BE SOFTWARE ARCHITECTURE WISE · KAPITEL 5

Architekturstile I: Monolithen und modulare Designs

Die meiste Software, darunter einige der größten und erfolgreichsten Produkte, läuft als eine einzige deploybare Anwendung. Dieses Kapitel erklärt, wie solche Anwendungen im Inneren strukturiert sein sollten. Es führt das Vokabular von Architekturstilen, Mustern, Taktiken und dem Architekturquantum ein, behandelt dann Schichtenarchitektur, den modularen Monolithen, Pipes and Filters, Microkernel- und Plug-in-Architektur, hexagonale, Onion- und Clean Architecture sowie strategisches Domain-Driven Design und endet mit der Frage, wann ein Monolith die richtige Antwort ist. Es richtet sich an Entwicklungsteams und Architekten, die Geschäftsanwendungen entwerfen oder warten und eine Struktur bewusst wählen und durchsetzen wollen.

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

AUF EINEN BLICK

Das lernen Sie

- Architekturstile, Muster, Taktiken und Referenzarchitekturen unterscheiden und erklären, warum die Randbedingungen eines Stils seine Qualitäten hervorbringen.
- Technische und fachliche Partitionierung vergleichen und vorhersagen, wie sich jede auf die Kosten einer typischen Änderung und auf die Organisation der Teams auswirkt.
- Die Architekturquanten in einem Entwurf erkennen und erklären, warum gemeinsam genutzte Datenbanken Services zu einem einzigen Quantum verbinden.
- Schichtenarchitektur, modularen Monolithen, Pipes and Filters und Microkernel beschreiben, ihre Merkmale bewerten und ihre Anti-Patterns erkennen.
- Hexagonale, Onion- oder Clean Architecture anwenden, um die Geschäftslogik unabhängig von Frameworks und Infrastruktur zu halten.
- Mit strategischem Domain-Driven Design Bounded Contexts finden und ihre Beziehungen in einer Context Map beschreiben.
- Strukturregeln im Code mit Architekturtests und Modulwerkzeugen wie import-linter, ArchUnit und Spring Modulith durchsetzen.
- Entscheiden, wann ein Monolith die richtige Wahl ist, wie man ihn skaliert und welche Anzeichen darauf hindeuten, dass es Zeit ist, Services herauszulösen.

Bevor Sie beginnen

Kapitel 3, insbesondere Modularität, Kopplung und Kohäsion sowie Abhängigkeiten und Schichtenbildung; Kapitel 4, insbesondere das C4-Modell. Die Qualitätsmerkmale aus Kapitel 2 werden durchgehend verwendet.

Dauer

Etwa 3 Stunden Lesezeit; 45 Minuten für den Test.

Schlüsselbegriffe

Architekturstil · Architekturmuster · Taktik · technische Partitionierung · fachliche Partitionierung · Architekturquantum · geschlossene und offene Schichten · Architecture Sinkhole · modularer Monolith · Pipes and Filters · Backpressure · Microkernel · Ports and Adapters · Bounded Context

Warum Stile wichtig sind

Die Kapitel 3 und 4 haben Ihnen die Prinzipien guter Struktur (Modularität, Kopplung und Kohäsion, Abhängigkeitsrichtung) und die Werkzeuge zu ihrer Beschreibung vermittelt. Dieses und das nächste Kapitel machen aus diesen Prinzipien die wiederkehrenden Formen, die reale Systeme annehmen. Ein Architekturstil ist ein benannter Satz von Randbedingungen dafür, welche Arten von Teilen ein System hat und wie sie verbunden sein dürfen. Auf die Randbedingungen kommt es an: Verbieten Sie den Teilen einer Pipeline, Zustand zu teilen, oder einem Geschäftsmodul, die Tabellen eines anderen anzufassen, und Sie erhalten

vorhersagbare Vorteile zu einem vorhersagbaren Preis. Wer die wichtigsten Stile kennt, kann von einer Form ausgehen, die sich für bestimmte Qualitäten bewährt hat, sie in einem Satz erklären und früh erkennen, wann ein System von ihr abdriftet.

Dieses Kapitel behandelt Stile, bei denen die gesamte Anwendung eine einzige Deployment-Einheit ist: ein Build, ein Release, meist eine Datenbank, und die Teile rufen einander im Arbeitsspeicher auf. Eine solche monolithische Architektur vermeidet die Netzwerkfehler, Teilausfälle und Konsistenzprobleme verteilter Systeme, denen sich Kapitel 6 widmet. Primitiv sind Monolithen deshalb nicht. Ob die Arbeit an einem Monolithen Freude oder Qual ist, hängt fast vollständig davon ab, wie er im Inneren aufgeteilt ist, und genau darum geht es in diesem Kapitel.

Wie das Kapitel zusammenhängt

Abschnitt 5.1 legt das Vokabular fest, das für jeden Stil in den Kapiteln 5 und 6 verwendet wird: Stile, Muster, Taktiken und Referenzarchitekturen; den Unterschied zwischen technischer Partitionierung (nach Schicht) und fachlicher Partitionierung (nach Geschäftsfähigkeit); monolithisches gegenüber verteiltem Deployment; und das Architekturquantum, die unabhängig deploybare Einheit, für die ein Satz von Qualitäten gilt. Er endet mit einer Übersichtstabelle aller Stile der beiden Kapitel.

Die Abschnitte 5.2 bis 5.5 beschreiben dann vier Stile in einheitlichem Format: Definition und Ursprung, ein Diagramm, Funktionsweise, ein Beispiel, eine Tabelle der Merkmale, wann man ihn einsetzt und wann nicht, Anti-Patterns und wie man ihn im Code durchsetzt. Die Schichtenarchitektur (5.2) ist der vertraute Stapel aus Präsentation, Geschäftslogik und Persistenz, mit geschlossenen und offenen Schichten und dem Sinkhole-Anti-Pattern. Der modulare Monolith (5.3) teilt eine deploybare Anwendung nach Fachdomänen auf, mit expliziten Modul-APIs und klarer Datenhoheit, veranschaulicht an den Erfahrungen von Shopify und einer Modulkarte einer Essenslieferplattform. Pipes and Filters (5.4) ordnet die Verarbeitung als Kette unabhängiger Schritte an, von Unix-Pipes bis zu Datenpipelines. Der Microkernel (5.5) baut einen minimalen Kern, den Plug-ins erweitern, wie bei IDEs, Browsern und Regel-Engines.

Die Abschnitte 5.6 und 5.7 handeln von der Struktur innerhalb von und zwischen Modulen statt vom Deployment. Hexagonale, Onion- und Clean Architecture (5.6) stellen die Geschäftslogik ins Zentrum und machen Datenbanken, Benutzeroberflächen und externe Dienste von ihr abhängig, nie umgekehrt; dadurch wird der Kern testbar und seine Infrastruktur austauschbar. Strategisches Domain-Driven Design (5.7) liefert die Methode, um gute Grenzen zu finden: Subdomänen, Bounded Contexts und ihre Beziehungen in einer Context Map. Schließlich fragt

Abschnitt 5.8, wann ein Monolith die richtige Antwort ist, wie weit er skalieren kann und welche Anzeichen sagen, dass es Zeit für eine Aufteilung ist, und bietet eine Entscheidungshilfe und eine Vergleichstabelle für die Stile des Kapitels.

Ein durchgehendes Beispiel

Wie in den früheren Kapiteln zieht sich eine Essenslieferplattform durch das ganze Kapitel: Kunden bestellen bei Restaurants, Kuriere liefern, Zahlungen laufen über einen externen Anbieter. In Abschnitt 5.1 wird sie anhand von vier Entwurfskandidaten eingeordnet; in 5.2 ist sie eine klassische Webanwendung in Schichtenarchitektur; in 5.3 wird sie zu acht Modulen mit einer Abhängigkeitsregel, die im Build geprüft wird; in 5.4 fließen ihre Bestellereignisse durch eine Pipeline in ein Data Warehouse, und den Engpass zu finden ist eine ausgearbeitete Rechnung. Spätere Abschnitte kehren zur selben Plattform zurück, am ausführlichsten in der Context Map von Abschnitt 5.7, damit sich die Stile an einem vertrauten System statt an zusammenhanglosen Spielzeugbeispielen vergleichen lassen.

Die Idee zum Mitnehmen

Wenn Sie sich nur eines aus diesem Kapitel merken, dann dies: **Die Grenzen innerhalb eines Systems sind wichtiger als die Grenzen zwischen Deployment-Einheiten**. Ein nach Domänen aufgeteilter Monolith mit expliziten Modul-APIs, einem Eigentümer für jede Tabelle und automatisch geprüften Abhängigkeitsregeln kann von mehreren Teams viele Jahre lang sicher verändert werden, und jedes Modul kann später zu einem Service werden, wenn es das wirklich braucht. Eine Gruppe von Services, die sich eine Datenbank teilen und gegenseitig ihre Interna aufrufen, ist immer noch ein einziges verworrenes System, nun zusätzlich mit Netzwerkfehlern. Stile sind Werkzeuge, um gute Grenzen zu schaffen und zu erhalten; keiner von ihnen ist ein Ziel an sich.

Daraus folgen zwei Gewohnheiten. Erstens: Wählen Sie den einfachsten Stil, dessen Merkmale die architekturrelevanten Anforderungen des Systems erfüllen, und halten Sie die Wahl und ihre Abwägungen in einem Entscheidungsdokument fest. Zweitens: Setzen Sie die Regeln des Stils mit Architekturtests im Build durch, denn jeder Stil erodiert, wenn seine Randbedingungen nur in den Köpfen der Beteiligten existieren.

Architekturstile, Muster und Taktiken

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.1 Funktionale und Qualitätsanforderungen
- 3.1 Modularität
- 3.7 Gesetz von Conway

Begriffe von früher, die er verwendet: Qualitätsmerkmal, Architekturrelevant, Konnektor, Kohäsion, Gesetz von Conway

DIE IDEE IN KÜRZE

Denken Sie an Haustypen: ein Reihenhaus, ein Bungalow, ein Hochhaus. Jeder Name verrät einem Bauunternehmen viel, bevor es überhaupt eine Zeichnung gibt: wie viele Stockwerke, wie die Räume verbunden sind, was es kostet, wo es gut funktioniert. Ein Hochhaus passt in eine dicht bebaute Stadt und ist in einem Dorf Verschwendung. Software hat ihre eigenen benannten Formen, die Architekturstile: eine Anwendung in Schichten, eine Reihe von Modulen in einem Programm, eine Kette von Verarbeitungsschritten, eine Gruppe getrennter Services. Wer die Stile kennt und weiß, was jeder gut und schlecht kann, kann als Team schnell eine sinnvolle Form wählen und in wenigen Worten darüber sprechen.

WAS SIE WISSEN MÜSSEN

- Ein Architekturstil ist ein benannter Satz von Elementtypen, Konnektortypen und Randbedingungen dafür, wie sie kombiniert werden (Garlan und Shaw); die Randbedingungen geben einem Stil seine vorhersagbaren Qualitäten.
- Ein Architekturmuster in der Tradition von *Pattern-Oriented Software Architecture* (Buschmann et al., 1996) ist eine bewährte Lösung für ein wiederkehrendes Strukturproblem in einem bestimmten Kontext; eine Taktik ist eine einzelne Entwurfsentscheidung, die auf ein Qualitätsmerkmal zielt; eine Referenzarchitektur ist eine Vorlage für eine Domäne.
- Technische Partitionierung teilt ein System nach technischer Rolle auf (Präsentation, Logik, Daten); fachliche Partitionierung teilt es nach Geschäftsfähigkeit auf (Bestellungen, Zahlungen, Lieferung). Die Wahl entscheidet, wie weit sich eine typische Feature-Änderung ausbreitet und wie sich Teams organisieren lassen.

- Ein monolithischer Stil wird als eine Einheit deployt; ein verteilter Stil wird als mehrere Einheiten deployt, die über ein Netzwerk zusammenarbeiten, und erbt alle Kosten des Netzwerks.
- Ein Architekturquantum ist ein unabhängig deploybarer Teil mit hoher funktionaler Kohäsion, Daten eingeschlossen; pro Quantum gilt ein Satz von Architekturmerkmalen, deshalb müssen Teile mit sehr unterschiedlichen Anforderungen getrennte Quanten sein.
- Jeder Stil ist eine Abwägung mit typischen Stärken und Schwächen; reale Systeme kombinieren Stile, und die Bewertungen in diesem Kurs sind relative Einschätzungen für typische Umsetzungen, keine Gesetze.

Warum Architekten Formen benennen

Die meisten Systeme werden nicht aus dem Nichts entworfen. Erfahrene Architekten erkennen, dass ein neues Problem einem bereits bekannten ähnelt, und greifen zu einer Form, die sich bewährt hat: ein paar Schichten für eine unkomplizierte Geschäftsanwendung, eine Kette von Verarbeitungsschritten für einen Datenfeed, ein Kern mit Plug-ins für ein Produkt, das Kunden erweitern. Diese Formen zu benennen gibt einem Team ein gemeinsames Vokabular. Der Satz „es ist ein modularer Monolith mit ereignisbasierter Kommunikation zwischen den Modulen“ vermittelt in wenigen Worten, wofür sonst eine Seite voller Diagramme nötig wäre, und er bringt ein ganzes Wissen darüber mit, was typischerweise gut und was schiefgeht.

David Garlan und Mary Shaw stellten dies in *An Introduction to Software Architecture* (technischer Bericht der Carnegie Mellon University, 1994) und in ihrem Buch *Software Architecture: Perspectives on an Emerging Discipline* (1996) auf ein solides Fundament. Sie beschrieben einen Architekturstil als Vokabular von Komponenten- und Konnektortypen zusammen mit einem Satz von Randbedingungen dafür, wie sie kombiniert werden dürfen. Pipes and Filters hat zum Beispiel eine Art von Komponente (einen Filter, der Daten umwandelt) und eine Art von Konnektor (eine Pipe, die sie in eine Richtung transportiert), und es verlangt, dass Filter keinen Zustand teilen und ihre Nachbarn nicht kennen. Roy Fielding, dessen Dissertation (2000) REST als Stil definierte, brachte es noch schärfer auf den Punkt: Ein Stil ist ein abgestimmter Satz von Randbedingungen, und die Randbedingungen erzeugen die Eigenschaften. Verbieten Sie Filtern, Zustand zu teilen, und Sie erhalten Filter, die sich unabhängig testen, wiederverwenden und neu anordnen lassen; das ist kein Glück, sondern eine Folge der Regel.

So liest man jeden Stil in diesem und im nächsten Kapitel am besten. Ein Stil ist kein Bild zum Abzeichnen. Er ist ein Satz von Regeln, denen man zu folgen zustimmt, gewählt, weil ihre Einhaltung bestimmte Architekturmerkmale einbringt, die Qualitätsmerkmale aus Kapitel 2, zu einem Preis. Ein Team, das das Vokabular eines Stils übernimmt, aber nicht seine Regeln, zahlt den Preis ohne den Nutzen.

Stile, Muster, Taktiken und Referenzarchitekturen

Für wiederverwendbares Entwurfswissen werden vier verwandte Begriffe verwendet, oft recht locker. Sie unterscheiden sich vor allem in ihrer Reichweite und darin, wie viel sie vorschreiben.

Ein Architekturmuster beschreibt eine bewährte Lösung für ein wiederkehrendes Strukturproblem in einem bestimmten Kontext. Die Idee der Muster stammt vom Gebäudearchitekten Christopher Alexander und kam über die Entwurfsmuster von Gamma, Helm, Johnson und Vlissides (1994), die auf der Ebene weniger Klassen arbeiten, in die Software. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns* von Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal (1996), meist POSA genannt, katalogisierte Muster auf der Ebene ganzer Systeme: Layers, Pipes and Filters und Blackboard (gruppiert als Weg vom Chaos zur Struktur, „From Mud to Structure"), Broker für verteilte Systeme, Model-View-Controller und Presentation-Abstraction-Control für interaktive Systeme sowie Microkernel und Reflection für anpassbare Systeme. Jedes POSA-Muster ist als Kontext, Problem, Kräfte und Lösung mit Konsequenzen beschrieben, was die Abwägungen explizit macht.

Eine Taktik im Sinne von Len Bass, Paul Clements und Rick Kazman in *Software Architecture in Practice* ist kleiner: eine Entwurfsentscheidung, die eine Reaktion eines Qualitätsmerkmals beeinflusst, etwa ein Heartbeat, um einen ausgefallenen Server zu erkennen, oder ein Cache, um die Latenz zu senken. Abschnitt 2.6 verwendete Änderbarkeitstaktiken wie Kapselung und den Einsatz eines Vermittlers. Muster bündeln mehrere Taktiken: Das Layers-Muster kombiniert Kapselung, eingeschränkte Abhängigkeiten und den Einsatz eines Vermittlers.

Eine Referenzarchitektur ist eine Vorlage für eine ganze Domäne oder Plattform: Sie benennt die typischen Bausteine und wie sie verbunden sind, sodass Projekte sie anpassen, statt mit einem leeren Blatt zu beginnen. AUTOSAR, die geschichtete Softwarearchitektur für Steuergeräte in Autos, ist ein bekanntes Beispiel; die Architecture Center der Cloud-Anbieter bieten viele kleinere. Eine Referenzarchitektur besteht meist aus mehreren Stilen und Mustern zugleich.

