÜRZE

Wenn Sie ein Paket aufgeben, geben Sie es am Schalter ab und erhalten eine Sendungsnummer. Sie wissen nicht, und es ist Ihnen egal, ob es per Transporter, Zug oder Flugzeug reist oder durch welches Depot es läuft. Die Post kann all das morgen ändern, ohne Ihnen Bescheid zu sagen, denn die Vereinbarung am Schalter bleibt dieselbe. Softwaremodule funktionieren am besten wie dieser Schalter: Sie geben ein kleines, klares Versprechen und verbergen, wie sie es einhalten. Ärger beginnt, wenn sich Kunden auf Details verlassen, die niemand versprochen hat, etwa dass das Paket immer vor zehn Uhr ankommt.

WAS SIE WISSEN MÜSSEN

- Information Hiding (Parnas, 1972) bedeutet, dass jedes Modul eine Designentscheidung, die schwierig ist oder sich wahrscheinlich ändert, hinter einer Schnittstelle verbirgt, die so wenig wie möglich preisgibt.
- Kapselung ist ein Sprachmechanismus (private Felder, Module, Zugriffskontrolle); Information Hiding ist ein Designprinzip darüber, welche Entscheidungen man verbirgt. Man kann kapseln, ohne etwas Nützliches zu verbergen.
- Eine Schnittstelle ist ein Vertrag; Design by Contract (Bertrand Meyer) macht ihn mit Vorbedingungen, Nachbedingungen und Invarianten präzise und weist die Schuld an jedem Fehlschlag dem Aufrufer oder dem Anbieter zu.
- Alle nicht trivialen Abstraktionen sind bis zu einem gewissen Grad undicht (Joel Spolsky, 2002); gute Designer legen daher Fehlerverhalten, Kosten und Performance-Eigenschaften ehrlich offen, statt so zu tun, als gäbe es sie nicht.
- Ousterhouts tiefe Module bieten viel Funktionalität hinter einer kleinen Schnittstelle; flache Module verursachen Schnittstellenkosten, ohne viel zu verbergen.

- Hyrums Gesetz besagt, dass sich jemand auf jedes beobachtbare Verhalten einer weit verbreiteten API verlassen wird; stabile APIs brauchen daher minimale Oberflächen, explizite Versionierung, additive Änderungen und Contract-Tests.

Das Prinzip: verbergen, was sich wahrscheinlich ändert

In Abschnitt 3.1 zeigte David Parnas' KWIC-Beispiel, dass eine Zerlegung eines Programms nach Verarbeitungsschritten jede Designentscheidung über mehrere Module verteilt. Seine Alternative, Information Hiding, gibt jedem Modul eine Entscheidung, die es geheim hält: wie Zeilen gespeichert werden, wie Verschiebungen berechnet werden, wann sortiert wird. Die Schnittstelle des Moduls wird so gewählt, dass sie so wenig wie möglich über diese Entscheidung verrät. Alles andere im System hängt nur von der Schnittstelle ab, sodass sich die Entscheidung ändern kann, ohne dass es jemand merkt.

Parnas' Liste dessen, was sich zu verbergen lohnt, ist auch ein halbes Jahrhundert später noch eine gute Checkliste: Datenstrukturen und wie sie gespeichert werden; die Reihenfolge, in der Operationen ausgeführt werden; die Details von Hardware, Geräten und Betriebssystemen; Ein- und Ausgabeformate; und Algorithmen, die durch bessere ersetzt werden könnten. Moderne Ergänzungen folgen derselben Logik: welcher Drittanbieter genutzt wird, welche Datenbank, welcher Message Broker, welche Version einer externen API und welche Geschäftsregeln gerade überprüft werden.

Das Prinzip gilt auf jeder Ebene gleichermaßen. Eine Klasse verbirgt ihre Felder; ein Paket verbirgt seine internen Module; ein Service verbirgt seine Datenbank, seine Sprache und sein Deployment; die öffentliche API einer Organisation verbirgt, welche Services dahinter existieren. In jedem Fall lässt sich das Verborgene günstig ändern, und das Offengelegte wird teuer zu ändern, weil andere davon abhängen.

Kapselung ist nicht Information Hiding

Die beiden Begriffe werden oft synonym verwendet, bedeuten aber Verschiedenes. Kapselung ist ein Mechanismus: Daten werden mit den Operationen darauf gebündelt und der direkte Zugriff wird eingeschränkt, durch private Felder, Modulgrenzen oder Zugriffsmodifikatoren (Abschnitt 0.3). Information Hiding ist ein Designprinzip: Man entscheidet, *welche* Entscheidungen geheim bleiben, und gestaltet die Schnittstelle so, dass sie sie nicht verrät. Kapselung ist eines der Werkzeuge, mit denen Sie Informationen verbergen; was zu verbergen ist, entscheidet sie nicht für Sie.

Man kann leicht perfekt kapseln und trotzdem nichts verbergen. Eine Klasse mit privaten Feldern und je einem öffentlichen Getter und Setter für jedes davon ist vollständig gekapselt, doch jeder Aufrufer weiß genau, welche Felder es gibt und welche Typen sie haben. Ändern Sie

die interne Darstellung, bricht jeder Aufrufer. Auch das Gegenteil kommt vor: Python erzwingt überhaupt keine Privatheit, und doch kann ein gut entworfenes Python-Modul seine Entscheidungen durch eine sorgfältig gewählte Menge von Funktionen sehr wirksam verbergen.

```
# Encapsulated, but the representation leaks through the interface
class Price:
    def __init__(self, pounds: float):
        self._pounds = pounds
    def get_pounds(self) → float:          # callers now depend on 'float pounds'
        return self._pounds
    def set_pounds(self, value: float):
        self._pounds = value

# Information hiding: callers ask for what they need, not for the field
class Money:
    def __init__(self, minor_units: int, currency: str):
        self._minor, self._currency = minor_units, currency
    def add(self, other: "Money") → "Money": ...
    def allocate(self, ratios: list[int]) → list["Money"]: ...
    def format(self, locale: str) → str: ...
# Switching from pence to a decimal type later changes only this class.
```

Beide Klassen sind gekapselt; nur die zweite verbirgt die Entscheidung, wie ein Preis gespeichert wird.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: private Felder, die nichts verbergen

Die Idee, die hier die meiste Sorgfalt verlangt, ist, dass eine Klasse all ihre Daten privat halten und trotzdem nichts Nützliches verbergen kann. Kapselung schließt die Daten weg; Information Hiding entscheidet, welche Designentscheidung die Schnittstelle nicht preisgeben darf. Verfolgen Sie die beiden Klassen oben durch eine Änderung.

- Der erste Entwurf des Shops ist die Klasse Price. Ihr Betrag liegt in einem privaten Feld, sie ist also ordentlich gekapselt, aber get_pounds gibt den Betrag als Gleitkommazahl in Pfund heraus, und set_pounds nimmt eine solche entgegen.
- Checkout, Rechnungen und Erstattungen rufen jeweils get_pounds auf und rechnen selbst mit dem Ergebnis: Sie addieren Preise oder teilen eine Erstattung auf mehrere Artikel auf.
- Kapitel 0 hat erklärt, dass Gleitkommazahlen nur Näherungen sind. Manche Summen liegen nach dem Runden um einen Penny daneben, also beschließt das Team, Geldbeträge stattdessen als ganze Pence zu speichern.
- Das private Feld zu ändern ist leicht, behebt aber nichts: get_pounds gibt weiterhin Gleitkommazahlen heraus, und die fehlerhafte Arithmetik steckt in jedem Aufrufer. Um das zu beheben, muss das Team die Schnittstelle und jeden rechnenden Aufrufer ändern. Das private Feld verbarg, wo die Zahl lag, aber nicht die Entscheidung, auf die es ankam: dass Geld eine Gleitkommazahl in Pfund ist.
- Im zweiten Entwurf, Money, erhalten Aufrufer nie die rohe Zahl. Sie bitten Money, zwei Beträge zu addieren, einen Betrag nach Verhältnissen aufzuteilen (allocate) oder sich für die Anzeige zu formatieren. Die Arithmetik mit Beträgen steckt in Money, sodass eine geänderte Darstellung nur diese eine Klasse betrifft.
- **Die Lehre:** Private Felder sind das Schloss; Information Hiding heißt zu wählen, was dahinter kommt. Benennen Sie zuerst die Entscheidung, die sich am wahrscheinlichsten ändert, und prüfen Sie dann, dass keine Operation der Schnittstelle sie verrät.

Abstraktion und Schnittstellen als Verträge

Eine Abstraktion ist eine vereinfachte Sicht auf etwas, die behält, was für einen Zweck wichtig ist, und den Rest weglässt. Eine Datei ist eine Abstraktion über Plattenblöcke, Caches und Treiber; eine *Zahlung* ist eine Abstraktion über ein Kartennetzwerk, die API eines Anbieters und den Abwicklungsprozess einer Bank. Abstraktion und Information Hiding sind zwei Seiten desselben Vorgangs: Die Wahl einer Abstraktion entscheidet, was die Schnittstelle zeigt; das Verbergen des Rests macht die Abstraktion verlässlich.

Eine Schnittstelle ist daher ein Vertrag zwischen einem Modul und seinen Nutzern. Ihr formaler Teil ist das, was ein Typprüfer oder ein API-Schema sehen kann: Operationsnamen, Parameter- und Rückgabetypen, Fehlertypen, Feldnamen in einem JSON-Body. Ihr informeller Teil ist genauso verbindlich: was die Operation tut, ihre Seiteneffekte, ob sie gefahrlos wiederholt werden kann, wie lange sie dauert, welche Fehler möglich sind und was sie bedeuten. John Ousterhout betont, dass sich der informelle Teil nur in der Dokumentation festhalten lässt und dass ein Modul keine echte Abstraktion bietet, wenn Nutzer seinen Code lesen müssen, um zu erfahren, wie man es verwendet.

Design by Contract

Design by Contract, von Bertrand Meyer in den 1980er-Jahren mit der Sprache Eiffel entwickelt und in seinem Buch *Object-Oriented Software Construction* sowie seinem Artikel *Applying 'Design by Contract'* von 1992 beschrieben, macht den Vertrag präzise. Jede Operation hat eine Vorbedingung, die der Aufrufer vor dem Aufruf erfüllen muss; eine Nachbedingung, die die Operation bei der Rückkehr garantiert, sofern die Vorbedingung galt; und jede Klasse oder jedes Modul hat Invarianten, Bedingungen, die zwischen je zwei Operationen gelten. Der Vertrag weist Verantwortung zu: Ist eine Vorbedingung verletzt, liegt der Fehler beim Aufrufer; ist eine Nachbedingung verletzt, liegt er beim Anbieter.

Meyer argumentierte gegen die entgegengesetzte Gewohnheit, die defensive Programmierung, bei der jede Routine alles prüft, falls jemand sie falsch aufruft. Redundante Prüfungen fügen Code, Komplexität und neue Fehlerpfade hinzu. Stattdessen sollte jede Bedingung klar in der Verantwortung einer Seite liegen. Eine Routine mit einer anspruchsvollen Vorbedingung ist einfacher; eine Routine mit einer toleranten ist leichter aufzurufen; die Wahl ist eine Designentscheidung, die ausdrücklich getroffen wird. An den Grenzen eines Systems, wo Eingaben von nicht vertrauenswürdigen Nutzern oder anderen Organisationen kommen, ist Validierung nicht optional: Dort besagt der Vertrag, dass das Grenzmodul selbst alles prüft (Kapitel 8).

```
class Account:
    """Invariant: balance_pence ≥ -overdraft_limit_pence."""

    def withdraw(self, amount_pence: int) → None:
        # Preconditions (the caller's obligations)
        if amount_pence ⩽ 0:
            raise ValueError("amount must be positive")
        if self.balance_pence - amount_pence < -self.overdraft_limit_pence:
            raise InsufficientFunds(self.id)
        old = self.balance_pence
        self.balance_pence -= amount_pence
        # Postcondition (the supplier's guarantee)
        assert self.balance_pence == old - amount_pence
        assert self.balance_pence ≥ -self.overdraft_limit_pence  # invariant
```

Ein schriftlich festgehaltener Vertrag: Vorbedingungen werden beim Eintritt geprüft, Nachbedingung und Invariante werden angegeben und per Assertion geprüft.

Dieselben drei Ideen lassen sich auf Services übertragen. Eine API-Spezifikation, etwa ein OpenAPI-Dokument, nennt die Vorbedingungen (Pflichtfelder, Formate, Authentifizierung), die Nachbedingungen (die Antwort und die Zustandsänderung, etwa *es existiert eine Bestellung mit dem Status PLACED*) und die Invarianten, auf die sich ein Client verlassen darf (*die Gesamtsumme einer Bestellung ändert sich nach der Zahlung nie*). Consumer-Driven-Contract-Tests, bei denen jeder Client die Erwartungen veröffentlicht, auf die er sich verlässt, und der Anbieter sie in seinem Build ausführt, prüfen diese Verträge automatisch.

Leaky Abstractions

Im November 2002 veröffentlichte Joel Spolsky *The Law of Leaky Abstractions*, das besagt: „Alle nicht trivialen Abstraktionen sind bis zu einem gewissen Grad undicht.“ Eine Leaky Abstraction ist eine, deren verborgene Details ihre Nutzer dennoch beeinflussen. TCP bietet einen zuverlässigen Datenstrom über ein unzuverlässiges Netzwerk, bis ein Kabel durchtrennt wird und es das nicht mehr kann. SQL lässt Sie angeben, was Sie wollen, nicht wie man es bekommt, bis zwei logisch gleichwertige Abfragen sich in der Geschwindigkeit um den Faktor tausend unterscheiden und Sie den Abfrageplan lesen müssen. Ein objektrelationaler Mapper verbirgt SQL, bis eine Schleife stillschweigend eine Abfrage pro Zeile absetzt.

Die Folgerung für Architekten ist nicht, dass Abstraktionen schlecht sind, sondern dass eine Schnittstelle ehrlich mit den Details umgehen sollte, die durchsickern werden. Das klassische Beispiel ist der entfernte Aufruf, der genau wie ein lokaler Funktionsaufruf aussieht. Netzwerkaufrufe sind um Größenordnungen langsamer, können in einen Timeout laufen, können auf dem Server gelingen, während die Antwort verloren geht, und können unabhängig vom Aufrufer ausfallen; Abschnitt 6.1 führt diese als Fallacies of Distributed Computing auf. Eine gute entfernte Schnittstelle legt das offen: Sie nimmt einen Timeout entgegen, liefert

Fehler, die *abgelehnt* von *unbekanntem Ergebnis* unterscheiden, und akzeptiert einen Idempotenzschlüssel, damit Wiederholungen gefahrlos sind. Das Netzwerk vollständig zu verbergen ist das Leck, das nur darauf wartet aufzutreten.

Tiefe und flache Module

John Ousterhouts *A Philosophy of Software Design* (Erstausgabe 2018; zweite Auflage 2021) bietet eine nützliche Art, eine Schnittstelle zu beurteilen. Stellen Sie sich jedes Modul als Rechteck vor: Die obere Kante ist seine Schnittstelle, die jeder Nutzer lernen muss, und die Fläche ist die Funktionalität, die es bietet. Ein tiefes Modul hat eine kleine Schnittstelle über viel Funktionalität und zahlt die Kosten des Erlernens daher um ein Vielfaches zurück. Ein flaches Modul hat eine Schnittstelle, die fast so komplex ist wie das, was es tut, und verursacht daher Kosten, ohne viel zu verbergen.

Sein Lieblingsbeispiel für Tiefe ist die Datei-Ein- und -Ausgabe unter Unix: eine Handvoll Aufrufe (open, read, write, seek, close) verbirgt Dateisysteme, Plattenlayout, Caching, Berechtigungen, Geräte und gleichzeitigen Zugriff. Sein Beispiel für Flachheit ist eine Klasse, deren Methoden jeweils einen Aufruf an eine andere Klasse weiterreichen, oder eine Linked-List-Klasse, deren Schnittstelle fast alles offenlegt, was sie tut. Ousterhout argumentiert, dass der verbreitete Rat, Klassen und Methoden sehr klein zu halten, zu dem führen kann, was er Classitis nennt: viele flache Klassen, deren Schnittstellen zusammen mehr Komplexität ergeben als ein einziges tieferes Modul.

Tiefe und flache Module



Ousterhouts Bild: Die Breite der oberen Kante ist die Schnittstelle, die Nutzer lernen müssen; die Fläche darunter ist die Funktionalität, die sie verbirgt.

Nicht alle stimmen zu, wie weit Ousterhout das treibt. Befürworter kleiner Klassen und Funktionen, etwa Robert C. Martin in *Clean Code*, argumentieren, dass winzige Einheiten leichter zu benennen, zu testen und wiederzuverwenden sind; Ousterhout und Martin haben die Frage in einem 2025 veröffentlichten schriftlichen Austausch diskutiert. Beide Seiten sind sich über das zugrunde liegende Ziel einig: die Komplexität zu verringern, die ein Leser im Kopf behalten muss. Auf Architekturebene ist der Tiefentest besonders wertvoll: Ein Service, dessen API seine Datenbanktabellen eins zu eins abbildet, mit Anlegen, Lesen, Ändern und Löschen für jede, ist flach und schiebt die gesamte eigentliche Logik auf seine Clients ab.

Hyrums Gesetz: Alles Beobachtbare wird zum Vertrag

Hyrums Gesetz, benannt nach Hyrum Wright, damals Softwareentwickler bei Google, durch seinen Kollegen Titus Winters und bekannt gemacht durch *Software Engineering at Google* (2020), stellt fest: Bei einer ausreichenden Zahl von Nutzern einer API ist es egal, was Sie im Vertrag versprechen; von allen beobachtbaren Verhaltensweisen Ihres Systems wird irgendjemand abhängen. Die Reihenfolge, in der eine Funktion Elemente aus einer ungeordneten Menge liefert, der genaue Wortlaut einer Fehlermeldung, ein Feld, das immer vorhanden ist, obwohl es als optional dokumentiert ist, die Tatsache, dass ein Aufruf 50 Millisekunden dauert: Wenn Nutzer es sehen können, werden sich einige darauf verlassen.

Das Gesetz erklärt, warum Information Hiding bewusst entworfen und nicht erhofft werden muss. Jedes Verhalten, das Sie versehentlich offenlegen, wird Teil des faktischen Vertrags, und wer es ändert, bricht etwas bei irgendjemandem, ganz gleich, wie deutlich die Dokumentation es als unspezifiziert bezeichnet hat. Manche Teams randomisieren unspezifiziertes Verhalten absichtlich, etwa die Iterationsreihenfolge, damit niemand davon abhängig werden kann; Go randomisiert aus diesem Grund die Iterationsreihenfolge von Maps. Andere halten Antwort-Payloads schlank, damit es weniger gibt, wovon man abhängen kann.

Postels Gesetz und seine Kritik

Während Hyrums Gesetz beschreibt, was mit Schnittstellen geschieht, sagt Ihnen Postels Gesetz, wie Sie sich an einer Schnittstelle verhalten sollen. Jon Postel gab die frühen Internet-Spezifikationen heraus, und die Idee erscheint 1980 in RFC 760 zum Internet Protocol: „Eine Implementierung sollte in ihrem Sendeverhalten konservativ und in ihrem Empfangsverhalten liberal sein.“ RFC 761 zu TCP lieferte die meist zitierte Form: „Sei konservativ in dem, was du tust, und liberal in dem, was du von anderen annimmst.“ Es heißt auch **Robustheitsprinzip**. Es ist eine Entwurfsmaxime, kein Naturgesetz. Es passte zu seiner Zeit: Viele unabhängige Implementierungen unvollkommener Spezifikationen mussten zusammenarbeiten.

In einer Form bleibt es nützlich. Martin Fowlers **Tolerant Reader** wendet es auf die Integration an: Ein Consumer sollte nur lesen, was er braucht, und den Rest ignorieren. Liest der Benachrichtigungsservice zwei Felder aus einem `OrderPlaced`-Ereignis und ignoriert unbekannte, kann der Bestellservice Felder ergänzen, ohne ihn zu brechen; genau das lässt die kompatiblen Änderungen aus Abschnitt 7.7 funktionieren.

Die Kritik betrifft, was mit der Zeit geschieht. Verbinden Sie einen toleranten Empfänger mit Hyrums Gesetz, und jeder akzeptierte Fehler wird zu etwas, worauf sich andere verlassen. Angenommen, die Integration eines Restaurants sendet Preise als `"12,50"`, und die Plattform akzeptiert das stillschweigend. Der Sender erfährt nie, dass es falsch ist, andere Partner kopieren es, und die Eigenheit ist zu einem ungeschriebenen Teil des Vertrags geworden, den jede Neuentwicklung nachbilden muss.

Das Internet Architecture Board legte dies in RFC 9413 dar, *Maintaining Robust Protocols* (Juni 2023), einem Informational-Dokument, keinem Standard, dessen Annahme breit, aber nicht einstimmig war. Es argumentiert, dass das Tolerieren unerwarteter Eingaben zum Verfall von Protokollen führt: Fehler bleiben verborgen, Fehlverhalten wird zum De-facto-Standard, und neue Implementierungen müssen die Bugs der alten kopieren, um mit ihnen zusammenzuarbeiten. Sein Beispiel sind TLS-Server, die an leeren Erweiterungen scheiterten, was Clients zu Umgehungen zwang. Es empfiehlt die aktive Pflege von Protokollen, sodass Mehrdeutigkeiten in der Spezifikation aufgelöst und nicht stillschweigend hingenommen werden, und *tugendhafte Intoleranz*: bei einem Verstoß gegen die Spezifikation einen fatalen Fehler auszulösen, damit Bugs früh sichtbar werden.

Eine praktische Regel versöhnt beide. Seien Sie nur gegenüber dem tolerant, was Sie laut Vertrag ignorieren dürfen, etwa unbekannten zusätzlichen Feldern. Seien Sie streng bei allem, was der Vertrag festlegt: Ein fehlendes Pflichtfeld, ein falscher Typ oder ein ungültiger Wert erhält einen klaren Fehler, keine Vermutung. Das zählt besonders bei Geld, Identität und Sicherheit, wo ein Empfänger, der „sein Bestes tut“, eine Nachricht anders auslegen kann als das System neben ihm.

Stabile APIs zwischen Komponenten entwerfen

Zwischen Komponenten und Services sind Schnittstellen das Teuerste, was man ändern kann, weil jede Änderung mit allen Konsumenten abgestimmt werden muss. Die folgenden Praktiken ergeben sich aus allem Obigen; Abschnitt 0.6 hat APIs eingeführt, und Abschnitt 7.7 behandelt Schemakompatibilität ausführlich.

Praktiken für stabile Komponenten- und Service-APIs

Praktik	Was sie bedeutet	Warum sie funktioniert
Minimale Oberfläche	Nur Operationen und Felder offenlegen, die ein bekannter Konsument braucht; mehr auf Anfrage ergänzen.	Weniger, wovon man abhängen kann, also weniger, was brechen kann (Hyrums Gesetz).
Die Domäne modellieren, nicht die Speicherung	Operationen wie `place_order` und `cancel_order` anbieten statt tabellenförmigem Anlegen, Ändern und Löschen.	Die Speicherung kann sich ändern, ohne dass sich der Vertrag ändert; das Modul bleibt tief.
Additive Änderung	Optionale Felder und neue Operationen hinzufügen; bestehende innerhalb einer Version nie umbenennen, entfernen oder in ihrer Bedeutung ändern.	Alte Clients funktionieren weiter; neue Clients nutzen die neuen Teile.
Tolerant Reader	Clients ignorieren Felder, die sie nicht verstehen, und hängen nicht von der Feldreihenfolge ab.	Anbieter können Felder hinzufügen, ohne jemanden zu brechen.
Explizite Versionierung	Für Breaking Changes eine neue Hauptversion einführen (in der URL, einem Header oder dem Event-Schema) und Alt und Neu für einen veröffentlichten Zeitraum parallel betreiben.	Konsumenten migrieren nach ihrem eigenen Zeitplan; Deployment-Kopplung wird vermieden.
Expand and Contract	Die neue Form hinzufügen, Konsumenten migrieren, per Monitoring bestätigen, dass niemand mehr die alte Form nutzt, und sie dann entfernen, jeweils als getrennte Releases.	Breaking Changes erfolgen ohne gemeinsames Release.
Contract-Tests	Konsumenten veröffentlichen ihre Erwartungen; der Build des Anbieters führt sie aus.	Versehentliche Breaking Changes werden vor dem Deployment erkannt.
Ehrliche Fehlersemantik	Timeouts, Wiederholbarkeit, Idempotenz und die Bedeutung von Fehlern als Teil des Vertrags dokumentieren.	Vermeidet die schlimmsten Lecks der Netzwerkabstraktion.

Ausgearbeitetes Beispiel: eine Zahlungsschnittstelle, die den Anbieter verbirgt

Ein Essenslieferdienst wickelt Zahlungen über Anbieter A ab. Die Finanzabteilung möchte Anbieter B wegen niedrigerer Gebühren in einem Land hinzunehmen und bei einem Ausfall eines Anbieters innerhalb eines Tages wechseln können. Heute ruft der Checkout-Code das

SDK von Anbieter A an elf Stellen direkt auf, nutzt dessen Fehlercodes, um zu entscheiden, was dem Kunden angezeigt wird, und speichert dessen Transaktionsobjekte in der Bestelltabelle. Jedes davon ist eine Entscheidung, die durchgesickert ist.

Der Entwurf führt eine Schnittstelle `PaymentGateway` ein, die dem Shop gehört, nicht einem Anbieter. Sie ist in den eigenen Begriffen des Shops formuliert: Beträge als ganze Zahlen in der kleinsten Währungseinheit mit einer Währung, die Bestellkennung des Shops, ein Idempotenzschlüssel und drei Ergebnisse. Jeder Anbieter erhält einen Adapter, der die Schnittstelle implementiert und in die API, die Fehlercodes und die Eigenheiten dieses Anbieters übersetzt und zurück. Die Konfiguration wählt den Adapter je Land. Das ist derselbe Schritt wie der Vermittler in Abschnitt 2.6, nun als Information Hiding betrachtet: Die verborgene Entscheidung ist, *welcher Anbieter und wie er aufgerufen wird*.

```
from typing import Protocol, Literal
from dataclasses import dataclass

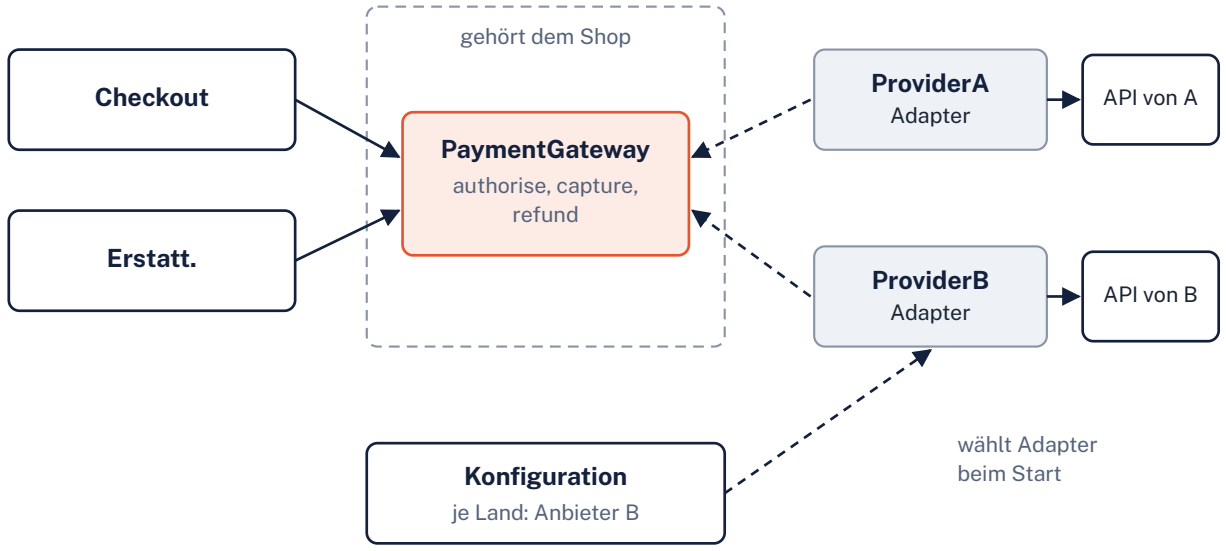
@dataclass(frozen=True)
class PaymentResult:
    outcome: Literal["approved", "declined", "unknown"]  # unknown: safe to retry
    reference: str | None        # our reference, not the provider's object

class PaymentGateway(Protocol):
    def authorise(self, order_id: str, amount_minor: int, currency: str,
                  idempotency_key: str) -> PaymentResult: ...
    def capture(self, reference: str) -> PaymentResult: ...
    def refund(self, reference: str, amount_minor: int) -> PaymentResult: ...

class ProviderBAdapter:            # implements PaymentGateway
    def authorise(self, order_id, amount_minor, currency, idempotency_key):
        resp = self._client.create_charge(value=amount_minor / 100,   # B wants decimals
                                          currency=currency, key=idempotency_key)
        return PaymentResult(_map_status(resp.code), _store_ref(resp.id))
```

Die Schnittstelle gehört dem Shop und spricht seine Sprache; nur die Adapter wissen, dass es einen Anbieter gibt.

Die Zahlungsschnittstelle und ihre Adapter



Aufrufer sehen nur die Schnittstelle; Anbieternamen, Fehlercodes, Retries und Währungsregeln bleiben in den Adaptern
Durchgezogen: hängt ab von oder ruft auf. Gestrichelt: implementiert oder wählt.

Aufrufer hängen von einer Schnittstelle ab, die dem Shop gehört; Adapter implementieren sie, und die Konfiguration wählt, welcher läuft.

AUSGEARBEITETES BEISPIEL

Was die Schnittstelle verbirgt und was sie bewusst nicht verbirgt

Den Entwurf an den Ideen dieses Abschnitts prüfen.

Verborgene Entscheidung	Welcher Anbieter, sein SDK und seine API-Version, seine Fehlercodes, seine Retry-Regeln, seine Dezimal- oder Ganzzahlbeträge, seine Transaktionskennungen.
Vertrag (Vorbedingungen)	Positiver Betrag in der kleinsten Währungseinheit, eine unterstützte Währung nach ISO 4217, ein eindeutiger Idempotenzschlüssel pro Zahlungsversuch.
Vertrag (Nachbedingungen)	`approved` bedeutet, dass der Betrag reserviert ist und eine Referenz zurückgegeben wird; `declined` bedeutet, dass kein Geld bewegt wurde; `unknown` bedeutet, dass sich das Ergebnis nicht feststellen ließ und derselbe Aufruf mit demselben Schlüssel wiederholt werden darf.
Bewusstes Nicht-Verbergen	Die Schnittstelle legt `unknown` offen, statt so zu tun, als würden Zahlungen immer gelingen oder scheitern: Das Netzwerkleck ist Teil des Vertrags, keine Überraschung.
Kosten eines neuen Anbieters	Vorher: Änderungen an 11 Aufrufstellen plus der Bestelltabelle. Nachher: eine neue Adapterklasse, ihre Tests und ein Konfigurationseintrag.
Tiefe	Drei Operationen verbergen eine beträchtliche Menge anbieterspezifischer Logik: eher tief als flach.

Achten Sie auf die Falle des kleinsten gemeinsamen Nenners: Bietet Anbieter B eine nützliche Funktion, die A fehlt, etwa Ratenzahlung, muss die Schnittstelle entweder um eine optionale Fähigkeit wachsen, oder die Funktion ist nicht verfügbar. Abstraktionen über Lieferanten tauschen immer einen Teil der Stärken jedes Lieferanten gegen die Freiheit zu wechseln.

Hyrums Gesetz gilt trotzdem: Beginnt der Checkout, von der typischen Latenz von Anbieter A abzuhängen, wird ein langsamerer Anbieter B das offenlegen. Nehmen Sie Timeouts in den Vertrag auf.

ACHTUNG

Häufige Fehler

- Getter und Setter für jedes Feld: gekapselt, aber die Darstellung liegt vollständig offen.
- Schnittstellen, die nach der aktuellen Implementierung benannt sind, etwa `StripePaymentService` oder `S3FileStore`, was den Zweck der Schnittstelle zunichtemacht.
- Durchreichende Schichten, die eine Schnittstelle hinzufügen, aber nichts verbergen, und das System flacher und mühsamer zu lesen machen.
- Fehler verbergen: entfernte Aufrufe, die als lokale getarnt sind, ohne Timeout und ohne Möglichkeit, ein unbekanntes Ergebnis von einer Ablehnung zu unterscheiden.
- Interne Objekte wie Datenbankzeilen oder die Antwort eines Anbieters über eine öffentliche API zurückgeben, was sie stillschweigend zum Teil des Vertrags macht.

Wie Fachleute es anwenden

Erfahrene Designer beginnen eine Schnittstelle bei den Bedürfnissen ihrer Aufrufer, formuliert in deren Sprache, und entwerfen erst dann die Implementierung. Sie führen eine Liste der Entscheidungen, die jedes wichtige Modul verbirgt, und prüfen neue Schnittstellen dagegen. Sie behandeln jede öffentliche API als Produkt mit einem Lebenszyklus (Vorschlag, Review, Versionierung, Abkündigung), weil nach Hyrums Gesetz alles Veröffentlichte genutzt wird. Und sie akzeptieren, dass Abstraktionen undicht sind, und machen die Lecks deshalb explizit: Latenzbudgets, Fehlersemantik und Retry-Regeln stehen im Vertrag.

Information Hiding sagt Ihnen, was jedes Modul für sich behalten sollte. Es sagt Ihnen nicht, in welche Richtung die Abhängigkeiten zwischen Modulen zeigen sollten oder was passiert, wenn sie Schleifen bilden. Das ist das Thema von Abschnitt 3.4.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum kann eine Klasse mit privaten Feldern und öffentlichen Gettern und Settern für alle davon beim Information Hiding trotzdem versagen?

Weil die Schnittstelle die genaue interne Darstellung preisgibt, sodass Aufrufer davon abhängen und jede Änderung der Darstellung sie bricht; Kapselung ist vorhanden, aber keine Entscheidung ist verborgen.

▼ **Wer ist bei Design by Contract schuld, wenn eine Vorbedingung verletzt wird?**

Der Aufrufer, denn die Vorbedingung zu erfüllen ist seine Pflicht; eine verletzte Nachbedingung weist auf einen Fehler beim Anbieter hin.

▼ **Was macht ein Modul im Sinne Ousterhouts tief?**

Es bietet viel Funktionalität hinter einer kleinen, einfachen Schnittstelle, sodass die Kosten, die Schnittstelle zu lernen, im Vergleich zu dem, was sie verbirgt, gering sind.

▼ **Warum spricht Hyrums Gesetz für kleine API-Antworten?**

Auf jedes beobachtbare Feld oder Verhalten wird sich irgendwann ein Client verlassen, sodass alles Offengelegte schwer zu ändern wird; weniger offenzulegen lässt mehr Freiheit.

▼ **Probieren Sie es aus: Kritisieren Sie diese Benachrichtigungsschnittstelle: sendSms(phone, text, vendorRetryCount), die ein VendorXResponse-Objekt zurückgibt, das die Aufrufer auf Fehlercodes des Anbieters untersuchen. Was gibt sie preis, und was würden Sie ändern, ohne bei Fehlern unehrlich zu werden?**

Sie legt die Retry-Einstellung, den Antworttyp und die Fehlercodes des Anbieters offen, sodass ein Anbieterwechsel jeden Aufrufer brechen würde. Definieren Sie eine Schnittstelle, die Ihrem eigenen Unternehmen gehört, etwa send_message(recipient, text), die gesendet, abgelehnt oder später versuchen zurückgibt. Verbergen Sie Wiederholungen, SDK und Codes in einem Adapter. Tun Sie nicht so, als könne sie nicht fehlschlagen: Der Vertrag sollte sagen, dass die Zustellung sich verzögern oder fehlschlagen kann und wie schnell sie antwortet. Abstraktionen sind undicht, legen Sie diese Fakten also ehrlich offen.

QUELLEN

- Joel Spolsky, The Law of Leaky Abstractions (2002): https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/
- Hyrum's Law: https://www.hyrumslaw.com/
- Bertrand Meyer, Applying 'Design by Contract' (IEEE Computer, 1992): https://se.inf.ethz.ch/~meyer/publications/computer/contract.pdf
- John Ousterhout, A Philosophy of Software Design (book page): https://web.stanford.edu/~ouster/cgi-bin/book.php
- Martin Fowler, Tolerant Reader: https://martinfowler.com/bliki/TolerantReader.html
- RFC 9413: Maintaining Robust Protocols: https://www.rfc-editor.org/rfc/rfc9413.html
- RFC 761: Transmission Control Protocol: https://www.rfc-editor.org/rfc/rfc761.html

Abhängigkeiten steuern: Richtung, Zyklen und Stabilität

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.3 Code organisieren
- 3.1 Modularität
- 3.2 Kopplung und Kohäsion

Begriffe von früher, die er verwendet: Afferente Kopplung (Ca), Efferente Kopplung (Ce), Inversion of Control, Fitness Function, Schnittstelle

DIE IDEE IN KÜRZE

Denken Sie an ein Gebäude. Die oberen Stockwerke ruhen auf den unteren, und das Fundament ruht auf nichts darüber. Sie können das oberste Stockwerk renovieren, ohne das Fundament anzurühren, aber wenn Sie unter dem Fundament graben, ist alles darüber in Gefahr. Stellen Sie sich nun zwei Stockwerke vor, die sich gegenseitig tragen: Sie könnten keines von beiden je umbauen. Softwaremodule ruhen auf dieselbe Weise aufeinander. Gute Strukturen setzen die Teile, die sich oft ändern, nach oben, auf solide, selten geänderte Teile darunter, und lassen nie zwei Teile sich gegenseitig in einer Schleife tragen.

WAS SIE WISSEN MÜSSEN

- Eine Abhängigkeit zeigt von dem Modul, das etwas braucht, zu dem Modul, das es braucht; Änderungen fließen entgegen den Pfeilen, daher entscheidet die Richtung der Abhängigkeiten, wer von wessen Änderungen betroffen ist.
- Das Acyclic Dependencies Principle verbietet Zyklen zwischen Komponenten, weil jede Komponente in einem Zyklus zusammen mit den anderen gebaut, getestet und ausgeliefert werden muss.
- Das Stable Dependencies Principle besagt: Abhängigkeiten in Richtung der Stabilität; die Instabilität ist I = Ce / (Ca + Ce), von 0 (nur eingehende Abhängigkeiten) bis 1 (nur ausgehende).
- Das Stable Abstractions Principle besagt, dass stabile Komponenten abstrakt sein sollten; die Abstraktheit A ist die Zahl der abstrakten Typen geteilt durch alle Typen, und die Distanz zur Hauptreihe ist D = |A + I - 1|.

- Zyklen werden durch Abhängigkeitsumkehr aufgelöst (das Modul, von dem abgehangen wird, besitzt eine Schnittstelle, die das andere implementiert) oder indem man den gemeinsamen Teil in eine neue, stabilere Komponente herauslöst.
- Strikte Schichtenbildung erlaubt Aufrufe nur in die nächsttiefere Schicht, gelockerte in jede tiefere Schicht; beide, wie jede andere Abhängigkeitsregel, sollten automatisch geprüft werden, mit Werkzeugen wie ArchUnit, dependency-cruiser oder import-linter.

Warum die Richtung zählt

Eine Abhängigkeit besteht, wenn ein Modul ein anderes braucht, um zu kompilieren, zu laufen oder sich korrekt zu verhalten. Zeichnen Sie sie als Pfeil von dem Modul, das abhängt, zu dem Modul, von dem es abhängt: Checkout zu Preisgestaltung. Der Pfeil sagt etwas Wichtiges über Änderungen aus. Ändert die Preisgestaltung ihre Schnittstelle oder ihr Verhalten, muss sich der Checkout womöglich ändern; ändert sich der Checkout, ist das der Preisgestaltung gleich. Änderungen laufen entgegen der Pfeilrichtung. Die Richtung jeder Abhängigkeit ist also eine Entscheidung darüber, wer durch wessen Änderungen gestört wird.

Architekten befassen sich mit zwei Arten von Richtung, die man leicht verwechselt. Der *Kontrollfluss* ist, wer wen zur Laufzeit aufruft. Die *Quellcode-Abhängigkeit* ist, wer wen importiert, referenziert oder gegen wen kompiliert. Meist zeigen beide in dieselbe Richtung, aber das muss nicht so sein. Ein Framework ruft Ihren Code auf, ohne von ihm abzuhängen (Abschnitt 0.3 nannte das Inversion of Control), und mit einer Schnittstelle kann ein übergeordnetes Modul ein untergeordnetes aufrufen, während die Quellcode-Abhängigkeit in die andere Richtung zeigt. Diese Freiheit, die Richtung der Quellcode-Abhängigkeit unabhängig vom Kontrollfluss zu wählen, ist das mächtigste Werkzeug dieses Abschnitts.

Zwischen Services zeigt sich Richtung als Wissen. Ruft Bestellungen die API der Abrechnung auf, hängt Bestellungen von der Abrechnung ab und muss sich ändern, wenn sich deren API ändert. Veröffentlicht Bestellungen ein Ereignis, das die Abrechnung konsumiert, kehrt sich die Abhängigkeit um: Die Abrechnung hängt vom Ereignisvertrag von Bestellungen ab, und Bestellungen weiß nicht, dass es die Abrechnung gibt (Abschnitt 3.2). Die Wahl zwischen einem Aufruf und einem Ereignis ist daher teilweise eine Wahl der Abhängigkeitsrichtung.

Das Acyclic Dependencies Principle

Robert C. Martin formulierte drei Prinzipien für Abhängigkeiten zwischen Komponenten, zuerst Mitte der 1990er-Jahre in Artikeln und Aufsätzen und später in *Agile Software Development: Principles, Patterns, and Practices* (2002) und *Clean Architecture* (2017). Das erste, das Acyclic Dependencies Principle (ADP), besagt, dass der Abhängigkeitsgraph zwischen Komponenten keine Zyklen enthalten darf: Er muss ein gerichteter azyklischer Graph sein.

Ein Abhängigkeitszyklus wie „Bestellungen hängt von Benachrichtigungen ab, Benachrichtigungen von Kunden und Kunden von Bestellungen“ macht aus drei Komponenten eine. Keine lässt sich ohne die anderen kompilieren oder testen; eine Änderung an einer kann sich durch die Schleife bis zu ihr selbst zurück fortpflanzen; keine kann eigenständig ausgeliefert werden; und niemand kann sagen, welche zuerst zu bauen ist. Martin führt das Prinzip mit dem ein, was er das *Morning-after-Syndrom* nennt: Sie gehen mit funktionierendem Code nach Hause und stellen am nächsten Tag fest, dass er kaputt ist, weil jemand anderes etwas geändert hat, wovon Sie abhängen. Sein Heilmittel ist, das System in getrennt releaste Komponenten aufzuteilen, und das funktioniert nur, wenn die Abhängigkeiten zwischen ihnen keine Zyklen enthalten. Zyklen machen es außerdem unmöglich, eine Komponente später als Bibliothek oder Service herauszulösen, weil sie die ganze Schleife mitzieht.

Das Java Platform Module System erzwingt ADP zwischen Modulen: Die Java Language Specification macht es zu einem Kompilierfehler, wenn ein Modul direkt oder indirekt von sich selbst abhängt. Die meisten anderen Sprachen erlauben Zyklen zwischen Paketen, daher prüfen Teams sie mit Werkzeugen. Zyklen zwischen Klassen innerhalb einer kleinen Komponente sind weniger schädlich, und unter anderem Martin Fowler toleriert sie dort; am wichtigsten ist das Prinzip zwischen Komponenten, die verschiedene Leute bauen und releasen.

Zyklen auflösen

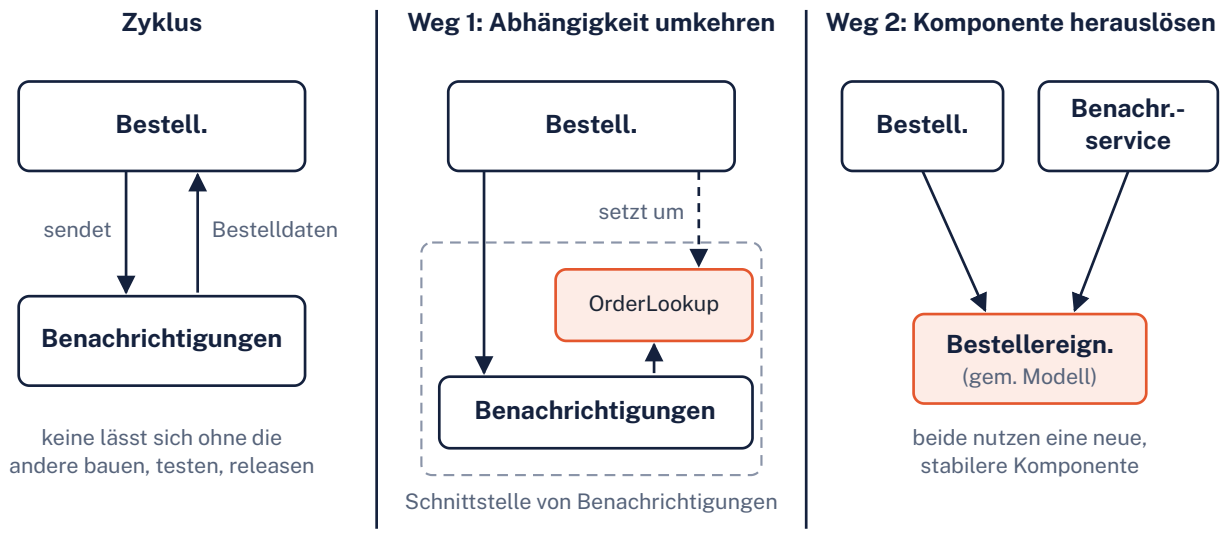
Jeder Zyklus lässt sich auflösen, und es gibt zwei Standardtechniken. Beide zeigt das Diagramm unten für einen Zyklus, in dem Bestellungen die Benachrichtigungen aufruft, um eine Bestätigung zu senden, und Benachrichtigungen in Bestellungen zurückruft, um die Details der Bestellung zu lesen.

Abhängigkeitsumkehr. Benachrichtigungen deklariert eine Schnittstelle, die genau beschreibt, was es braucht, etwa `OrderLookup` mit einer Methode, und Bestellungen implementiert sie. Der Pfeil von Benachrichtigungen zu Bestellungen zeigt nun in die andere Richtung: Bestellungen hängt von Benachrichtigungen ab (um es aufzurufen und seine Schnittstelle zu implementieren), und Benachrichtigungen hängt von nichts in Bestellungen ab. Das ist Abhängigkeitsumkehr, die Abschnitt 3.5 als das D in SOLID behandelt. Das entscheidende Detail ist die Eigentümerschaft: Die Schnittstelle gehört dem Modul, das sie *nutzt*, nicht dem, das sie implementiert.

Eine neue Komponente herauslösen. Finden Sie heraus, was beide Seiten tatsächlich voneinander brauchen, oft ein kleines gemeinsames Modell oder ein Vertrag, und verschieben Sie es in eine neue Komponente, von der beide abhängen. Hier veröffentlicht Bestellungen einen Ereignistyp, der in einer Komponente *Bestellereignisse* definiert ist, und

Benachrichtigungen liest die Bestelldetails aus dem Ereignis, statt zurückzurufen. Von der neuen Komponente hängen beide ab, und sie hängt von keiner ab, ist also stabiler als jede von ihnen: genau das, was das nächste Prinzip verlangt.

Zwei Wege, einen Zyklus aufzulösen



Weg 1: Bestellungen ruft Benachr. auf, das seinen Bedarf als Schnittstelle nennt; Bestellungen implementiert sie.
Weg 2: Der gemeinsame Teil kommt in eine eigene Komponente; kein Lesen mehr aus Bestellungen.

Pfeile zeigen vom abhängigen Modul zu dem Modul, von dem es abhängt. Gestrichelt: implementiert.

Ein Zyklus lässt sich auflösen, indem man eine Abhängigkeit über eine Schnittstelle umkehrt, die ihrem Nutzer gehört, oder indem man das, was beide Seiten teilen, in eine neue Komponente herauslöst.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: derselbe Aufruf, umgekehrter Pfeil

Der Punkt, an dem hier die meisten stolpern, ist, dass „wer wen aufruft, wenn das Programm läuft“ und „wessen Code wen erwähnt“ zwei getrennte Dinge sind und dass Sie mit einer Schnittstelle das Zweite wählen können. „Zyklen auflösen“ hat gerade die Lösung für Bestellungen und Benachrichtigungen gezeigt; verfolgen Sie eine Bestätigungs-E-Mail zweimal hindurch, einmal so, wie sie abläuft, und einmal so, wie sie geschrieben ist.

- Vor der Lösung rief Bestellungen die Benachrichtigungen auf, um die Bestätigung zu senden, und Benachrichtigungen rief in Bestellungen zurück, um die Details der Bestellung zu holen. Der Code jedes Moduls nannte das andere: ein Abhängigkeitszyklus.
- Nach der Lösung beobachten Sie den Ablauf. Bestellungen ruft Benachrichtigungen auf wie zuvor. Beim Start hat Benachrichtigungen ein OrderLookup übergeben bekommen, das in Wirklichkeit die eigene Implementierung von Bestellungen ist. Fragt es also nach den Details der Bestellung, läuft wie zuvor Code von Bestellungen: Der Kontrollfluss hat sich nicht geändert.
- Lesen Sie jetzt stattdessen den Code. Benachrichtigungen nennt nur OrderLookup, das es selbst deklariert, und nennt Bestellungen nie. Zur Laufzeit geht der Aufruf also von Benachrichtigungen in den Code von Bestellungen, während der Quellcode-Pfeil von Bestellungen zu Benachrichtigungen zeigt.
- Dieser Unterschied ist der Gewinn. Benachrichtigungen lässt sich jetzt ohne irgendetwas aus Bestellungen im Build kompilieren und testen, mit einem kleinen gefälschten OrderLookup, das erfundene Bestelldetails liefert.
- Die Eigentümerschaft sorgt dafür, dass es funktioniert. Wäre OrderLookup innerhalb von Bestellungen deklariert, müsste Benachrichtigungen weiterhin den Code von Bestellungen nennen, um es zu nutzen, und der Zyklus bliebe bestehen. Die Schnittstelle muss bei Benachrichtigungen liegen, der Seite, die sie braucht.
- **Die Lehre:** Der Aufruf zur Laufzeit und die Quellcode-Abhängigkeit können in entgegengesetzte Richtungen zeigen. Eine Schnittstelle, die ihrem Nutzer gehört, lässt Sie die Abhängigkeit so ausrichten, wie Sie es brauchen; das ist die Abhängigkeitsumkehr, auf die dieser Abschnitt immer wieder zurückkommt.

Das Stable Dependencies Principle und Instabilität

Manche Komponenten ändern sich oft (Benutzeroberflächen, Integrationen mit Partnern, neue Features); andere sollten sich selten ändern (zentrale Domänenkonzepte, gemeinsame Verträge, der Umgang mit Geld). Das Stable Dependencies Principle (SDP) besagt:

Abhängigkeiten in Richtung der Stabilität. Von einer Komponente, die leicht zu ändern sein soll, sollte keine Komponente abhängen, die schwer zu ändern ist, sonst wird sie ebenfalls schwer zu ändern.

Martin misst Stabilität über Abhängigkeiten, nicht darüber, wie oft der Code bearbeitet wird. Eine Komponente, von der viele andere abhängen, ist *stabil* in dem Sinne, dass sie teuer zu ändern ist: Jede Änderung riskiert, ihre abhängigen Komponenten zu brechen. Eine Komponente, die von vielen anderen abhängt und keine abhängigen Komponenten hat, ist *instabil*: Nichts hindert sie daran, sich zu ändern, und ihre Abhängigkeiten können sie dazu zwingen. Mit der afferenten Kopplung (Ca, eingehend) und der efferenten Kopplung (Ce, ausgehend) aus Abschnitt 3.2 ist die Instabilität:

I = Ce / (Ca + Ce)

I reicht von 0 bis 1. I = 0 bedeutet nur eingehende Abhängigkeiten: maximal stabil, vielen gegenüber verantwortlich, von nichts abhängig. I = 1 bedeutet nur ausgehende Abhängigkeiten: maximal instabil, nichts hängt davon ab. Als Metrik formuliert besagt das SDP, dass I entlang jedes Abhängigkeitspfeils abnehmen sollte: Jede Komponente sollte von Komponenten mit einem niedrigeren I als ihrem eigenen abhängen. Eine Abhängigkeit von einer stabilen Komponente (I = 0,1) zu einer volatilen (I = 0,8) ist ein Warnsignal; die übliche Lösung ist wiederum Abhängigkeitsumkehr, also eine Schnittstelle an einer stabilen Stelle.

Das Stable Abstractions Principle und die Hauptreihe

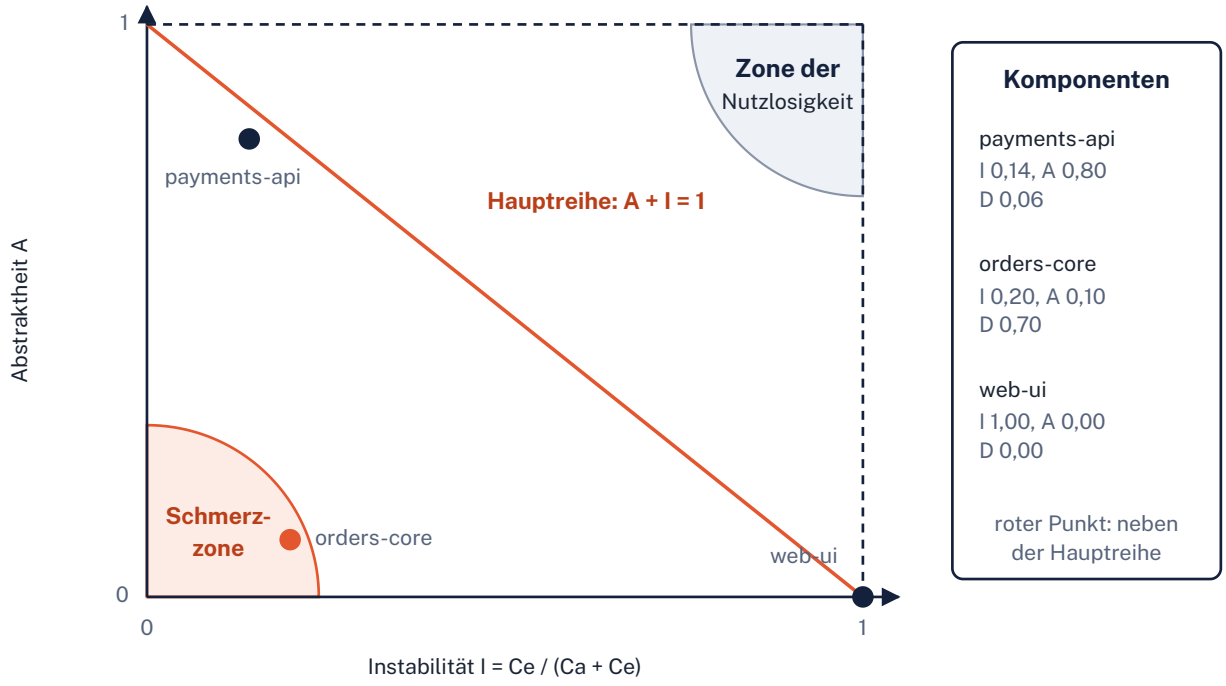
Stabilität hat ihren Preis: Eine Komponente mit I nahe 0 ist schwer zu ändern. Besteht sie aus viel konkretem Code, wird das System starr. Das Stable Abstractions Principle (SAP) löst das: Eine Komponente sollte so abstrakt sein, wie sie stabil ist. Stabile Komponenten sollten überwiegend aus Schnittstellen und abstrakten Typen bestehen, damit sich ihr Verhalten durch neue Implementierungen an anderer Stelle erweitern lässt, ohne sie zu ändern; instabile Komponenten dürfen konkret sein, weil sie ohnehin leicht zu ändern sind. SDP und SAP zusammen sind das Dependency Inversion Principle, angewandt auf ganze Komponenten: Abhängigkeiten zeigen zur Stabilität, und Stabilität geht mit Abstraktion einher.