Fachleute sind sich nicht einig, wo die Grenze zwischen Stil und Muster verläuft. POSA nennt Layers ein Muster; Garlan und Shaw nennen Schichtensysteme einen Stil; Mark Richards und Neal Ford reservieren in *Fundamentals of Software Architecture* „Stil" für die Gesamttopologie eines Systems und „Muster" für wiederverwendbare Strukturen auf tieferer Ebene, die in vielen Stilen vorkommen können. Dieser Kurs folgt dem üblichen Sprachgebrauch: **Stil** für die Gesamtform eines Systems oder eines großen Teils davon, **Muster** für eine wiederverwendbare Lösung, die auf jeder Ebene sitzen kann. Verwenden Sie keine Energie auf die Grenze; verwenden Sie sie auf die Randbedingungen und ihre Folgen.

Vier Arten wiederverwendbaren Entwurfswissens

Begriff	Bereich	Beispiel	Beantwortete Frage
Architekturstil	Ganzes System oder ein großer Teil	Schichten, Pipes and Filters, Microservices	Welche Arten von Teilen und Verbindungen sind erlaubt, und nach welchen Regeln?
Architekturmuster	Jede Strukturebene	Broker, Model-View-Controller, Microkernel	Was ist eine bewährte Lösung für dieses wiederkehrende Strukturproblem, und was kostet sie?
Taktik	Eine Entwurfsentscheidung	Heartbeat, Cache, Kapselung, Wiederholung mit Back-off	Welche einzelne Entscheidung bewegt dieses Qualitätsmerkmal in die Richtung, die wir brauchen?
Referenzarchitektur	Eine Domäne oder Plattform	AUTOSAR für Fahrzeugelektronik; veröffentlichte Architekturen der Cloud-Anbieter	Was enthalten Systeme dieser Art üblicherweise, und wie ist es üblicherweise angeordnet?

Zwei Arten, ein System zu schneiden: technische oder fachliche Partitionierung

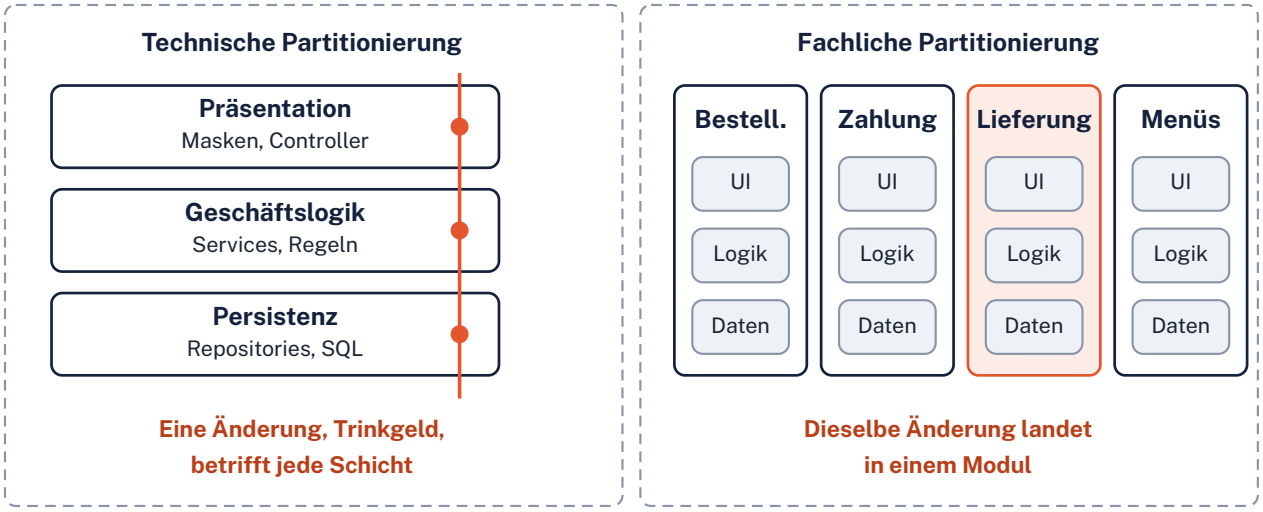
Die erste Strukturentscheidung in jedem Stil ist, wie das System auf oberster Ebene aufgeteilt wird. Es gibt zwei grundlegende Antworten. Technische Partitionierung gruppiert Code nach technischer Rolle: alles zur Benutzeroberfläche an einem Ort, alle Geschäftsregeln an einem anderen, der gesamte Datenbankzugriff an einem dritten. Fachliche Partitionierung gruppiert Code nach Geschäftsfähigkeit: alles zu Bestellungen an einem Ort, Zahlungen an einem anderen, Lieferung an einem dritten, jeweils mit der Benutzeroberfläche, Logik und dem Datenzugriff, die dafür nötig sind.

Der Unterschied zeigt sich, sobald sich ein Feature ändert. Angenommen, ein Essenslieferunternehmen lässt Kunden ein Trinkgeld für den Kurier hinzufügen. In einem technisch partitionierten System braucht diese Änderung ein neues Feld auf einer Bildschirmmaske (Präsentation), eine neue Regel für Trinkgeldgrenzen und Auszahlungen (Geschäftslogik) sowie eine neue Spalte und Abfrage (Persistenz): drei Teile, oft im Besitz von drei verschiedenen Personen oder Teams. In einem fachlich partitionierten System landet dieselbe Änderung fast vollständig im Liefermodul. Die meisten fachlichen Änderungswünsche sind in Begriffen der Domäne formuliert, daher hält fachliche Partitionierung eine Änderung meist an einem Ort. Technische Partitionierung hält eine *technische* Änderung an einem Ort, etwa den Austausch der Datenbankzugriffsbibliothek, aber solche Änderungen sind seltener.

Die Wahl prägt auch die Organisation. Nach dem Gesetz von Conway wird ein technisch partitioniertes System meist von einem Frontend-Team, einem Backend-Team und einem Datenbankteam gebaut, und jedes Feature braucht eine Übergabe zwischen ihnen. Ein fachlich partitioniertes System kann von Teams gebaut werden, die jeweils eine Geschäftsfähigkeit durchgängig verantworten. Martin Fowler gab 2015 dieselbe Empfehlung: Präsentations-, Domänen- und Datenschicht sind eine gute Art, ein kleines Stück Software zu organisieren, aber die oberste Ebene eines größeren Systems sollte in domänenorientierte Module aufgeteilt werden, jedes intern in Schichten, wenn das hilft.

Keines von beiden ist falsch. Technische Partitionierung ist einfach, vertraut und wird von der Standard-Ordnerstruktur fast jedes Web-Frameworks unterstützt, und sie passt zu kleinen Systemen, in denen ein Team alles macht. Fachliche Partitionierung erfordert vorab mehr Überlegung, weil jemand gute Domänengrenzen finden muss (das Thema von Abschnitt 5.7), aber sie skaliert besser mit der Größe des Systems und der Zahl der Teams.

Dasselbe Feature bei zwei Partitionierungen



Bei technischer Partitionierung durchquert ein typisches Feature jede Schicht; bei fachlicher Partitionierung landet es in einem Modul, das intern trotzdem in Schichten aufgebaut sein kann.

Monolithische und verteilte Stile

Die zweite grundlegende Unterscheidung betrifft das Deployment. Eine Deployment-Einheit ist das, was als ein Stück gebaut, released und gestartet wird: eine einzelne ausführbare Datei, ein Container-Image, ein Webarchiv. In einer monolithischen Architektur ist die ganze Anwendung eine Deployment-Einheit, meist mit einer Datenbank. Sie kann als viele identische Kopien hinter einem Load Balancer laufen, aber jede Kopie enthält den gesamten Code, und jede Änderung auszuliefern heißt, alles auszuliefern. In einer verteilten Architektur besteht das System aus mehreren Deployment-Einheiten, die zusammenarbeiten, indem sie Nachrichten über ein Netzwerk senden, und jede kann separat released und skaliert werden.

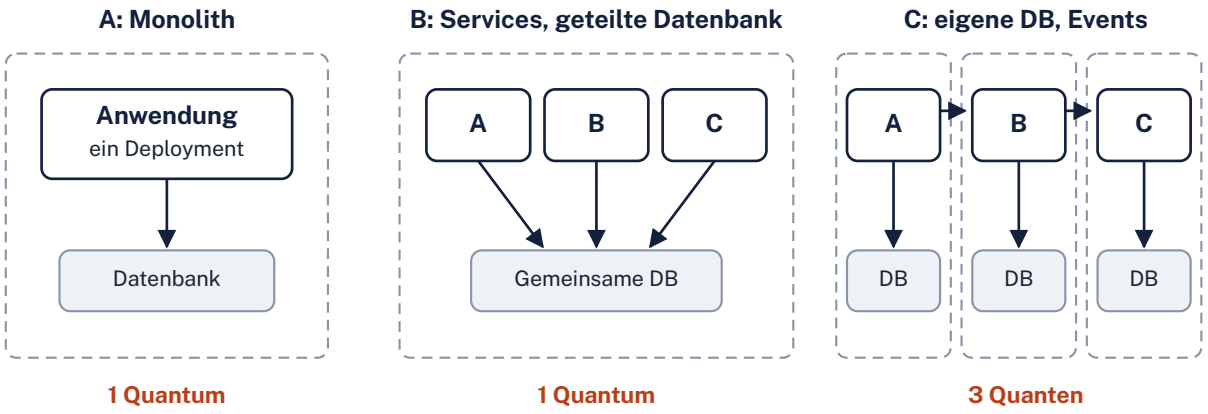
Zwei Klarstellungen vermeiden viel Verwirrung. Erstens beschreibt „Monolith“ das Deployment, nicht die Qualität. Ein Monolith kann wunderbar modular sein oder ein Big Ball of Mud (der Name, den Brian Foote und Joseph Yoder einem System ohne erkennbare Struktur gaben), und dasselbe gilt für ein verteiltes System: Viele Teams haben gelernt, dass die Aufteilung eines Big Ball of Mud in Services einen verteilten Big Ball of Mud ergibt, und der ist schlimmer. Zweitens ist Verteilung kein kostenloses Upgrade. Jeder Aufruf über ein Netzwerk kann langsam sein, auf halbem Weg scheitern oder doppelt ankommen, Daten lassen sich nicht mehr in einer lokalen Transaktion ändern, und das System braucht Monitoring, Deployment-Automatisierung und Rufbereitschaftskompetenz, ohne die ein Monolith auskommt. Kapitel 6 behandelt diese Kosten im Detail. Dieses Kapitel behandelt die Stile, die sie vermeiden: Alles läuft in einer Deployment-Einheit, und die Architektur steckt darin, wie der Code darin organisiert ist.

Das Architekturquantum

Monolithisch und verteilt ist ein nützlicher erster Schnitt, aber zu grob für echte Entscheidungen. Neal Ford, Rebecca Parsons und Patrick Kua führten in *Building Evolutionary Architectures* (2017) eine schärfere Idee ein, die Ford, Richards, Sadalage und Dehghani in *Software Architecture: The Hard Parts* (2021) verfeinerten: das Architekturquantum. Ein Quantum ist ein Teil des Systems, der unabhängig deployt werden kann, hohe funktionale Kohäsion hat (er erledigt eine zusammenhängende Aufgabe für das Geschäft) und alles enthält, was er zum Funktionieren braucht, insbesondere seine Datenbank. Die spätere Definition ergänzt, dass die Teile eines Quantums statisch gekoppelt sind (sie teilen Code, Schemas oder Infrastruktur, die gemeinsam deployt werden müssen) und dass synchrone Aufrufe Quanten für die Dauer des Aufrufs betrieblich aneinander binden.

Warum ist das wichtig? Weil Architekturmerkmale pro Quantum gelten. Ein Quantum wird als Ganzes skaliert, deployt, abgesichert und wiederhergestellt, also hat es ein Niveau an Skalierbarkeit, ein Niveau an Verfügbarkeit, einen Release-Rhythmus. Brauchen zwei Teile eines Systems sehr unterschiedliche Merkmale, etwa ein Kurier-Standortfeed, der Tausende Aktualisierungen pro Sekunde verkraften muss, und ein monatlicher Rechnungslauf für Restaurants, dann bekommen sie diese nur, wenn sie in verschiedenen Quanten liegen. Umgekehrt entstehen durch die Aufteilung von Code in mehrere Services keine mehreren Quanten, wenn sie sich weiterhin eine Datenbank teilen: Keiner von ihnen kann das Schema ändern oder allein wiederhergestellt werden, also stehen und fallen sie gemeinsam. Ein Monolith ist per Definition ein Quantum.

Quanten zählen



Gestrichelte Grenzen markieren jedes Quantum: was samt Daten eigenständig deployt und betrieben wird

Eine gemeinsame Datenbank bindet Services zu einem Quantum; getrennte Quanten brauchen getrenntes Deployment und getrennte Daten.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: vier Entwürfe einordnen

Ein Essenslieferunternehmen prüft vier Entwurfskandidaten für seine Plattform. Für jeden stellt die Architektin drei Fragen: Wie ist die oberste Ebene partitioniert, wie viele Deployment-Einheiten gibt es, und wie viele Quanten?

Entwurf 1: eine Django-Anwendung mit den Ordnern `views`, `services` und `models`, eine PostgreSQL-Datenbank	Technische Partitionierung; eine Deployment-Einheit; ein Quantum. Einfach, aber jedes Feature durchquert alle drei Ordner.
Entwurf 2: eine Anwendung mit den Paketen `orders`, `payments`, `delivery` und `menus`, jedes mit eigenen Tabellen, ein Datenbankserver	Fachliche Partitionierung; eine Deployment-Einheit; ein Quantum. Änderungen bleiben lokal, und die Module ließen sich später trennen.
Entwurf 3: vier Services (Bestellungen, Zahlungen, Lieferung, Speisekarten), die alle dieselben Datenbanktabellen lesen und schreiben	Fachliche Partitionierung dem Namen nach; vier Deployment-Einheiten, aber ein Quantum, weil eine Schemaänderung oder ein Datenbankausfall alle vier betrifft. Alle Kosten der Verteilung, kaum ihre Vorteile.
Entwurf 4: dieselben vier Services, jeder mit eigener Datenbank, die Events über einen Message Broker austauschen	Fachliche Partitionierung; vier Deployment-Einheiten; vier Quanten. Jeder lässt sich unabhängig skalieren und releasen, um den Preis von Netzwerkfehlern und Eventual Consistency zwischen ihnen.

Die Entwürfe 1 und 2 kosten im Betrieb gleich viel, aber Entwurf 2 macht typische Änderungen billiger und hält künftige Optionen offen.

Entwurf 3 ist der Warnfall: Die Zahl der Deployment-Einheiten sagt wenig; die Zahl der Quanten, die von Daten und Kopplung abhängt, sagt, was sich wirklich unabhängig weiterentwickeln kann.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: ein Quantum, ein Satz von Qualitäten

Der Punkt, an dem hier die meisten stolpern, ist, dass Architekturmerkmale zu einem ganzen Architekturquantum gehören, nicht zu jedem einzelnen Feature darin. Ein Quantum wird als Ganzes deployt, skaliert und wiederhergestellt, daher teilt alles darin ein Niveau an Skalierbarkeit, ein Niveau an Verfügbarkeit und einen Release-Rhythmus. Verfolgen Sie die beiden gegensätzlichen Features aus diesem Abschnitt durch einen Monolithen.

- Die Essenslieferplattform läuft als eine Deployment-Einheit mit einer Datenbank. Zwei ihrer Features brauchen sehr Unterschiedliches: ein Kurier-Standortfeed, der Tausende Aktualisierungen pro Sekunde verkraften muss, und ein monatlicher Rechnungslauf für Restaurants.
- Zur Abendessenszeit ist der Standortfeed überlastet. Kapazität lässt sich nur hinzufügen, indem mehr Kopien der ganzen Anwendung laufen, und jede Kopie trägt auch den Rechnungscode mit, der überhaupt keine zusätzliche Kapazität braucht.
- Hat der Rechnungslauf einen Bug, der den Arbeitsspeicher des Prozesses aufbraucht, kann er auch den Standortfeed stoppen, weil beide im selben Prozess laufen.
- Braucht der Rechnungscode eine kleine Korrektur, kann sie nicht allein ausgeliefert werden: Die ganze Anwendung wird gebaut, getestet und neu deployt, Standortfeed eingeschlossen.
- Damit der Feed seine eigene Skalierung, seine eigenen Ausfälle und seine eigenen Releases bekommt, muss er ein separates Quantum werden, eigenständig deployt mit eigenen Daten. Seinen Code in einen separaten Service zu verschieben reicht nicht, wenn er sich weiterhin die Datenbank teilt, denn dann kann keiner der Teile das Schema ändern oder allein wiederhergestellt werden.
- **Die Lehre:** Fragen Sie nicht, wie viele Services ein Entwurf hat; fragen Sie, welche Teile sehr unterschiedliche Merkmale brauchen. Diese Teile brauchen getrennte Quanten, und alles andere kann in einem zusammenbleiben.

Wie Sie die folgenden Stilabschnitte lesen

Die Abschnitte 5.2 bis 5.5 und der größte Teil von Kapitel 6 beschreiben jeweils einen Stil, immer in derselben Reihenfolge, damit sich die Stile vergleichen lassen: eine Definition und die Herkunft des Stils; ein Diagramm; die Funktionsweise; ein konkretes Beispielsystem; eine Tabelle, die seine wichtigsten Merkmale bewertet; wann man ihn einsetzt und wann nicht; häufige Anti-Patterns; und wie man ihn im Code durchsetzt, denn ein Stil, der nicht geprüft wird, erodiert, wie Abschnitt 1.7 beschrieben hat.

Die Merkmalsbewertungen (Niedrig, Mittel, Hoch, jeweils mit Begründung) folgen dem Ansatz der Stilvergleiche von Richards und Ford, sind aber die eigenen Einschätzungen dieses Kurses. Lesen Sie sie als relativ und typisch: Sie vergleichen Stile so, wie sie üblicherweise gebaut werden, von einem kompetenten Team, für ein typisches Geschäftssystem. Ein diszipliniertes Team kann einen Schichtenmonolithen bauen, der viele Male am Tag deployt wird, und ein nachlässiges Microservices, die sich überhaupt nicht deployen lassen. Die Bewertungen zeigen Ihnen, wo der Stil hilft und wo Sie gegen ihn arbeiten müssen.

Zwei Abschnitte in diesem Kapitel behandeln keine Deployment-Stile. Hexagonale, Onion- und Clean Architecture (Abschnitt 5.6) beschreiben, wie man das Innere einer Anwendung oder eines Service so organisiert, dass die Geschäftslogik nicht von der Technologie abhängt. Domain-Driven Design (Abschnitt 5.7) ist ein Weg, die Domänengrenzen zu finden, auf die modulare Monolithen und Microservices angewiesen sind. Beide gelten gleichermaßen innerhalb eines Monolithen wie eines verteilten Systems.

Die Stile der Kapitel 5 und 6 auf einen Blick

Stil	Hauptpartitionierung	Deployment	Typischer Einsatz
Schichten (5.2)	Technisch	Monolithisch	Kleine bis mittlere Geschäftsanwendungen, gebaut von einem Team
Modularer Monolith (5.3)	Fachlich	Monolithisch	Wachsende Produkte mit mehreren Teams, die ein Deployment wollen
Pipes and Filters (5.4)	Technisch (Verarbeitungsschritte)	Monolithisch oder verteilt	Datenverarbeitung: ETL, Medien, Logs, Compiler
Microkernel (5.5)	Technischer Kern, Plug-ins oft fachlich	Meist monolithisch	Produkte, die Kunden oder Dritte erweitern; Regel-Engines
Hexagonal, Onion, Clean (5.6)	Innere Struktur: Domäne im Zentrum	Beides	Jede Anwendung, deren Logik ihre Frameworks überdauern muss
Servicebasiert (6.2)	Fachlich	Verteilt, wenige grobe Services	Ein pragmatischer Schritt über den Monolithen hinaus, oft mit gemeinsamer Datenbank
Serviceorientierte Architektur (6.2)	Technisch (Orchestrierung, Enterprise-Services)	Verteilt	Unternehmensintegration, heute weitgehend historisch
Microservices (6.3)	Fachlich	Verteilt, viele feingranulare Services	Viele Teams, die im großen Maßstab unabhängig releasen müssen
Ereignisgesteuert (6.4)	Technisch (Ereignisverarbeitung)	Verteilt	Hochreaktive, entkoppelte Reaktionen auf Geschäftsereignisse
Serverless und Space-Based (6.8)	Unterschiedlich	Verteilt	Spitzenlasten oder extreme Last; Pay-per-Use-Workloads

Stile lassen sich kombinieren, und sie sind keine Ziele

Reale Systeme sind Mischformen. Ein modularer Monolith kann innerhalb jedes Moduls in Schichten aufgebaut sein; eines seiner Module kann einen nächtlichen Pipes-and-Filters-Job ausführen; ein Microservice kann im hexagonalen Stil gebaut sein; ein ereignisgesteuertes System kann einen Microkernel nutzen, damit Kunden eigene Regeln hinzufügen können. Den Stil jedes Teils zu benennen ist trotzdem nützlich, weil jeder Teil dann einen bekannten Satz von Regeln und Abwägungen erbt.

Der häufige Fehler ist, einen Stil als Ziel an sich zu behandeln: „Wir stellen auf Microservices um“ als Strategie statt als Mittel zu einer benannten Qualität wie unabhängigem Deployment für zwölf Teams. Gehen Sie von den architekturrelevanten Anforderungen aus Kapitel 2 aus, wählen Sie den einfachsten Stil, dessen Merkmale sie erfüllen, halten Sie die Wahl als Entscheidungsdokument fest und setzen Sie seine Regeln durch. Abschnitt 5.8 bietet eine Entscheidungshilfe für die Stile in diesem Kapitel.

ACHTUNG

Häufige Missverständnisse

- **Monolith heißt Altsystem.** Er heißt eine Deployment-Einheit. Viele erfolgreiche Produkte laufen als gut strukturierte Monolithen.
- **Mehr Services heißt modularer.** Bei Modularität geht es um Grenzen und Abhängigkeiten; Services, die sich eine Datenbank teilen, sind ein Quantum, egal wie viele es sind.
- **Ein Stil wird einmal gewählt.** Stile entwickeln sich: Eine Anwendung in Schichten kann zu einem modularen Monolithen werden und dann ein Modul an einen separaten Service abgeben, wenn dessen Anforderungen auseinanderlaufen.
- **Die Ordner des Frameworks zu verwenden heißt, einen Stil zu wählen.** Eine Standard-Ordnerstruktur ist eine technische Partitionierung, die die Autoren des Frameworks für kleine Anwendungen gewählt haben, keine Entscheidung über Ihr System.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Welche zwei Dinge definieren laut Garlan und Shaw einen Architekturstil?**

Ein Vokabular von Komponenten- und Konnektortypen und ein Satz von Randbedingungen dafür, wie sie kombiniert werden dürfen. Die Randbedingungen erzeugen die typischen Qualitäten des Stils.

▼ **Was ist der Unterschied zwischen einer Taktik und einem Muster?**

Eine Taktik ist eine Entwurfsentscheidung, die auf eine Reaktion eines Qualitätsmerkmals zielt, etwa ein Heartbeat oder ein Cache. Ein Muster ist eine größere, dokumentierte Lösung, die typischerweise mehrere Taktiken kombiniert.

▼ **Warum können vier Services, die sich eine Datenbank teilen, als ein einziges Architekturquantum zählen?**

Weil keiner von ihnen das gemeinsame Schema ändern, wiederhergestellt oder auf Datenebene skaliert werden kann, ohne die anderen zu beeinflussen; in der Praxis werden sie gemeinsam deployt und fallen gemeinsam aus, also teilen sie einen Satz betrieblicher Merkmale.

▼ **Welche Art von Änderung hält technische Partitionierung an einem Ort?**

Eine technische Änderung, etwa den Austausch der Datenbankzugriffsbibliothek oder des UI-Frameworks. Typische fachliche Änderungen ziehen sich dagegen durch alle technischen Schichten.

▼ **Probieren Sie es aus: Eine Ticketing-Plattform hat drei separat deployte Services, Booking, Payments und Reports, die alle eine gemeinsame Datenbank nutzen. Wie ist sie partitioniert, wie viele Deployment-Einheiten und wie viele Quanten hat sie? Warum könnte die hohe Last von Reports wichtig sein?**

Es ist nach Domäne partitioniert, mit drei Deployment-Einheiten, aber nur einem Quantum, weil die gemeinsame Datenbank sie zusammenbindet: Zu einem Quantum gehören seine Daten. Das System ist also nicht so unabhängig, wie es aussieht. Die schweren Abfragen von Reports können Booking am Vorverkaufstag ausbremsen, und für alle drei gilt ein einziger Satz von Qualitätsmerkmalen. Damit Reports andere Anforderungen haben kann, machen Sie es zu einem eigenen Quantum, etwa mit eigenem Reporting-Datenspeicher.

QUELLEN

- Garlan and Shaw: An Introduction to Software Architecture (SEI, 1994): https://www.sei.cmu.edu/library/an-introduction-to-software-architecture/
- Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures, chapter 1: https://ics.uci.edu/~fielding/pubs/dissertation/software_arch.htm
- Martin Fowler: Presentation Domain Data Layering: https://martinfowler.com/bliki/PresentationDomainDataLayering.html
- Microsoft Azure Architecture Center: Architecture styles: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/

Schichtenarchitektur

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 3.4 Abhängigkeiten
- 5.1 Stile und Muster

Begriffe von früher, die er verwendet: Technische Partitionierung, Deployment-Einheit, Architekturquantum, Gesetz von Conway

DIE IDEE IN KÜRZE

Denken Sie an ein Restaurant. Die Gäste sprechen nur mit dem Servicepersonal. Das Servicepersonal gibt Bestellungen an die Küche weiter und geht nie selbst ins Lager. Die Köche holen Zutaten beim Lagerpersonal, und nur das Lagerpersonal hat mit den Lieferanten zu tun. Jede Gruppe hat eine Aufgabe und spricht nur mit der Gruppe neben ihr, sodass Sie das Servicepersonal umschulen können, ohne die Küche anzurühren. Eine Anwendung in Schichtenarchitektur funktioniert genauso: Die Bildschirmmasken sprechen mit den Geschäftsregeln, die Geschäftsregeln mit dem Datenzugriffscode, und nur dieser Code spricht mit der Datenbank.

WAS SIE WISSEN MÜSSEN

- Eine Schichtenarchitektur ordnet Code in horizontalen Schichten nach technischer Rolle an, typischerweise Präsentation, Geschäftslogik, Persistenz und Datenbank, und lässt jede Schicht nur von Schichten unter ihr abhängen.
- Geschlossene Schichten dürfen nicht übersprungen werden, was Isolationsschichten ergibt: Eine Änderung innerhalb einer Schicht greift nicht auf andere über, solange ihre Schnittstelle hält. Eine offene Schicht, etwa eine Schicht gemeinsamer Dienste, darf umgangen werden.
- Schichtenbildung ist technische Partitionierung, daher betrifft ein typisches fachliches Feature jede Schicht; das verlangsamt Änderungen und legt Teamstrukturen nach Schichten nahe.
- Das Anti-Pattern Architecture Sinkhole bezeichnet Anfragen, die durch Schichten laufen, die sie nur weiterreichen; als Faustregel ist eine Minderheit solcher Anfragen normal, eine große Mehrheit deutet auf den falschen Stil hin.

- Der Stil schneidet bei Einfachheit und niedrigen Kosten gut ab und bei Deploybarkeit, Skalierbarkeit, Elastizität und Fehlertoleranz schlecht, weil das ganze System eine Deployment-Einheit ist.
- Schichtenregeln erodieren, wenn sie nicht automatisch durchgesetzt werden, zum Beispiel mit import-linter in Python oder ArchUnit in Java, als Tests, die den Build scheitern lassen.

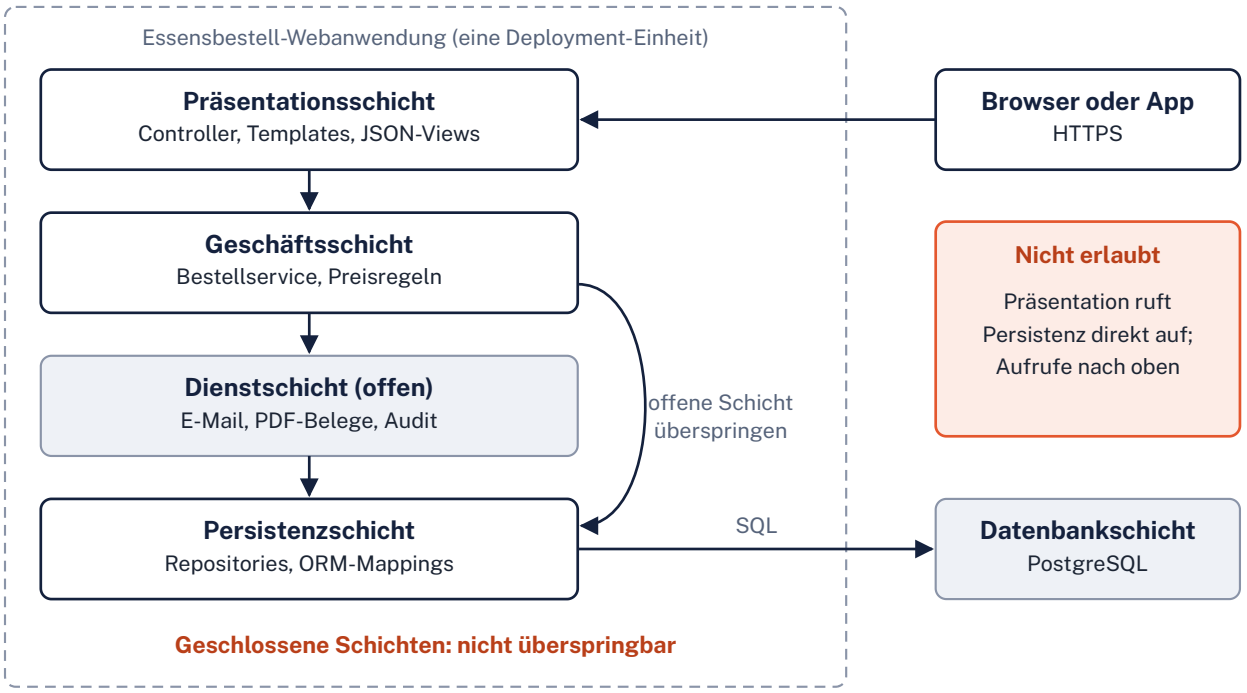
Definition und Ursprung

Eine Schichtenarchitektur gliedert eine Anwendung in horizontale Schichten, von denen jede eine technische Verantwortung hat, der Schicht darüber Dienste anbietet und die Dienste der Schicht darunter nutzt. Sie ist der am weitesten verbreitete Stil für Geschäftsanwendungen, oft eher aus Gewohnheit als aus einer Entscheidung heraus, weil die meisten Web-Frameworks ein Projekt mit Ordnern für Views, Geschäftslogik und Models erzeugen.

Die Idee ist alt und stammt aus der Systemsoftware. Edsger Dijkstra beschrieb das Multiprogramming-Betriebssystem THE (Communications of the ACM, 1968) als Hierarchie von Ebenen, von denen jede nur auf den darunterliegenden aufbaut, sodass sich jede Ebene für sich betrachten und testen ließ. Netzwerkprotokollstapel wie das OSI-Referenzmodell nutzen dasselbe Prinzip. In der Geschäftssoftware kam die Schichtenbildung mit Client-Server- und dreistufigen Systemen in den 1990er-Jahren an, und POSA (1996) katalogisierte sie als Layers-Muster. Garlan und Shaw führten Schichtensysteme unter ihren grundlegenden Stilen auf, mit der Randbedingung, dass Interaktionen auf benachbarte Schichten beschränkt sind.

Die typische Geschäftsanwendung hat vier Schichten. Die **Präsentationsschicht** kümmert sich um die Benutzeroberfläche und das Protokoll: HTTP-Anfragen, Bildschirmmasken, JSON. Die **Geschäftsschicht** enthält die Regeln und Anwendungsfälle: wie eine Bestellung bepreist wird, wann sie storniert werden darf. Die **Persistenzschicht** wandelt Geschäftsobjekte in gespeicherte Daten und zurück, meist über Repositories und einen objektrelationalen Mapper. Die **Datenbankschicht** ist die Datenbank selbst. Kleine Anwendungen legen Geschäftslogik und Persistenz manchmal zusammen; größere fügen Schichten hinzu, etwa eine Anwendungsschicht, die Anwendungsfälle über einer Domänenschicht mit dem Kernmodell koordiniert. Der Stil legt nicht die Anzahl fest, sondern die Richtung der Abhängigkeit.

Eine Essensbestell-Anwendung in Schichten



Aufrufe fließen nur nach unten; geschlossene Schichten können nicht übersprungen werden, die offene Dienstschicht dagegen schon.

Funktionsweise: geschlossene Schichten, offene Schichten und Isolation

Jede Schicht kann geschlossen oder offen sein. Eine Anfrage muss eine geschlossene Schicht durchlaufen: Die Präsentationsschicht darf nicht an der Geschäftsschicht vorbei in die Persistenzschicht greifen, auch wenn das schneller zu schreiben wäre. Eine offene Schicht darf umgangen werden. Das übliche Beispiel ist eine Schicht gemeinsamer technischer Dienste, etwa für den E-Mail-Versand, das Erzeugen von PDF-Belegen oder das Schreiben von Audit-Einträgen. Nur manche Geschäftsoperationen brauchen sie, daher würde es sinnlosen Weiterleitungscode erzeugen, jede Anfrage durch diese Schicht zu zwingen. Im Diagramm darf die Geschäftsschicht die Dienstschicht aufrufen oder direkt an ihr vorbei zur Persistenz gehen.

Geschlossene Schichten ergeben Isolationsschichten (Layers of Isolation). Weil die Präsentationsschicht nur die Schnittstelle der Geschäftsschicht kennt, kann das Team den objektrelationalen Mapper ersetzen, die Tabellen umstrukturieren oder von einem Datenbankprodukt zu einem anderen wechseln, ohne einen einzigen Controller oder ein Template anzurühren, solange die Schnittstelle der Geschäftsschicht gleich bleibt. Das ist Information Hiding (David Parnas, 1972), angewandt auf eine ganze Ebene des Codes, und es ist der Hauptnutzen, den der Stil bietet. Jede offene Schicht und jede Ausnahme, die einer Schicht erlaubt, eine andere zu überspringen, schwächt ihn: Fragen Controller die Datenbank direkt ab, greift eine Schemaänderung nun auch auf die Präsentationsschicht über.