Die Abstraktheit wird gemessen als A = Na / Nc, wobei Na die Zahl der abstrakten Klassen und Schnittstellen in der Komponente ist und Nc die Gesamtzahl der Klassen. A = 0 bedeutet vollständig konkret; A = 1 bedeutet nichts als Abstraktionen. Trägt man A gegen I auf, ergibt sich ein Quadrat. Die Linie von (I = 0, A = 1) bis (I = 1, A = 0) ist die Hauptreihe: stabil und abstrakt an einem Ende, instabil und konkret am anderen, mit ausgewogenen Mischungen dazwischen. Die Distanz zu ihr ist:

D = |A + I - 1|

D reicht von 0 (auf der Linie) bis 1 (in einer entfernten Ecke). Die Ecke nahe (0, 0) ist die *Zone des Schmerzes*: stabil, weil viel davon abhängt, und konkret, sodass jede Änderung sowohl nötig als auch schmerzhaft ist. Ein Datenbankschema, das jedes Modul direkt liest, liegt hier. Martin merkt an, dass manche Komponenten harmlos dorthin gehören, etwa eine ausgereifte konkrete Hilfsbibliothek, die sich nie ändern muss; die Zone schmerzt nur bei volatilen Komponenten. Die Ecke nahe (1, 1) ist die *Zone der Nutzlosigkeit*: Abstraktionen, von denen nichts abhängt, typischerweise Schnittstellen, die nach einem Refactoring übrig geblieben sind oder für eine Erweiterung entworfen wurden, die nie kam.

Die Hauptreihe



Komponenten nahe der Hauptreihe halten Abstraktheit und Stabilität im Gleichgewicht; die Zone des Schmerzes enthält stabilen konkreten Code, die Zone der Nutzlosigkeit Abstraktionen, die niemand nutzt.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: I, A und D für drei Komponenten berechnen

Ein Shop hat drei Komponenten. Nach Martins Definition zählt Ca die Klassen außerhalb der Komponente, die von Klassen in ihr abhängen, und Ce die Klassen in ihr, die von Klassen außerhalb abhängen. Die Zahlen stammen aus einem Abhängigkeitswerkzeug.

payments-api (5 Klassen, davon 4 Schnittstellen)	Ca = 6, Ce = 1. I = 1 / (6 + 1) = 0,14. A = 4 / 5 = 0,80. D = |0,80 + 0,14 - 1| = 0,06.
orders-core (20 Klassen, 2 abstrakt)	Ca = 8, Ce = 2. I = 2 / (8 + 2) = 0,20. A = 2 / 20 = 0,10. D = |0,10 + 0,20 - 1| = 0,70.
web-ui (12 Klassen, keine abstrakt)	Ca = 0, Ce = 9. I = 9 / (0 + 9) = 1,00. A = 0 / 12 = 0,00. D = |0 + 1 - 1| = 0,00.
SDP entlang der Pfeile prüfen	web-ui (1,00) hängt von orders-core (0,20) und von payments-api (0,14) ab; orders-core (0,20) hängt von payments-api (0,14) ab. I nimmt entlang jedes Pfeils ab, also gilt das SDP.
Diagnose	orders-core hat D = 0,70: stabil (viele abhängige Klassen), aber konkret. Ändern sich seine Regeln oft, gefährdet jede Änderung acht abhängige Klassen. Es liegt in der Zone des Schmerzes.
Abhilfe und Neuberechnung	Die vier Typen, die seine abhängigen Klassen nutzen, in orders-api herauslösen (3 Schnittstellen und 1 Werttyp: A = 0,75; Ca ist 8 oder mehr, da nun auch die implementierenden Klassen davon abhängen, und Ce = 0, also I = 0; D = |0,75 + 0 - 1| = 0,25). Die übrigen 16 konkreten Klassen werden zu orders-impl, von dem nur die Start-Verdrahtung abhängt (Ca = 1) und das von orders-api und zwei weiteren abhängt (Ce = 3): I = 3 / 4 = 0,75, A = 0, D = 0,25.

Die Abhilfe rückt beide Teile viel näher an die Hauptreihe (jeweils von 0,70 auf 0,25). Abhängige Klassen verlassen sich nun auf eine kleine, stabile, überwiegend abstrakte Komponente, und der volatile konkrete Code liegt dort, wo er sich günstig ändern lässt.

Die Zahlen sind Heuristiken, keine Zielwerte. Ein D von 0,70 ist ein Grund hinzusehen, kein Beweis für ein Problem; würde sich orders-core nie ändern, richtete es dort, wo es ist, keinen Schaden an.

ACHTUNG

Die Metriken vernünftig nutzen

- Werkzeuge unterscheiden sich darin, was sie zählen (Klassen, Dateien, Pakete) und ob sie Bibliotheken von Drittanbietern einbeziehen; vergleichen Sie nur Zahlen aus demselben Werkzeug und derselben Konfiguration.
- In Sprachen ohne abstrakte Klassen, etwa Python mit Duck Typing, ist A nur ein Näherungswert; zählen Sie Protocols und abstrakte Basisklassen und behandeln Sie die Metrik als Anstoß für eine Diskussion.
- Trends sind aussagekräftiger als Momentaufnahmen: Eine Komponente, deren D von Release zu Release wächst, gerät allmählich in Schwierigkeiten.
- Metriken lassen sich austricksen, etwa indem man ungenutzte Schnittstellen hinzufügt, um A zu erhöhen. Abschnitt 3.8 kommt auf Goodharts Gesetz zurück.

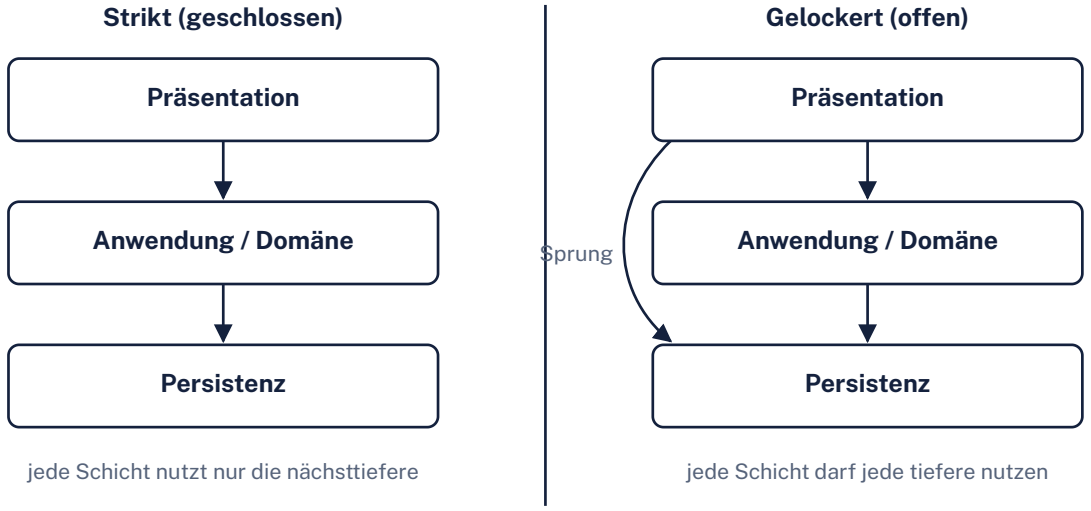
Strikte und gelockerte Schichtenbildung

Schichtenbildung ist die älteste und häufigste Abhängigkeitsregel: Module werden in Schichten angeordnet, etwa Präsentation, Anwendung oder Domäne und Persistenz, und jede Schicht darf nur Schichten unter ihr nutzen. Untere Schichten wissen nichts über höhere, sodass sich eine neue Benutzeroberfläche hinzufügen lässt, ohne die Domäne zu ändern. Schichtenbildung ist ein eigener Architekturstil (Abschnitt 5.2); hier interessiert die Abhängigkeitsregel, die sie vorgibt.

Bei **strikter** (geschlossener) Schichtenbildung darf jede Schicht nur die unmittelbar darunterliegende Schicht nutzen. Jede Schicht verbirgt dann alles unter ihr, und das Ersetzen einer unteren Schicht betrifft nur eine andere Schicht; der Preis ist durchreichender Code, bei dem die mittlere Schicht bloß Aufrufe weiterleitet, zu denen sie nichts beizutragen hat. Bei **gelockerter** (offener) Schichtenbildung darf eine Schicht jede Schicht unter ihr nutzen. Das beseitigt den Boilerplate-Code, bedeutet aber, dass eine Änderung an der untersten Schicht jede Schicht darüber betreffen kann. Viele reale Systeme mischen beides und markieren bestimmte Schichten als offen, etwa eine gemeinsame Logging- oder Hilfsschicht, die alle nutzen dürfen.

Die klassische Schichtenbildung hat eine berüchtigte Schwäche: Die Domänenschicht hängt von der Persistenzschicht ab, sodass die wichtigsten Geschäftsregeln von Datenbankdetails abhängen. Hexagonale, Onion- und Clean Architecture (Abschnitt 5.6) wenden Abhängigkeitsumkehr an, um das zu beheben: Die Domäne besitzt Schnittstellen wie `OrderRepository`, und der Persistenzcode implementiert sie, sodass alle Quellcode-Abhängigkeiten nach innen zur Domäne zeigen.

Strikte und gelockerte Schichtenbildung



Strikte Schichtenbildung erlaubt Aufrufe nur in die nächsttiefere Schicht; gelockerte Schichtenbildung erlaubt einer Schicht auch, darunterliegende Schichten zu überspringen.

Abhängigkeitsregeln automatisch prüfen

Abhängigkeitsregeln, die nur in einem Wiki stehen, werden innerhalb von Wochen gebrochen. Der verlässliche Weg, sie einzuhalten, ist, sie als ausführbare Prüfungen zu formulieren, die im Build laufen und ihn fehlschlagen lassen, wenn eine Regel verletzt wird. Eine solche Prüfung ist eine architektonische Fitness Function, eine Idee, die in Abschnitt 2.8 eingeführt und in Abschnitt 11.3 weiterentwickelt wird. Drei weit verbreitete Open-Source-Werkzeuge decken die wichtigsten Ökosysteme ab.

ArchUnit (Java und andere JVM-Sprachen) lässt Sie Architekturregeln als gewöhnliche Unit-Tests schreiben: welche Pakete auf welche zugreifen dürfen, Schichtdefinitionen und Zyklenfreiheit zwischen Teilen der Codebasis. **dependency-cruiser** (JavaScript und TypeScript) analysiert Importe und prüft sie gegen Regeln in einer Konfigurationsdatei, und seine Startkonfiguration verbietet bereits zirkuläre Abhängigkeiten. **import-linter** (Python) prüft in einer Konfigurationsdatei deklarierte *Contracts*; zu seinen Contract-Typen gehören verbotene Importe, Schichten, Unabhängigkeit zwischen benachbarten Modulen, geschützte Module und azyklische Geschwistermodule.

```
@ArchTest
static final ArchRule layers = layeredArchitecture()
    .consideringAllDependencies()
    .layer("Web").definedBy("..web..")
    .layer("Domain").definedBy("..domain..")
    .layer("Persistence").definedBy("..persistence..")
    .whereLayer("Web").mayNotBeAccessedByAnyLayer()
    .whereLayer("Domain").mayOnlyBeAccessedByLayers("Web", "Persistence")
    .whereLayer("Persistence").mayNotBeAccessedByAnyLayer();

@ArchTest
static final ArchRule noCycles =
    slices().matching("com.shop.(*)..").should().beFreeOfCycles();
```

ArchUnit (Java): eine Schichtenregel und eine Zyklenprüfung, als Tests geschrieben.

```
// .dependency-cruiser.js
module.exports = {
  forbidden: [
    { name: "no-circular", severity: "error",
      from: {}, to: { circular: true } },
    { name: "domain-not-to-db", severity: "error",
      comment: "Domain code must use repository interfaces, not the database",
      from: { path: "^src/domain" },
      to: { path: "^src/db" } },
  ],
};
```

dependency-cruiser (JavaScript und TypeScript): Zyklen verbieten und verhindern, dass die Domäne die Datenbankschicht importiert.

```
# .importlinter
[importlinter]
root_package = shop

[importlinter:contract:layers]
name = Web above domain above adapters
type = layers
layers =
    shop.web
    shop.domain
    shop.adapters

[importlinter:contract:features-independent]
name = Feature packages do not import each other
type = independence
modules =
    shop.catalogue
    shop.ordering
    shop.payments
```

import-linter (Python): ein Schichten-Contract und ein Unabhängigkeits-Contract, geprüft durch Ausführen von lint-imports in der Pipeline.

Beachten Sie, dass die ArchUnit-Regel oben der Persistenz den Zugriff auf die Domäne erlaubt, aber nicht umgekehrt: die eben beschriebene domänenzentrierte Anordnung, nicht die klassische Schichtenbildung. Solche Regeln sind schnell hinzugefügt, laufen in Sekunden und machen aus einem Architekturdiagramm etwas, das der Build verteidigt. Verletzt eine bestehende Codebasis eine Regel bereits, können die meisten Werkzeuge die aktuellen Verstöße als Baseline erfassen und nur bei neuen fehlschlagen, sodass die Regel ab dem Tag ihrer Einführung verhindert, dass es schlimmer wird.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: was tun mit einer unerwünschten Abhängigkeit

Nutzen Sie dies, wenn eine Abhängigkeitsprüfung fehlschlägt oder ein Review eine Abhängigkeit findet, die in die falsche Richtung zeigt.

Q1 **Ist die Abhängigkeit Teil eines Zyklus zwischen Komponenten?**

- **Ja** Lösen Sie zuerst den Zyklus auf (ADP): Lesen Sie unten weiter, um zu entscheiden, wie.
- **Nein** Gehen Sie zur Frage nach der Stabilität.

Q2 **Braucht eine Seite von der anderen nur einen kleinen, klar umrissenen Dienst?**

- **Ja** Kehren Sie die Abhängigkeit um: Die Seite, die den Dienst nutzt, besitzt eine Schnittstelle, und die andere Seite implementiert sie.
- **Nein** Weiter zur nächsten Frage.

Q3 **Hängen beide Seiten von einem gemeinsamen Modell, Vertrag oder Stück Logik ab?**

- **Ja** Lösen Sie es in eine neue Komponente heraus, von der beide abhängen, und machen Sie diese Komponente klein, stabil und überwiegend abstrakt.
- **Nein** Prüfen Sie, ob die beiden Komponenten in Wahrheit eine sind: Führen Sie sie zusammen und teilen Sie sie dann entlang einer besseren Grenze neu auf (Abschnitt 3.1).

Q4 **Hängt eine stabile Komponente (niedriges I) von einer volatilen (hohes I) ab?**

- **Ja** Das verletzt das SDP. Legen Sie eine Schnittstelle in die stabile Komponente oder in eine neue stabile und verschieben Sie den volatilen Code dahinter.
- **Nein** Die Richtung ist akzeptabel. Halten Sie die Regel in einer Fitness Function fest, damit es so bleibt.

Wie Fachleute Abhängigkeiten steuern

Erfahrene Architekten haben ein Bild des Abhängigkeitsgraphen zwischen Komponenten und sehen es sich regelmäßig an, aus dem Code erzeugt statt von Hand gezeichnet. Sie entscheiden, welche Komponenten stabil sein sollen (Domänenkonzepte, gemeinsame Verträge) und welche volatil (Benutzeroberflächen, Integrationen), und lassen die Pfeile von volatil zu stabil zeigen. Neue Zyklen behandeln sie als Build-Fehler, nicht als Stilfragen. Und sie wenden dasselbe Denken über Services hinweg an: Gemeinsame Bibliotheken und

gemeinsame Event-Schemas sind die stabilen Komponenten eines verteilten Systems und verdienen minimale Oberflächen und sorgfältige Versionierung, während die Interna jedes Service frei veränderbar bleiben.

Nichts davon erfordert, dass die Metriken genau berechnet werden. Ihr Wert liegt in den Fragen, die sie erzwingen: In welche Richtung sollte dieser Pfeil zeigen, was passiert mit allem, was davon abhängt, wenn sich das ändert, und ist diese Komponente so abstrakt, wie es die Zahl der Dinge verlangt, die von ihr abhängen? Abschnitt 3.5 geht weiter mit den SOLID-Prinzipien, die dieselben Ideen innerhalb von Komponenten und zwischen ihnen anwenden.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Warum müssen alle Komponenten in einem Abhängigkeitszyklus zusammen ausgeliefert werden?**

Jede hängt direkt oder indirekt von allen anderen ab, sodass eine Änderung an einer Änderungen an den übrigen erfordern kann; keine lässt sich ohne die anderen bauen oder testen.

▼ **Eine Komponente hat Ca = 3 und Ce = 9. Wie hoch ist ihre Instabilität, und was bedeutet das?**

I = 9 / 12 = 0,75. Sie hängt von viel mehr ab, als von ihr abhängt, ist also relativ instabil: leicht zu ändern, aber den Änderungen ihrer Abhängigkeiten ausgesetzt.

▼ **Was liegt in der Zone des Schmerzes, und wann schmerzt sie tatsächlich?**

Komponenten, die zugleich stabil (viele abhängige Komponenten) und konkret (wenige Abstraktionen) sind. Es schmerzt, wenn sich eine solche Komponente oft ändern muss; eine konkrete Komponente, die sich nie ändert, richtet dort wenig Schaden an.

▼ **Was ist der Unterschied zwischen strikter und gelockerter Schichtenbildung?**

Bei strikter Schichtenbildung darf eine Schicht nur die direkt darunterliegende Schicht nutzen; bei gelockerter darf sie jede Schicht unter ihr nutzen.

▼ **Probieren Sie es aus: Die Komponente pricing-rules hat Ca = 8, Ce = 0 und 12 Klassen, keine davon abstrakt. Die Komponente reports hat Ca = 2, Ce = 6 und 10 Klassen, davon 5 abstrakt. Berechnen Sie I, A und D für beide, und sagen Sie, welche Ihnen mehr Sorgen macht.**

pricing-rules: I = 0 / 8 = 0, A = 0, D = |0 + 0 - 1| = 1, mitten in der Zone des Schmerzes: Vieles hängt davon ab, doch es ist konkret, jede Änderung schmerzt also. reports: I = 6 / 8 = 0,75, A = 0,5, D = 0,25, recht nah an der Hauptreihe. pricing-rules ist die Sorge. Extrahieren Sie Schnittstellen, damit Abhängige sich auf stabile Abstraktionen stützen. Regeln, die sich oft ändern, machen das dringend.

QUELLEN

- Robert C. Martin, OO Design Quality Metrics: An Analysis of Dependencies (1994): https://bccdev.ime.usp.br/tccs/2010/lucianna-joao/arquivos/referencias/oodmetrics.pdf
- ArchUnit User Guide: https://www.archunit.org/userguide/html/000_Index.html
- Import Linter: contract types: https://import-linter.readthedocs.io/en/stable/contract_types/
- dependency-cruiser (GitHub): https://github.com/sverweij/dependency-cruiser
- The Java Language Specification, Chapter 7: Packages and Modules: https://docs.oracle.com/javase/specs/jls/se21/html/jls-7.html

Die SOLID-Prinzipien auf Komponenten angewandt

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 3.2 Kopplung und Kohäsion
- 3.3 Information Hiding
- 3.4 Abhängigkeiten

Begriffe von früher, die er verwendet: Design by Contract, Vorbedingung, Invariante, Hyrums Gesetz, Stable Dependencies Principle

DIE IDEE IN KÜRZE

Denken Sie an eine gut geführte Restaurantgruppe. Jede Küche untersteht einem Küchenchef, sodass ihr nie zwei Vorgesetzte widersprüchliche Anweisungen geben. Neue Gerichte kommen auf die Speisekarte, ohne dass die Küche umgebaut wird. Ein Vertretungskoch muss jedes Gericht genau so zubereiten wie der Stammkoch, sonst merken es die Gäste. Jeder Lieferant bekommt nur das Bestellformular, das er braucht, nicht das ganze Firmenhandbuch. Und die Zentrale setzt die Standards, während sich die Lieferanten danach richten, nicht umgekehrt. SOLID sind diese fünf Gewohnheiten, aufgeschrieben für Software.

WAS SIE WISSEN MÜSSEN

- SOLID bezeichnet fünf von Robert C. Martin zusammengetragene Prinzipien: Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation und Dependency Inversion; jedes gilt für Klassen, Komponenten und Services.
- Single Responsibility bedeutet einen Akteur pro Modul: Code, der auf Wunsch verschiedener Personengruppen geändert wird, sollte getrennt werden, und es bedeutet nicht, dass ein Modul nur eine Sache tun darf.
- Liskov Substitution verlangt Verhaltenskompatibilität: keine stärkeren Vorbedingungen, keine schwächeren Nachbedingungen und erhaltene Invarianten, und genau das ist der Test für eine abwärtskompatible API-Version.
- Interface Segregation spricht für kleine Rollenschnittstellen im Code und, zwischen Services, für clientspezifische APIs, fokussierte Events und kleine Bibliotheken.

- Dependency Inversion lässt Details von Abstraktionen abhängen, die der Seite der Fachlogik gehören; wem die Schnittstelle gehört, ist wichtiger als die Tatsache, dass es eine gibt.
- REP, CCP und CRP entscheiden, was in eine Komponente gehört, und ziehen in entgegengesetzte Richtungen; das richtige Gleichgewicht verschiebt sich mit zunehmender Reife eines Systems von CCP zu REP und CRP.
- Dogmatisch angewandt erzeugt SOLID flache, spekulative Abstraktionen; am besten dient es als Vokabular, um Entwürfe zu diagnostizieren, die echte Probleme verursachen.

Fünf Prinzipien mit Geschichte

SOLID ist eine Merkhilfe für fünf Designprinzipien, die Robert C. Martin Ende der 1990er-Jahre zusammentrug und verfeinerte, am ausführlichsten in seinem Aufsatz *Design Principles and Design Patterns* aus dem Jahr 2000. Michael Feathers bemerkte, dass sich ihre Anfangsbuchstaben zum Wort SOLID umstellen ließen, und das Akronym setzte sich durch. Die Prinzipien wurden für objektorientierte Klassen geschrieben, doch Martins späteres Buch *Clean Architecture* (2017) argumentiert, dass jedes von ihnen auch für Komponenten gilt, und in der Praxis wendet man sie ganz selbstverständlich auch auf Services und APIs an.

Deshalb gehören sie in einen Architekturkurs. Jedes Prinzip ist im Kern eine Aussage darüber, **wo Änderungen landen sollten** und **in welche Richtung Abhängigkeiten zeigen sollten**, also über die beiden Anliegen der vorigen Abschnitte zu Kopplung und Kohäsion und zum Steuern von Abhängigkeiten. Dieser Abschnitt nimmt sich die Prinzipien nacheinander vor, zeigt jedes in wenigen Zeilen Python und dann, was es bedeutet, wenn die „Klasse“ eine deploybare Komponente oder ein Service eines anderen Teams ist. Zum Schluss kommen die Kohäsionsprinzipien für Komponenten und die Kritik an SOLID, die erheblich ist und die man kennen sollte.

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist

Sie brauchen drei Begriffe aus früheren Abschnitten. Ein **Modul** ist jede Codeeinheit mit einer Grenze: eine Funktion, eine Klasse, ein Paket, eine Komponente oder ein Service. Eine **Schnittstelle** ist die Menge der Operationen, die ein Modul anderen anbietet, ein Vertrag, der verbirgt, wie die Arbeit erledigt wird. **Kopplung** beschreibt, wie viel ein Modul über ein anderes wissen muss; je mehr es weiß, desto eher bricht eine Änderung auf der einen Seite die andere. Pythons `Protocol`, das unten verwendet wird, ist einfach eine Möglichkeit, eine Schnittstelle zu benennen: Jede Klasse mit passenden Methoden erfüllt sie.

S: Single Responsibility bedeutet ein Akteur

Das Single Responsibility Principle ist das meistzitierte und am häufigsten missverstandene der fünf. Es besagt *nicht*, dass ein Modul nur eine Sache tun sollte; das ist eine gute Regel für Funktionen, aber eine andere Regel. Martins eigene Formulierung entwickelte sich von der Aussage, eine Klasse solle nur einen Grund zur Änderung haben, hin zu der Aussage in *Clean Architecture*, ein Modul solle einem und nur einem Akteur gegenüber verantwortlich sein. Ein Akteur ist eine Gruppe von Personen (Nutzer oder Stakeholder), die das System auf dieselbe Weise ändern wollen. Das Prinzip handelt von Menschen und von der Richtung künftiger Änderungen.

Das klassische Beispiel ist eine Klasse `Employee` mit drei Methoden: `calculate_pay`, vorgegeben von der Finanzabteilung; `report_hours`, vorgegeben vom Betrieb; und `save`, vorgegeben von den Datenbankadministratoren. Teilen sich die Gehaltsberechnung und der Stundenbericht eine Hilfsfunktion zur Berechnung der regulären Stunden, verändert eine von der Finanzabteilung gewünschte Änderung stillschweigend den Bericht, auf den sich der Betrieb verlässt. Niemand war nachlässig; der Code vermischte einfach die Anliegen zweier Akteure, sodass eine Änderung auf Wunsch des einen dem anderen schaden konnte.

```
from dataclasses import dataclass

@dataclass
class EmployeeRecord:          # shared data, no behaviour
    id: str
    hourly_rate: float
    hours: list[float]

class PayCalculator:           # actor: finance
    def pay(self, e: EmployeeRecord) → float:
        regular = sum(min(h, 8) for h in e.hours)
        overtime = sum(max(h - 8, 0) for h in e.hours)
        return (regular + 1.5 * overtime) * e.hourly_rate

class HoursReporter:           # actor: operations
    def weekly_report(self, e: EmployeeRecord) → str:
        return f"{e.id}: {sum(e.hours):.1f} hours"

class EmployeeRepository:      # actor: database administrators
    def save(self, e: EmployeeRecord) → None: ...
```

Aufteilung nach Akteur: Jede Klasse ändert sich nur, wenn ihre eigenen Stakeholder es wünschen, und die gemeinsamen Daten bleiben ein einfacher Datensatz.

Auf der Ebene von Komponenten und Services wird das Prinzip zu einem Test für Grenzen: **Wessen Wünsche ändern diese Komponente?** Auf einer Essenslieferplattform wird ein einzelner „Restaurant“-Service, der Speisekarten (geändert auf Wunsch der

Restaurantbetreiber), Provisionsregeln (geändert vom Finanzteam) und das Suchranking (geändert vom Growth-Team) enthält, aus drei voneinander unabhängigen Gründen geändert, von drei Personengruppen, in drei Release-Rhythmen. Jedes Release gefährdet die beiden anderen Anliegen, und das verantwortliche Team wird zur Warteschlange. Eine Aufteilung entlang der Akteure, etwa in einen Speisekarten-Service, eine Provisionskomponente in der Finanzdomäne und einen Ranking-Service, richtet jede Grenze an einer einzigen Änderungsquelle aus. Das ist auch der Gedanke dahinter, Services an Geschäftsfähigkeiten auszurichten und, im Domain-Driven Design, an Bounded Contexts (behandelt in Kapitel 5).

Beachten Sie, dass das Prinzip nicht sagt: „Machen Sie Services klein.“ Ein großer Service, der einem einzigen Akteur gut dient, ist in Ordnung; ein winziger Service, den zwei Abteilungen ständig in gegensätzliche Richtungen ändern, ist es nicht.

O: offen für Erweiterung, geschlossen für Änderung

Das Open-Closed Principle wurde zuerst von Bertrand Meyer in *Object-Oriented Software Construction* (1988) formuliert: Ein Modul sollte offen für Erweiterung sein (man kann neues Verhalten hinzufügen), aber geschlossen für Änderung (sein bestehender, getesteter Quellcode muss dafür nicht bearbeitet werden). Meyer dachte an Vererbung; Martin deutete es im Sinne von Schnittstellen und Polymorphie neu. Die praktische Form lautet: **Wenn eine Art von Variation zu erwarten ist, geben Sie ihr einen Erweiterungspunkt**, damit die nächste Variante neuer Code ist und nicht ein weiterer Zweig in altem Code.

```
from typing import Protocol

class Carrier(Protocol):
    def quote(self, weight_kg: float) → int: ...   # price in pence

class RoyalMail:
    def quote(self, weight_kg: float) → int:
        return 350 + int(weight_kg * 120)

class Courier:
    def quote(self, weight_kg: float) → int:
        return 600 + int(weight_kg * 80)

CARRIERS: dict[str, Carrier] = {"royal_mail": RoyalMail(), "courier": Courier()}

def cheapest(weight_kg: float) → tuple[str, int]:
    return min(((name, c.quote(weight_kg)) for name, c in CARRIERS.items()),
               key=lambda pair: pair[1])
```

Einen Versanddienstleister hinzuzufügen heißt, eine neue Klasse zu registrieren; der Code für das Angebot ändert sich nie.

Auf Architekturebene erscheint dieselbe Idee als **Erweiterungspunkte, die andere Teams nutzen können, ohne Sie um eine Änderung Ihres Codes zu bitten**. Ein Bestellservice, der ein `OrderPlaced`-Event veröffentlicht, ist offen für Erweiterung: Das Team des Treueprogramms kann einen neuen Consumer hinzufügen, der Punkte vergibt, und der Bestellservice wird weder geändert noch erneut getestet oder neu deployt. Eine Plug-in-Architektur (der Microkernel-Stil) gibt dasselbe Versprechen für Features. Bei APIs übersetzt sich das Prinzip in additive Weiterentwicklung: Neue optionale Felder, neue Endpoints und neue Event-Typen erweitern einen Vertrag, ohne zu ändern, worauf sich bestehende Clients verlassen.

Der Haken ist die Vorhersage. Ein Erweiterungspunkt ist nur für die Arten von Änderungen wertvoll, die tatsächlich eintreffen. Eine Registry von Versanddienstleistern lohnt sich, wenn der Shop alle paar Monate neue hinzufügt; eine generische „Preisregel-Engine“, gebaut, weil die Preisgestaltung eines Tages variieren *könnte*, ist spekulative Maschinerie. Fachleute fügen den Erweiterungspunkt beim zweiten oder dritten Auftreten einer Variation hinzu, nicht vorher.

L: Liskov Substitution und Verhaltensverträge

Das Liskov Substitution Principle geht auf Barbara Liskovs Keynote *Data Abstraction and Hierarchy* auf der OOPSLA-Konferenz 1987 zurück und wurde von Liskov und Jeannette Wing in *A Behavioral Notion of Subtyping* (ACM Transactions on Programming Languages and Systems, 1994) präzisiert. Informell: Wenn Code mit Objekten vom Typ T funktioniert, muss er weiter funktionieren, wenn er ein Objekt eines beliebigen Subtyps S erhält, ohne zu wissen, welchen er bekommen hat. Die richtigen Methodennamen reichen nicht; das *Verhalten* muss kompatibel sein.

Die Regeln der verhaltensbasierten Subtypisierung hängen direkt mit Design by Contract zusammen (Abschnitt 3.3). Ein Subtyp darf **Vorbedingungen nicht verstärken** (vom Aufrufer mehr verlangen), darf **Nachbedingungen nicht abschwächen** (im Gegenzug weniger versprechen) und muss **Invarianten erhalten** (die Tatsachen, die für den Typ immer gelten). Liskov und Wing ergänzten eine History Constraint: Ein Subtyp darf keine Zustandsänderungen zulassen, die der Supertyp verbietet, weshalb sich ein veränderlicher Punkt nicht als unveränderlicher ausgeben kann.

```
class Discount:
    """Postcondition: 0 ⩽ result ⩽ total."""
    def apply(self, total: int) → int:
        return total

class PercentOff(Discount):
    def __init__(self, pct: int):
        self.pct = pct
    def apply(self, total: int) → int:
        return total - total * self.pct // 100      # keeps the contract

class FixedVoucher(Discount):
    def __init__(self, pence: int):
        self.pence = pence
    def apply(self, total: int) → int:
        return total - self.pence                    # can go below zero!

# FixedVoucher(500).apply(300) == -200: the checkout now issues a refund.
# Fix: return max(total - self.pence, 0), honouring the base postcondition.
```

Der Basisvertrag besagt, dass ein Rabatt die Summe nie negativ macht. Ein Subtyp, der das bricht, lässt Aufrufer scheitern, die dem Vertrag vertraut haben.

Das Symptom einer LSP-Verletzung sind Typprüfungen in Aufrufern: `if isinstance(d, FixedVoucher): ...`. Jede solche Prüfung bedeutet, dass einer Abstraktion nicht mehr zu trauen ist, und das Wissen um ihre Ausnahmen verbreitet sich in der gesamten Codebasis.

Auf Service-Ebene ist Ersetzbarkeit die Frage hinter jeder **kompatiblen API-Änderung**. Soll Version 2.3 einer Zahlungs-API die Version 2.2 ersetzen, ohne dass Clients es merken, ist sie ein verhaltensbasierter Subtyp von 2.2, und es gelten dieselben Regeln. Sie darf Vorbedingungen nicht verstärken: kein neues Pflichtfeld in der Anfrage, keine strengere Validierung bestehender Felder, kein niedrigeres Rate Limit. Sie darf Nachbedingungen nicht abschwächen: keine entfernten Antwortfelder, keine langsamere garantierte Antwortzeit, kein Verlust der Idempotenz. Sie muss Invarianten erhalten: Ein Betrag ist weiterhin in der kleinsten Währungseinheit angegeben, eine Bestellkennung ändert sich weiterhin nie. Einen neuen Wert in eine Aufzählung in einer *Antwort* aufzunehmen, ist ein subtiler Fall: Es erweitert, was der Client erhalten kann, und bricht damit Clients, die die alte Menge für vollständig hielten. Dieselbe Überlegung gilt, wenn eine Implementierung eines Service eine andere hinter einem Load Balancer ersetzt oder wenn Sie einen Adapter für einen Zahlungsanbieter gegen einen anderen austauschen. Consumer-Driven-Contract-Tests sind im Grunde automatisierte Liskov-Prüfungen, und Hyrums Gesetz (Abschnitt 3.3) erinnert daran, dass sich Clients auf Verhalten verlassen können, das Sie nie versprochen haben.

I: Interface Segregation und clientspezifische APIs

Das Interface Segregation Principle besagt, dass kein Client gezwungen sein sollte, von Operationen abzuhängen, die er nicht nutzt. Eine fette Schnittstelle koppelt jeden Client an jede Operation: Ändert sich eine Operation, sind alle Clients betroffen, zumindest dadurch, dass sie neu kompiliert, getestet oder deployt werden müssen, selbst jene, die sie nie aufrufen. In kompilierten Sprachen ist diese Kopplung wörtlich zu nehmen (eine Änderung an einem Header oder einem gemeinsamen JAR erzwingt den Neubau aller davon abhängigen Module); in Python zeigt sie sich in Test Doubles, die zwanzig Methoden implementieren müssen, um Code zu testen, der zwei davon aufruft.

```
from typing import Protocol

class OrderReader(Protocol):
    def get(self, order_id: str) → dict: ...
    def list_for_customer(self, customer_id: str) → list[dict]: ...

class OrderWriter(Protocol):
    def save(self, order: dict) → None: ...
    def cancel(self, order_id: str) → None: ...

def monthly_statement(orders: OrderReader, customer_id: str) → int:
    return sum(o["total"] for o in orders.list_for_customer(customer_id))

# The reporting job cannot cancel an order, and its tests
# need a fake with two read methods, not the whole repository.
```

Jeder Client hängt nur von der Rolle ab, die er braucht; eine Klasse kann trotzdem beide Rollen implementieren.

Auf Service-Ebene ist Interface Segregation das Argument für **clientspezifische APIs**. Eine einzige Allzweck-API, die die mobile App, den Webshop, das Lagersystem und Partnerintegrationen bedient, wächst meist zu einem Entwurf auf dem kleinsten gemeinsamen Nenner heran: Jeder Client erhält Felder, die er ignoriert, und jede Änderung muss mit allen abgestimmt werden. Das Muster Backend for Frontend (bei SoundCloud benannt und von Phil Calçado und Sam Newman beschrieben) gibt jeder Art von Client eine eigene dünne API-Schicht, die dem Team gehört, das diesen Client baut, und die Aufrufe an die darunterliegenden Services zusammensetzt. GraphQL geht dasselbe Problem anders an, indem jeder Client genau die Felder auswählt, die er braucht. Für Events legt es nahe, mehrere fokussierte Event-Typen zu veröffentlichen (`OrderPlaced`, `OrderShipped`) statt eines einzigen `OrderChanged`-Events mit allen Feldern, das jeden Consumer zwingt, alles zu parsen und zu filtern.

Die architektonische Fassung gilt auch für **gemeinsame Bibliotheken**. Eine unternehmensweite `common`-Bibliothek mit Logging, Geldbeträgen, PDF-Erzeugung und HTTP-Clients koppelt jeden Service an jeden ihrer Teile: Ein Sicherheitsfix im PDF-Code erzwingt ein neues Release von Services, die nie PDFs erzeugen. Sie in kleine Bibliotheken aufzuteilen, die Clients unabhängig voneinander übernehmen, ist Interface Segregation auf Paketebene und nimmt das Common Reuse Principle weiter unten vorweg.

D: Dependency Inversion und wem die Schnittstelle gehört

Das Dependency Inversion Principle hat zwei Teile: High-Level-Module (Fachlogik) sollten nicht von Low-Level-Modulen abhängen (Details wie Datenbanken, Message Broker und Web-Frameworks); beide sollten von Abstraktionen abhängen. Und Abstraktionen sollten nicht von Details abhängen. Das Wort „Inversion“ bezieht sich auf die Richtung der Quellcode-Abhängigkeit: In einem naiven Entwurf ruft die Fachlogik das Detail auf und importiert es deshalb; nach der Umkehr importiert das Detail eine Schnittstelle, die die Fachlogik definiert, sodass der Quellcode-Pfeil entgegen dem Kontrollfluss zeigt.

Der entscheidende und oft übersehene Punkt ist die **Zuständigkeit**. Die Schnittstelle gehört der High-Level-Komponente und ist in deren Begriffen formuliert (`OrderRepository.save(order)`, nicht `PostgresClient.execute(sql)`). Liegt die Schnittstelle im Persistenzpaket, haben Sie eine Abstraktion hinzugefügt, aber nichts umgekehrt: Die Fachlogik hängt weiterhin vom Persistenzpaket ab.

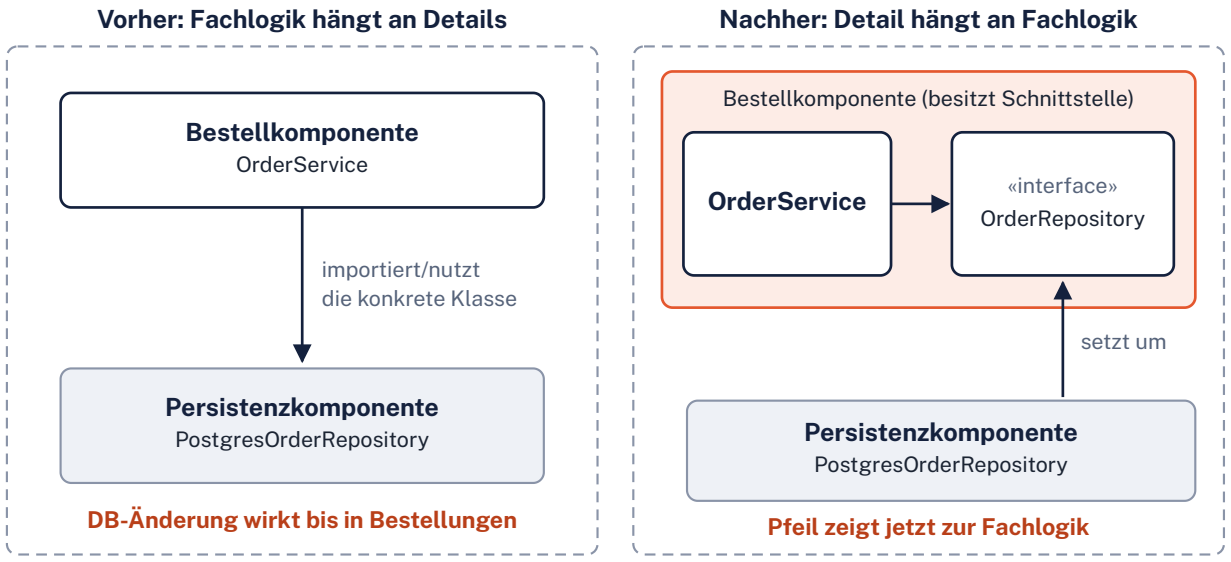
```
# orders/ports.py  (high-level policy owns this)
from typing import Protocol
class PaymentPort(Protocol):
    def charge(self, order_id: str, pence: int) → str: ...  # returns payment id

# orders/service.py
class Checkout:
    def __init__(self, payments: PaymentPort):
        self.payments = payments
    def pay(self, order_id: str, pence: int) → str:
        return self.payments.charge(order_id, pence)

# infrastructure/stripe_adapter.py  (detail depends on policy)
from orders.ports import PaymentPort
class StripeAdapter(PaymentPort):
    def charge(self, order_id: str, pence: int) → str:
        ...  # call the provider's API, map its errors to ours
```

Das Bestellpaket definiert den Port, den es braucht; die Infrastruktur implementiert ihn und wird beim Start eingebunden.

Eine Abhängigkeit über eine Komponentengrenze hinweg umkehren



Wird die Schnittstelle in die Bestellkomponente verlegt, kehrt sich die Quellcode-Abhängigkeit um, sodass sich die Persistenz ändern kann, ohne die Fachlogik zu berühren.

Dependency Inversion ist nicht dasselbe wie Dependency Injection, auch wenn beide meist zusammen auftreten. Injection ist eine Technik, einem Modul seine Kollaborateure von außen zu übergeben (das Konstruktorargument `payments` oben); Inversion ist eine Entscheidung darüber, in welche Richtung Quellcode-Abhängigkeiten zeigen. Sie können eine konkrete Klasse injizieren, ohne irgendetwas umzukehren.

Auf Architekturebene ist Dependency Inversion die ordnende Regel der hexagonalen, Onion- und Clean Architecture: Die Domäne sitzt im Zentrum und definiert Ports; Adapter für Datenbanken, Queues und Benutzeroberflächen sitzen außen und hängen nach innen ab. Zwischen Services lautet dieselbe Frage: „Wem gehört der Vertrag?“ Definiert ein zentraler Abrechnungsservice ein `BillingEvent`-Schema und muss sich jedes Produktteam daran anpassen, ist die Abrechnung die stabile Fachlogik und die Produkte hängen von ihr ab, was oft richtig ist. Liest die Abrechnung stattdessen die internen Tabellen jedes Produkts, zeigt die Abhängigkeit vom Stabilen zum Volatilen, und jede Schemaänderung eines Produkts bricht die Abrechnung. Abschnitt 3.4 hat gezeigt, wie sich das mit dem Stable Dependencies Principle quantifizieren lässt.

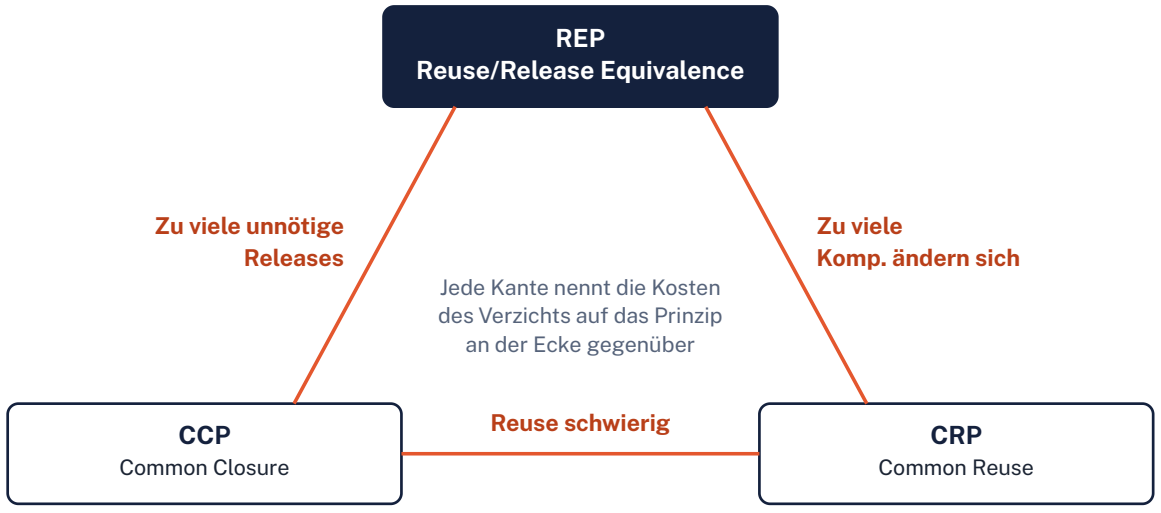
Die Kohäsionsprinzipien für Komponenten: REP, CCP und CRP

SOLID sagt Ihnen, wie Sie Klassen und ihre Abhängigkeiten gestalten. Martin stellt ihm drei Prinzipien zur Seite, mit denen sich entscheiden lässt, **welche Klassen zusammen in eine Komponente gehören**, wobei eine Komponente eine Einheit für Release und Deployment ist, etwa ein JAR, ein Python-Paket oder eine .NET-Assembly.

Das Reuse/Release Equivalence Principle (REP) besagt, dass die Einheit der Wiederverwendung die Einheit des Release ist: Sollen andere eine Komponente wiederverwenden, braucht sie Versionsnummern, Release Notes und ein stimmiges Thema, also sollte es sinnvoll sein, ihre Klassen gemeinsam zu releasen. Das Common Closure Principle (CCP) ist das Single Responsibility Principle für Komponenten: Fassen Sie in einer Komponente die Klassen zusammen, die sich aus denselben Gründen zu denselben Zeiten ändern, damit eine typische Änderung nur eine Komponente berührt und nur diese ein Release braucht. Das Common Reuse Principle (CRP) ist das Interface Segregation Principle für Komponenten: Klassen, die zusammen genutzt werden, gehören zusammen, und Klassen, bei denen das nicht so ist, sollten nicht zusammen verpackt werden, denn von einer Komponente abzuhängen heißt, von ihr als Ganzes abzuhängen.

Die drei ziehen in entgegengesetzte Richtungen. REP und CCP sind *inklusiv* (sie machen Komponenten größer); CRP ist *exklusiv* (es macht sie kleiner). Martin zeichnet sie als Dreieck, in dem jede Kante den Preis dafür nennt, das Prinzip an der gegenüberliegenden Ecke aufzugeben.

Das Spannungsdreieck



Wer sich auf zwei beliebige Kohäsionsprinzipien konzentriert, zahlt den Preis, der auf der Kante zwischen ihnen steht; Architekten wählen eine Position, und die Position wandert, während das Projekt reift.

Martin beobachtet, dass sich die Position eines Projekts im Dreieck mit der Zeit verschieben sollte. Am Anfang zählt die Entwickelbarkeit am meisten, also bevorzugen Teams CCP: nach Änderung gruppieren und in Kauf nehmen, dass Komponenten schwer wiederzuverwenden sind. Wenn ein System reift und andere Teams beginnen, von seinen Teilen abzuhängen, verschiebt sich das Gleichgewicht in Richtung REP und CRP. Dieselbe Spannung tritt zwischen Services auf. Zwei Services zusammenzulegen, die sich immer gemeinsam ändern (CCP),

verringert koordinierte Releases, macht das Ergebnis aber weniger wiederverwendbar; einen Service so aufzuteilen, dass Consumer nur von dem abhängen, was sie nutzen (CRP), erhöht die Zahl der deploybaren Einheiten, die eine einzelne fachliche Änderung berührt.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine gemeinsame Komponente in einem Onlineshop prüfen

Ein Onlineshop hat ein Paket `shop-core`, das von fünf Services genutzt wird. Es enthält Preisregeln, Steuerberechnung, Adressvalidierung, die Erzeugung von PDF-Rechnungen und einen Datenbank-Helper. Im letzten Quartal hatte es 40 Releases. Wenden Sie die Prinzipien an.

Akteure (SRP, CCP)	Die Preise ändert das Vertriebsteam, die Steuern die Finanzabteilung, wenn sich Steuersätze ändern, das PDF-Layout das Marketing, den Datenbank-Helper das Plattformteam. Vier Akteure bedeuten vier unabhängige Änderungsgründe, also ist das Paket nicht im Sinne des CCP geschlossen.
Nutzer (ISP, CRP)	Der Lagerservice nutzt nur die Adressvalidierung, und doch zwang ihn jedes der 40 Releases zu einem Upgrade und erneuten Tests. Nach dem CRP waren für ihn rund 40 Releases abzüglich der wenigen, die Adressen betrafen, unnötig.
Releases (REP)	Die Release Notes haben kein stimmiges Thema, sodass Consumer nicht beurteilen können, ob ein Release für sie relevant ist.
Abhängigkeiten (DIP, SDP)	Der Datenbank-Helper importiert die ORM-Modelle der Services, sodass ein stabiles, weit verbreitetes Paket von volatilem Service-Code abhängt: Der Pfeil zeigt in die falsche Richtung.
Vorgeschlagene Aufteilung	`pricing` (gehört der Vertriebsdomäne), `tax` (Finanzen), `address`, `invoice-pdf`, und der Datenbank-Helper wandert in jeden Service oder in eine Plattformbibliothek ohne Domänenimporte.
Erwartete Wirkung	Jeder Service hängt nur von den Paketen ab, die er nutzt. Eine Steuersatzänderung führt zum Release eines einzigen Pakets und berührt nur die Services, die Steuern berechnen.

Die Prinzipien haben die Antwort nicht vorgegeben; sie lieferten ein Vokabular, um zu erklären, warum das aktuelle Paket Probleme macht und was jede Option kostet.

Die Aufteilung erhöht die Zahl der Pakete, die versioniert und veröffentlicht werden müssen: die Kante „zu viele Komponenten ändern sich“ im Dreieck. Das ist hier akzeptabel, weil die Änderungen überwiegend unabhängig voneinander sind.

Kritik und Grenzen

SOLID wird viel gelehrt und viel kritisiert, und beide Reaktionen haben ihre Berechtigung. Die gewichtigsten Kritikpunkte sind diese.

Die Prinzipien sind so vage, dass sie fast alles rechtfertigen. Dan North hat das in Konferenzvorträgen und in seinem Essay von 2022 vertreten, mit dem er CUPID vorstellte (composable, Unix philosophy, predictable, idiomatic, domain-based). Er argumentiert, „ein Grund zur Änderung“ sei willkürlich, da sich fast jedem Code viele Änderungsgründe zuschreiben lassen, und dass eine eifrige Anwendung künstliche Nahtstellen zwischen Code erzeugt, der sich tatsächlich gemeinsam ändert, etwa zwischen Inhalt und Format eines Berichts. Er schlägt Eigenschaften vor, die man anstreben sollte, statt Regeln, die man befolgen muss. Martin antwortete auf diese Kritik 2020 in einem Beitrag über die Relevanz von SOLID: Die Prinzipien blieben gültig, weil sich die zugrunde liegenden Probleme der Abhängigkeitssteuerung nicht geändert hätten, und wer Leuten ohne Anleitung sage, sie sollten „es einfach einfach halten“, erzeuge ein Chaos.

Dogmatische Anwendung führt zu Überabstraktion. Eine Schnittstelle für jede Klasse, eine Factory für jede Schnittstelle und ein Erweiterungspunkt für jede denkbare Variation führen zu Code, bei dem man zehn Dateien öffnen muss, um einem einzigen Verhalten zu folgen. John Ousterhouts Unterscheidung zwischen tiefen und flachen Modulen (Abschnitt 3.3) ist ein nützliches Gegengewicht: Eine Abstraktion sollte viel hinter einer kleinen Schnittstelle verbergen, und viele von SOLID inspirierte Klassen sind flach. Spekulative Erweiterungspunkte verstoßen gegen YAGNI (Abschnitt 3.8) und verursachen echte Kosten.

Die Prinzipien wurden für ein einziges Paradigma geschrieben. In Sprachen mit First-Class-Funktionen und struktureller Typisierung schrumpfen viele SOLID-Muster darauf zusammen, eine Funktion zu übergeben oder sich auf Duck Typing zu verlassen. Die zugrunde liegenden Ideen (nach Akteur trennen, von stabilen Verträgen abhängen, Schnittstellen klein halten) gelten weiterhin; das Zeremoniell oft nicht.

Die ausgewogene professionelle Sicht behandelt SOLID als **diagnostisches Vokabular**: Wenn ein Entwurf Probleme macht, helfen die Prinzipien zu erklären, warum, und zeigen eine Richtung auf. Sie sind keine Checkliste, die jede Klasse erfüllen muss, und „verstößt gegen das OCP“ ist für sich allein kein Argument; „jeder neue Versanddienstleister erforderte bisher Änderungen an dieser Funktion und verursachte zwei Incidents“ dagegen schon.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: Sollte ich hier eine Abstraktion einführen?

Nutzen Sie diese Hilfe, wenn ein Prinzip nach einer neuen Schnittstelle, einem Erweiterungspunkt oder einer Aufteilung zu verlangen scheint.

Q1 **Ist diese Art von Variation tatsächlich mindestens zweimal aufgetreten, oder ist sie sicher (zum Beispiel, weil ein Vertrag mit einem zweiten Anbieter unterzeichnet ist)?**

- **Ja** Ein Erweiterungspunkt ist wahrscheinlich gerechtfertigt; weiter.
- **Nein** Behalten Sie den konkreten Code, halten Sie ihn leicht refaktorierbar und warten Sie auf echte Belege (YAGNI).

Q2 **Werden die Teile, die Sie trennen wollen, von verschiedenen Akteuren oder in verschiedenen Rhythmen geändert?**

- **Ja** Trennen Sie sie (SRP, CCP); die Grenze verringert den Abstimmungsaufwand.
- **Nein** Lassen Sie sie zusammen; eine Aufteilung würde Nahtstellen schaffen, die jede Änderung überqueren muss.

Q3 **Importiert eine High-Level-Fachlogik derzeit ein volatiles Detail wie eine Datenbank, ein Anbieter-SDK oder ein Framework?**

- **Ja** Definieren Sie eine Schnittstelle, die der Seite der Fachlogik gehört, und implementieren Sie sie im Detail (DIP).
- **Nein** Keine Umkehr nötig; eine direkte Abhängigkeit von einem stabilen Modul ist in Ordnung.

Q4 **Wird die neue Schnittstelle erhebliche Komplexität hinter wenigen Operationen verbergen?**

- **Ja** Es ist ein tiefes Modul; es lohnt sich.
- **Nein** Es ist ein flacher Durchreicher; überdenken Sie es oder führen Sie es mit seinem Aufrufer zusammen.