Die Richtung der Abhängigkeit ist die andere Hälfte der Regel. Untere Schichten dürfen nichts von höheren wissen: Die Persistenzschicht darf nie einen Controller importieren, und die Geschäftsschicht darf nie HTML erzeugen. Muss eine untere Schicht eine höhere benachrichtigen, geschieht das über eine Abstraktion, die der unteren Schicht gehört, etwa eine Callback-Schnittstelle oder ein Event, sodass die Abhängigkeit im Code weiterhin nach unten zeigt. Abschnitt 3.4 behandelt Abhängigkeitsrichtung, Zyklen und strikte gegenüber gelockerter Schichtenbildung ausführlich.

Es hilft, Schichten und Tiers auseinanderzuhalten. Eine Schicht ist eine logische Gruppierung im Code. Ein Tier ist eine physische Grenze zur Laufzeit, etwa ein separater Server oder Prozess. Ein klassisches Three-Tier-System betreibt Browser, Anwendungsserver und Datenbankserver als drei Tiers, während der Anwendungsserver allein drei oder vier Schichten enthalten kann. Macht man aus einer Schicht ein Tier, liegt ein Netzwerk zwischen zwei Schichten, die einander vorher im Arbeitsspeicher aufgerufen haben, was die Kosten jedes Aufrufs vervielfacht; das lohnt sich selten um seiner selbst willen.

Beispiel: eine klassische Webanwendung

Nehmen wir die Bestellwebsite eines regionalen Essenslieferunternehmens: Kunden sehen sich Speisekarten an, geben Bestellungen auf und bezahlen; das Restaurantpersonal sieht eingehende Bestellungen. Ein Team aus fünf Personen baut sie mit einem verbreiteten Web-Framework und einer PostgreSQL-Datenbank. Eine Schichtenstruktur ist der naheliegende Ausgangspunkt, und die Codeaufteilung macht die Schichten sichtbar.

```
foodshop/
  presentation/          # HTTP only: parse requests, render responses
    order_views.py
    menu_views.py
    templates/
  business/              # rules and use cases; no HTTP, no SQL
    order_service.py
    pricing.py
  services/              # open layer: shared technical services
    email.py
    receipts.py
  persistence/           # all database access
    order_repository.py
    menu_repository.py
    models.py
  settings.py
```

Paketstruktur der Essensbestell-Anwendung in Schichten: ein Paket auf oberster Ebene pro Schicht.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine Anfrage durch die Schichten verfolgen

Ein Kunde drückt auf **Bestellen**. Die Anfrage kommt als `POST /orders` mit dem Warenkorb und der Lieferadresse an.

Präsentation	`order_views.place_order` parst das JSON, prüft, ob es wohlgeformt ist und ob der Benutzer angemeldet ist, und ruft `order_service.place(customer_id, items, address)` auf. Es weiß nichts über Tabellen oder Preise.
Geschäftslogik	`order_service.place` prüft, ob das Restaurant geöffnet hat, bepreist den Warenkorb mit `pricing.py` (Rabatte, Liefergebühr, Steuer) und bittet `order_repository`, die Bestellung zu speichern. Es ruft `receipts.send` in der offenen Dienstschicht auf, um einen Beleg per E-Mail zu senden.
Persistenz	`order_repository.save` bildet die Bestellung auf Zeilen in `orders` und `order_lines` ab und schreibt sie in einer Datenbanktransaktion.
Datenbank	PostgreSQL committet die Transaktion und kehrt zurück. Das Ergebnis fließt nach oben zurück: Der Service gibt die neue Bestellung zurück, die View rendert `201 Created` mit der Bestellnummer.
Änderungswunsch	Das Produktmanagement wünscht sich ein Kurier-Trinkgeld. Die Entwicklerin fügt ein Trinkgeldfeld in Anfrage und Antwort hinzu (Präsentation), eine Regel für das maximale Trinkgeld und das Trinkgeld in der Gesamtsumme (Geschäftslogik) sowie eine Spalte und ein Mapping (Persistenz), dazu eine Migration (Datenbank). Vier Schichten für ein kleines Feature.

Jede Schicht erledigte eine Art von Aufgabe, und jede ließ sich allein testen, wobei die Schicht darunter durch ein Test Double ersetzt wurde.

Das Trinkgeld zeigt die Kosten technischer Partitionierung: Selbst eine kleine fachliche Änderung zieht sich durch jede Schicht, betrifft also mehr Dateien, mehr Reviewer und, wenn die Teams nach Schichten aufgeteilt sind, mehr Teams.

Die Folgen technischer Partitionierung

Weil Schichtenbildung technische Partitionierung ist, wird die Fachdomäne über die Schichten verschmiert. Es gibt keinen einzelnen Ort namens „Lieferung“: Es gibt Liefer-Controller, Liefer-Services und Liefer-Repositories, vermischt mit allem anderen in jeder Schicht. Wer verstehen will, wie Trinkgeld funktioniert, muss quer durch alle lesen. Mit dem Wachstum des Systems wird jede Schicht zu einer großen Sammlung unzusammenhängenden Codes mit geringer Kohäsion: Die Geschäftsschicht enthält Regeln für Bestellungen, Zahlungen, Speisekarten und

Kuriere nebeneinander, und nichts hindert den Zahlungscode daran, tief in den Speisekartencode hineinzurufen. Mit der Zeit bleiben die Schichten sauber, aber das Innere jeder Schicht wird zu einem Big Ball of Mud.

Es gibt auch organisatorische Kosten. Teams nach Schichten zu organisieren (ein Frontend-Team, ein Backend-Team, ein Datenbankteam) bedeutet, dass fast jedes Feature eine Übergabe und eine Warteschlange zwischen Teams braucht, was das Gesetz von Conway dann in der Architektur festschreibt. Martin Fowler bezeichnet nach Schichten organisierte Teams aus diesem Grund als Anti-Pattern: Die einzelnen Entwickler müssen nicht Full-Stack sein, die Teams aber schon. Eine Schichtenarchitektur zwingt Teams nicht, sich nach Schichten aufzuteilen, macht es aber zum Weg des geringsten Widerstands.

Schließlich ist ein Schichtenmonolith ein einziges Architekturquantum: eine Deployment-Einheit und meist eine Datenbank. Jede Änderung, so klein sie auch ist, bedeutet, die ganze Anwendung zu bauen, zu testen und neu zu deployen, und die ganze Anwendung skaliert und fällt als Einheit aus. Das ist oft völlig akzeptabel, setzt aber eine Obergrenze.

Ein alternativer Schnitt: Vertical-Slice-Architektur

Jimmy Bogards **Vertical-Slice-Architektur** (2018) organisiert Code um einzelne Anfragen statt um Schichten. Jedes Feature, etwa *Bestellung aufgeben* oder *Bestellung stornieren*, enthält seine eigene Verarbeitung, Validierung und seinen Datenzugriff vom Anfang der Anfrage bis zur Datenbank. Sein Ziel ist, „die Kopplung zwischen Slices zu minimieren und die Kopplung innerhalb eines Slice zu maximieren“, sodass eine Änderung ein Feature berührt statt jede Schicht. Der Preis ist die gemeinsame Struktur, die Schichten boten: Duplikate können sich ansammeln und Handler anschwellen, und Bogard sagt, dass der Ansatz ein Team voraussetzt, das Code Smells erkennt und refaktoriert. Slices lassen sich auch gut mit Modulen verbinden: Module teilen die Domänen auf, Slices die Features innerhalb eines Moduls.

Das Architecture Sinkhole und andere Anti-Patterns

Das Anti-Pattern Architecture Sinkhole, ein von Mark Richards bekannt gemachter Name, beschreibt Anfragen, die durch Schichten fließen, die nichts tun, als sie weiterzureichen. Eine Speisekarte anzusehen ist der typische Fall: Der Controller ruft eine Servicemethode auf, die eine Repository-Methode aufruft, die eine Abfrage ausführt, und Controller und Service fügen nichts hinzu. Jede durchreichende Schicht bringt Code mit sich, der geschrieben, getestet und gelesen werden muss, und ein wenig Latenz. Ein gewisses Maß an Durchreichen ist der Preis der Isolation und in Ordnung. Richards bietet eine Faustregel an: Ist etwa ein Fünftel der Anfragen einfaches Durchreichen, ist das normal; sind es die meisten, besteht die Anwendung

wahrscheinlich überwiegend aus Datenzugriff mit wenig Geschäftslogik, und die Schichtenbildung ist der falsche Stil oder die Schichten sollten geöffnet werden. Schichten zu öffnen gibt allerdings Isolation auf, also entscheiden Sie bewusst und halten Sie fest, warum.

Weitere häufige Anti-Patterns sollte man früh erkennen. **Undichte Schichten**: SQL in Controllern, HTTP-Anfrageobjekte, die an Services übergeben werden, oder ORM-Entitäten, die direkt als JSON gerendert werden, sodass eine Änderung in einer Schicht auf andere übergreift. **Abhängigkeiten nach oben**: ein Repository, das einen Service aufruft, oder ein Domänenobjekt, das HTML formatiert. **Die anämische Geschäftsschicht**: Services, die nur Daten umherschieben, während die eigentlichen Regeln in Controllern oder Stored Procedures leben; Fowler benannte 2003 das verwandte Anti-Pattern Anemic Domain Model. **Der Gott-Service**: ein `OrderService` mit mehreren tausend Zeilen, den jeder andere Service aufruft. **Die Utility-Schicht, von der alles abhängt**: ein `common`-Paket, das nach und nach Geschäftslogik ansammelt und alles mit allem koppelt.

Sinkhole oder nützliche Schichtenbildung



Durchreichende Anfragen kosten Code und Latenz, ohne Verhalten hinzuzufügen; einige sind normal, eine Mehrheit ist ein Warnsignal.

Merkmale des Schichtenstils

Merkmal	Bewertung	Begründung
Einfachheit	Hoch	Fast jedem Entwicklungsteam vertraut und die Standardstruktur der meisten Frameworks; leicht zu erklären und zu beginnen.
Gesamtkosten	Hoch (günstig)	Eine Codebasis, ein Deployment, eine Datenbank; minimale Infrastruktur und minimaler Betrieb.
Testbarkeit	Mittel	Jede Schicht lässt sich testen, indem die Schicht darunter durch ein Test Double ersetzt wird, aber realistische Tests brauchen oft den ganzen Stack und eine Datenbank.
Änderbarkeit bei fachlichen Änderungen	Niedrig bis mittel	Technische Änderungen bleiben innerhalb einer Schicht, aber die meisten Feature-Änderungen ziehen sich durch jede Schicht.
Deploybarkeit	Niedrig	Jede Änderung bedeutet, die ganze Anwendung neu zu deployen; das Release-Risiko wächst mit der Größe.
Skalierbarkeit und Elastizität	Niedrig	Die ganze Anwendung skaliert als eine Einheit; man kann nicht nur den ausgelasteten Teil skalieren, und der Start einer großen Anwendung bremst elastisches Skalieren.
Fehlertoleranz	Niedrig	Ein Speicherleck oder eine außer Kontrolle geratene Abfrage kann den ganzen Prozess lahmlegen; es gibt keine Isolation zwischen Features.
Performance	Mittel	Aufrufe im selben Prozess sind schnell, aber zusätzliche Schichten und Objekt-Mapping verursachen Overhead, und Sinkholes noch mehr.
Weiterentwickelbarkeit im großen Maßstab	Niedrig	Geringe Kohäsion innerhalb jeder Schicht und Kopplung über Features hinweg machen große Schichtensysteme schwer umzustrukturieren.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: eine Schicht, die scheinbar nichts tut

Der schwierigste Gedanke dieses Abschnitts ist die Abwägung hinter dem Architecture Sinkhole. Eine Schicht, die eine Anfrage nur weiterreicht, wirkt wie verschwendeter Code, und doch ist es die Regel, die Anfragen durch sie zwingt, die die Schichten isoliert hält. Verfolgen Sie zwei Anfragen durch die Essensbestell-Anwendung.

- Ein Kunde sieht sich eine Speisekarte an. Der Controller ruft eine Methode der Geschäftsschicht auf, die einfach das Speisekarten-Repository aufruft, das die Abfrage ausführt. Controller und Geschäftsmethode reichen die Anfrage nur weiter.
- Eine Entwicklerin schlägt vor, den Controller das Repository direkt aufrufen zu lassen. Das würde Code sparen, aber die Geschäftsschicht zu einer offenen Schicht machen: einer Schicht, die Anfragen überspringen dürfen.
- Monate später strukturiert das Team die Speisekartentabellen um. Über die Geschäftsschicht müsste sich der Controller nicht ändern, solange die Schnittstelle dieser Schicht gleich bleibt. Mit der Abkürzung erreicht die Änderung auch den Controller.
- Eine Bestellung aufzugeben ist anders: Der Controller prüft die Eingabe, die Geschäftsschicht bepreist den Warenkorb und prüft die Regeln, und das Repository speichert die Bestellung in einer Transaktion. Jede Schicht leistet echte Arbeit.
- Die nützliche Frage ist also, wie viele Anfragen der Speisekarte gleichen. Richards' Faustregel: Etwa ein Fünftel einfaches Durchreichen ist normal. Sind es die meisten, besteht die Anwendung wahrscheinlich überwiegend aus Datenzugriff mit wenigen Regeln. Dann passt ein anderer Stil womöglich besser, oder das Team öffnet bewusst einige Schichten, hält fest, warum, und nimmt den Verlust an Isolation in Kauf.
- **Die Lehre:** Ein gewisses Maß an Durchreichen ist der Preis von Isolationsschichten. Zählen Sie es, statt es sofort zu entfernen, und werten Sie eine große Mehrheit durchreichender Anfragen als Zeichen, dass der Stil nicht passt.

Wann sie gut passt und wann nicht

Die Schichtenarchitektur ist eine gute Wahl für kleine und mittlere Anwendungen, die ein Team baut, für Produkte, deren Domäne noch nicht gut verstanden ist (fachliche Partitionierung braucht Grenzen, die Sie noch nicht kennen), für knappe Budgets und Fristen und für Anwendungen, die überwiegend aus Formularen über Daten mit überschaubaren Regeln bestehen. Sie ist auch eine vernünftige innere Struktur für ein einzelnes Modul oder einen Service, der selbst Teil eines fachlich partitionierten Systems ist: Ein geschichtetes `delivery`-Modul innerhalb eines modularen Monolithen bekommt das Beste aus beiden Welten.

Als oberste Struktur eines großen Systems mit mehreren Teams ist sie eine schlechte Wahl, weil jedes Team in jeder Schicht arbeitet und Features kollidieren. Sie ist schlecht, wenn Teile des Systems sehr unterschiedliche Skalierbarkeit oder Verfügbarkeit brauchen, weil es nur ein Quantum gibt. Und sie ist schlecht, wenn die Geschäftslogik reichhaltig und zentral ist, weil die Regeln über Controller, Services und Stored Procedures verstreut werden; dort hält eine domänenzentrierte innere Struktur wie hexagonale oder Clean Architecture den Kern unabhängig von der Datenbank und dem Web-Framework.

Der häufigste Entwicklungspfad führt von einem Schichtenmonolithen zu einem modularen Monolithen: ein Deployment behalten, aber die oberste Ebene nach Domänen neu ordnen und die Schichten innerhalb jedes Moduls beibehalten.

Die Schichtenregeln im Code durchsetzen

Schichtenregeln verfallen schnell. Unter Termindruck importiert jemand das Repository direkt in einen Controller, weil das zwanzig Minuten spart, im Review fällt es niemandem auf, und der Nächste kopiert das Muster. Die Abwehr ist ein Architekturtest: eine automatisierte Prüfung von Abhängigkeiten, die im Build läuft und ihn scheitern lässt, wenn eine Regel verletzt wird, wodurch die Regel zu einer Fitness Function wird.

In Python liest import-linter Verträge aus einer Konfigurationsdatei. Sein `layers`-Vertrag listet die Schichten von der höchsten zur niedrigsten auf und verbietet jeder niedrigeren Schicht, eine höhere zu importieren. Für sich allein macht er die Schichten nicht geschlossen (eine höhere Schicht darf jede niedrigere importieren), daher fügt ein `forbidden`-Vertrag die Regel hinzu, dass die Präsentation die Persistenz nicht direkt importieren darf; dafür muss `allow_indirect_imports` eingeschaltet sein, weil sonst die normale Kette von der Präsentation über die Geschäftslogik zur Persistenz als Verstoß zählen würde. In Java drückt ArchUnit dieselben Regeln als Unit-Test aus: `layeredArchitecture()` deklariert die Schichten nach Paketen und legt zum Beispiel fest, dass auf die Persistenzschicht nur die Serviceschicht zugreifen darf. Ähnliche Werkzeuge gibt es für andere Ökosysteme, etwa dependency-cruiser für JavaScript und TypeScript.

Zwei Praktiken sorgen dafür, dass die Prüfungen wirken. Führen Sie sie mit den vorhandenen Verstößen als bekannte Ausnahmen ein, sodass der Build sofort grün ist und nur neue Verstöße ihn scheitern lassen; bauen Sie die Ausnahmen dann nach und nach ab. Und halten Sie die Konfiguration neben dem Code und prüfen Sie Änderungen daran wie Code, denn eine Regel zu lockern ist eine Architekturentscheidung, die ein Entscheidungsdokument verdient.

```
[importlinter]
root_package = foodshop

[importlinter:contract:layers]
name = Layers only depend downwards
type = layers
layers =
    foodshop.presentation
    foodshop.business
    foodshop.services
    foodshop.persistence

[importlinter:contract:closed-business]
name = Presentation must go through business
type = forbidden
source_modules = foodshop.presentation
forbidden_modules = foodshop.persistence
# presentation → business → persistence is the normal path
allow_indirect_imports = True
```

Eine import-linter-Konfiguration (.importlinter) für die Essensbestell-Anwendung: Schichten dürfen nur nach unten abhängen, und die Präsentation darf die Geschäftsschicht nicht überspringen.

VERTIEFUNG

Schichten in anderen Stilen

Schichtenbildung lebt in den meisten anderen Stilen weiter. Jedes Modul eines modularen Monolithen, jeder Microservice und jedes Plug-in kann intern in Schichten aufgebaut sein. Hexagonale und Clean Architecture lassen sich als Schichtenbildung mit einer entscheidenden Änderung lesen: Die Abhängigkeit zwischen Geschäftslogik und Persistenz ist umgekehrt, sodass der Datenbankcode von Schnittstellen abhängt, die die Geschäftsschicht definiert, statt umgekehrt. Diese Umkehrung lässt den Geschäftskern ohne Datenbank testbar sein und seine Frameworks überdauern.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Was ist der Unterschied zwischen einer geschlossenen und einer offenen Schicht?**

Eine Anfrage muss eine geschlossene Schicht durchlaufen und kann sie nicht überspringen; eine offene Schicht darf von der Schicht darüber umgangen werden, was typisch für eine Schicht gemeinsamer Dienste ist, die nur manche Anfragen brauchen.

▼ Warum braucht eine Anwendung in Schichtenarchitektur für ein neues Feature meist Änderungen in jeder Schicht?

Weil sie technisch partitioniert ist: Der Code jeder Geschäftsfähigkeit ist auf Präsentations-, Geschäfts- und Persistenzschicht verteilt, und die meisten Features sind in fachlichen Begriffen definiert.

▼ Was ist der Unterschied zwischen einer Schicht und einem Tier?

Eine Schicht ist eine logische Gruppierung von Code nach Verantwortung; ein Tier ist eine physische Laufzeitgrenze wie ein separater Server. Mehrere Schichten laufen meist innerhalb eines Tiers.

▼ Warum macht ein layers-Vertrag von import-linter die Schichten für sich allein nicht geschlossen?

Er verbietet nur niedrigeren Schichten, höhere zu importieren; eine höhere Schicht darf weiterhin jede niedrigere importieren, daher braucht es einen separaten forbidden-Vertrag, der verhindert, dass die Präsentation die Persistenz importiert.

▼ Probieren Sie es aus: Sie verfolgen 500 Anfragen durch eine geschichtete Ticketing-App, und 400 laufen ohne zusätzliche Logik direkt durch die Geschäfts- und die Persistenzschicht. Wie nennt man das, was deutet es an, und was würden Sie prüfen, bevor Sie handeln?

Das ist das Anti-Pattern Architecture Sinkhole. Etwas Durchreichen ist der normale Preis der Isolation, nach der üblichen Faustregel etwa ein Fünftel, aber hier sind es 80 %, was darauf hindeutet, dass die Schichtung der falsche Stil sein könnte. Prüfen Sie zuerst, ob diese Anfragen meist einfache Lesevorgänge sind und ob der Rest des Systems wirklich von der Isolation profitiert. Zu den Optionen gehört, eine Schicht zu öffnen oder zur Domänenpartitionierung überzugehen. Schreiben Sie nicht aufgrund einer einzigen Stichprobe neu.

QUELLEN

- Martin Fowler: Presentation Domain Data Layering: https://martinfowler.com/bliki/PresentationDomainDataLayering.html
- Microsoft Azure Architecture Center: N-tier architecture style: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier
- import-linter documentation: Layers contract: https://import-linter.readthedocs.io/en/stable/contract_types/layers/
- ArchUnit user guide (layer checks): https://www.archunit.org/userguide/html/000_Index.html

- Jimmy Bogard: Vertical Slice Architecture (2018): https://www.jimmybogard.com/vertical-slice-architecture/

Der modulare Monolith

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 3.3 Information Hiding
- 5.1 Stile und Muster
- 5.2 Schichten

Begriffe von früher, die er verwendet: Deployment-Einheit, Fachliche Partitionierung, Big Ball of Mud, Architekturtest, Transaktion

DIE IDEE IN KÜRZE

Denken Sie an ein großes Kaufhaus. Es ist ein Gebäude mit einem Eingang, einheitlichen Öffnungszeiten und einem Sicherheitsdienst, doch innen ist es in Abteilungen gegliedert: Lebensmittel, Kleidung, Elektronik. Jede Abteilung hat eigenes Personal, ein eigenes Lager und eine eigene Kasse, und wenn die Kleidungsabteilung etwas aus der Lebensmittelabteilung braucht, fragt sie an deren Theke, statt einfach in deren Lager zu gehen. Ein modularer Monolith ist Software, die wie dieses Kaufhaus gebaut ist: ein Programm, als ein Stück deployt, aber innen in fachliche Abteilungen gegliedert, die ihre eigenen Daten verwalten und miteinander nur über vereinbarte Theken verkehren.

WAS SIE WISSEN MÜSSEN

- Ein modularer Monolith ist eine einzige Deployment-Einheit, deren oberste Ebene nach Fachdomänen in Module unterteilt ist, jedes mit einer kleinen, expliziten öffentlichen API, privaten Interna und eigenen Daten.
- Er bewahrt die betriebliche Einfachheit eines Monolithen (ein Build, ein Deployment, prozessinterne Aufrufe, lokale Transaktionen) und bietet zugleich viele der Modularitätsvorteile, die man sich von Microservices verspricht.
- Grenzen erodieren, wenn sie nicht durchgesetzt werden: durch Sichtbarkeitsregeln der Sprache, Architekturtests (ArchUnit, import-linter), die Verifikation von Spring Modulith für Spring-Anwendungen, Packwerk für Ruby und getrennte Datenbankschemas je Modul.
- Jede Tabelle gehört genau einem Modul; andere Module nutzen dessen API oder reagieren auf dessen Ereignisse, greifen aber nie auf dessen Tabellen zu. Joins über Modulgrenzen hinweg werden durch API-Aufrufe oder Lesemodelle ersetzt.

- Module können über prozessinterne Aufrufe kommunizieren (einfach, unmittelbar, transaktional) oder über prozessinterne Ereignisse (lockerere Kopplung, aber die Arbeit des Listeners läuft getrennt ab und kann getrennt fehlschlagen).
- Shopify hat seinen großen Ruby-on-Rails-Monolithen ab 2017 in fachliche Komponenten umgebaut und dies 2019 beschrieben; spätere Beiträge zeigen sowohl den Wert des Ansatzes als auch, wie schwer vollständige Isolation ist.
- Ein gut abgegrenztes Modul lässt sich später mit überschaubarem Aufwand als Service herauslösen, weil seine API, seine Daten und seine Abhängigkeiten bereits explizit sind.

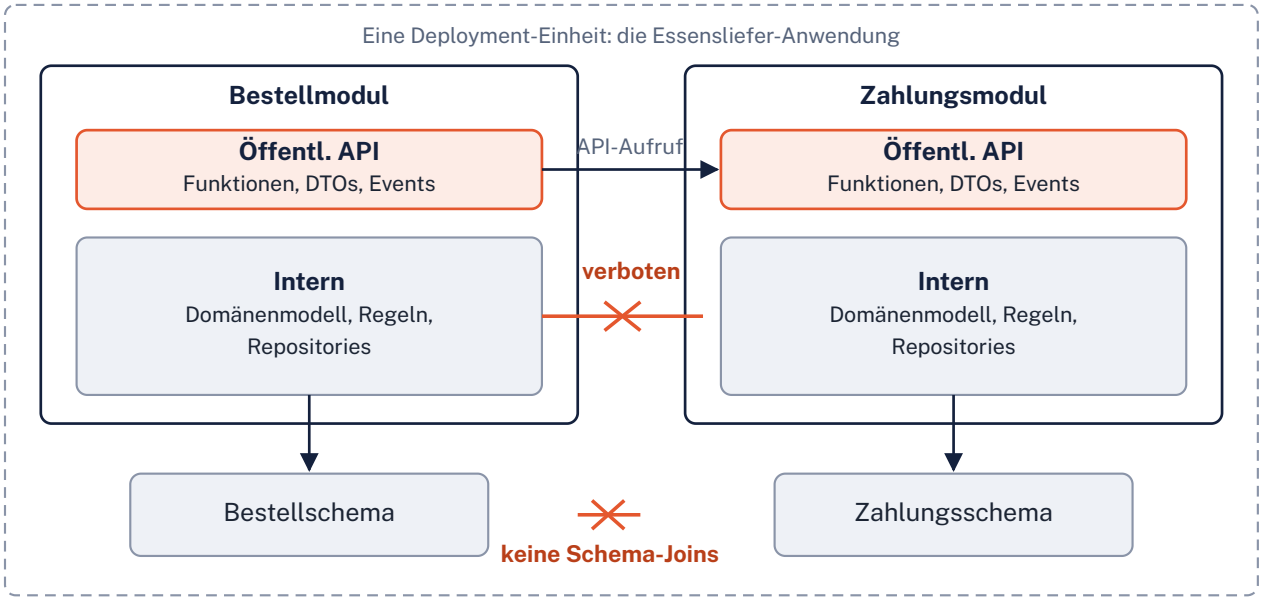
Definition: ein Deployment, viele Domänen

Ein modularer Monolith ist eine Anwendung, die als eine Einheit gebaut und deployt wird, deren Code aber auf oberster Ebene in Module unterteilt ist, die fachlichen Grenzen folgen: Bestellungen, Zahlungen, Lieferung, Speisekarten. Jedes Modul hat eine kleine, bewusst gestaltete Modul-API, die andere Module nutzen dürfen, einen größeren internen Teil, den niemand von außen anfassen darf, und eigene Daten. Er verbindet die fachliche Partitionierung von Microservices mit der einzigen Deployment-Einheit eines klassischen Monolithen.

Die Idee ist nicht neu. Module mit verborgenen Interna gehen auf David Parnas' Information Hiding (1972) zurück, und große, gut strukturierte Monolithen hat es immer gegeben. Neu sind der Name und die Aufmerksamkeit. Nach einem Jahrzehnt, in dem viele Organisationen ihre Systeme in Microservices zerlegt und dabei die Kosten der Verteilung entdeckt haben, argumentierten Fachleute wie Simon Brown und Sam Newman, der modulare Monolith sei eine unterschätzte Option und viele Teams wären mit ihm besser bedient als mit einer Menge von Services. Richards und Ford widmen ihm in der zweiten Auflage von *Fundamentals of Software Architecture* (2025) ein eigenes Kapitel.

Der Unterschied zum Schichtenstil liegt im Schnitt auf oberster Ebene. Eine Schichtenanwendung ist in Präsentation, Geschäftslogik und Persistenz unterteilt, und jede Domäne verteilt sich über alle drei. Ein modularer Monolith ist nach Domänen unterteilt, und jedes Modul kann innen geschichtet sein. Der Unterschied zu Microservices liegt im Deployment: Die Module teilen sich einen Prozess, ein Release und meist einen Datenbankserver, und sie rufen einander im Arbeitsspeicher auf.

Aufbau eines modularen Monolithen



Jedes Modul bietet eine kleine öffentliche API an und besitzt ein eigenes Schema; der Zugriff auf die Interna oder Tabellen eines anderen Moduls ist verboten.

Modul-APIs und interner Code

Die mit Abstand wichtigste Regel lautet: Ein Modul wird nur über seine API genutzt. Die API ist die Menge der Funktionen, Data Transfer Objects und Ereignisse, deren Stabilität die Verantwortlichen des Moduls zusagen; alles andere, einschließlich Domänenmodell, Repositories und Datenbanktabellen, ist intern und darf sich jederzeit ändern. In der Praxis ist die API klein: Ein Zahlungsmodul bietet vielleicht `authorise_payment`, `refund`, einen Typ `PaymentResult` und ein Ereignis `PaymentCaptured` an und verbirgt Dutzende Klassen für Zahlungsanbieter, Wiederholungen, Buchungsjournale und Abstimmung.

Gute Modul-APIs haben einige Eigenschaften gemeinsam. Sie sprechen die Sprache der Domäne (`authorise_payment`, nicht `insert_payment_row`). Sie übergeben einfache Daten wie Kennungen und kleine unveränderliche Datensätze statt interner Objekte, damit Aufrufer nicht über ein zurückgegebenes Objekt in die Interna des Moduls greifen können. Sie sind grob genug, dass ein typischer Anwendungsfall ein oder zwei Aufrufe braucht, nicht zwanzig. Und sie sind so entworfen, als könnte eines Tages ein Netzwerk zwischen Aufrufer und Aufgerufenem liegen: keine gemeinsam genutzten veränderlichen Objekte, keine Abhängigkeit von der Datenbanktransaktion des Aufrufers, explizite Fehler. Diese letzte Eigenschaft hält die Option offen, das Modul später herauszulösen.

Innerhalb der Grenze ist alles erlaubt, was das Team des Moduls für nützlich hält: eine Schichtenstruktur, eine hexagonale, ein reichhaltiges Domänenmodell oder einfache Transaction Scripts. Verschiedene Module können sich unterschiedlich entscheiden, weil nichts

außerhalb davon abhängt.

```
# delivery_platform/payments/__init__.py  (the module's public API)
from .api import authorise_payment, refund, PaymentResult
from .events import PaymentCaptured

__all__ = ["authorise_payment", "refund", "PaymentResult", "PaymentCaptured"]

# delivery_platform/ordering/internal/checkout.py
from delivery_platform import payments             # allowed: public API
# from delivery_platform.payments.internal import ledger   # forbidden

def place_order(order, card_token):
    result = payments.authorise_payment(order.id, order.total, card_token)
    if not result.ok:
        raise PaymentDeclined(result.reason)
    order.confirm(result.payment_id)
    events.publish(OrderPlaced(order.id, order.restaurant_id))
```

Ein Python-Paket als Modul: Das Paket payments exportiert eine kleine API, und ordering importiert nur diese API.

Die Grenzen durchsetzen

In einem Monolithen kann technisch jede Klasse jede andere aufrufen, daher halten Grenzen nur, wenn etwas sie durchsetzt. Ohne Durchsetzung bleibt eine unter Termindruck gebaute Abkürzung in die Interna eines anderen Moduls im Review unsichtbar, und ein modularer Monolith verfällt innerhalb von ein, zwei Jahren zu einem Big Ball of Mud. Es gibt mehrere Verteidigungslinien, die man am besten gemeinsam einsetzt.

Sichtbarkeit in der Sprache. Nutzen Sie die stärkste Kapselung, die die Sprache bietet: paketprivate Klassen in Java, `internal` in Kotlin und C# (das Grenzen nur zwischen Build-Modulen oder Assemblies durchsetzt, nicht zwischen Paketen innerhalb eines Projekts), die `exports` des Java Platform Module System, getrennte Build-Module (Maven- oder Gradle-Teilprojekte, .NET-Projekte), deren Abhängigkeiten explizit deklariert werden. Python und Ruby bieten nur Konventionen (einen führenden Unterstrich, ein Unterpaket `internal`), daher müssen Werkzeuge die Arbeit übernehmen.

Architekturtests. Architekturtests prüfen Abhängigkeiten im Build. ArchUnit (Java) kann sicherstellen, dass keine Klasse außerhalb eines Moduls dessen interne Pakete berührt und dass die Modulabhängigkeiten zyklenfrei sind. import-linter (Python) bietet einen `forbidden`-Vertrag, der andere Module daran hindern kann, das interne Paket eines Moduls zu importieren, ihnen aber die Nutzung seiner öffentlichen API weiterhin erlaubt, einen `layers`-Vertrag, dessen Schichten mehrere voneinander unabhängige Geschwister enthalten können, getrennt

durch `|`, womit sich eine Abhängigkeitsrichtung zwischen Modulen prüfen lässt, und einen `independence`-Vertrag für Pakete, die überhaupt nicht voneinander abhängen dürfen, auch nicht indirekt.

Unterstützung durch Frameworks. Spring Modulith für Spring-Boot-Anwendungen behandelt jedes direkte Unterpaket des Hauptpakets der Anwendung als Modul, das Basispaket des Moduls als dessen API und seine Unterpakete als intern, und bietet eine einzeilige Verifikation (`ApplicationModules.of(Application.class).verify()`), die fehlschlägt, wenn ein Modul auf die Interna eines anderen Moduls verweist, wenn Modulabhängigkeiten einen Zyklus bilden oder, falls ein Modul seine erlaubten Abhängigkeiten auflistet, wenn es eine nicht aufgeführte nutzt. Es kann außerdem Komponentendiagramme der Module im C4-Stil erzeugen. In der Ruby-Welt analysiert Shopifys quelloffenes Packwerk statisch die Verweise zwischen Paketen, die jeweils durch eine Datei `package.yml` deklariert sind, und meldet nicht deklarierte Abhängigkeiten.