SOLID auf drei Ebenen

Prinzip	Klassenebene	Komponenten- oder Service-Ebene
Single Responsibility	Ein Akteur pro Klasse	Service oder Komponente, ausgerichtet an einer Geschäftsfähigkeit und einem verantwortlichen Team
Open-Closed	Neues Verhalten durch neue Implementierungen einer Schnittstelle	Events, Plug-ins und additive API-Änderungen, mit denen andere erweitern können, ohne Sie zu ändern
Liskov Substitution	Unterklassen halten den Vertrag der Basisklasse ein	Neue API-Versionen und Ersatzimplementierungen halten den alten Vertrag ein: keine neuen Pflicht-eingaben, keine entfernten Ausgaben
Interface Segregation	Kleine Rollenschnittstellen	Clientspezifische APIs (Backend for Frontend), fokussierte Event-Typen, kleine gemeinsame Bibliotheken
Abhängigkeitsumkehr	Von eigenen Abstraktionen abhängen	Die Domäne definiert Ports; Adapter und Infrastruktur hängen nach innen ab; stabile Eigentümer definieren Verträge

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Eine Klasse formatiert eine Rechnung und berechnet auch deren Steuer. Verstößt das automatisch gegen das Single Responsibility Principle?

Nicht automatisch. Es verstößt dagegen, wenn die Formatierung und die Steuerregeln von verschiedenen Akteuren geändert werden (etwa Marketing und Finanzabteilung), weil dann Änderungen auf Wunsch des einen den anderen beschädigen können; gehört beides einer Gruppe, kann die Klasse in Ordnung sein.

▼ Warum bricht ein neues Pflichtfeld in einer API-Anfrage die Ersetzbarkeit?

Es verstärkt die Vorbedingung: Aufrufer, die den alten Vertrag erfüllten, scheitern jetzt, sodass die neue Version die alte nicht mehr vertreten kann.

▼ Was ist der Unterschied zwischen Dependency Inversion und Dependency Injection?

Bei Inversion geht es um die Richtung von Quellcode-Abhängigkeiten, wobei die Abstraktion der Seite der Fachlogik gehört; Injection ist eine Technik, Kollaborateure von außen zu übergeben. Sie können eine konkrete Klasse injizieren, ohne irgendetwas umzukehren.

▼ **Ein Team gruppiert Klassen strikt danach, was sich gemeinsam ändert, und andere Teams klagen, die Komponenten seien unmöglich wiederzuverwenden. Auf welcher Kante des Spannungsdreiecks befindet sich das Team?**

Auf der Kante CCP-CRP, beschriftet mit „Reuse schwierig“: Es hat das Reuse/Release Equivalence Principle vernachlässigt.

▼ **Probieren Sie es aus: Version 2 einer internen Erstattungs-API lehnt jetzt Beträge über £100 ab, die Version 1 akzeptiert hat, und gibt das Feld reason nicht mehr zurück. Das Team nennt das kompatibel, weil die URL unverändert ist. Beurteilen Sie das mithilfe der Liskov Substitution.**

Sie ist nicht kompatibel. Liskovs Substitutionsprinzip fragt, ob die neue Version die alte ersetzen kann, ohne dass Aufrufer es bemerken. Beträge über £100 abzulehnen verschärft eine Vorbedingung, und das Feld reason zu streichen schwächt eine Nachbedingung; beides kann bestehende Aufrufer brechen. Eine unveränderte URL beweist nichts. Behalten Sie entweder das alte Verhalten bei und ändern Sie nur durch Hinzufügen, oder veröffentlichen Sie eine neue Version und migrieren Sie die Aufrufer, wobei das £100-Limit nur für diejenigen gilt, die sich dafür entscheiden.

QUELLEN

- Robert C. Martin: The Single Responsibility Principle (Clean Coder blog): https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html
- Robert C. Martin: Solid Relevance (Clean Coder blog, 2020): https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html
- Barbara Liskov and Jeannette Wing: A Behavioral Notion of Subtyping (1994): https://www.cs.columbia.edu/~wing/publications/LiskovWing94.pdf
- Dan North: CUPID, for joyful coding (2022): https://dannorth.net/blog/cupid-for-joyful-coding/
- Sam Newman: Backends for Frontends pattern: https://samnewman.io/patterns/architectural/bff/

Connascence: ein feineres Maß für Kopplung

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 3.2 Kopplung und Kohäsion
- 3.3 Information Hiding

Begriffe von früher, die er verwendet: Kopplung, Kapselung, Semantische Kopplung, Zeitliche Kopplung, Invariante

DIE IDEE IN KÜRZE

Wenn zwei Freunde verabreden, sich „am üblichen Ort zur üblichen Zeit“ zu treffen, bindet sie eine Absprache, die sonst niemand sehen kann; ändert einer von ihnen den Plan, muss der andere es erfahren. Software ist voll von solchen Absprachen: Namen, die Reihenfolge von Werten, was eine Zahl insgeheim bedeutet, welcher Schritt zuerst geschehen muss. Connascence ist eine Methode, diese Absprachen aufzulisten, zu beurteilen, wie riskant jede ist, und die riskanten schwächer, sichtbarer oder näher beieinander zu machen.

WAS SIE WISSEN MÜSSEN

- Zwei Elemente sind connascent, wenn eine Änderung am einen eine Änderung am anderen erfordern würde, damit das System korrekt bleibt; Meilir Page-Jones führte die Idee 1992 ein.
- Statische Formen (Name, Typ, Bedeutung, Position, Algorithmus) sind im Quellcode sichtbar; dynamische Formen (Ausführung, Timing, Wert, Identität) zeigen sich erst zur Laufzeit und sind im Allgemeinen stärker.
- Die Stärke beschreibt, wie schwer eine Form zu finden und zu ändern ist, die Lokalität, wie weit die Elemente voneinander entfernt sind, und der Grad, wie viele Elemente die Absprache teilen.
- Wandeln Sie starke Formen in schwächere um, halten Sie starke Connascence lokal und verringern Sie den Grad, indem Sie eine Konvention über eine einzige Definition oder einen einzigen Eigentümer leiten.
- Werden Positionsparameter durch benannte Parameter und magische Werte durch benannte Aufzählungen ersetzt, werden aus starken, unsichtbaren Absprachen schwache, die Werkzeuge prüfen können.

> - Zwischen Services sind nur schwache Formen wie Name und Typ in einem veröffentlichten Schema gesund; Connascence von Wert oder Identität zwischen Services weist auf eine falsch gezogene Grenze hin.

Warum Kopplung ein feineres Vokabular braucht

Abschnitt 3.2 hat Kopplung mit den Stufen aus dem Structured Design (Inhalts-, globale, externe, Steuerungs-, Datenstruktur- und Datenkopplung) und mit den verteilten Formen beschrieben, die zwischen Services eine Rolle spielen. Diese Stufen ordnen ein, *wie* Module verbunden sind. Weniger sagen sie zu zwei Fragen, vor denen Entwicklungsteams täglich stehen: **Wie schmerzhaft wird eine Änderung**, und **welches Refactoring würde den Schmerz verringern**? Connascence beantwortet beide.

Connascence (lateinisch etwa „zusammen geboren“) wurde von Meilir Page-Jones 1992 in einem Artikel in den *Communications of the ACM*, „Comparing techniques by means of encapsulation and connascence“, eingeführt und in seinem Buch *What Every Programmer Should Know About Object-Oriented Design* (Dorset House, 1995) ausgearbeitet, später überarbeitet als *Fundamentals of Object-Oriented Design in UML* (1999). Seine Definition ist einfach: Zwei Elemente sind connascent, wenn irgendeine Änderung am einen eine Änderung am anderen erfordern würde, damit das System korrekt bleibt. Die Elemente können alles Mögliche sein: zwei Funktionen, eine Funktion und ihre Aufrufer, ein Producer und die Consumer seiner Nachrichten, eine Datenbankspalte und jeder Bericht, der sie liest.

Zwei Eigenschaften machen die Idee für Architekten nützlich. Erstens gibt es Connascence in benannten **Formen**, die sich nach Stärke ordnen lassen, sodass man nicht nur sagen kann „diese sind gekoppelt“, sondern „diese sind über die Position gekoppelt, was schlimmer ist als eine Kopplung über den Namen, und so wandelt man das eine ins andere um“. Zweitens hat sie zwei weitere Dimensionen, **Lokalität** und **Grad**, die zeigen, ob eine bestimmte Form an ihrer Stelle akzeptabel ist.

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist

Immer wenn sich zwei Codestücke über etwas einig sein müssen, sind sie connascent. Ruft eine Funktion eine andere über deren Namen auf, müssen sich beide über diesen Namen einig sein: Benennen Sie die eine um, müssen Sie auch die andere umbenennen. Das ist unvermeidlich und harmlos. Die Gefahr liegt in Absprachen, die schwer zu sehen sind (eine Zahl, die insgeheim „versandt“ bedeutet), schwer zu prüfen (zwei Services, die in einer bestimmten Reihenfolge laufen müssen) oder über viele Stellen verstreut. Connascence ist eine Methode, diese verborgenen Absprachen zu benennen, damit Sie sie sichtbarer, schwächer oder näher beieinander machen können.

Statische Formen: im Code sichtbar

Statische Connascence lässt sich durch Lesen des Quellcodes finden, oft mit Werkzeugen, ohne das Programm auszuführen. Von der schwächsten zur stärksten, in der am häufigsten genannten Reihenfolge:

Connascence des Namens: Elemente müssen sich über einen Namen einig sein. Ein Aufrufer verwendet `calculate_tax`; ein JSON-Consumer liest `customer_id`. Es ist die schwächste Form, weil Umbenennen mechanisch ist und Werkzeuge jede Verwendung finden.

Connascence des Typs: Elemente müssen sich über einen Typ einig sein. Eine Funktion erwartet eine ganze Zahl von Pence; die Übergabe eines Floats läuft in Python durch, verfälscht aber das Ergebnis. Typprüfer und Schemas machen sie sichtbar.

Connascence der Bedeutung (oder Konvention): Elemente müssen sich darüber einig sein, was ein Wert bedeutet. Status `3` bedeutet „versandt“; `-1` bedeutet „nicht gefunden“; eine leere Zeichenkette bedeutet „Standardwert verwenden“. Solche magischen Werte sind für Werkzeuge unsichtbar: Nichts verbindet die `3` im Lagercode mit der `3` im E-Mail-Service.

Connascence der Position: Elemente müssen sich über die Reihenfolge von Werten einig sein. `create_user("Ann", "Smith", "ann@example.com")` hängt von der Parameterreihenfolge ab; eine CSV-Datei hängt von der Spaltenreihenfolge ab; ein Tupel `(lat, lon)` bricht stillschweigend, wenn jemand `(lon, lat)` annimmt, wie es GeoJSON tut.

Connascence des Algorithmus: Elemente müssen sich über einen Algorithmus einig sein. Sender und Empfänger eines Webhooks müssen dieselbe HMAC-Signatur berechnen; Client und Server müssen dasselbe Passwort-Hashing-Verfahren oder dieselbe Prüfsumme anwenden; jeder Service, der nach Kunde routet, muss dieselbe Hashfunktion verwenden, um einen Shard auszuwählen.

Dynamische Formen: erst zur Laufzeit sichtbar

Dynamische Connascence hängt davon ab, wie sich das System zur Laufzeit verhält, daher ist sie schwerer zu finden und meist stärker.

Connascence der Ausführung: Die Reihenfolge der Ausführung ist wichtig. Eine Verbindung muss geöffnet werden, bevor Daten gesendet werden; `begin()` muss vor `commit()` kommen; eine Bestellung muss angelegt sein, bevor ihre Zahlung eingezogen wird.

Connascence des Timings: Der Zeitpunkt der Ausführung ist wichtig. Das Timeout eines Clients muss länger sein als die normale Verarbeitungszeit des Servers; ein Test, der zwei Sekunden schläft, hofft, dass ein asynchroner Job fertig ist; zwei Threads liefern sich ein Wettrennen um die Aktualisierung eines Kontostands.

Connascence des Werts: Mehrere Werte müssen sich gemeinsam ändern, meist wegen einer Invariante. Die Liste gültiger Ländercodes im Web-Frontend und im Backend muss übereinstimmen; die Summe einer Bestellung muss der Summe ihrer Positionen entsprechen; Soll und Haben einer Buchung müssen ausgeglichen sein.

Connascence der Identität: Elemente müssen sich auf dieselbe Entität beziehen, nicht nur auf eine gleiche. Zwei Repositories müssen dieselbe Datenbanktransaktion teilen, damit ihre Schreibvorgänge atomar sind; zwei Komponenten müssen in *dieselbe* Queue-Instanz veröffentlichen, nicht in zwei Queues mit gleicher Konfiguration.

Page-Jones benannte auch die Gegenbeziehung, die **Contranascence**: Elemente, die sich *unterscheiden* müssen, etwa zwei Klassen in einem Namensraum, die nicht denselben Namen tragen dürfen, oder zwei Services, die nicht denselben Port verwenden dürfen. Für sie gelten dieselben Regeln.

Die neun Formen mit je einem Beispiel

Form	Art	Beispiel in einer Essensliefer-App	Typischer Weg, sie abzuschwächen
Name	Statisch	Der Bestellservice ruft `restaurant.get_menu()` auf	Bereits die schwächste Form; Namen aussagekräftig halten
Typ	Statisch	Liefergebühr als ganze Zahl in Pence übergeben, nicht als Float in Pfund	Type Hints, Schemas, ein `Money`-Typ
Bedeutung	Statisch	`status == 4` bedeutet „Fahrer zugewiesen“	Benannte Konstanten oder eine Aufzählung, über eine einzige Definition geteilt
Position	Statisch	`quote(2.5, 51.5, -0.12, True)`	Schlüsselwortargumente, ein Parameterobjekt, benannte JSON-Felder
Algorithmus	Statisch	App und Server berechnen dieselbe Warenkorb-Prüfsumme	Eine gemeinsame Bibliothek, oder eine Seite berechnet das Ergebnis und sendet es
Ausführung	Dynamisch	Die Zahlung muss autorisiert sein, bevor das Restaurant benachrichtigt wird	Die Abfolge an einer Stelle kapseln (ein Workflow oder eine Saga)
Timing	Dynamisch	Die Fahrer-App gibt nach 5 s auf, die Disposition braucht bis zu 8 s	Die Interaktion asynchron machen; Timeouts aus SLOs ableiten
Wert	Dynamisch	Frontend und Backend halten jeweils die Liste der Liefergebiete	Ein Eigentümer der Daten; andere fragen sie ab oder abonnieren sie
Identität	Dynamisch	Zwei Module müssen in derselben Datenbanktransaktion schreiben	Einem Modul die Zuständigkeit für die Unit of Work geben

Stärke, Lokalität und Grad

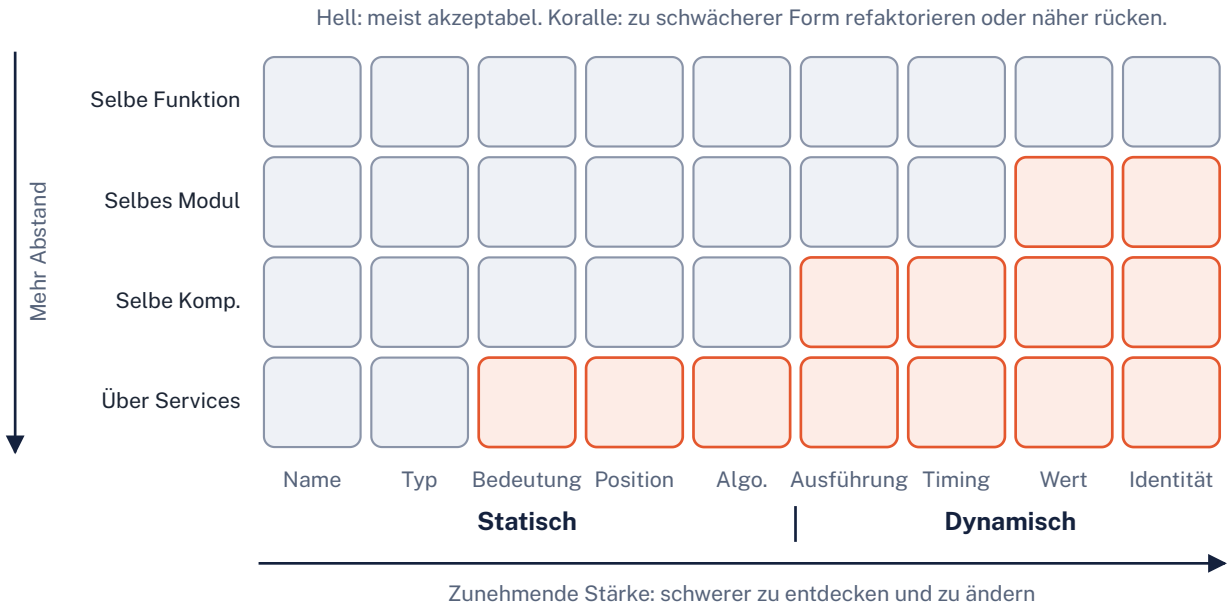
Jedes Auftreten von Connascence lässt sich anhand von drei Eigenschaften beurteilen. Die Stärke beschreibt, wie schwer die Form zu entdecken und zu ändern ist: Name ist schwach, weil Werkzeuge ihn finden und umbenennen; Bedeutung ist stärker, weil nichts die magischen

Werte kennzeichnet; Timing und Identität sind am stärksten, weil sie sich womöglich erst unter Last oder in der Produktion zeigen. Die Reihenfolge in der Tabelle ist die übliche Rangfolge, auch wenn die Autoren bei den statischen Formen leicht voneinander abweichen (manche ordnen Position unter Bedeutung ein).

Die Lokalität beschreibt, wie nah die connascenten Elemente beieinander liegen. Positionsparameter zwischen zwei privaten Hilfsfunktionen in derselben Datei sind harmlos: Wer die eine ändert, sieht die andere. Positionsfelder in einer Nachricht zwischen Services verschiedener Teams sind gefährlich, weil die Menschen, die gemeinsam ändern müssen, nicht zusammensitzen, nicht gemeinsam releasen und womöglich nicht einmal voneinander wissen.

Der Grad beschreibt, wie viele Elemente beteiligt sind. Eine magische Zahl, die zwei Funktionen teilen, ist ein Ärgernis; dieselbe Zahl in vierzig Dateien in sechs Services ist ein Migrationsprojekt. Der Grad lässt sich zählen: Kommt der Statuscode `3` an 14 Stellen im Lagerservice, an 9 im Benachrichtigungsservice und an 5 in den Views der Reporting-Datenbank vor, beträgt der Grad 28, und seine Bedeutung zu ändern heißt, 28 Änderungen über drei Teams und mindestens zwei Deployments hinweg zu koordinieren, mit einer Datenmigration für die Views.

Die Stärke muss mit wachsender Entfernung sinken



Starke Formen der Connascence sind innerhalb einer Funktion oder Klasse vertretbar; über Komponenten- und Service-Grenzen hinweg sollten Sie die schwächsten Formen anstreben.

Die Leitlinien

Page-Jones' Rat lässt sich in drei Aussagen zusammenfassen: Minimieren Sie die Connascence insgesamt, indem Sie ein System in gekapselte Elemente zerlegen; minimieren Sie die Connascence, die Kapselungsgrenzen überschreitet; und maximieren Sie die Connascence innerhalb dieser Grenzen. Die letzte überrascht viele, ist aber einfach Kohäsion in anderen Worten: Was sich gemeinsam ändern muss, sollte zusammen liegen.

Jim Weirich, der Ruby-Entwickler, der vor allem als Autor des Build-Werkzeugs Rake bekannt ist, belebte die Idee ab 2009 in Vorträgen für eine neue Generation wieder, darunter „The Building Blocks of Modularity" (2009 auf mehreren Ruby-Konferenzen gehalten, unter ihnen die MountainWest RubyConf) (den er scherzhaft eine große vereinheitlichte Theorie der Softwareentwicklung nannte) und „Connascence Examined". Er verdichtete die Praxis zu zwei Regeln. Die **Regel der Lokalität**: Je weiter zwei Elemente voneinander entfernt sind, desto schwächer sollte die Connascence zwischen ihnen sein. Die **Regel des Grades**: Wandeln Sie starke Formen der Connascence in schwächere um, besonders wenn viele Elemente beteiligt sind.

In der Praxis ergeben sich daraus drei Schritte, grob nach Vorrang geordnet:

Die Form abschwächen. Ersetzen Sie Bedeutung durch Name (eine benannte Konstante), Position durch Name (Schlüsselwortargumente, benannte Felder), Algorithmus durch Name (eine gemeinsame Funktion, die beide Seiten aufrufen), Ausführung durch Kapselung (eine Methode, die die Schritte in der richtigen Reihenfolge ausführt).

Die Lokalität erhöhen. Müssen zwei Elemente eine starke Form teilen, rücken Sie sie näher zusammen: in dieselbe Klasse, dasselbe Modul, dieselbe Komponente oder denselben Service, verantwortet vom selben Team.

Den Grad verringern. Teilen viele Elemente eine Konvention, leiten Sie sie über eine einzige Definition, einen einzigen Eigentümer oder einen einzigen Adapter, sodass nur wenige Stellen das Detail kennen.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Positionsparameter und magische Werte ersetzen

Eine Fahrdienst-App hat eine Funktion, die von der Buchungs-API, dem Scheduler und einem Admin-Werkzeug genutzt wird. Ihre Signatur ist über zwei Jahre gewachsen: `book(rider_id, 51.501, -0.142, 51.515, -0.072, 2, True, 0)`. Lesen Sie die Connascence ab und refaktorieren Sie die Funktion.

Position	Acht Positionsargumente. Werden Abhol- und Zielkoordinaten vertauscht, besteht der Code trotzdem die Typprüfung und schickt den Fahrer stillschweigend an den falschen Ort.
Bedeutung	`2` ist die Fahrzeugklasse (1 Standard, 2 groß, 3 barrierefrei); `True` bedeutet „mit Karte zahlen“; `0` bedeutet „sofort buchen“, während jede andere Zahl eine geplante Zeit in Epochensekunden ist.
Grad	Eine Suche findet 31 Aufrufstellen in drei Komponenten, sodass jeder neue Parameter 31 Änderungen an einer festen Position bedeutet.
Schritt 1: Name statt Position	Führen Sie einen Typ `Location` und reine Schlüsselwortargumente ein, sodass Aufrufer `pickup=Location(lat=51.501, lon=-0.142)` schreiben.
Schritt 2: Name statt Bedeutung	Ersetzen Sie `2` durch `VehicleClass.LARGE`, `True` durch `Payment.CARD` und `0` durch `when=None` mit der Bedeutung „sofort“.
Schritt 3: Grad verringern	Fassen Sie die Argumente in einem Objekt `BookingRequest` zusammen; ein später hinzugefügtes optionales Feld berührt nur den Code, der es braucht.
Ergebnis	Connascence der Position und der Bedeutung (stark, Grad 31) wird zu Connascence des Namens und des Typs (schwach), die Typprüfer und IDE verifizieren und automatisch umbenennen können.

Am Verhalten hat sich nichts geändert; das Refactoring hat nur geändert, welche Absprachen Aufrufer einhalten müssen und wie sichtbar diese Absprachen sind.

Nehmen Sie die Änderung in Schritten vor, die alte Aufrufer funktionsfähig halten, etwa indem Sie ein Release lang beide Formen akzeptieren und die Verwendung der alten protokollieren.

```
from dataclasses import dataclass
from datetime import datetime
from enum import Enum

class VehicleClass(Enum):
    STANDARD = "standard"
    LARGE = "large"
    ACCESSIBLE = "accessible"

class Payment(Enum):
    CARD = "card"
    CASH = "cash"

@dataclass(frozen=True, kw_only=True)
class Location:
    lat: float
    lon: float

@dataclass(frozen=True, kw_only=True)
class BookingRequest:
    rider_id: str
    pickup: Location
    dropoff: Location
    vehicle: VehicleClass = VehicleClass.STANDARD
    payment: Payment = Payment.CARD
    when: datetime | None = None          # None means now

def book(request: BookingRequest) → str: ...
```

Nach dem Refactoring: Jede Absprache zwischen Aufrufer und aufgerufener Funktion ist ein Name oder ein Typ (kw_only macht Aufrufe mit Positionsargumenten zum Fehler).

Connascence über Service-Grenzen hinweg

Mark Richards und Neal Ford stellen Connascence in *Fundamentals of Software Architecture* (2020; zweite Auflage 2025) neben Kohäsion und Kopplung als eines der zentralen Maße für Modularität vor und argumentieren, dass sie zwischen Services ebenso nützlich ist wie im Code. Liest man die Formen mit einem verteilten System im Kopf, entspricht jede einem vertrauten Problem:

Name und Typ über Services hinweg sind die Absprachen, die ein veröffentlichtes Schema explizit macht: Feldnamen und -typen in einem OpenAPI-Dokument, einem Avro- oder Protobuf-Schema oder einem Event-Katalog. Diese Formen wollen Sie, und Contract-Tests können sie prüfen.

Bedeutung über Services hinweg ist der gemeinsame Statuscode oder das unerklärte Flag: was Abschnitt 3.2 semantische Kopplung nannte. Bevorzugen Sie selbstbeschreibende Werte (`"status": "rider_assigned"`) gegenüber Zahlen und Einheiten in Feldnamen (`fee_pence`, `timeout_ms`) gegenüber Annahmen. Einheiten verdienen besondere Sorgfalt: Der Mars Climate Orbiter der NASA ging 1999 verloren, weil die Bodensoftware eines Teams Impulswerte in Pound-Force-Sekunden lieferte, während die Navigationssoftware, die sie verarbeitete, Newtonsekunden erwartete, eine Connascence der Bedeutung zwischen Organisationen, die niemand explizit gemacht hatte.

Position über Services hinweg zeigt sich beim Austausch von Dateien mit fester Feldbreite oder CSV-Dateien und bei positionsbasierten Nachrichtenformaten; benannte Felder beseitigen sie. **Algorithmus** zeigt sich, wenn zwei Services unabhängig voneinander denselben Preis, Rabatt oder Hash berechnen müssen; die Abhilfe besteht meist darin, dass ein Service ihn berechnet und das Ergebnis sendet.

Ausführung und Timing über Services hinweg sind die harten Probleme verteilter Systeme, und Timing ist die Connascence hinter der zeitlichen Kopplung aus Abschnitt 3.2: ein Consumer, der annimmt, dass der Datenbank-Schreibvorgang des Producers sichtbar ist, bevor das Event eintrifft; eine Kette synchroner Aufrufe, deren Timeouts nicht korrekt verschachtelt sind; ein Workflow, der nur funktioniert, wenn Nachrichten in der richtigen Reihenfolge eintreffen. Interaktionen asynchron, idempotent und reihenfolgetolerant zu machen, schwächt diese Formen ab; Muster wie Sagas und die Outbox (Kapitel 6) gibt es vor allem, um sie zu beherrschen.

Wert und Identität über Services hinweg sind Anzeichen für eine Grenze an der falschen Stelle. Müssen zwei Services einen Wert konsistent halten (die Bestellsumme im Bestellservice und die Rechnungssumme in der Abrechnung) oder dieselbe Zeile in derselben Transaktion bearbeiten, dann sollte entweder ein Service diese Daten besitzen und der andere sie anfragen, oder die beiden Services sollten wahrscheinlich einer sein. Starke Connascence über ein Netzwerk hinweg ist geradezu die Definition eines verteilten Monolithen.

ACHTUNG

Häufige Fehler

- Jede Kopplung als schlecht ansehen. Connascence des Namens ist der Preis jeder Zusammenarbeit; das Ziel ist schwache, lokale Connascence mit geringem Grad, nicht gar keine.
- Die Form abschwächen, aber den Grad erhöhen, etwa indem man jede Konstante jedes Service in eine gemeinsame „constants“-Bibliothek packt, die alle Services gemeinsam aktualisieren müssen.
- Statische Formen beheben und dynamische ignorieren. Eine wunderbar typisierte API, deren Aufrufer drei Endpoints in einer bestimmten Reihenfolge aufrufen müssen, hat weiterhin starke Connascence der Ausführung.
- Elemente außerhalb des Codes vergessen: Konfigurationsdateien, Datenbank-Views, Dashboards und Alarmregeln sind connascent mit dem Code, den sie beschreiben.

Balanced Coupling: Stärke, Distanz und Volatilität

Connascence beschreibt die Art des Wissens, das zwei Teile teilen. Vlad Khononovs *Balancing Coupling in Software Design* (2024), erwähnt in Abschnitt 3.2, baut auf Structured Design und Connascence auf und stellt eine andere Frage: Ist diese Kopplung *hier* ein Problem? Er argumentiert, dass Kopplung kein zu minimierendes Übel ist, sondern der Klebstoff, der ein System zum System macht, und er beurteilt sie anhand von drei Dimensionen zugleich.

Integrationsstärke gibt an, wie viel Wissen die Teile teilen. In absteigender Reihenfolge sind seine vier Stufen *intrusiv* (Abstützen auf private Interna eines anderen, etwa seine Datenbanktabellen), *funktional* (geteiltes Wissen über Geschäftsanforderungen, etwa bei einer duplizierten Regel), *Modell* (ein geteiltes Domänenmodell) und *Vertrag* (nur eine explizite, stabile Schnittstelle wird geteilt, die den Rest verbirgt). **Distanz** gibt an, wie weit die Teile voneinander entfernt sind: dieselbe Datei, dasselbe Modul, verschiedene Services. Sie erhöht die Kosten gemeinsamer Änderungen und ist teilweise organisatorisch, denn zwei Services eines Teams liegen faktisch näher beieinander als zwei Services verschiedener Teams, wie das Gesetz von Conway erwarten lässt. **Volatilität** gibt an, wie wahrscheinlich sich ein Teil ändert. Domain-Driven Design hilft bei der Schätzung: Core Subdomains ändern sich am stärksten, Supporting und Generic Subdomains viel weniger.

Das Modell wird oft in einer vereinfachten Formel zusammengefasst:

```
BALANCE = (STRENGTH XOR DISTANCE) OR NOT VOLATILITY
```

Lesen Sie sie in zwei Schritten. Stärke und Distanz sollten sich ausgleichen: Teile, die viel teilen, gehören nah zusammen (das ist Kohäsion), und weit entfernte Teile sollten wenig teilen. Sind beide hoch, ist eine Änderung teuer und häufig; das ist der verteilte Monolith. Sind beide niedrig, stehen unzusammenhängende Dinge grundlos beieinander, was ebenfalls Komplexität ist. Der zweite Schritt ist die Ausnahmeklausel: Wenn sich keiner der Teile ändern wird, spielt ein Ungleichgewicht womöglich keine Rolle.

Angenommen, auf der Essenslieferplattform liest Lieferung direkt die Tabellen von Bestellungen. Das ist intrusive Stärke über eine Service- und Teamgrenze hinweg, und das Bestellen ist Kern des Geschäfts, also ist es unausgewogen. Es gibt zwei ehrliche Lösungen: die Stärke verringern, indem ein Vertrag veröffentlicht wird, oder die Distanz verringern, indem beide zusammengelegt werden. Dagegen kann eine dünne Integration mit einem selten geänderten Altsystem der Finanzabteilung intrusiv und distanziert sein, aber tragbar, solange dieses System stillhält.

Behandeln Sie das Modell als Denkwerkzeug, nicht als Messung. Die Formel ist eine binäre Vereinfachung der ausführlicheren Darstellung im Buch. Volatilität ist eine Prognose, und die Commit-Historie kann in die Irre führen: Häufige Commits können auf schlechten Entwurf hindeuten, und selten angefasster Code kann schlicht zu beängstigend für Änderungen sein. Eine Einschätzung niedriger Volatilität sollte daher festgehalten werden, zum Beispiel in einem Entscheidungsdokument, und überprüft werden.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Eine Funktion erwartet `(width, height)`, und ein Aufrufer übergibt `(height, width)`. Welche Form der Connascence wurde verletzt, und wie lässt sie sich am einfachsten abschwächen?**

Connascence der Position. Verwenden Sie Schlüsselwortargumente oder ein Parameterobjekt, damit die Absprache zu einer des Namens wird.

▼ **Warum ist Connascence der Bedeutung stärker als Connascence des Namens?**

Werkzeuge und Lesende können jede Verwendung eines Namens finden, aber nichts verbindet die einzelnen Verwendungen eines magischen Werts wie 3, sodass Änderungen leicht übersehen werden.

▼ **Zwei Services müssen das Kreditlimit eines Kunden immer in derselben Transaktion aktualisieren. Was deutet das an?**

Starke Connascence der Identität oder des Werts über eine Netzwerkgrenze hinweg: Die Daten brauchen wahrscheinlich einen einzigen Eigentümer, oder die beiden Services gehören zusammen.

▼ **Warum kann starke Connascence innerhalb einer einzelnen kleinen Funktion völlig akzeptabel sein?**

Weil die Lokalität hoch ist: Wer ein Element ändert, sieht das andere sofort, sodass Kosten und Risiko, sie konsistent zu halten, winzig sind.

▼ **Probieren Sie es aus: Der Aufruf hold(customer, "A", 14, 1, True) wird von der Website, der Theaterkasse und einem Admin-Werkzeug verwendet, wobei 1 einen Rollstuhlplatz bedeutet. Benennen Sie die beteiligten Arten von Connascence, und sagen Sie, wie Sie sie abschwächen würden.**

Connascence of Position: Jeder Aufrufer muss die Argumente in der richtigen Reihenfolge halten, und row und seat könnten unbemerkt vertauscht werden. Connascence of Meaning: 1 und True sind magische Werte, die alle drei Aufrufer gleich deuten müssen. Der Grad ist drei, und die Elemente liegen weit auseinander, was es verschlimmert. Verwenden Sie benannte Parameter oder einen kleinen Anfragetyp und eine benannte Enumeration wie SeatType.WHEELCHAIR. Beides wird zu Connascence of Name, die Tools prüfen können.

QUELLEN

- connascence.io: reference to the forms and properties of connascence: https://connascence.io/
- connascence.io: connascence of position: https://connascence.io/position.html
- Practicing Ruby: Connascence as a software design metric (on Jim Weirich's talk): https://practicingruby.com/articles/connascence
- Balanced Coupling (coupling.dev): https://coupling.dev/
- Balanced Coupling: Balance: https://coupling.dev/posts/core-concepts/balance/
- Balanced Coupling: Integration strength: https://coupling.dev/posts/dimensions-of-coupling/integration-strength/
- Balanced Coupling: Distance: https://coupling.dev/posts/dimensions-of-coupling/distance/
- Balanced Coupling: Volatility: https://coupling.dev/posts/dimensions-of-coupling/volatility/
- Pearson: Balancing Coupling in Software Design: https://www.pearson.com/en-nz/subject-catalog/p/balancing-coupling-in-software-design-successful-software-architecture-in-general-and-distributed-systems/P200000000372/9780137353576

Das Gesetz von Conway und die Gestalt von Organisationen

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 1.3 Warum sie zählt
- 3.1 Modularität
- 3.2 Kopplung und Kohäsion

Begriffe von früher, die er verwendet: Schnittstelle, Kopplung, Abhängigkeit

DIE IDEE IN KÜRZE

Wenn vier Gruppen von Handwerkern, die nie miteinander sprechen, jeweils einen Teil eines Hauses bauen, erhalten Sie vier Teile, die ungeschickt zusammenpassen, mit den Nahtstellen genau dort, wo die Gruppen aufeinandertrafen. Software funktioniert genauso: Die Teile eines Systems und die Verbindungen zwischen ihnen bilden meist nach, welche Teams miteinander sprechen. Wenn Sie ein System entwerfen, entwerfen Sie also, ob Sie wollen oder nicht, auch, wer mit wem sprechen muss, und umgekehrt gilt das genauso.

WAS SIE WISSEN MÜSSEN

- Melvin Conways Datamation-Artikel von 1968 argumentierte, dass Organisationen, die Systeme entwerfen, gezwungen sind, Entwürfe hervorzubringen, die ihre eigenen Kommunikationsstrukturen nachbilden.
- Der Grund ist mechanisch: Eine Schnittstelle zwischen zwei Subsystemen existiert nur dort, wo ihre Designer sie ausgehandelt haben, daher spiegelt der Graph des Systems den Kommunikationsgraphen der Organisation.
- MacCormack, Baldwin und Rusnak (2012) fanden, dass Produkte lose gekoppelter Organisationen durchweg modularer waren als vergleichbare Produkte eng gekoppelter, auch wenn eine Übersichtsarbeit von 2016 die Spiegelung als häufig, aber nicht universell einstufte.
- Eine Gruppe von n Personen hat n(n-1)/2 Kommunikationspfade, daher wachsen die Koordinationskosten ungefähr mit dem Quadrat der Teamgröße, und kleine Teams, die jeweils passende Teile des Systems verantworten, halten sie beherrschbar.
- Das Inverse Conway Maneuver gestaltet Teams und ihre Zusammenarbeit passend zur gewünschten Architektur, wirkt gegen eine bestehende, eng gekoppelte Codebasis aber nur langsam.

- Die kognitive Last des Teams begrenzt, wie viel von einem System ein Team verantworten kann; Grenzen sollten jedem Team einen Teil geben, den es vollständig verstehen, bauen und betreiben kann.

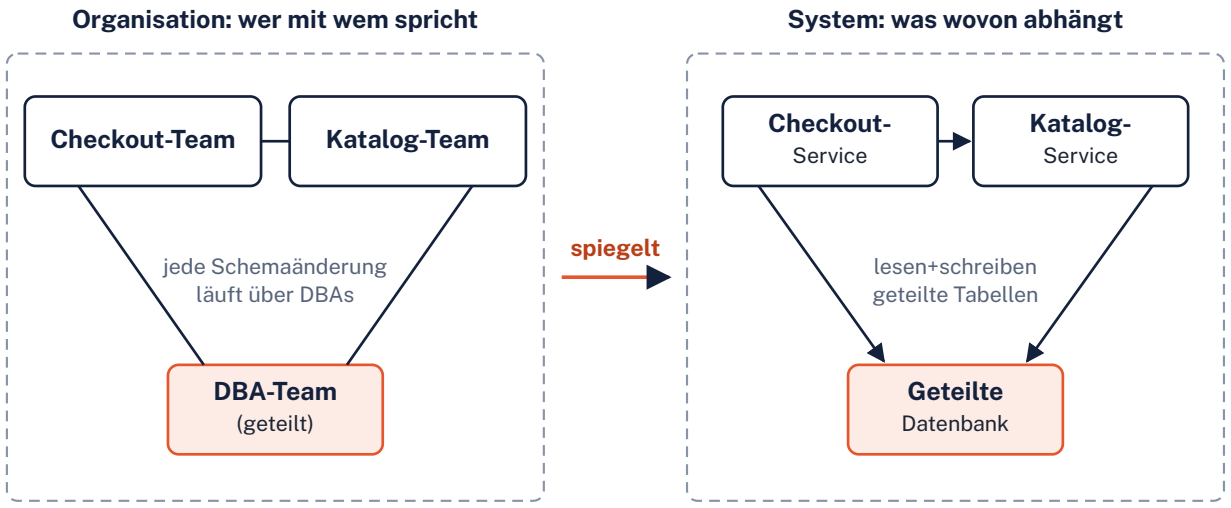
Was Conway tatsächlich sagte

Im April 1968 veröffentlichte die Zeitschrift *Datamation* „How Do Committees Invent?“ von Melvin Conway, einem Programmierer und Berater. Sein Fazit ist der Satz, der heute als Gesetz von Conway bekannt ist: Organisationen, die Systeme entwerfen, sind „gezwungen, Entwürfe hervorzubringen, die Kopien der Kommunikationsstrukturen dieser Organisationen sind“. Der Name kam später: Fred Brooks zitierte den Artikel in *The Mythical Man-Month* (1975) und nannte die Idee Conway’s law.

Conways Argument ist präziser als das Schlagwort. Ein System besteht aus Subsystemen, die über Schnittstellen verbunden sind. Damit zwei Subsysteme sich verbinden, müssen die Menschen, die sie entwerfen, diese Schnittstelle ausgehandelt und vereinbart haben; sie müssen also kommuniziert haben. Wenn die Designer zweier Subsysteme nicht miteinander kommunizieren, gibt es keine Schnittstelle, auf die sie sich einigen könnten. Es gibt daher eine strukturerhaltende Entsprechung (Conway verwendete den mathematischen Begriff Homomorphismus) zwischen dem Graphen des Systems und dem Graphen der Organisation, die es entworfen hat. Sein Beispiel: Eine Auftragsforschungsorganisation hatte acht Personen, die einen COBOL- und einen ALGOL-Compiler schreiben sollten. Fünf wurden COBOL zugeteilt, drei ALGOL. Der COBOL-Compiler lief in fünf Phasen, der ALGOL-Compiler in drei.

Zwei weitere Punkte aus dem Artikel werden oft vergessen. Erstens stellte Conway ausdrücklich klar, dass „System“ weit gemeint ist: Es gilt für jedes entworfene Ding, nicht nur für Software. Zweitens betraf sein praktisches Fazit die **Flexibilität**. Weil der erste Entwurf selten der beste ist und weil eine Organisation ihre Struktur meist früh festlegt (sobald Teams gebildet sind, ist der Entwurf schon teilweise entschieden), forderte er, Designorganisationen schlank und flexibel zu halten und nach der Kommunikation zu organisieren, die der Entwurf tatsächlich braucht.

Der Spiegel



Die Struktur des Systems bildet die Struktur der Kommunikation nach

Wenn zwei Teams sich nur über ein gemeinsames Datenbankteam abstimmen können, koppelt sich das System, das sie bauen, meist über eine gemeinsame Datenbank.

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist

Stellen Sie sich vor, drei Ämter einer Stadtverwaltung, die nie miteinander sprechen, bauen jeweils einen Teil einer Website. Sie erhalten wahrscheinlich drei Bereiche mit unterschiedlichen Menüs, unterschiedlichen Anmeldungen und Formularen, die dreimal nach Ihrer Adresse fragen. Niemand hat das geplant; so kommt es, wenn die einzigen Nahtstellen eines Produkts dort liegen, wo die Menschen, die es bauen, tatsächlich miteinander gesprochen haben. Das Gesetz von Conway besagt, dass dies kein Zufall ist, sondern eine Tendenz, die man vorhersagen und deshalb nutzen kann.

Die Belege: die Mirroring Hypothesis

Jahrzehntelang stützte sich das Gesetz von Conway vor allem auf Anekdoten. Seit den 2000er-Jahren haben Forschende es überprüft, meist unter dem Namen Mirroring Hypothesis: Die technischen Abhängigkeiten in einem Produkt entsprechen den organisatorischen Verbindungen (Kommunikation, gemeinsamer Standort, Anstellung) zwischen den Menschen, die es bauen.

Die bekannteste Softwarestudie stammt von Alan MacCormack, Carliss Baldwin und John Rusnak, 2012 in *Research Policy* veröffentlicht als „Exploring the duality between product and organizational architectures: a test of the mirroring hypothesis“. Sie verglichen passende Paare

von Produkten mit ähnlichen Funktionen, eines von einer eng gekoppelten Organisation gebaut (ein kommerzielles Unternehmen mit einem Team am selben Standort) und eines von einer lose gekoppelten Organisation (eine verteilte Open-Source-Community): zum Beispiel zwei Finanzverwaltungspakete, zwei Textverarbeitungen, zwei Datenbanken sowie Linux im Vergleich mit kommerziellen Betriebssystemkernen. Modularität maßen sie über die *Propagation Cost*, den durchschnittlichen Anteil der Dateien eines Systems, den eine Änderung an einer Datei direkt oder indirekt betreffen kann. In jedem Paar war das Produkt der lose gekoppelten Organisation modularer, und im extremsten Paar betrug der Unterschied ein Mehrfaches. Die Autoren deuten dies als Beleg dafür, dass verteilte Teams, die sich nicht leicht abstimmen können, zu Entwürfen gedrängt werden, die weniger Abstimmung brauchen.

Eine Studie von Microsoft Research zu Windows Vista von Nachiappan Nagappan, Brendan Murphy und Victor Basili (2008) näherte sich der Frage von der Qualitätsseite. Kennzahlen, die die Organisation hinter jedem Binary beschrieben (wie viele Fachleute daran gearbeitet hatten, wie viele das Unternehmen verlassen hatten, wie weit die Beteiligten in der Managementhierarchie auseinanderlagen), sagten besser vorher, welche Binaries fehleranfällig waren, als herkömmliche Codemetriken wie Churn, Komplexität, Testabdeckung und Abhängigkeiten.

Die Belege sind nicht einhellig, und die ehrliche Zusammenfassung zählt. Eine Übersichtsarbeit von Lyra Colfer und Carliss Baldwin aus dem Jahr 2016 über 142 empirische Studien fand Spiegelung in Studien zu Branchen und Unternehmen häufig, aber nicht universell, und stellte fest, dass Studien zu offenen kollaborativen Projekten sie oft *nicht* stützten, vermutlich weil gemeinsame Code-Repositories, Issue-Tracker und offene Diskussion es verteilten Beteiligten erlauben, sich ohne die üblichen organisatorischen Verbindungen abzustimmen. Spiegelung ist eine starke Tendenz, kein Naturgesetz, und gute Werkzeuge und Praktiken können sie teilweise durchbrechen.

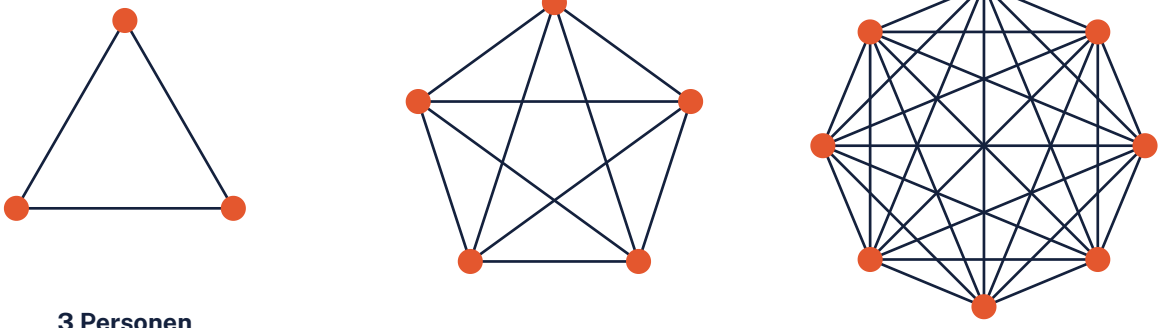
Kommunikationspfade und warum die Teamgröße zählt

Eine Gruppe von n Personen hat $n(n - 1) / 2$ mögliche Kommunikationspfade zwischen je zwei Personen. Die Zahl wächst ungefähr mit dem Quadrat der Gruppe: Ein doppelt so großes Team hat viermal so viele Pfade. Brooks erklärte mit derselben Arithmetik, warum zusätzliche Leute ein verspätetes Projekt noch weiter verzögern: Jede neue Person muss eingearbeitet werden und bringt neue Pfade mit, die alle pflegen müssen.

Architektonisch erklärt die Arithmetik zwei Dinge. Erstens kann ein einzelnes Team nicht unbegrenzt wachsen und dennoch ein gemeinsames Verständnis seines Codes behalten; ab einer bestimmten Größe zerfällt es in informelle Untergruppen, und nach dem Gesetz von Conway zerfällt der Code entlang derselben Linien, ob jemand diese Nahtstellen entworfen hat

oder nicht. Zweitens zählt die Zahl der Pfade zwischen *Teams* ebenso wie die innerhalb von Teams: Wenn jedes Team mit jedem anderen sprechen muss, um eine Änderung auszuliefern, ist das System faktisch eine große, gekoppelte Einheit, egal wie viele Services es hat.

Pfade wachsen viel schneller als die Zahl der Menschen



3 Personen
3 Pfade

5 Personen
10 Pfade

8 Personen
28 Pfade

Jedes Paar von Personen ist ein möglicher Kommunikationspfad: 3 Personen haben 3, 5 haben 10 und 8 haben 28.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Kommunikationspfade in einer wachsenden Produktgruppe

Das Bestellprodukt eines Essenslieferunternehmens wird von einem Team aus 8 Entwicklern gebaut. Es soll auf 24 Entwickler wachsen. Vergleichen Sie ein großes Team mit drei Teams zu je 8, die klare Zuständigkeiten haben.

Ein Team aus 8	8 × 7 / 2 = 28 Pfade.
Ein Team aus 24	24 × 23 / 2 = 276 Pfade: etwa zehnmal so viele bei dreimal so vielen Menschen.
Drei Teams zu je 8, Pfade innerhalb	3 × 28 = 84 Pfade innerhalb der Teams.
Drei Teams, Pfade zwischen den Teams	Hat jedes Team genau eine Ansprechperson für jedes andere Team (etwa den Tech Lead), kommen 3 × 2 / 2 = 3 Pfade zwischen den Teams hinzu. Insgesamt etwa 87.
Drei Teams, alle stimmen sich mit allen ab	Sind die Zuständigkeiten unklar und braucht jede Änderung alle Entwickler, ist der Graph faktisch wieder der Graph mit 276 Pfaden.
Was es zeigt	Die Aufteilung von Menschen in Teams hilft nur, wenn das System passend aufgeteilt wird, sodass die meisten Änderungen nur Kommunikation innerhalb eines Teams brauchen. Erst die Architektur macht den kleineren Graphen möglich.

Die Formel liefert mögliche Pfade, keine tatsächlichen Gespräche; es geht darum, wie schnell die Koordinationskosten wachsen können, nicht um eine genaue Vorhersage.

Team Topologies (Skelton und Pais) empfiehlt langlebige Teams von etwa 5 bis 9 Personen, was zur Arithmetik passt: klein genug, dass die Pfade beherrschbar bleiben.

Das Inverse Conway Maneuver

Wenn Organisationen dazu neigen, Systeme hervorzubringen, die ihre Struktur spiegeln, können Sie diese Tendenz bewusst nutzen: **Gestalten Sie die Teamstruktur und die Kommunikationspfade für die Architektur, die Sie wollen**, und lassen Sie das System in diese Form hineinwachsen. Jonny LeRoy und Matt Simons nannten dies im Dezember 2010 in einem Artikel im *Cutter IT Journal* das Inverse Conway Maneuver, und es ist seither eine Standardtechnik in Microservice-und Plattformprogrammen.

In der Praxis heißt das: zuerst die beabsichtigten Grenzen festlegen (zum Beispiel Bestellungen, Zahlungen, Lieferverfolgung und Restaurant-Onboarding als getrennte Services), dann um jede ein langlebiges Team bilden, ihm die Verantwortung für Code, Daten und Deployment-Pipeline übertragen und dafür sorgen, dass Teams vor allem über veröffentlichte APIs und Events zusammenarbeiten statt über gemeinsamen Code oder gemeinsame Datenbanken. Wo Kontakt über Teamgrenzen hinweg wirklich nötig ist, machen Sie ihn explizit: ein Plattformteam, das ein Self-Service-Produkt anbietet, ein Enabling Team, das für begrenzte Zeit hilft.

Das Manöver hat Grenzen. Gegen die Struktur eines bestehenden Systems wirkt es nur langsam: Teams umzuorganisieren strukturiert keine eng gekoppelte Codebasis um, und eine Zeit lang werden die neuen Teams gegen die alte Architektur ankämpfen. Martin Fowler merkt an, dass es ein starres bestehendes System nicht schnell reparieren kann. Außerdem setzt es voraus, dass Sie die richtigen Grenzen kennen; ziehen Sie sie falsch und verfestigen sie dann in Teamstrukturen, wird das Gesetz von Conway den Fehler getreu reproduzieren und seine Korrektur teuer machen. Deshalb beginnen Teams oft mit einem modularen Monolithen und verschieben Grenzen, bevor sie sie organisatorisch verankern.

Kognitive Last des Teams

Das Gesetz von Conway sagt Ihnen, dass Teamgrenzen zu Systemgrenzen werden. Eine zweite Randbedingung sagt Ihnen, wie groß der Teil jedes Teams sein darf. Die kognitive Last des Teams, ein Konzept, das Matthew Skelton und Manuel Pais mit *Team Topologies* (2019; zweite Auflage 2025) in die Gestaltung von Softwareorganisationen eingeführt haben, ist die Gesamtmenge dessen, was ein Team im Kopf behalten muss, um das zu bauen, zu betreiben und zu ändern, was es verantwortet: die fachliche Domäne, den Code, die Infrastruktur, die Werkzeuge, die Rufbereitschaft.

Sie übernehmen aus der Psychologie die Unterscheidung zwischen **intrinsischer** Last (der inhärenten Schwierigkeit der Aufgabe, etwa die Domäne zu verstehen), **extrinsischer** Last (Aufwand, den die Umgebung auferlegt, etwa ein manueller Deployment-Ablauf oder ein undurchsichtiger Build) und **lernbezogener** Last (dem Lernaufwand, der künftige Arbeit verbessert). Ziel der Gestaltung ist es, die extrinsische Last zu minimieren, zum Beispiel durch Plattformen und Automatisierung, damit die Kapazität in die intrinsische und die lernbezogene Last fließt.

Für Architekten ist die Folge unmittelbar. **Eine Service- oder Komponentengrenze sollte dem verantwortlichen Team einen Teil des Systems geben, den es vollständig verstehen kann.** Verantwortet ein Team zwölf Services in vier Domänen, ist es überlastet, und entweder Qualität, Tempo oder die Gesundheit der Rufbereitschaft leiden, egal wie sauber der Code jedes Service ist. Umgekehrt kann eine Domäne, die so kompliziert ist, dass kein Stream-

aligned Team sie bewältigen kann (eine Preis-Engine mit versicherungsmathematischen Modellen, ein Video-Codec), ein Spezialistenteam und eine klare Grenze um ihr Subsystem rechtfertigen.

Grenzen von der Teamstruktur prägen lassen

Zusammengenommen geben diese Ideen Architekten eine Reihe von Arbeitsregeln:

Richten Sie Grenzen an der Teamverantwortung aus. Jedes Modul, jeder Service, jeder Datenspeicher und jede Pipeline sollte genau ein verantwortliches Team haben. Geteilte Verantwortung heißt in der Praxis keine Verantwortung, und der Code wird widerspiegeln, was auch immer die letzte beitragende Person brauchte.

Minimieren Sie die Kommunikation, die jede Änderung braucht. Eine Grenze liegt an der richtigen Stelle, wenn die meisten Änderungen an einer fachlichen Fähigkeit von einem Team umgesetzt, getestet und deployt werden können. Achten Sie auf Änderungen, für die sich regelmäßig drei Teams abstimmen müssen: Sie zeigen, dass die Struktur des Systems und die Teamstruktur nicht übereinstimmen.

Machen Sie Schnittstellen zwischen Teams explizit und stabil. Schnittstellen zwischen Teams verdienen Versionierung, Dokumentation und Contract Tests; Schnittstellen innerhalb eines Teams können informell bleiben und sich frei ändern.

Überprüfen Sie beides gemeinsam. Umorganisationen sind Architekturentscheidungen; Architekturentscheidungen sind organisatorische Entscheidungen. Ein Architekt, der zu Teamänderungen nicht gefragt wird, oder eine Führungskraft, die umorganisiert, ohne den Code zu bedenken, ignoriert eine Kraft, die trotzdem wirkt.

Die Muster für Teamtypen und Interaktionsmodi und wie man sie weiterentwickelt, behandelt Team Topologies und soziotechnisches Design.