Datengrenzen. Geben Sie jedem Modul ein eigenes Datenbankschema (oder zumindest eigene, klar mit Präfix versehene Tabellen) und idealerweise einen eigenen Datenbankbenutzer, der nur auf dieses Schema zugreifen kann. Ein Code-Review allein verhindert keinen bequemen `JOIN` über Modulgrenzen hinweg; Datenbankberechtigungen schon.

```
[importlinter:contract:payments-internals]
name = Only payments may use payments internals
type = forbidden
source_modules =
    delivery_platform.ordering
    delivery_platform.delivery
    delivery_platform.reporting
forbidden_modules =
    delivery_platform.payments.internal
# going through the public API (which itself uses internals) is fine
allow_indirect_imports = True

[importlinter:contract:module-order]
name = Module dependencies point one way
type = layers
layers =
    delivery_platform.reporting | delivery_platform.notifications
    delivery_platform.delivery
    delivery_platform.ordering | delivery_platform.restaurants
    delivery_platform.payments | delivery_platform.menus | delivery_platform.customers
```

import-linter-Verträge (in der Praxis einer pro Modul): Andere Module dürfen keine Interna von payments importieren, und Modulabhängigkeiten müssen einer Richtung folgen, wobei Module in derselben Zeile voneinander unabhängig sind.

Datenhoheit innerhalb eines Monolithen

Bei der Datenhoheit scheitern modulare Monolithen am häufigsten. Einen Datenbankserver zu teilen ist in Ordnung; Tabellen zu teilen nicht. Liest das Liefermodul die Tabelle `orders` direkt, kann das Bestellteam diese Tabelle nicht mehr ändern, ohne sich mit dem Lieferteam abzustimmen, und die beiden Module sind so eng gekoppelt, als wären sie eins. Die Regel ist einfach formuliert: Jede Tabelle hat genau ein besitzendes Modul, nur der Besitzer schreibt sie, und andere Module beziehen die Daten über die API des Besitzers oder aus Ereignissen, die er veröffentlicht.

Die Regel hat Folgen, die Teams bewusst akzeptieren müssen. Ein Bericht, der früher eine einzige SQL-Abfrage über Bestellungen, Zahlungen und Kuriere war, braucht nun entweder Aufrufe an drei Modul-APIs oder ein Lesemodell: ein Reporting-Modul, das Ereignisse der anderen abonniert und eine eigene, denormalisierte, für Abfragen optimierte Kopie hält. Fremdschlüssel über Modulschemas hinweg werden meist entfernt, weil sie das Schema eines Moduls von dem eines anderen abhängig machen würden; die Integrität über Module hinweg sichert stattdessen die Anwendung. Diese Kosten sind real, aber es sind dieselben Kosten, die ein späterer Umstieg auf Services verursachen würde, nur früher und in einem günstigeren Umfeld bezahlt.

Ein Vorteil bleibt, den verteilte Systeme verlieren: Module in einem Prozess können, wenn es wirklich sein muss, eine einzige lokale Datenbanktransaktion teilen. Nutzen Sie das sparsam. Jede solche Nutzung koppelt die Daten der Module im Betrieb und muss rückgängig gemacht werden, falls ein Modul jemals herausgelöst wird.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: ein Besitzer für jede Tabelle

Die Idee, die hier die meiste Sorgfalt verlangt, ist die Datenhoheit innerhalb eines Programms. Die Module teilen sich einen Datenbankserver, doch jede Tabelle hat einen einzigen Besitzer: Nur dieses Modul darf sie direkt lesen oder schreiben. Verfolgen Sie eine Abkürzung und einen Bericht durch die Essenslieferplattform.

- Das Liefermodul braucht die Details jeder neuen Bestellung. Die Bestelltabelle liegt in derselben Datenbank, also ist der schnellste Code eine Abfrage, die sie direkt liest.
- Monate später baut das Bestellmodul diese Tabelle um und ersetzt die Statusspalte durch getrennte Zeitpunkte für Aufgabe und Bezahlung. Die Abfrage der Lieferung schlägt nun fehl, und von da an müssen die beiden Teams jede Änderung an der Tabelle gemeinsam planen, so eng gekoppelt, als wären sie ein Modul.
- Die Regel beseitigt die Abkürzung. Nur das Bestellmodul greift auf seine Tabellen zu; andere Module nutzen seine Modul-API oder die prozessinternen Ereignisse, die es veröffentlicht: Nachrichten innerhalb des Programms, die melden, dass etwas geschehen ist. Im Entwurf weiter unten in diesem Abschnitt lauscht die Lieferung auf OrderPlaced, statt abzufragen.
- Berichte sind schwieriger. Ein Stornierungsbericht, der früher eine einzige SQL-Abfrage über Bestellungen, Erstattungen und Kunden war, würde in die Tabellen dreier Module greifen.
- Es gibt zwei Auswege: die APIs der drei Module aufrufen und die Antworten zusammenführen, oder das Reporting ein Lesemodell führen lassen: Es lauscht auf Ereignisse der Module, etwa OrderCancelled, und speichert eine eigene Kopie genau der Daten, die der Bericht braucht.
- Auch Fremdschlüssel zwischen den Tabellen verschiedener Module werden meist entfernt; die Anwendung, nicht die Datenbank, hält diese Verknüpfungen korrekt.
- **Die Lehre:** Einen Datenbankserver zu teilen ist in Ordnung; Tabellen zu teilen nicht. Ereignisse und Lesemodelle bedeuten Mehrarbeit, aber es ist dieselbe Arbeit, die ein späterer Umstieg auf Services erfordern würde, nur früher und günstiger erledigt.

Kommunikation zwischen Modulen: Aufrufe oder Ereignisse

Module kommunizieren auf zwei Arten, beide im Arbeitsspeicher. Ein **prozessinterner Aufruf** der API eines anderen Moduls ist einfach, schnell und unmittelbar: Das Bestellmodul ruft `payments.authorise_payment` auf und erhält sofort das Ergebnis, und eine Exception wird ganz normal weitergereicht. Das erzeugt eine direkte Abhängigkeit vom Aufrufer zum

Aufgerufenen, und der Aufrufer kann erst fertig werden, wenn der Aufgerufene fertig ist. Nutzen Sie Aufrufe, wenn der Aufrufer eine Antwort braucht, um weiterzumachen, etwa ob die Zahlung autorisiert wurde.

Ein prozessinternes Ereignis kehrt die Abhängigkeit um. Das Bestellmodul veröffentlicht `OrderPlaced` und weiß nicht und kümmert sich nicht darum, wer zuhört; Lieferung, Benachrichtigungen und Reporting abonnieren es. Eine neue Reaktion hinzuzufügen heißt, einen Listener hinzuzufügen, ohne das Bestellmodul anzufassen. Nutzen Sie Ereignisse, wenn der Veröffentlichende keine Antwort braucht und wenn mehrere Module auf dieselbe Tatsache reagieren.

Ereignisse bringen Feinheiten mit sich, die Teams ohne Erfahrung damit oft übersehen. Läuft ein Listener innerhalb der Transaktion des Veröffentlichenden, kann ein Fehler im Listener die Bestellung zurückrollen; läuft er nach dem Commit, ist die Bestellung gespeichert, aber die Arbeit des Listeners kann verloren gehen, wenn der Prozess dazwischen abstürzt. Spring Modulith begegnet dem mit einer Event Publication Registry, die jedes Paar aus Ereignis und Listener in derselben Transaktion wie die fachliche Änderung festhält und erst als abgeschlossen markiert, wenn der Listener erfolgreich war, sodass unvollständige erneut versucht werden können; ein selbst gebautes Gegenstück ist die Transactional Outbox, beschrieben in Abschnitt 6.7. Listener sollten außerdem idempotent sein, damit ein wiederholtes Ereignis keinen Schaden anrichtet. Keiner der beiden Stile ist besser; die meisten modularen Monolithen nutzen Aufrufe für Abfragen und Befehle, die eine Antwort brauchen, und Ereignisse für Benachrichtigungen über bereits Geschehenes.

Migrationen und Tests pro Modul

Zuständigkeit hält nur, wenn die Tools sie respektieren. Geben Sie jedem Modul ein eigenes Datenbankschema und einen eigenen Migrationsordner, sodass die Änderungen des Bestellmoduls bei seinem Code liegen und nicht unbemerkt die Tabellen des Liefermoduls ändern können; Flyway kann zum Beispiel einmal pro Schema mit getrennten Skriptorten und Verlaufstabellen laufen. Ein Datenbank-Login pro Modul, das nur für dessen eigenes Schema berechtigt ist, macht aus einer verirrten Abfrage einen Fehler. Tests folgen denselben Linien: Testen Sie jedes Modul über seine öffentliche API und ersetzen Sie seine Nachbarn durch Platzhalter. Spring Moduliths `@ApplicationModuleTest` startet zum Beispiel standardmäßig ein Modul allein. Behalten Sie einige Tests der ganzen Anwendung für die Verdrahtung. Weil alles weiterhin gemeinsam deployt wird, gilt Expand and Contract auch während eines Rolling Release.

Shopifys modularer Monolith

Shopifys Kernanwendung ist eine der größten Ruby-on-Rails-Codebasen überhaupt, an der über mehr als ein Jahrzehnt mehr als tausend Entwickler gearbeitet haben. In „Deconstructing the Monolith“ (Shopify Engineering, Februar 2019) beschrieb Kirsten Westeinde, warum das Unternehmen sie nicht in Microservices zerlegte, sondern ab 2017 zu einem modularen Monolithen umbaute. Die Probleme waren bekannt: Code für voneinander unabhängige Geschäftsbereiche war verflochten, Änderungen hatten unerwartete Auswirkungen an anderer Stelle, und die Einarbeitung dauerte lange. Microservices wurden erwogen und als mit eigenen Kosten verbunden eingeschätzt (getrennte Pipelines und Infrastruktur, Netzwerklatenz und -ausfälle, schwierigeres Refactoring über Services hinweg), daher entschied sich das Team für Modularität ohne mehrere Deployment-Einheiten.

Der erste Schritt bestand darin, den Code in einer großen automatisierten Änderung vom Standardaufbau von Rails (Models, Views und Controller) in Komponenten umzuordnen, die nach Fachdomänen benannt waren, etwa Bestellungen, Versand, Lagerbestand und Abrechnung. Der nächste war die Isolation: Jede Komponente sollte eine öffentliche API und die alleinige Hoheit über ihre Daten erhalten, und ein internes Werkzeug, Wedge, verfolgte den Fortschritt jeder Komponente, indem es während der Testsuite Aufrufe über Grenzen hinweg aufzeichnete. Ein Update vom September 2020 meldete 37 Komponenten in einem Kernmonolithen von etwa 2,8 Millionen Zeilen Ruby, und im selben Monat veröffentlichte Shopify Packwerk als Open Source, ein Werkzeug zur statischen Analyse, mit dem sich Abhängigkeiten zwischen Paketen deklarieren und prüfen lassen.

Die Geschichte ist lehrreich, weil sie kein einfacher Erfolg ist. Das Update von 2020 hielt fest, dass die ursprünglichen Ideen noch galten, sich aber fast jedes Detail geändert hatte und kaum eine Komponente bereits eine vollständige, starke Grenze hatte. Ein Rückblick auf Packwerk von 2024 berichtete, dass dessen Privacy-Prüfungen entfernt worden waren, dass manche nach Namen statt nach Laufzeitverhalten organisierte Pakete Entwickler in die Irre führten und dass selbst die vollständige Isolation eines Basispakets viele Monate dauerte; Packwerk war zu einem Wächter über zentrale Grenzen geworden statt zum Kern der Strategie. Die Lehre für Architekten: Ein modularer Monolith ist eine Entwicklungsrichtung, die anhaltende Investitionen braucht, und Grenzen nachträglich in eine große bestehende Codebasis einzuziehen ist viel schwerer, als sie von Anfang an zu wahren.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: die Essenslieferplattform als modularer Monolith

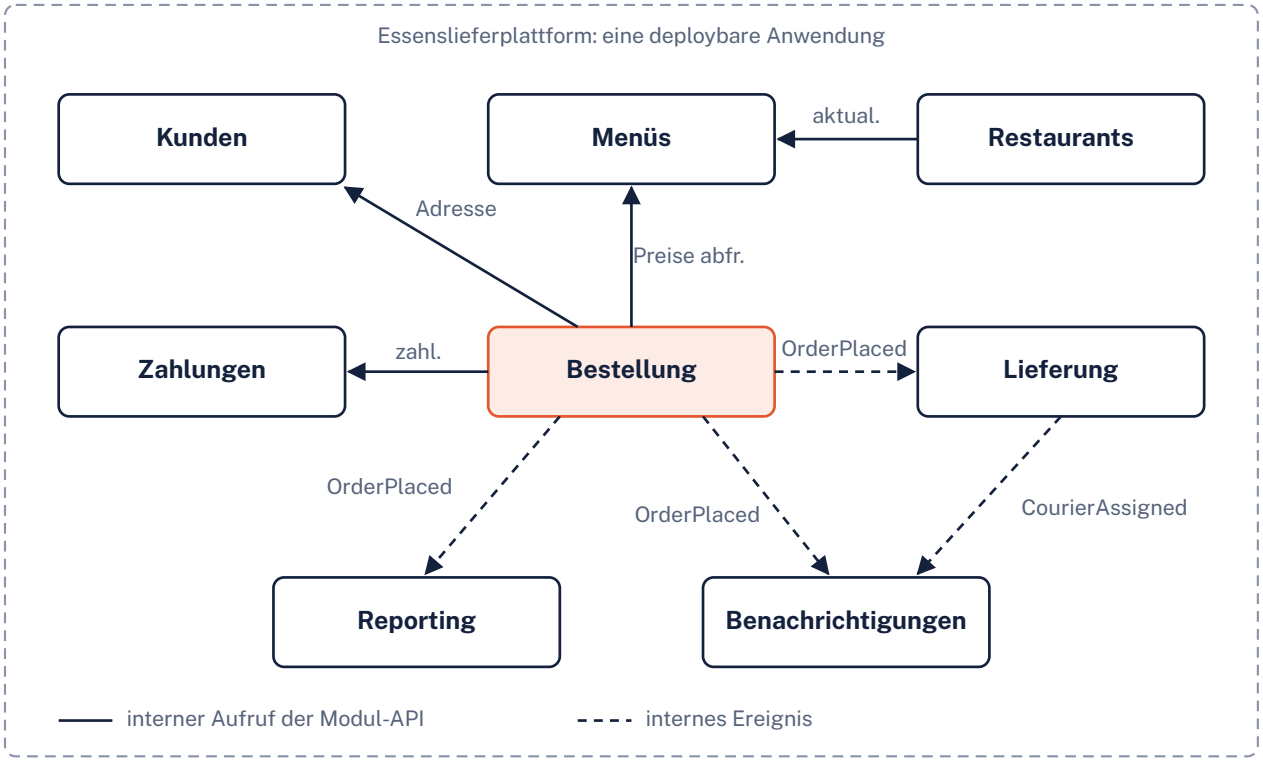
Ein Essensliefer-Start-up mit vier Teams (etwa 25 Entwicklerinnen und Entwicklern) möchte unabhängige Verantwortung ohne die Kosten des Betriebs von Microservices. Der Architekt schlägt acht Module in einer deploybaren Python-Anwendung vor, mit einem PostgreSQL-Server und einem Schema pro Modul.

Kunden	Konten, Adressen, Präferenzen. API: `get_customer`, `get_delivery_address`. Team A.
Restaurants; Speisekarten (zwei Module)	Restaurants verwaltet Restaurantkonten und Öffnungszeiten und aktualisiert Speisekarten über die Speisekarten-API; Speisekarten liefert allen anderen Preise und Verfügbarkeit (`price_items`, `is_open`). Team B.
Bestellung	Warenkorb, Checkout, Lebenszyklus der Bestellung. Ruft Kunden, Speisekarten und Zahlungen auf; veröffentlicht `OrderPlaced` und `OrderCancelled`. Team C.
Zahlungen	Autorisierung, Einzug und Erstattungen über einen externen Anbieter. API: `authorise_payment`, `refund`; veröffentlicht `PaymentCaptured`. Team C.
Zustellung	Kurierzuordnung und Sendungsverfolgung. Lauscht auf `OrderPlaced`; veröffentlicht `CourierAssigned` und `Delivered`. Team D.
Benachrichtigungen und Reporting	Reine Listener: versenden Nachrichten und pflegen Lesemodelle. Sie rufen kein anderes Modul auf. Team A.
Abhängigkeitsregel (Dependency Rule)	Zahlungen, Speisekarten und Kunden hängen von niemandem ab; Bestellungen und Restaurants rufen sie auf; Lieferung reagiert auf die Ereignisse von Bestellungen; Benachrichtigungen und Reporting hören nur zu. Geprüft durch import-linter im Build.
Deployment	Ein Container-Image, betrieben als sechs Replikate hinter einem Load Balancer, plus ein Worker-Prozess (derselbe Code), der Event-Listener und Hintergrundjobs abarbeitet.

Jedes Team besitzt ganze Module, daher landen die meisten Änderungen in einem Modul und bei einem Team.

Die Abhängigkeitsregel macht Zyklen unmöglich: Muss Zahlungen jemals etwas über Bestellungen wissen, erhält es eine Bestellkennung, keinen Import des Bestellmoduls.