Wenn Organigramm und Architektur nicht übereinstimmen

Situation	Was Sie im System sehen	Typische Abhilfe
Teams nach technischer Schicht organisiert (Frontend, Backend, Datenbank)	Jedes Feature braucht drei Teams und drei Übergaben; Schichten entwickeln dicke, generische Schnittstellen; Releases werden koordiniert	Cross-funktionale Teams, an fachlichen Fähigkeiten ausgerichtet, die jeweils einen vertikalen Schnitt verantworten
Microservices mit je einem Team pro Service, aber ein zentrales Datenteam verantwortet alle Schemas	Ein verteiltes System, das über eine gemeinsame Datenbank gekoppelt ist; Schemaänderungen warten wochenlang	Jedem Service-Team die Hoheit über seine Daten geben; das Datenteam stellt eine Plattform und Standards bereit
Zwei Teams teilen sich die Verantwortung für einen Service	Widersprüchliche Konventionen in einer Codebasis, Merge-Konflikte, niemand in Rufbereitschaft für die schwierigen Teile	Den Service entlang der Anliegen der Teams aufteilen oder ihn einem Team übergeben
Ein Team verantwortet viele unzusammenhängende Services	Vernachlässigte Services, veraltete Abhängigkeiten, langsame Reaktion auf Incidents: kognitive Überlastung	Services zusammenlegen, einige an andere Teams abgeben oder in Plattform-Tooling investieren
Ein externer Dienstleister baut ein Subsystem unter einem Festpreisvertrag	Eine starre, stark spezifizierte Schnittstelle, deren Änderung sehr teuer ist	Die Grenze als Einwegtür behandeln; sie dort ziehen, wo Änderungen am wenigsten wahrscheinlich sind
Eine Umorganisation überträgt eine Fähigkeit an ein neues Team, aber der Code bleibt im Repository des alten Teams	Das neue Team umgeht Code, den es nicht verantwortet; Erosion der beabsichtigten Grenze	Code, Pipelines und Rufbereitschaft zusammen mit der Verantwortung übertragen

ACHTUNG

Häufige Missverständnisse

- Das Gesetz von Conway ist keine Empfehlung, das aktuelle Organigramm zu kopieren; es beschreibt eine Kraft. Die Entscheidung ist, ob Sie die bestehende Organisation das System prägen lassen oder die Organisation für das System gestalten, das Sie brauchen.
- Microservices entkommen dem Gesetz nicht. Hundert Services, gebaut von Teams, die sich alle abstimmen müssen, verhalten sich wie ein System, mit den zusätzlichen Kosten eines Netzwerks.
- Das Gesetz handelt von der Kommunikationsstruktur, nicht von den formalen Berichtslinien. Zwei Teams unter verschiedenen Führungskräften, die zusammensitzen und täglich miteinander sprechen, können eng integrierte Teile hervorbringen.
- Remote- und verteiltes Arbeiten hebt das Gesetz nicht auf; es verändert, welche Kommunikation günstig ist. Asynchrone, schriftliche, gut dokumentierte Schnittstellen werden zu den natürlichen Grenzen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum muss nach Conways Argument die Struktur des Systems der Struktur der Organisation ähneln?

Weil jede Schnittstelle zwischen Subsystemen von den Menschen ausgehandelt werden muss, die sie entwerfen; wo Menschen nicht kommunizieren, wird keine Schnittstelle entworfen, daher folgen die Verbindungen im System den Verbindungen zwischen Menschen.

▼ Wie viele Kommunikationspfade hat ein Team aus 12 Personen?

$12 \times 11 / 2 = 66$.

▼ Ein Unternehmen organisiert sich in Teams pro Microservice um, behält aber eine gemeinsame Datenbank, die jedes Team ändert. Was sagt das Gesetz von Conway vorher?

Die Services bleiben über die Datenbank gekoppelt, und Änderungen erfordern Abstimmung zwischen allen Teams, weil sich die Kommunikationsstruktur (alle arbeiten am selben Schema) nicht geändert hat.

▼ Was ist extrinsische kognitive Last, und wie kann Architektur sie verringern?

Aufwand, den die Umgebung statt des Problems auferlegt, etwa manuelle Deployments oder undurchsichtige Werkzeuge; Plattformen, Automatisierung und klare Grenzen verringern ihn, damit Teams sich auf ihre Domäne konzentrieren können.

▼ **Probieren Sie es aus: Eine Ticketing-Gruppe von 10 Personen wird auf 30 Ingenieure wachsen. Vergleichen Sie die Kommunikationspfade für ein Team mit 30 Personen und für drei Teams mit je 10 Personen und einer Verbindung zwischen jedem Teampaar. Was muss sich außerdem in der Software ändern?**

Ein Team aus 10 hat 10 x 9 / 2 = 45 Pfade. Ein Team aus 30 hat 30 x 29 / 2 = 435. Drei Teams aus 10 haben 3 x 45 = 135 innerhalb der Teams plus 3 Verbindungen, also 138. Das funktioniert aber nur, wenn jedes Team einen passenden Teil des Systems mit klaren Schnittstellen verantwortet. Sonst müssen sich die Teams ständig abstimmen, planen Sie also die Modulgrenzen und die Teamaufteilung gemeinsam.

QUELLEN

- Melvin Conway: How Do Committees Invent? (Datamation, April 1968): https://www.melconway.com/Home/Committees_Paper.html
- Martin Fowler: Conway's Law: https://martinfowler.com/bliki/ConwaysLaw.html
- MacCormack, Baldwin and Rusnak: Exploring the duality between product and organizational architectures (Research Policy, 2012): https://doi.org/10.1016/j.respol.2012.04.011
- Team Topologies: key concepts: https://teamtopologies.com/key-concepts
- Thoughtworks Technology Radar: Inverse Conway Maneuver: https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver

Einfachheit, Umkehrbarkeit und das Messen von Struktur

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 1.6 Entscheidungen und Abwägungen
- 1.7 Technische Schulden
- 3.4 Abhängigkeiten

Begriffe von früher, die er verwendet: Afferente Kopplung (Ca), Instabilität (I), Abhängigkeitszyklus, Kopplung, Refactoring

DIE IDEE IN KÜRZE

Manche Dinge sind schwer, weil die Aufgabe schwer ist: Eine Steuererklärung ist kompliziert, weil das Steuerrecht kompliziert ist. Andere Dinge sind schwer, weil wir sie so angehen: ein Ablagesystem mit sieben Ordnern, wo zwei genügen würden. Gute Designer akzeptieren die erste Art und bekämpfen die zweite. Sie bauen, was jetzt gebraucht wird, halten jede Tatsache an genau einer Stelle fest, vermeiden Entscheidungen, die sie nicht rückgängig machen können, und prüfen ihre Arbeit mit ein paar ehrlichen Messungen, im Bewusstsein, dass sich eine Zahl manipulieren lässt, wenn sie zum Ziel wird.

WAS SIE WISSEN MÜSSEN

- Fred Brooks unterschied essenzielle Komplexität, die zum Problem gehört, von akzidenteller Komplexität, die aus der Art entsteht, wie die Lösung gebaut ist; Architekten platzieren die erste und beseitigen die zweite.
- YAGNI warnt davor, für vermutete Bedürfnisse zu bauen, wegen der Kosten für Bau, Verzögerung, Mitschleppen und Reparatur, entschuldigt aber nicht, Tests, Refactoring oder Einwegtür-Entscheidungen zu vernachlässigen.
- Bei DRY, wie Hunt und Thomas es definieren, geht es darum, dass jedes Stück Wissen eine einzige maßgebliche Darstellung hat; Code zusammenzulegen, der nur ähnlich aussieht, erzeugt schädliche Kopplung.
- Flexibilität ist eine Option mit Preis: Zahlen Sie dafür, wenn die Änderung wahrscheinlich ist oder ohne sie sehr teuer wäre und die Option günstig ist, und verlassen Sie sich im Übrigen auf gute Modularität.

- Die zyklomatische Komplexität, M = E − N + 2P, zählt die unabhängigen Pfade durch eine Funktion und entspricht bei strukturiertem Code der Zahl der Entscheidungen plus eins; McCabe schlug eine Grenze von etwa 10 vor.
- LCOM, Kopplungs- und Stabilitätsmetriken, Change Coupling und Hotspots decken jeweils andere strukturelle Probleme auf, und alle werden irreführend, sobald man sie zu Zielvorgaben macht (Goodharts Gesetz).
- Fitness Functions machen aus strukturellen Entscheidungen, etwa verbotenen Abhängigkeiten, automatisierte Prüfungen, die den Build scheitern lassen, wenn sie verletzt werden.

Zwei Arten von Komplexität

In „No Silver Bullet: Essence and Accidents of Software Engineering“ (vorgestellt 1986 auf dem IFIP World Computer Congress und 1987 in *IEEE Computer* veröffentlicht) teilte Fred Brooks die Schwierigkeiten der Softwareentwicklung in zwei Arten ein. Essenzielle Komplexität gehört zum Problem: die tatsächlichen Regeln der Mehrwertsteuer, die tatsächliche Vielfalt von Versicherungsprodukten, die tatsächlichen Arten, auf die eine Lieferung scheitern kann. Kein Entwurf kann sie beseitigen, ohne zu ändern, was das System tut. Akzidentelle Komplexität entsteht aus der Art, wie wir die Lösung bauen: Sprachen, Werkzeuge, Frameworks, Deployment-Maschinerie, Workarounds und Strukturen, die komplizierter sind, als das Problem verlangt. Brooks argumentierte, dass höhere Programmiersprachen und andere Fortschritte bereits viel akzidentelle Schwierigkeit beseitigt hätten, sodass keine einzelne neue Technik innerhalb eines Jahrzehnts eine Verbesserung um das Zehnfache bringen würde.

Für Architekten ist die Unterscheidung ein tägliches Werkzeug. Essenzielle Komplexität muss *platziert* werden: dort, wo die Menschen, die sie verstehen, daran arbeiten können, hinter einer Grenze, die verhindert, dass sie in alles andere durchsickert (ein Preismodul, ein Steuerservice). Akzidentelle Komplexität muss *beseitigt oder verhindert* werden: Jeder zusätzliche Service, jede Queue, jeder Cache, jede Abstraktionsschicht und jede Konfigurationsoption fügt etwas davon hinzu, und jedes Element muss seinen Platz verdienen, indem es ein echtes Problem löst. Der Essay „Out of the Tar Pit“ (2006) von Ben Moseley und Peter Marks trieb die Idee weiter und benannte veränderlichen Zustand und Kontrollfluss als wichtige Quellen akzidenteller Komplexität; das ist ein Grund, warum funktionale und ereignisgesteuerte Entwürfe, die geteilten Zustand begrenzen, beliebt sind.

Der Rest dieses Abschnitts behandelt die Prinzipien, die akzidentelle Komplexität gering halten (KISS, YAGNI, richtig verstandenes DRY, Umkehrbarkeit), und die Messungen, die Ihnen helfen, Struktur objektiv zu sehen, samt ihren Grenzen.

KISS und YAGNI

KISS, meist ausgeschrieben als „keep it simple, stupid“, ist eine Designmaxime, die 1960 in der US Navy belegt ist und mit Kelly Johnson in Verbindung gebracht wird, dem leitenden Ingenieur der Skunk Works von Lockheed, der verlangt haben soll, dass Düsenflugzeuge im Feld von einem durchschnittlichen Mechaniker mit einer Handvoll Werkzeuge repariert werden können. Für Software gilt dasselbe: Ein Entwurf ist gut, wenn die Menschen, die ihn betreiben und reparieren müssen, oft um 3 Uhr morgens unter Druck, ihn verstehen können. Einfach heißt nicht leicht oder naiv. Es heißt wenige Konzepte, wenige bewegliche Teile und wenige Sonderfälle für das zu lösende Problem.

YAGNI („you aren’t gonna need it“, „Sie werden es nicht brauchen“) stammt aus dem Extreme Programming der späten 1990er-Jahre: Bauen Sie keine Fähigkeiten für Bedürfnisse, die Sie nur vermuten. Martin Fowlers Analyse ist nützlich, weil sie die Kosten benennt, die entstehen, wenn man es ignoriert. Da sind die **Kosten des Bauens** des vermuteten Features; die **Kosten der Verzögerung**, weil stattdessen etwas Wertvolles nicht gebaut wurde; die **Kosten des Mitschleppens**, weil jede spätere Änderung den zusätzlichen Code berücksichtigen muss; und, falls die Vermutung falsch war, die **Kosten der Reparatur**. Selbst wenn die Vermutung sich als richtig erweist, wurde das Feature mit weniger Wissen gebaut, als Sie später gehabt hätten.

Zwei Einschränkungen verhindern, dass YAGNI zur Ausrede wird. Erstens gilt es für *Features und spekulative Flexibilität*, nicht für die Praktiken, die Änderungen günstig machen. Automatisierte Tests, Refactoring, Continuous Delivery und klare Modulgrenzen sind es, die das Aufschieben von Features sicher machen; wer sie im Namen von YAGNI weglässt, verwandelt Aufschub in Schulden. Zweitens sind manche Entscheidungen Einwegtüren (Abschnitt 1.6). Die Wahl eines Datenmodells, der Form einer öffentlichen API, eines Mandantenmodells oder einer Sicherheitsgrenze ist kein Feature, das Sie später zu denselben Kosten ergänzen können. Dort ist Vorausdenken keine Spekulation, sondern die Aufgabe des Architekten.

DRY meint Wissen, nicht Text

Andy Hunt und Dave Thomas führten DRY in *The Pragmatic Programmer* (1999; Jubiläumsausgabe zum 20. Jahrestag 2019) mit dieser Definition ein: „Jedes Stück Wissen muss innerhalb eines Systems eine einzige, eindeutige, maßgebliche Darstellung haben.“ Das Schlüsselwort ist *Wissen*. Wenn der Mehrwertsteuersatz, die Regel für eine gültige Postleitzahl oder die Bedeutung eines Bestellstatus an mehreren Stellen steht, muss man für eine Änderung jede Kopie finden, und die Kopien werden auseinanderdriften. Die Ausgabe von 2019 betont, dass es bei DRY um die Duplizierung von Wissen und Absicht geht, nicht um Codezeilen, die zufällig gleich aussehen.

Der häufige Missbrauch geht in die andere Richtung. Zwei Funktionen, die gleich aussehen, werden zusammengelegt, weil sie gleich aussehen, obwohl sie unterschiedliches Wissen darstellen, das sich aus unterschiedlichen Gründen ändern wird: Die Liefergebührenregel und die Restaurantprovisionsregel multiplizieren heute beide eine Summe mit einem Prozentsatz, aber sie gehören zu verschiedenen Akteuren (Finanzen und Partnermanagement) und werden auseinanderlaufen. Sie zusammenzulegen koppelt zwei Konzepte, die die Domäne getrennt hält: Beide hängen nun von einer gemeinsamen Funktion ab, sodass eine für die eine Regel gewünschte Änderung stillschweigend die andere ändert. In den Begriffen von Abschnitt 3.6 erzeugt das Connascence, wo das Problem keine hat (eine gemeinsame Funktion ist nur dann die richtige Lösung, wenn beide Seiten wirklich dasselbe berechnen müssen); es ist außerdem ein Verstoß gegen das Single-Responsibility-Prinzip (Abschnitt 3.5). Sandi Metz' bekannter Rat lautet, Duplizierung der falschen Abstraktion vorzuziehen, weil eine falsche gemeinsame Abstraktion Parameter und Bedingungen ansammelt, bis niemand mehr wagt, sie zu ändern. Viele Teams nutzen die Dreierregel: eine zweite Kopie tolerieren und eine Abstraktion herausziehen, wenn die dritte auftaucht und das gemeinsame Wissen klar ist.

Auf Architekturebene wird die Abwägung explizit. Eine gemeinsame Bibliothek, die jeder Service nutzen muss, beseitigt duplizierten Code, koppelt aber die Release-Zyklen der Services. Viele Microservice-Organisationen akzeptieren bewusst duplizierten *Code* (kleine Hilfsfunktionen, in jeden Service kopiert), bestehen aber auf genau einem Verantwortlichen für dupliziertes *Wissen* (den kanonischen Kundendatensatz, die Steuerregeln). Fragen Sie bei jeder Duplizierung: Wenn sich diese Regel ändert, müssen sich dann alle Kopien gemeinsam ändern? Wenn ja, ist es dupliziertes Wissen und sollte einen Verantwortlichen haben. Wenn nein, ist es zufällige Ähnlichkeit und kann bleiben.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: wenn das Zusammenlegen ähnlich aussehenden Codes nach hinten losgeht

Der schwierigste Gedanke dieses Abschnitts ist, dass es bei DRY um Wissen geht, nicht um Code, der gleich aussieht. Kapitel 0 hat gezeigt, warum eine Regel nur einmal geschrieben werden sollte; die Schwierigkeit liegt darin, eine zweimal geschriebene Regel von zwei Regeln zu unterscheiden, die sich nur ähneln. Folgen Sie dem Beispiel aus dem Abschnitt selbst.

- Auf der Essenslieferplattform multiplizieren die Liefergebührenregel und die Restaurantprovisionsregel heute beide eine Bestellsumme mit einem Prozentsatz. Dieselbe Rechnung taucht zweimal auf.
- Eingedenk der Regel, dass eine Regel an einer Stelle leben sollte, legt ein Entwickler sie zu einer gemeinsamen Funktion zusammen. Der Code ist kürzer.
- Monate später wird eine Änderung nur für die Provisionsregel gewünscht: Die Provision wird nun pro Bestellung auf einen Höchstbetrag begrenzt. Der Entwickler baut die Obergrenze in die gemeinsame Funktion ein, und stillschweigend werden auch die Liefergebühren begrenzt. Das hat niemand verlangt.
- Zur Reparatur wird ein Flag ergänzt, damit die Funktion erkennt, welche Regel sie berechnet: die Kontrollkopplung aus Abschnitt 3.2. Jeder spätere Wunsch fügt einen weiteren Parameter oder eine weitere Bedingung hinzu, bis niemand mehr wagt, die Funktion zu ändern.
- Die beiden Regeln waren nie dasselbe Wissen: Sie gehören zu verschiedenen Teams (Finanzen und Partnermanagement) und ändern sich aus verschiedenen Gründen. Zwei kurze, getrennte Funktionen waren der bessere Entwurf.
- Vergleichen Sie damit den Mehrwertsteuersatz, der im Checkout, in der Rechnung und auf dem Erstattungsbildschirm steht: Jede Kopie muss sich ändern, sobald sich der Satz ändert. Das ist ein Stück Wissen, dreimal geschrieben, und es braucht einen Verantwortlichen.
- **Die Lehre:** Der Test ist, ob sich alle Kopien gemeinsam ändern müssen, wenn sich die Regel ändert. Wenn ja, geben Sie der Regel ein Zuhause. Wenn nicht, ist die Ähnlichkeit Zufall, und ein wenig Duplizierung ist meist günstiger als die falsche gemeinsame Abstraktion.

Umkehrbarkeit und die Kosten von Optionen

Einfachheit hat eine zeitliche Dimension. Ein Entwurf, der heute einfach, aber unmöglich zu ändern ist, ist nicht günstig; ein Entwurf voller Flexibilität, die niemand nutzt, ist es auch nicht. Das nützliche Konzept ist Umkehrbarkeit: Wie teuer wäre es, diese Entscheidung später

rückgängig zu machen oder zu ändern? Abschnitt 1.6 hat Einwegtüren und Zweiwegtüren eingeführt. Hier geht es darum, dass sich Flexibilität wie eine Finanzoption verhält: Sie hat jetzt einen Preis (zusätzlicher Code, zusätzliche Abstraktion, zusätzliche Betriebskosten) und einen Wert, der davon abhängt, wie wahrscheinlich und wie teuer die Änderung ist, gegen die sie absichert.

Daraus ergibt sich eine Faustregel, die YAGNI mit vorausschauendem Denken versöhnt. **Bezahlen Sie für eine Option nur, wenn die Änderung einigermaßen wahrscheinlich ist oder ohne die Option sehr teuer wäre, und wenn die Option selbst günstig ist.** Einen Zahlungsanbieter hinter einer eigenen Schnittstelle zu kapseln kostet einen Tag und schützt vor einer Änderung, die sowohl plausibel als auch teuer ist: eine gute Option. Eine Multi-Cloud-Abstraktionsschicht für ein Start-up mit einem einzigen Kunden zu bauen ist eine teure Option gegen eine unwahrscheinliche Änderung. Oft ist die günstigste Option einfach gute Modularität: Gut gesetzte Grenzen halten viele künftige Änderungen lokal, ohne dass jemand sie vorhersagen muss; das ist Parnas' ursprüngliches Argument für Information Hiding (Abschnitt 3.1).

Struktur messen: zyklomatische Komplexität

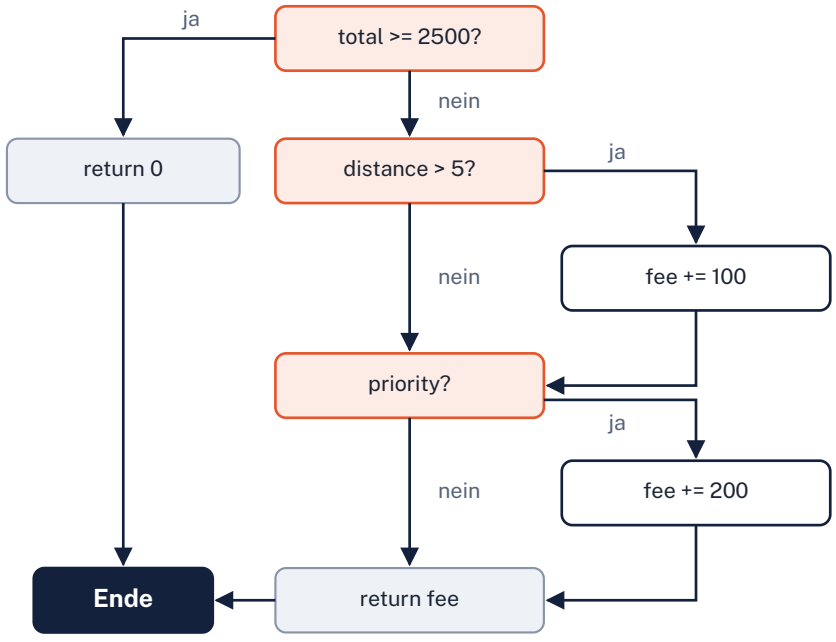
Das Urteil über Einfachheit wird besser, wenn es sich auf Messungen stützt. Die älteste weit verbreitete Codemetrik ist die zyklomatische Komplexität, die Thomas McCabe in „A Complexity Measure“ (*IEEE Transactions on Software Engineering*, 1976) einführte. Sie betrachtet eine Funktion als Kontrollflussgraphen, mit Knoten für Codeblöcke und Kanten für mögliche Kontrollübergänge, und zählt die linear unabhängigen Pfade durch ihn: $M = E - N + 2P$, wobei E die Zahl der Kanten, N die Zahl der Knoten und P die Zahl der Zusammenhangskomponenten ist (1 für eine einzelne Funktion). Bei strukturiertem Code mit einem Einstiegspunkt entspricht dies der Zahl der Entscheidungspunkte plus eins.

McCabe schlug 10 als vernünftige Obergrenze für ein Modul vor. Der spätere NIST-Bericht zum strukturierten Testen von Arthur Watson und McCabe (1996) merkt an, dass erfahrene Teams Grenzen bis 15 erfolgreich verwendet haben, und empfiehlt eine schriftliche Begründung, wann immer ein Modul die gewählte Grenze überschreitet. Werkzeuge zur statischen Analyse wie `radon` oder das `mccabe`-Plugin für flake8 in Python, die ESLint-Regel `complexity` in JavaScript und SonarQube berechnen sie automatisch. Die Werkzeuge unterscheiden sich in Details, etwa ob jedes `and` oder `or` in einer Bedingung als zusätzliche Entscheidung zählt; vergleichen Sie daher nur Zahlen aus demselben Werkzeug.

```
def delivery_fee(total: int, distance_km: float, priority: bool) → int:
    if total ⩾ 2500:          # decision 1: free delivery over 25 pounds
        return 0
    fee = 299
    if distance_km > 5:        # decision 2
        fee += 100
    if priority:               # decision 3
        fee += 200
    return fee
```

Drei Entscheidungen, also zyklomatische Komplexität 4.

Der Kontrollflussgraph und die Zählung



Zählung

Knoten N = 8

Kanten E = 10

M = E − N + 2

M = 10 − 8 + 2 = 4

Probe: 3 Entsch. + 1 = 4

4 unabhängige Pfade

also mindestens 4 Testfälle

für Basispfad-Tests

Acht Knoten und zehn Kanten ergeben M = 10 − 8 + 2 = 4, passend zu drei Entscheidungen plus eins.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: wenn die Funktion wächst

Ein Jahr später hat `delivery_fee` eine Schleife über den Warenkorb und eine zusammengesetzte Bedingung dazubekommen. Ermitteln Sie ihre Komplexität und entscheiden Sie, was zu tun ist.

Neuer Code	`for item in items:` mit `if item.bulky: fee += 50` darin, und `if priority and rider_available: fee += 200` ersetzt `if priority`.
Entscheidungen	Ursprünglich 3, plus die `for`-Schleife (1), plus das `if item.bulky` (1), plus das `and` (1, falls das Werkzeug boolesche Operatoren zählt): 6.
Zyklomatische Komplexität	6 + 1 = 7. Unter McCabes Grenze von 10, aber fast verdoppelt.
Folgen für das Testen	Basispfad-Tests brauchen mindestens 7 Testfälle, um jeden unabhängigen Pfad einmal zu durchlaufen; die alte Suite mit 4 deckt nicht mehr jeden Pfad ab.
Deutung für den Entwurf	Die Funktion vermischt nun drei Preisanliegen: Schwellenwert für die Warenkorbgröße, Entfernungs- und Artikelzuschläge sowie Priorität. Sie ändern sich auf Wunsch verschiedener Stellen (Marketing, Betrieb, Produkt).
Refactoring	Ziehen Sie `distance_surcharge`, `bulky_surcharge` und `priority_surcharge` heraus, jeweils mit Komplexität 2 oder 3, und lassen Sie `delivery_fee` sie summieren. Die Gesamtkomplexität ist ähnlich, aber jedes Teil ist klein, benannt und für sich testbar.

Refactoring senkt die Gesamtkomplexität selten; es verteilt sie auf Teile, die leichter zu verstehen, zu testen und unabhängig zu ändern sind. Die essenzielle Komplexität der Preisregeln bleibt.

Die Metrik hat einen Trend angezeigt, der ein Gespräch wert war; die Entscheidung zur Aufteilung fiel beim Lesen des Codes und mit der Frage, wer ihn ändert.

Struktur jenseits von Funktionen messen

Die zyklomatische Komplexität sieht jeweils nur eine Funktion. Architekten brauchen Maße dafür, wie Module zueinander stehen.

Kopplungs- und Stabilitätsmetriken. Abschnitt 3.4 (Abhängigkeiten steuern) hat afferente und efferente Kopplung eingeführt, die Instabilität $I = Ce / (Ca + Ce)$, die Abstraktheit und den Abstand von der Hauptreihe. Sie lassen sich günstig aus Importgraphen berechnen und decken

gut Komponenten auf, von denen viele abhängen, die aber selbst von volatilem Code abhängen; außerdem finden sie Abhängigkeitszyklen.

Kohäsionsmetriken. Die Familie der LCOM-Metriken (lack of cohesion of methods, mangelnde Kohäsion der Methoden) schätzt ab, ob eine Klasse eine Verantwortlichkeit trägt oder mehrere. Die ursprüngliche Version von Shyam Chidamber und Chris Kemerer (1994) vergleicht Methodenpaare danach, ob sie gemeinsame Felder nutzen. Eine weit verbreitete spätere Variante, LCOM4 von Martin Hitz und Behzad Montazeri, baut einen Graphen, in dem Methoden verbunden sind, wenn sie ein Feld teilen oder die eine die andere aufruft, und zählt die zusammenhängenden Gruppen. Eine Klasse mit LCOM4 = 1 ist in sich geschlossen; eine Klasse mit LCOM4 = 3 sind drei Klassen, die sich eine Datei teilen. Zum Beispiel hat eine Klasse `Customer`, deren `update_address` und `format_address` die Adressfelder nutzen, deren `add_points` und `points_balance` die Treuefelder nutzen und deren `send_newsletter` nur das E-Mail-Feld nutzt, LCOM4 = 3, was auf die Verantwortlichkeiten Adresse, Treueprogramm und Marketing hindeutet.

Historienbasierte Metriken. Die Versionsverwaltung zeichnet auf, wie sich das System tatsächlich ändert, was die statische Analyse nicht sehen kann. Change Coupling findet Dateien oder Services, die wiederholt gemeinsam committet werden: Wenn sich `pricing.py` und `invoice_template.html` in 70 % der Commits gemeinsam ändern, sind sie gekoppelt, ob eine die andere importiert oder nicht, und die Kopplung kann eine Servicegrenze überschreiten, die sie eigentlich trennen sollte. Hotspots kombinieren Änderungshäufigkeit mit Komplexität, um Code zu finden, der zugleich schwer zu bearbeiten ist und häufig bearbeitet wird; Adam Tornhills *Your Code as a Crime Scene* (2015; zweite Auflage 2024) hat die Technik bekannt gemacht. Abschnitt 1.7 über technische Schulden hat gezeigt, wie Hotspots die Tilgung auf die Schulden lenken, die die höchsten Zinsen kosten.

Strukturmetriken auf einen Blick

Metrik	Bereich	Was sie aufdeckt	Größter blinder Fleck
Zyklomatische Komplexität	Funktion	Verzweigungslogik, die schwer zu testen und zu durchschauen ist	Sagt nichts über Benennung, Daten oder das Zusammenspiel von Funktionen
Ca, Ce, Instabilität, Abstand	Paket oder Komponente	Abhängigkeitsrichtung, Stabilität, Zyklen	Zählt Abhängigkeiten, nicht ihre Stärke oder Bedeutung
LCOM (zum Beispiel LCOM4)	Klasse	Klassen mit mehreren unzusammenhängenden Verantwortlichkeiten	Empfindlich gegenüber Zugriffsmethoden und Sprachmerkmalen; verrauscht bei kleinen Klassen
Change Coupling	Datei, Modul oder Service	Verborgene Kopplung, die sich nur darin zeigt, wie sich das System ändert	Braucht genügend Historie; Sammel-Commits erzeugen Rauschen
Hotspots	Datei oder Modul	Wo sich Verbesserungen am schnellsten auszahlen	Priorisiert; sagt nicht, was falsch ist
Connascence (per Review beurteilt)	Beliebig	Form und Stärke jeder Abhängigkeit	Weitgehend manuell; wenige Werkzeuge messen die dynamischen Formen

Goodharts Gesetz und die Grenzen von Metriken

Jede der obigen Metriken ist ein Stellvertreter für etwas, das uns wichtig ist, das wir aber nicht direkt messen können: wie leicht sich das System verstehen und ändern lässt. Goodharts Gesetz, benannt nach dem Ökonomen Charles Goodhart (1975) und am bekanntesten in der Formulierung der Anthropologin Marilyn Strathern von 1997, warnt: Wenn ein Maß zum Ziel wird, hört es auf, ein gutes Maß zu sein. Menschen optimieren die Zahl, nicht die Eigenschaft.

Softwareteams erleben das ständig. Eine harte Grenze von zyklomatischer Komplexität 10 erzeugt Funktionen, die an willkürlichen Stellen aufgeteilt sind und jeweils die Prüfung bestehen, während die Logik schwerer nachvollziehbar wird. Ein Abdeckungsziel von 80 % erzeugt Tests, die Code ausführen, ohne irgendetwas zu prüfen. Eine Regel, dass keine Klasse 200 Zeilen überschreiten darf, erzeugt Klassen, die in `OrderHelper` und `OrderHelper2` aufgeteilt sind. Ein Ziel von „null Zyklen“ lässt sich erreichen, indem man Code in ein Sammelbecken-Paket `common` verschiebt, von dem alles abhängt.

Fachleute nutzen Metriken auf drei Arten, die Manipulation erschweren. Sie betrachten **Trends** statt absoluter Schwellenwerte (die Komplexität im Zahlungsmodul hat sich in sechs Monaten verdoppelt: warum?). Sie nutzen Metriken, um **Gespräche anzustoßen und Reviews zu priorisieren**, nicht, um Menschen zu benoten. Und sie **kombinieren** Metriken, die in verschiedene Richtungen ziehen, etwa Komplexität mit Änderungshäufigkeit oder Kopplung mit Teamverantwortung, sodass sich keine einzelne Zahl auf Kosten der anderen optimieren lässt.

Fitness Functions: Struktur prüfbar machen

Manche strukturellen Regeln sind klar genug, um sie zu automatisieren, und dann hängen sie nicht mehr von Wachsamkeit ab. *Building Evolutionary Architectures* von Neal Ford, Rebecca Parsons und Patrick Kua (2017; zweite Auflage 2022 mit Pramod Sadalage) nennt solche Prüfungen Fitness Functions: objektive, automatisierte Tests eines Architekturmerkmals, die in der Build-Pipeline oder gegen das laufende System laufen. Beispiele für Struktur sind: keine Abhängigkeitszyklen zwischen Paketen der obersten Ebene; das Domänenpaket importiert nichts aus den Web- oder Datenbankpaketen; keine Funktion über Komplexität 15 ohne ausdrückliche Ausnahme; kein Service liest die Datenbanktabellen eines anderen Service.

Werkzeuge wie ArchUnit (Java), import-linter (Python) und dependency-cruiser (JavaScript und TypeScript) drücken diese Regeln als Code aus; Abschnitt 3.4 hat Beispiele gegeben. Der Unterschied zu einer Dashboard-Metrik ist, dass eine Fitness Function eine *Entscheidung* kodiert, die das Team bereits getroffen hat; wird sie verletzt, scheitert der Build und erzwingt eine bewusste Wahl: den Code korrigieren oder die Regel ändern und festhalten, warum. Die vollständige Behandlung, einschließlich Fitness Functions für Performance, Sicherheit und andere Qualitäten, finden Sie in Fitness Functions und evolutionäre Architektur.

```
[importlinter]
root_package = shop

[importlinter:contract:domain-is-pure]
name = Domain must not depend on infrastructure
type = forbidden
source_modules =
    shop.domain
forbidden_modules =
    shop.infrastructure
    shop.web
```

Ein import-linter-Vertrag in setup.cfg oder .importlinter, der den Build scheitern lässt, wenn die Domänenschicht Infrastruktur importiert.

Checkliste: den Entwurf eines Moduls prüfen

Nutzen Sie diese Liste, wenn Sie ein neues Modul, eine Komponente oder einen Service prüfen oder eine wesentliche Änderung daran. Sie greift auf jeden Abschnitt dieses Kapitels zurück.

- [ ] **Können Sie seine Verantwortlichkeit in einem Satz angeben und den Akteur nennen, der seine Änderungen verlangt?**
  Braucht der Satz ein „und“ oder ändern es mehrere Akteure, erwägen Sie eine Aufteilung (Modularität, SRP, CCP).
- [ ] **Verbirgt es eine Designentscheidung, die sich wahrscheinlich ändert?**
  Ein Modul sollte ein Geheimnis besitzen, etwa ein Datenformat, einen Anbieter oder einen Algorithmus (Information Hiding).
- [ ] **Ist die Schnittstelle klein im Verhältnis zu dem, was sie verbirgt?**
  Tiefe Module verbergen viel hinter wenig; durchreichende Schichten verursachen Kosten, ohne etwas zu verbergen.
- [ ] **Zeigen seine Abhängigkeiten auf stabilere, abstraktere Module?**
  Prüfen Sie auf Zyklen und auf stabile Module, die von volatilen abhängen (ADP, SDP, DIP).
- [ ] **Was muss ein Client wissen, um es korrekt zu nutzen?**
  Achten Sie auf Positionsparameter, magische Werte, vorgeschriebene Aufrufreihenfolgen und Annahmen über das Timing; schwächen Sie sie ab oder halten Sie sie lokal (Connascence).
- [ ] **Wer verantwortet es, und kann dieses Team es ändern und deployen, ohne sich mit anderen abzustimmen?**
  Ein verantwortliches Team pro Modul; Schnittstellen zwischen Teams sollten explizit und versioniert sein (Gesetz von Conway).
- [ ] **Ist ein Teil seiner Komplexität akzidentell?**
  Hinterfragen Sie jede zusätzliche Schicht, Option und jeden generischen Mechanismus: Welcher echten Anforderung dient er (KISS, YAGNI)?
- [ ] **Ist Wissen dupliziert oder zufällige Ähnlichkeit fälschlich geteilt?**
  Dupliziertes Wissen braucht einen Verantwortlichen; ähnlich aussehender Code mit verschiedenen Verantwortlichen kann getrennt bleiben (DRY).

- [ ] **Welche Entscheidungen darin sind schwer umkehrbar, und sind sie bewusst getroffen?**
  Formen öffentlicher APIs, Datenmodelle und Grenzen verdienen mehr Prüfung als interne Details.

- [ ] **Was sagen die Metriken und die Historie?**
  Prüfen Sie Komplexitätstrends, Kopplungsmetriken, Change Coupling und Hotspots, und ergänzen Sie eine Fitness Function für jede Regel, die es wert ist, beibehalten zu werden.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Eine Funktion hat zwei `if`-Anweisungen und eine `while`-Schleife und keine booleschen Operatoren. Wie hoch ist ihre zyklomatische Komplexität?**

Drei Entscheidungen plus eins, also 4.

▼ **Zwei Services enthalten jeweils eine identische 20-zeilige Funktion zum Formatieren von Datumsangaben. Ist das ein Verstoß gegen DRY?**

Wahrscheinlich nicht. Es ist duplizierter Text, kein dupliziertes Fachwissen, und eine gemeinsame Nutzung würde die Releases der Services koppeln; duplizierte Geschäftsregeln, etwa die Steuerberechnung, sind etwas anderes.

▼ **Warum gilt YAGNI nicht für das Schreiben automatisierter Tests?**

Tests gehören zu dem, was künftige Änderungen günstig und sicher macht; bei YAGNI geht es darum, keine spekulativen Features und keine spekulative Flexibilität zu bauen, und es ist auf solche Praktiken angewiesen, um zu funktionieren.

▼ **Einem Team wird gesagt, dass keine Funktion Komplexität 10 überschreiten darf, und die Komplexität sinkt, während die Fehler zunehmen. Was ist wahrscheinlich passiert?**

Goodharts Gesetz: Funktionen wurden mechanisch aufgeteilt, um die Metrik zu erfüllen, wodurch die Logik schwerer nachvollziehbar wurde, ohne einfacher zu werden.

▼ **Probieren Sie es aus: Zwei Services haben jeweils eine 20-zeilige Adressprüfung, die fast identisch aussieht. Die eine prüft Restaurantadressen auf die rechtliche Registrierung, die andere, ob ein Kurier eine Haustür erreichen kann. Ein Kollege möchte eine gemeinsame Bibliothek. Stimmen Sie zu?**

Wahrscheinlich nicht. Bei DRY geht es darum, dass ein Stück Wissen einen einzigen Ort hat, und hier sieht der Code ähnlich aus, drückt aber zwei verschiedene Regeln aus, die sich aus verschiedenen Gründen ändern. Eine Zusammenführung würde sie koppeln, und der gemeinsame Code würde bald Flags für jeden Fall ansammeln. Behalten Sie die zwei Kopien, und teilen Sie allenfalls einen neutralen Postleitzahl-Parser, wenn das wirklich dasselbe Wissen ist. Prüfen Sie es erneut, wenn eine dritte Kopie auftaucht und das gemeinsame Wissen klar ist.

QUELLEN

- Fred Brooks: No Silver Bullet (UNC technical report version, 1986): https://techreports.cs.unc.edu/papers/86-020.pdf
- Martin Fowler: Yagni: https://www.martinfowler.com/bliki/Yagni.html
- Sandi Metz: The Wrong Abstraction: https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction
- Watson and McCabe: Structured Testing (NIST Special Publication 500-235, 1996): https://www.mccabe.com/pdf/mccabe-nist235r.pdf
- Martin Fowler: Beck Design Rules: https://martinfowler.com/bliki/BeckDesignRules.html

Weiterführende Literatur

- *On the Criteria To Be Used in Decomposing Systems into Modules*, David L. Parnas (1972), Communications of the ACM, volume 15, issue 12. Sechs kurze Seiten, die Information Hiding am KWIC-Beispiel einführten; nach wie vor das beste Argument für eine Zerlegung nach wahrscheinlichen Änderungen.
  https://doi.org/10.1145/361598.361623
- *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Edward Yourdon and Larry L. Constantine (1979), Yourdon Press / Prentice-Hall. Das Buch, das Kopplung und Kohäsion zum Standardvokabular machte, mit der ursprünglichen Diskussion ihrer Stufen.
- *How Do Committees Invent?*, Melvin E. Conway (1968), Datamation (April 1968). Die ursprüngliche Formulierung des Gesetzes von Conway, frei verfügbar auf der Website des Autors und differenzierter als die einzeilige Zusammenfassung.
  https://www.melconway.com/Home/Committees_Paper.html
- *A Philosophy of Software Design (2nd edition)*, John Ousterhout (2021), Yaknyam Press. Ein kurzes, meinungsstarkes Buch über das Reduzieren von Komplexität, die Quelle der tiefen und flachen Module.
  https://web.stanford.edu/~ouster/cgi-bin/book.php
- *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Robert C. Martin (2017), Prentice Hall. Fasst die SOLID-Prinzipien, die Kohäsions- und Kopplungsprinzipien für Komponenten sowie die Metriken Instabilität und Abstraktheit an einem Ort zusammen.
- *Balancing Coupling in Software Design*, Vlad Khononov (2024), Addison-Wesley. Eine moderne Synthese aus Structured Design, Connascence und Domain-Driven Design, die Kopplung nach Stärke, Distanz und Volatilität zugleich beurteilt.
- *Software Engineering at Google: Lessons Learned from Programming Over Time*, Titus Winters, Tom Manshreck and Hyrum Wright (eds) (2020), O'Reilly Media. Kostenlos online lesbar; behandelt Hyrums Gesetz und beschreibt hervorragend, wie man Schnittstellen und Abhängigkeiten im großen Maßstab gesund hält.
  https://abseil.io/resources/swe-book
- *Building Evolutionary Architectures (2nd edition)*, Neal Ford, Rebecca Parsons, Patrick Kua and Pramod Sadalage (2022), O'Reilly Media. Zeigt, wie man strukturelle Entscheidungen, etwa Abhängigkeitsregeln, mit automatisierten Fitness Functions schützt.

Glossar

Abhängigkeitsumkehr
Ein High-Level-Modul und ein Low-Level-Modul hängen beide von einer Abstraktion ab, die meist der High-Level-Seite gehört, sodass die Quellcode-Abhängigkeit entgegen dem Kontrollfluss zeigt.

Abstraktheit (A)
Martins Metrik: die Zahl der abstrakten Klassen und Schnittstellen einer Komponente geteilt durch die Gesamtzahl ihrer Klassen, von 0 (vollständig konkret) bis 1 (vollständig abstrakt).

Abstraktion
Eine vereinfachte Sicht auf etwas, die für einen bestimmten Zweck das Wesentliche behält und den Rest weglässt, etwa eine „Datei“, die Festplatten, Sektoren und Caching verbirgt.

Acyclic Dependencies Principle
Das Prinzip von Robert C. Martin, nach dem der Abhängigkeitsgraph zwischen Komponenten keine Zyklen enthalten darf.

Afferente Kopplung (Ca)
Die Zahl der Elemente außerhalb einer Komponente, die von ihr abhängen: ihre eingehenden Abhängigkeiten. Eine hohe Ca bedeutet, dass viele andere betroffen sind, wenn sie sich ändert.

Akzidentelle Komplexität
Komplexität, die durch die Art entsteht, wie eine Lösung gebaut ist (Werkzeuge, Struktur, Notlösungen), und nicht durch das Problem selbst. Sie lässt sich grundsätzlich beseitigen.

Backend for Frontend (BFF)
Eine serverseitige API, die für eine bestimmte Art von Client gebaut ist, etwa die mobile App, sodass jeder Client Operationen und Daten erhält, die auf seine Bedürfnisse zugeschnitten sind.

Common Closure Principle (CCP)
Klassen, die sich aus denselben Gründen und zu denselben Zeitpunkten ändern, gehören in dieselbe Komponente; das Single Responsibility Principle, angewandt auf Komponenten.

Common Reuse Principle (CRP)
Klassen, die zusammen verwendet werden, gehören zusammen, und Nutzer einer Komponente sollten nicht gezwungen sein, von Klassen abzuhängen, die sie nicht verwenden.

Connascence
Zwei Elemente sind connascent, wenn eine Änderung an einem eine Änderung am anderen erfordern würde, damit das System korrekt bleibt. Eine feinere Art, Kopplung zu beschreiben, eingeführt von Meilir Page-Jones.

Datenhoheit
Die Regel, dass jedes Datum von genau einem Modul oder Service geschrieben wird, den andere fragen müssen, statt seinen Speicher direkt zu lesen oder zu ändern.

Dependency Inversion Principle (DIP)
High-Level-Policy soll nicht von Low-Level-Details abhängen; beide sollen von Abstraktionen abhängen, und die Abstraktion soll der Policy-Seite gehören.

Deployment-Kopplung
Eine Situation, in der zwei Teile gemeinsam oder in fester Reihenfolge ausgeliefert werden müssen, weil eine Änderung am einen den anderen bricht.

Design by Contract
Die Methode von Bertrand Meyer, jede Operation mit Vorbedingungen, die der Aufrufer erfüllen muss, Nachbedingungen, die die Operation garantiert, und Invarianten, die immer gelten, zu spezifizieren.

DRY
Don’t repeat yourself: das Prinzip aus „The Pragmatic Programmer“, dass jedes Stück Wissen in einem System eine einzige maßgebliche Darstellung haben soll. Es geht um Wissen, nicht um gleich aussehenden Text.

Dynamische Connascence
Connascence, die sich erst zeigt, wenn das Programm läuft, etwa die Abhängigkeit von Reihenfolge oder Zeitpunkt der Ausführung, von zusammenhängenden Werten oder davon, dass zwei Teile dieselbe Objektinstanz teilen.

Efferente Kopplung (Ce)
Die Zahl der Elemente außerhalb einer Komponente, von denen sie abhängt: ihre ausgehenden Abhängigkeiten. Eine hohe Ce bedeutet, dass sie den Änderungen vieler anderer ausgesetzt ist.

Essenzielle Komplexität
Komplexität, die zum Problem selbst gehört, etwa die tatsächlichen Regeln des Steuer- oder Versicherungswesens, und die sich nicht beseitigen lässt, ohne zu ändern, was das System tut.

Goodharts Gesetz
Die Beobachtung, oft als „Wenn eine Kennzahl zum Ziel wird, hört sie auf, eine gute Kennzahl zu sein“ formuliert, dass Menschen die Zahl optimieren statt der Sache, die sie anzeigen sollte.

Grad (der Connascence)
Wie viele Elemente eine bestimmte Connascence teilen. Ein Wert, auf den sich zwei Funktionen geeinigt haben, ist weit leichter zu ändern als einer, auf den sich vierzig Services geeinigt haben.

Granularität
Die Größe von Modulen oder Services im Verhältnis zum Gesamtsystem. Grobgranular heißt wenige große Teile, feingranular heißt viele kleine. Beides hat Kosten.

Hauptreihe (Main Sequence)
Die Gerade $A + I = 1$ in einem Diagramm, das Abstraktheit gegen Instabilität aufträgt und auf der ausgewogene Komponenten liegen. Der Abstand von ihr ist $D = |A + I - 1|$.

Hyrums Gesetz
Die nach Hyrum Wright benannte Beobachtung, dass bei genügend Nutzern einer API jedes beobachtbare Verhalten von irgendjemandem vorausgesetzt wird, gleichgültig, was der Vertrag verspricht.

Information Hiding
Das Prinzip von David Parnas (1972), dass jedes Modul eine Designentscheidung verbergen soll, die schwierig ist oder sich wahrscheinlich ändert, damit sich die Entscheidung ändern kann, ohne andere Module zu beeinflussen.

Instabilität (I)
Martins Metrik I = Ce / (Ca + Ce), von 0 (maximal stabil: nur eingehende Abhängigkeiten) bis 1 (maximal instabil: nur ausgehende Abhängigkeiten).

Interface Segregation Principle (ISP)
Clients sollen nicht gezwungen sein, von Operationen abzuhängen, die sie nicht nutzen; mehrere kleine, clientspezifische Schnittstellen sind einer großen vorzuziehen.

Invariante
Eine Bedingung, die für ein Modul oder Objekt zwischen Operationen immer wahr ist, etwa „ein Kontostand fällt nie unter sein Überziehungslimit“.

Inverse Conway Maneuver
Teams und ihre Kommunikationswege bewusst so gestalten, dass sie der gewünschten Architektur entsprechen, damit das Gesetz von Conway für Sie statt gegen Sie arbeitet.

KISS
Keep it simple: eine Faustregel für den Entwurf aus der Ingenieurspraxis der 1960er-Jahre, nach der unnötige Komplexität zu vermeiden ist, weil sie ein System schwerer verständlich, schwerer zu betreiben und schwerer zu reparieren macht.

Kognitive Last des Teams
Die gesamte Menge an Wissen und geistiger Anstrengung, die ein Team braucht, um die Teile eines Systems, für die es zuständig ist, zu bauen, zu betreiben und zu ändern. Übersteigt sie die Kapazität des Teams, sinken Qualität und Geschwindigkeit.

Kommunikationspfade
Die möglichen Eins-zu-eins-Verbindungen zwischen den Mitgliedern einer Gruppe. Eine Gruppe von n Personen hat n(n-1)/2 davon, die Zahl wächst also ungefähr mit dem Quadrat der Gruppengröße.

LCOM (mangelnde Kohäsion der Methoden)
Eine Familie von Metriken auf Klassenebene, die die Kohäsion danach schätzen, wie die Methoden einer Klasse deren Felder gemeinsam nutzen; ein hoher Wert deutet darauf hin, dass die Klasse mehrere voneinander unabhängige Verantwortlichkeiten trägt.

Leaky Abstraction
Eine Abstraktion, deren verborgene Details sich dennoch auf ihre Nutzer auswirken, zum Beispiel ein entfernter Aufruf, der wie ein lokaler aussieht, bis das Netzwerk langsam ist oder ausfällt. Von Joel Spolsky 2002 so benannt.

Liskov Substitution Principle (LSP)
Alles, was sich als Subtyp ausgibt, muss überall dort verwendbar sein, wo sein Supertyp erwartet wird, ohne dass der Aufrufer einen Unterschied im Verhalten bemerkt. Benannt nach Barbara Liskov.

Logisches Modul
Eine Verantwortungseinheit im Entwurf, etwa Bestellungen oder Zahlungen, unabhängig davon, wie ihr Code paketiert oder deployt wird.

Lokalität (der Connascence)
Wie nah die connascenten Elemente beieinanderliegen: innerhalb einer Funktion, eines Moduls, einer Komponente oder über ein Netzwerk hinweg. Starke Connascence ist nur dann tragbar, wenn sie lokal ist.

Magischer Wert
Eine literale Zahl oder Zeichenkette, deren Bedeutung im Code nicht angegeben ist, etwa ein Status 3 für „versandt“. Sie erzeugt Connascence of Meaning zwischen allem, was sie verwendet.

Mirroring Hypothesis
Die wissenschaftliche Fassung des Gesetzes von Conway: Die technischen Abhängigkeiten in einem Produkt entsprechen tendenziell den Kommunikations- und Organisationsbeziehungen zwischen den Menschen, die es bauen.

Modularität
Der Grad, in dem ein System in Teile gegliedert ist, die sich weitgehend für sich allein verstehen, ändern und testen lassen. Ein Teilmerkmal der Wartbarkeit in ISO/IEC 25010.

Nachbedingung
Eine Bedingung, die eine Operation bei ihrem Abschluss garantiert, sofern ihre Vorbedingung erfüllt war.

Open-Closed Principle (OCP)
Ein Modul sollte offen für Erweiterung, aber geschlossen für Änderung sein: Neues Verhalten sollte durch Hinzufügen von Code entstehen, etwa durch eine neue Implementierung einer Schnittstelle, nicht durch Bearbeiten funktionierenden Codes. Erstmals 1988 von Bertrand Meyer formuliert.

Physisches Modul
Eine Einheit, die gebaut, versioniert oder deployt wird, etwa eine JAR-Datei, ein Python-Paket, ein Container-Image oder ein Service. Ein logischer Entwurf lässt sich auf viele Arten in physische Module verpacken.

Reuse/Release Equivalence Principle (REP)
Die Einheit der Wiederverwendung ist die Einheit des Release: Klassen, die zu einer Komponente zusammengefasst sind, sollten sich sinnvoll gemeinsam releasen, versionieren und wiederverwenden lassen.

Schichtenbildung (Layering)
Die Anordnung von Modulen in Schichten, wobei jede Schicht nur die Schichten unter ihr nutzen darf. Strikte Schichtenbildung erlaubt nur die direkt darunterliegende Schicht; lockere Schichtenbildung erlaubt jede tiefere Schicht.

Semantische Kopplung
Die Abhängigkeit von gemeinsamer Bedeutung oder gemeinsamen Geschäftsregeln, die in keiner Schnittstelle sichtbar ist, etwa bei zwei Services, die sich einig sein müssen, was „aktiver Kunde“ bedeutet.

Single Responsibility Principle (SRP)
Ein Modul sollte einem einzigen Akteur verantwortlich sein: einer Gruppe von Menschen, die seine Änderungen anfordern. Code, der sich aus verschiedenen Gründen auf Wunsch verschiedener Personen ändert, sollte getrennt gehalten werden.

SOLID
Ein Merkwort für fünf von Robert C. Martin zusammengestellte Prinzipien des objektorientierten Entwurfs: Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation und Dependency Inversion.

Stable Abstractions Principle
Robert C. Martins Prinzip, dass eine Komponente so abstrakt sein sollte, wie sie stabil ist, damit stabile Komponenten trotzdem erweitert werden können.

Stable Dependencies Principle
Robert C. Martins Prinzip, dass eine Komponente nur von Komponenten abhängen sollte, die stabiler sind als sie selbst: Abhängigkeiten in Richtung der Stabilität.

Stärke (der Connascence)
Wie schwer eine Form der Connascence zu entdecken und zu refaktorisieren ist. Name ist die schwächste Form; dynamische Formen wie Timing, Value und Identity gehören zu den stärksten.

Statische Connascence
Connascence, die im Quellcode erkennbar ist, ohne ihn auszuführen, etwa die Übereinstimmung bei einem Namen, einem Typ, der Bedeutung eines Werts, der Reihenfolge von Parametern oder einem Algorithmus.

Tiefes Modul
Der Begriff von John Ousterhout für ein Modul, dessen einfache Schnittstelle viel Funktionalität verbirgt. Ein flaches Modul bietet für die Komplexität seiner Schnittstelle wenig Verhalten.

Verhaltensbasierte Subtypisierung

Die Regel hinter der Liskovschen Substitution: Ein Subtyp darf von Aufrufern nicht mehr verlangen (stärkere Vorbedingungen), nicht weniger versprechen (schwächere Nachbedingungen) und die Invarianten des Supertyps nicht verletzen.

Vorbedingung

Eine Bedingung, die vor dem Aufruf einer Operation erfüllt sein muss; sie zu erfüllen ist Pflicht des Aufrufers.

YAGNI

You aren’t gonna need it: eine Praxis aus Extreme Programming, keine Fähigkeiten für nur vermutete Bedürfnisse zu bauen, weil sie jetzt Aufwand und später Komplexität kosten.

Zeitliche Kopplung

In verteilten Systemen die Abhängigkeit davon, dass ein anderer Teil im selben Moment verfügbar ist und antwortet, wie bei einem synchronen Aufruf. Anderswo kann der Begriff eine vorgeschriebene Reihenfolge von Aufrufen bezeichnen oder (bei der Analyse der Versionskontrolle) Dateien, die sich gemeinsam ändern, was dieses Kapitel Change Coupling nennt.

Zerlegung

Das Aufteilen eines Systems in Teile (Module) und die Festlegung, wofür jeder Teil zuständig ist. Das Kriterium der Aufteilung ist wichtiger als die Zahl der Teile.

Zyklomatische Komplexität

Das 1976 von Thomas McCabe eingeführte Maß für die Zahl unabhängiger Pfade durch ein Stück Code, berechnet aus seinem Kontrollflussgraphen; bei einfachem Code entspricht es der Zahl der Entscheidungen plus eins.