Modulkarte der Essenslieferplattform



Durchgezogene Pfeile sind Aufrufe der API eines anderen Moduls; gestrichelte Pfeile sind prozessinterne Ereignisse. Bestellungen ruft drei Module auf, und seine Listener hängen nur von den Ereignissen ab, die es veröffentlicht.

Merkmale des modularen Monolithen

Merkmal	Bewertung	Begründung
Einfachheit	Mittel bis hoch	Ein Deployment und prozessinterne Aufrufe, aber gute Modulgrenzen zu finden erfordert Nachdenken.
Gesamtkosten	Hoch (günstig)	Infrastruktur und Betrieb eines Monolithen; etwas Mehraufwand für Werkzeuge zur Grenzprüfung.
Änderbarkeit bei fachlichen Änderungen	Hoch	Die fachliche Partitionierung hält die meisten Änderungen innerhalb eines Moduls, das einem Team gehört.
Testbarkeit	Mittel bis hoch	Module lassen sich über ihre APIs testen, wobei Nachbarn ersetzt werden; Tests des Gesamtsystems laufen in einem Prozess.
Deploybarkeit	Niedrig bis mittel	Weiterhin eine Deployment-Einheit: Jedes Release liefert alle Module aus, auch wenn ein kleinerer Blast Radius pro Änderung hilft.
Skalierbarkeit und Elastizität	Niedrig bis mittel	Replikate skalieren die ganze Anwendung; ein einzelnes Modul lässt sich nicht allein skalieren, doch Worker-Prozesse können Hintergrundlast abtrennen.
Fehlertoleranz	Niedrig	Ein Prozess: Ein Absturz oder Speicherleck in irgendeinem Modul betrifft alle.
Performance	Hoch	Aufrufe zwischen Modulen laufen im Arbeitsspeicher, ohne Serialisierung oder Netzwerklatenz.
Evolvierbarkeit	Hoch	Explizite APIs und Datenhoheit machen es leicht, Module später umzubauen oder herauszulösen.

Wann man ihn einsetzt, wann nicht, und Anti-Patterns

Ein modularer Monolith ist eine starke Standardwahl für neue Produkte mit einigermaßen klarer Domäne, für Organisationen mit bis zu einer Handvoll Teams, für Systeme, die starke Konsistenz über Geschäftsbereiche hinweg brauchen, und für Teams mit begrenzter Betriebskapazität. Er ist auch das natürliche Ziel bei der Modernisierung eines verworrenen Monolithen: Grenzen darin einzuziehen ist meist billiger und weniger riskant, als ihn auf einen Schlag in Services zu zerlegen. Weniger geeignet ist er, wenn Teile des Systems wirklich unterschiedliche Qualitätsmerkmale brauchen (ein Strom von Kurierpositionen, der riesige Schreibvolumen aufnehmen muss, und ein Backoffice-Modul mit mäßiger Last), wenn Dutzende Teams viele Male am Tag unabhängig voneinander releasen müssen oder wenn Teile

in verschiedenen Sprachen geschrieben sein oder aus Gründen der Datenresidenz an verschiedenen Orten laufen müssen. Diese Anforderungen verlangen getrennte Architekturquanten.

Häufige Anti-Patterns: **Module nur dem Namen nach**, bei denen die Ordner den Domänen folgen, aber alles alles importiert; **die gemeinsame Datenbank als Selbstbedienungsladen**, bei der Module ihren eigenen Code behalten, aber gegenseitig ihre Tabellen joinen; **die geschwätzige Grenze**, eine API, so feingranular, dass ein Anwendungsfall Dutzende Aufrufe braucht, was schmerzhaft wird, falls das Modul jemals herausgelöst wird; **der wuchernde Shared Kernel**, ein Modul `common`, das nach und nach Geschäftslogik aufsaugt und jedes Modul an sich koppelt; und **Entitätsmodule**, ein Schnitt nach Datenentitäten (ein Kundenmodul, ein Bestellmodul, ein Produktmodul, die jeder Anwendungsfall alle zugleich braucht) statt nach fachlichen Fähigkeiten, was Module hervorbringt, die allein nichts tun können.

Die Option offenhalten, Services herauszulösen

Ein modularer Monolith wird mal als Zwischenschritt zu Microservices dargestellt, mal als Ziel. Er kann beides sein, und der Punkt ist, dass sich die Entscheidung bis zum Last Responsible Moment aufschieben lässt. Muss ein Modul später getrennt skalieren oder unabhängig ausgeliefert werden, ist das Herauslösen viel leichter, wenn das Modul bereits eine explizite API hat, seine Daten besitzt und von anderen nur über APIs und Ereignisse abhängt. Die Arbeit wird dann weitgehend mechanisch: eine Netzwerkschnittstelle vor die API setzen, das Schema in eine eigene Datenbank verschieben, prozessinterne Ereignisse durch einen Message Broker ersetzen und jede gemeinsame Transaktion durch eine explizite Saga oder Kompensation ersetzen.

Sam Newman vertritt in *Monolith to Microservices* (2019) dasselbe Argument: Ein gut modularisierter Monolith ist der beste Ausgangspunkt für eine Zerlegung, weil der schwierige Teil, Grenzen zu finden und durchzusetzen, bereits erledigt ist. Abschnitt 5.8 behandelt die Anzeichen dafür, dass es Zeit zum Herauslösen ist, und Abschnitt 11.6 behandelt Techniken der schrittweisen Migration wie die Strangler Fig.

Eine gröbere Aufteilung: Self-Contained Systems

Self-Contained Systems (SCS) teilen ein großes System in einige autonome Webanwendungen auf, jede mit eigenen Daten, eigener Logik und eigener Benutzeroberfläche und jede im Besitz eines Teams. Sie sind meist größer und weniger zahlreich als Microservices: Die SCS-Website nennt für einen Onlineshop vielleicht 5 bis 25, wo Microservices in die Hunderte gehen könnten. Jedes System sollte seine Hauptaufgabe allein erledigen, Aufrufe anderer Systeme bleiben also außerhalb des Anfrage-Antwort-Zyklus, und Oberflächen

werden über Links verbunden statt über gemeinsamen Code. Auf der Essenslieferplattform könnten Bestellen, das Restaurantportal und die Kurier-Tools je eines sein. Der Stil passt zu klar abgegrenzten Domänen mit Weboberflächen, und jedes System kann selbst ein modularer Monolith sein. Bildschirme, die Daten aus vielen Domänen brauchen, sind schwieriger.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Was unterscheidet einen modularen Monolithen von einem geschichteten Monolithen?

Seine oberste Ebene ist nach Fachdomänen statt nach technischen Schichten unterteilt; jedes Modul hat eine explizite API, private Interna und eigene Daten und kann innen geschichtet sein.

▼ Warum sollten Modul-APIs Kennungen und einfache Datensätze übergeben statt interner Domänenobjekte?

Zurückgegebene interne Objekte lassen Aufrufer in die Interna des Moduls greifen und koppeln sie an dessen Modell; einfache Daten lassen die Interna frei änderbar und halten die API bereit, später ein Netzwerk zu überqueren.

▼ Ein Bericht braucht Daten aus den Schemas von Bestellungen und Zahlungen. Welche zwei Optionen respektieren die Datenhoheit?

Die APIs beider Module aufrufen und die Ergebnisse zusammenführen, oder in einem Reporting-Modul ein Lesemodell aufbauen, das auf die Ereignisse beider Module lauscht und eine eigene denormalisierte Kopie hält.

▼ Welches Risiko bringt ein prozessinterner Event-Listener mit sich, der nach dem Commit der Transaktion des Veröffentlichenden läuft?

Stürzt der Prozess nach dem Commit, aber vor dem Lauf des Listeners ab, geht die Arbeit des Listeners verloren, sofern das Ereignis nicht dauerhaft festgehalten wird, etwa in einer Event Publication Registry oder einer Outbox, und erneut versucht wird.

▼ Probieren Sie es aus: In einem modularen Monolithen für die Essenslieferung greift das Modul Reports für eine nächtliche Zusammenfassung per Join direkt auf die Tabellen des Moduls Payments zu. Es funktioniert und ist schnell. Warum erheben Sie Einwände, wodurch würden Sie es ersetzen, und wie würden Sie verhindern, dass es wieder vorkommt?

Jede Tabelle gehört genau einem Modul. Der Join koppelt Reports an das private Schema von Payments, sodass eine Änderung an Payments ihn unbemerkt brechen kann und eine spätere Auslagerung schwieriger wird. Ersetzen Sie ihn durch einen Aufruf der öffentlichen API von Payments oder ein aus den Ereignissen von Payments gespeistes Lesemodell, und nehmen Sie etwas Mehraufwand und veraltete Daten in Kauf. Geben Sie jedem Modul sein eigenes Schema, und ergänzen Sie einen Architekturtest, etwa import-linter oder Datenbankberechtigungen, damit der Build bei jeder Wiederholung fehlschlägt.

QUELLEN

- Shopify Engineering: Deconstructing the Monolith (2019): https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity
- Shopify Engineering: Under Deconstruction: The State of Shopify's Monolith (2020): https://shopify.engineering/shopify-monolith
- Shopify Engineering: A Packwerk Retrospective (2024): https://shopify.engineering/a-packwerk-retrospective
- Spring Modulith reference documentation: Fundamentals: https://docs.spring.io/spring-modulith/reference/fundamentals.html
- import-linter documentation: Forbidden contract: https://import-linter.readthedocs.io/en/stable/contract_types/forbidden/
- Redgate: Handling multiple schemas in the same database with Flyway: https://www.red-gate.com/blog/handling-multiple-schemas-in-the-same-database-with-flyway/
- Spring Modulith reference: integration testing application modules: https://docs.spring.io/spring-modulith/reference/testing.html
- Self-Contained Systems: https://scs-architecture.org/
- SCS compared with microservices: https://scs-architecture.org/vs-ms.html

Pipes and Filters und Pipeline-Architekturen

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.4 Performance und Skalierbarkeit
- 5.1 Stile und Muster

Begriffe von früher, die er verwendet: Durchsatz, Idempotent, Architekturtest, Schema

DIE IDEE IN KÜRZE

Denken Sie an eine Autowaschanlage. Jedes Auto durchläuft eine Reihe von Stationen: Einweichen, Bürsten, Spülen, Trocknen. Jede Station erledigt eine Aufgabe, weiß nichts von den anderen und gibt das Auto einfach über ein Förderband an die nächste weiter. Sie können eine Wachsstation ergänzen, ohne die anderen zu ändern, und wenn das Trocknen langsam ist, stauen sich die Autos davor. Pipes and Filters funktioniert für Daten genauso: Jeder Datensatz durchläuft eine Kette einfacher Schritte (die Filter), von einem zum nächsten befördert durch ein Förderband (die Pipe).

WAS SIE WISSEN MÜSSEN

- Pipes and Filters ordnet die Verarbeitung als Kette unabhängiger Filter an, die durch Einweg-Pipes verbunden sind; Filter teilen keinen Zustand und kennen ihre Nachbarn nicht, nur das Datenformat, das sie lesen und schreiben.
- Es gibt vier Arten von Filtern: Producer (Quellen), Transformer, Tester (die Datensätze durchlassen oder verwerfen) und Consumer (Senken).
- Doug McIlroy schlug 1964 in einem Memo vor, Programme wie Gartenschläuche zu koppeln; Ken Thompson fügte Unix 1973 Pipes hinzu, und die Unix-Philosophie kleiner, durch Pipes verbundener Werkzeuge machte den Stil berühmt.
- Typische Einsatzgebiete sind ETL- und Datenpipelines, Compilerphasen, Medienverarbeitung und Log-Verarbeitung; dieselbe Struktur läuft als Batch (begrenzte Eingabe) oder als Stream (unbegrenzte Eingabe).
- Der Durchsatz einer Pipeline wird von ihrem langsamsten Filter bestimmt, und ein schneller Producer überrollt einen langsamen Consumer, sofern die Puffer nicht begrenzt sind und Backpressure den Producer bremst.

> - Der Stil schneidet bei Einfachheit, Modularität und Wiederverwendung von Filtern gut ab, bei Interaktivität und beim Umgang mit gemeinsamem Zustand oder komplexer Fehlerbehebung über mehrere Schritte hinweg schlecht.

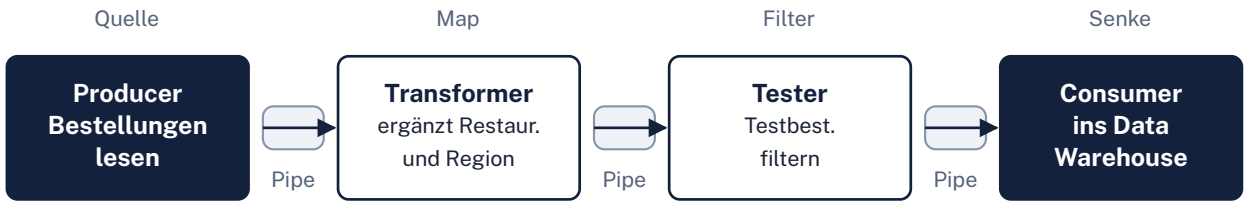
Definition und Ursprung

Im Stil Pipes and Filters ist die Verarbeitung eine Kette unabhängiger Schritte. Jeder Filter liest Daten von seiner Eingabe, erledigt eine Aufgabe damit und schreibt das Ergebnis in seine Ausgabe. Jede Pipe transportiert Daten in eine Richtung von einem Filter zum nächsten, meist mit einem Puffer dazwischen. Die Beschreibung von Garlan und Shaw (1994) nennt die Einschränkungen, die dem Stil seine Eigenschaften geben: Filter müssen unabhängig sein, dürfen keinen Zustand mit anderen Filtern teilen und dürfen die Identität der Filter an beiden Enden ihrer Pipes nicht kennen. Ein Filter kennt nur das Format dessen, was er liest und was er schreibt.

Die bekannteste Heimat des Stils ist Unix. In einem Memo vom 11. Oktober 1964 schrieb Doug McIlroy, es solle Möglichkeiten geben, Programme wie einen Gartenschlauch zu koppeln und ein weiteres Segment anzuschrauben, wenn die Daten auf andere Weise bearbeitet werden müssen. Ken Thompson implementierte 1973 in Version 3 Unix den Systemaufruf `pipe` und Pipes in der Shell, und die Notation `|` folgte in Version 4. Die daraus entstandene Unix-Philosophie, Programme zu schreiben, die eine Sache gut machen und über eine universelle Textschnittstelle zusammenarbeiten, ist der Stil Pipes and Filters, angewandt auf den Werkzeugsatz eines ganzen Betriebssystems. POSA (1996) katalogisierte Pipes and Filters als Architekturmuster, und Hohpe und Woolfs *Enterprise Integration Patterns* (2003) wandten es auf das Messaging zwischen Systemen an.

Richards und Ford nennen denselben Stil Pipeline-Architektur und unterscheiden vier Arten von Filtern. Ein **Producer** (oder Quelle) startet den Fluss und liest aus einer Datei, einer Datenbank oder einer Queue. Ein **Transformer** verändert jeden Datensatz: parsen, umwandeln, anreichern. Ein **Tester** prüft eine Bedingung und gibt den Datensatz weiter oder verwirft ihn, manchmal leitet er ihn an eine von mehreren Ausgaben. Ein **Consumer** (oder Senke) beendet den Fluss und schreibt in eine Datenbank, eine Datei oder auf einen Bildschirm.

Filter und Pipes



Jeder Filter kennt nur sein Ein- und Ausgabeformat, nie seine Nachbarn

Pipes puffern Daten, Transport in eine Richtung

Vier Arten von Filtern, verbunden durch Pipes; jeder Filter lässt sich für sich entwickeln, testen und ersetzen.

So funktioniert es

Die Einschränkungen erzeugen die Vorteile unmittelbar. Weil ein Filter keinen Zustand teilt und seine Nachbarn nicht kennt, lässt er sich für sich schreiben und testen, mit einer Liste von Eingabedatensätzen und einer erwarteten Liste von Ausgaben; er lässt sich in anderen Pipelines wiederverwenden; und Filter lassen sich hinzufügen, entfernen oder umordnen, indem man nur ändert, wie die Pipeline zusammengesetzt ist. Weil Datensätze inkrementell fließen, kann ein Filter mit dem ersten Datensatz beginnen, bevor der vorherige Filter den letzten fertig hat; so laufen die Stufen nebenläufig, und die ganze Pipeline braucht Speicher für nur wenige Datensätze auf einmal, nicht für den gesamten Datenbestand.

Die Pipe verbirgt, wie Daten sich bewegen. In einer Unix-Shell ist sie ein Puffer des Betriebssystems zwischen zwei Prozessen; unter Linux fasst er standardmäßig 64 KiB, und ein Schreiber, der ihn füllt, blockiert einfach, bis der Leser aufholt. In einem Python-Programm kann sie ein Generator sein, sodass jeder Filter den nächsten Datensatz vom vorherigen abruft. In einer Datenplattform kann sie eine Message Queue oder ein Topic in einem Log wie Apache Kafka sein, wodurch Filter auf verschiedenen Maschinen laufen und Neustarts überstehen können. Der Filtercode ändert sich dabei kaum; die betrieblichen Eigenschaften ändern sich sehr stark.

Es gibt Varianten der strengen Form. Eine Pipeline kann sich auffächern (ein Tester leitet Datensätze an zwei verschiedene Consumer) oder zusammenführen (zwei Producer speisen einen Transformer). Sie kann linear sein, was am einfachsten nachzuvollziehen ist, oder ein gerichteter azyklischer Graph von Schritten, wie in Workflow-Werkzeugen wie Apache Airflow. Streng genommen bricht ein Filter mit internem Zustand, etwa einer, der zählt oder sortiert, die

inkrementelle Eigenschaft, weil er die gesamte Eingabe sehen muss, bevor er Ausgabe erzeugt; `sort` in einer Unix-Pipeline ist ein solcher Filter. Solche Filter sind erlaubt, aber sie werden zu den Stellen, an denen Speicher, Latenz und Neustarts Nachdenken erfordern.

Beispiele: wo Pipelines vorkommen

ETL- und Datenpipelines. ETL-Jobs (Extract, Transform, Load) bringen Daten aus operativen Systemen in ein Data Warehouse: die Bestellungen von gestern extrahieren, bereinigen und umwandeln, mit Restaurantdaten zusammenführen, in Berichtstabellen laden. Jeder Schritt ist ein Filter, und moderne Datenplattformen (Apache Beam, Spark, dbt-Modelle, Cloud-Dataflow-Dienste) sind im Wesentlichen Werkzeuge, um solche Pipelines in großem Maßstab zu bauen und zu betreiben.

Compiler. Ein klassischer Compiler ist eine Pipeline aus Phasen: Die lexikalische Analyse macht aus Zeichen Tokens, das Parsen macht aus Tokens einen Syntaxbaum, die semantische Analyse prüft Typen und Namen, die Optimierung schreibt eine Zwischendarstellung um, und die Codegenerierung erzeugt Maschinencode. Jede Phase verarbeitet eine Darstellung und erzeugt die nächste, sodass Phasen getrennt entwickelt und ersetzt werden können; das LLVM-Projekt etwa lässt viele Sprach-Frontends einen Optimierer und eine Reihe von Backends gemeinsam nutzen. Echte Compiler lockern den strengen Stil, weil Phasen eine Symboltabelle teilen.

Medienverarbeitung. Videotranskodierung und Audioverarbeitung verketten Decoder, Skalierer, Filter und Encoder. GStreamer baut Medienanwendungen ausdrücklich als Pipelines aus Elementen, die über Pads verbunden sind, und die Filtergraphen von FFmpeg funktionieren genauso.

Log- und Ereignisverarbeitung. Log-Shipper wie Logstash und Fluent Bit werden als Eingabe-, Filter- und Ausgabestufen konfiguriert: Logzeilen lesen, in Felder parsen, das Rauschen von Health-Checks verwerfen, personenbezogene Daten maskieren und alles an einen Suchindex senden. Die Observability-Daten der Essenslieferplattform fließen auf diese Weise.

Feature- und Inferenzpipelines im maschinellen Lernen. Die Aufbereitung von Trainingsdaten sowie die Vor- und Nachverarbeitung rund um ein Modell werden üblicherweise als Pipelines gebaut, worauf Kapitel 12 zurückkommt.

```
# Top ten URLs that returned HTTP 500 in today's access log
grep '" 500 ' access.log \
  | awk '{print $7}' \
  | sort \
  | uniq -c \
  | sort -rn \
  | head -10
```

Eine Unix-Pipeline: Sechs kleine Werkzeuge, jedes ein Filter, finden die URLs, die am häufigsten mit HTTP 500 fehlschlugen (im Common-Access-Log-Format folgt der Statuscode auf die in Anführungszeichen stehende Anfragezeile).

```
import json

def read_lines(path):                 # producer
    with open(path) as f:
        for line in f:
            yield line

def parse(lines):                     # transformer
    for line in lines:
        yield json.loads(line)

def only_failed(events):              # tester
    for e in events:
        if e["status"] == "payment_failed":
            yield e

def count_by_provider(events):        # consumer
    counts = {}
    for e in events:
        counts[e["provider"]] = counts.get(e["provider"], 0) + 1
    return counts

events = only_failed(parse(read_lines("orders.jsonl")))
print(count_by_provider(events))
```

Dieselbe Idee mit Python-Generatoren: Jede Funktion ist ein Filter, und Datensätze fließen einzeln, sodass die Datei nie auf einmal in den Speicher geladen wird.

Batch und Streaming

Dieselbe Struktur läuft in zwei Modi. Die Stapelverarbeitung verarbeitet eine begrenzte Eingabe, etwa alle Bestellungen von gestern, läuft bis zum Ende und stoppt. Sie ist leicht nachzuvollziehen: Schlägt ein Lauf fehl, beheben Sie das Problem und starten ihn erneut, vorausgesetzt, jeder Lauf ersetzt seine Ausgabe, statt an sie anzuhängen. Nächtliche

Auszahlungen an Restaurants, monatliche Rechnungen und Warehouse-Ladevorgänge sind typische Batch-Pipelines. Ihre Schwäche ist die Verzögerung: Die Ergebnisse sind so alt wie der letzte Lauf.

Stream Processing verarbeitet eine unbegrenzte Eingabe und jeden Datensatz kurz nach seinem Eintreffen: Kurierpositionen, Zahlungsereignisse, Klicks. Die Ergebnisse sind frisch, doch es tauchen neue Fragen auf. Ein Stream endet nie, daher müssen Aggregate über Zeitfenster berechnet werden (Bestellungen pro Restaurant je fünf Minuten); Datensätze können verspätet oder in falscher Reihenfolge eintreffen; und die Pipeline muss ununterbrochen laufen, sich von Abstürzen erholen, ohne Datensätze zu verlieren oder doppelt zu zählen, und Spitzen der Eingaberate verkraften. Frameworks wie Kafka Streams, Apache Flink und Apache Beam bieten Zeitfenster, Checkpoints und State Stores für diese Probleme. Das Modell von Beam behandelt Batch als Sonderfall von Streaming mit begrenzter Eingabe, sodass eine Pipeline-Definition auf beide Arten laufen kann.

Viele Organisationen betreiben beides: einen Stream für frische, näherungsweise operative Dashboards und einen Batch-Job, der über Nacht die exakten Zahlen neu berechnet. Abschnitt 7.8 behandelt die Seite der Datenarchitektur bei dieser Wahl.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: den Engpass finden

Die Essenslieferplattform lädt Bestell-Events über vier Filter in ihr Data Warehouse. Gemessen auf einem Worker kann jeder Filter für sich die folgende Anzahl Datensätze pro Sekunde verarbeiten.

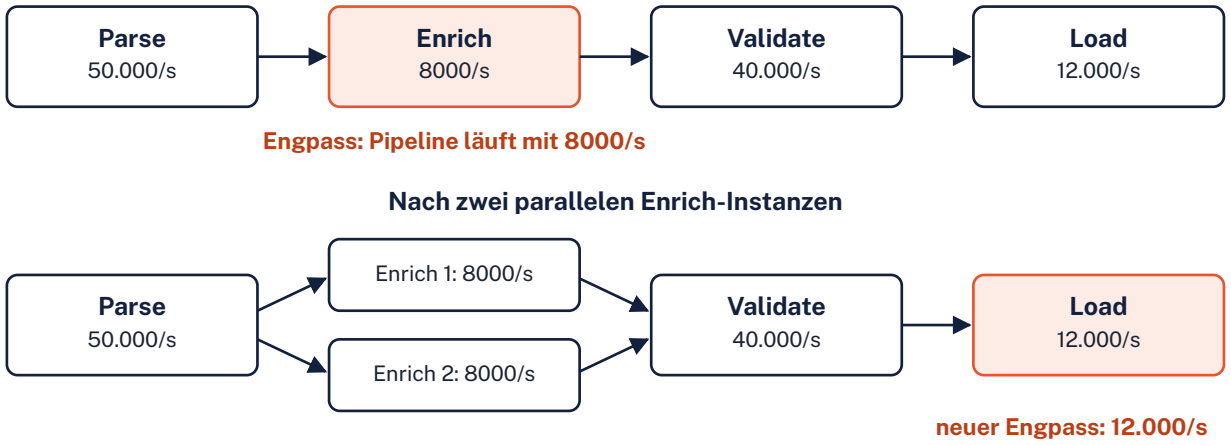
Kapazitäten	Parse 50.000/s; Enrich (schlägt Restaurant und Region nach) 8000/s; Validate 40.000/s; Load 12.000/s.
Durchsatz der Pipeline	Eine Pipeline kann nicht schneller sein als ihr langsamster Filter: min(50.000, 8000, 40.000, 12.000) = 8000 Datensätze pro Sekunde.
Zeit für die Daten eines Tages	An einem geschäftigen Tag erzeugt die Plattform 1.000.000 Bestell-Events. 1.000.000 ÷ 8000 = 125 Sekunden.
Den Engpass skalieren	Enrich ist zustandslos, daher können zwei Instanzen parallel laufen, die jeweils die Hälfte der Datensätze übernehmen: 2 × 8000 = 16.000/s auf dieser Stufe.
Neuer Durchsatz	min(50.000, 16.000, 40.000, 12.000) = 12.000/s. Jetzt ist Load der Engpass. 1.000.000 ÷ 12.000 ≈ 83 Sekunden, eine Ersparnis von 33 % statt der 50 %, die eine naive Lesart nahelegen würde.
Was die Puffer bewirken	Ohne Backpressure würde Parse Datensätze mit 50.000/s in die Pipe vor Enrich schieben, während Enrich 8000/s abarbeitet; der Puffer würde also um 42.000 Datensätze pro Sekunde wachsen, bis der Speicher erschöpft ist.

Einen anderen Filter als den Engpass zu beschleunigen ändert nichts; messen Sie zuerst.

Nur zustandslose Filter lassen sich so einfach parallelisieren. Ein Filter mit Zustand, etwa laufenden Summen pro Restaurant, muss seine Eingabe nach Schlüssel partitionieren, damit jede Instanz alle Datensätze zu ihren Schlüsseln sieht.

Das ist dieselbe Überlegung wie bei Amdahls Gesetz und der Auslastung aus Abschnitt 2.4: Die langsamste serielle Stufe setzt die Grenze.

Eine Pipeline zu skalieren verschiebt den Engpass



Der Durchsatz wird vom langsamsten Filter bestimmt; ihn zu parallelisieren hilft, bis ein anderer Filter der langsamste wird.

Backpressure und Fehlerbehandlung

Ist ein Producer schneller als ein Consumer, muss etwas nachgeben. Die Optionen sind: puffern (was das Problem nur aufschiebt, sofern der Puffer nicht begrenzt ist), Datensätze verwerfen oder Stichproben nehmen (für manche Metriken akzeptabel, für Zahlungen nie) oder Backpressure anwenden: die schnelle Stufe auf das Tempo der langsamen bremsen. Unix-Pipes tun das automatisch, weil eine volle Pipe den Schreiber blockiert. Pull-basierte Entwürfe wie Python-Generatoren tun es konstruktionsbedingt, weil nichts erzeugt wird, bevor die nächste Stufe danach fragt. Verteilte Pipelines brauchen es als bewusstes Designelement: begrenzte Queues, Consumer, die in ihrem eigenen Tempo aus einem Log abrufen, oder Protokolle wie die Reactive-Streams-Spezifikation der JVM, in der ein Subscriber signalisiert, für wie viele Datensätze er bereit ist. Abschnitt 9.3 behandelt Queues und Backpressure ausführlich.

Fehler brauchen für jeden Filter eine Strategie, die vorab festgelegt wird. Ein **fehlerhafter Datensatz** (eine Poison Message) sollte die Pipeline nicht anhalten: Leiten Sie ihn mit dem Grund an eine Dead Letter Queue oder Fehlerdatei weiter, zählen Sie ihn und schlagen Sie Alarm, wenn die Rate steigt. Ein **vorübergehender Fehler**, etwa ein Timeout bei einem Nachschlagedienst, wird mit Back-off wiederholt. Ein **Absturz** der Pipeline braucht einen Wiederaufsetzpunkt: Batch-Jobs laufen von vorn oder ab einem Checkpoint neu, und Streaming-Frameworks sichern ihre Position und ihren Zustand in Checkpoints. Weil Neustarts bedeuten, dass manche Datensätze doppelt verarbeitet werden, sollten Consumer, die Ergebnisse schreiben, idempotent sein, etwa durch einen Upsert auf einen natürlichen Schlüssel statt blinden Einfügens. Bei Fehlern ist der Stil am schwächsten: Filter wissen

bewusst nichts voneinander, daher kann ein in Schritt vier entdeckter Fehler nicht ohne Weiteres rückgängig machen, was Schritt zwei schon getan hat, und eine Pipeline, die ein solches koordiniertes Rollback braucht, ist vermutlich der falsche Stil.

Merkmale von Pipes and Filters (einzelnes Deployment)

Merkmal	Bewertung	Begründung
Einfachheit	Hoch	Jeder Filter tut eine Sache; der Fluss ist leicht zu zeichnen, zu lesen und zu erklären.
Modularität und Wiederverwendung	Hoch	Filter teilen keinen Zustand und kennen nur Datenformate, daher lassen sie sich wiederverwenden, umordnen und ersetzen.
Testbarkeit	Hoch	Ein Filter wird mit Beispieleingaben und erwarteten Ausgaben getestet, ohne seine Nachbarn.
Gesamtkosten	Hoch (günstig)	Ein Programm oder Job; wenig Infrastruktur, sofern nicht verteilt.
Durchsatz	Mittel bis hoch	Die Stufen laufen nebenläufig, und zustandslose Stufen lassen sich parallelisieren, doch der langsamste Filter deckelt alles.
Latenz und Interaktivität	Niedrig	Gebaut für Datenflüsse, nicht für Anfrage-Antwort-Dialoge mit Nutzern; jede Stufe fügt Verzögerung hinzu.
Deploybarkeit	Niedrig bis mittel	Als ein Programm oder Job deployt jede Änderung die ganze Pipeline neu; über Queues verteilte Filter lassen sich getrennt releasen.
Fehlertoleranz	Niedrig bis mittel	In einem Prozess stoppt ein Absturz die ganze Pipeline; Checkpoints, Wiederholungen und Dead Letter Queues müssen ergänzt werden.
Gemeinsamer Zustand und Transaktionen	Niedrig	Filter sind bewusst zustandslos und unabhängig; Konsistenz oder Rollback über Schritte hinweg ist umständlich.
Skalierbarkeit	Niedrig bis mittel in einem Prozess	Höher, wenn Filter über Queues verteilt sind, auf Kosten von Betriebsaufwand und Zustellgarantien.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: wenn eine Pipeline auf halbem Weg ausfällt

Der Punkt, an dem hier die meisten stolpern, ist, was geschieht, wenn eine Pipeline mittendrin ausfällt. Filter wissen nichts voneinander, daher gibt es keinen einfachen Weg, bereits erledigte Arbeit rückgängig zu machen, und ein Neustart verarbeitet manche Datensätze doppelt. Verfolgen Sie den Warehouse-Ladevorgang eines Tages durch drei Arten von Fehlern.

- Einmal am Tag lädt die Essenslieferplattform die Datensätze abgeschlossener Lieferungen ihrer Kuriere über vier Filter ins Data Warehouse: Parse, Enrich (das die Zone jedes Kuriers nachschlägt), Validate und Load.
- Ein Datensatz ist fehlerhaft, und Parse kann ihn nicht lesen. Statt den ganzen Job wegen eines schlechten Datensatzes anzuhalten, legt Parse ihn in einer Dead Letter Queue beiseite, vermerkt den Grund, zählt ihn und macht weiter.
- Als Nächstes antwortet der Nachschlagedienst, von dem Enrich abhängt, ein paar Sekunden lang nicht. Das ist ein vorübergehender Fehler, also wiederholt Enrich mit Back-off und wartet jedes Mal etwas länger.
- Dann stürzt der Job ab, nachdem Load 300.000 der 450.000 Datensätze des Tages geschrieben hat. Diese Zeilen bleiben: Die Filter koordinieren sich nicht, daher kann nichts in der Pipeline sie ohne Weiteres zurückrollen.
- Die einfachste Wiederherstellung ist, den Job von vorn neu zu starten. Fügt Load die Zeilen aber einfach ein, erscheinen die ersten 300.000 Lieferungen nun doppelt, und jeder darauf aufbauende Bericht zählt sie doppelt.
- Also wird Load idempotent gemacht: Es führt einen Upsert auf die eigene Kennung jeder Lieferung aus, fügt eine Zeile hinzu, wenn die Lieferung neu ist, und ersetzt sie, wenn sie schon vorhanden ist. Ein erneuter Lauf des Jobs hinterlässt dann genau eine Zeile pro Lieferung.
- **Die Lehre:** Legen Sie die Fehlerstrategie jedes Filters vor dem ersten Fehler fest. Legen Sie schlechte Datensätze beiseite, wiederholen Sie bei kurzen Ausfällen und machen Sie den letzten Schritt wiederholbar, denn Neustarts werden manche Datensätze doppelt verarbeiten.

Wann man ihn einsetzt, wann nicht, und Anti-Patterns

Setzen Sie Pipes and Filters ein, wenn das Problem von Natur aus ein Datenfluss in eine Richtung durch klar getrennte Transformationen ist: Aufnahme, Umwandlung, Anreicherung, Validierung, Aggregation, Export. Es ist die Standardwahl im Data Engineering, für die Verarbeitung von Dateien und Medien und für jeden Job, bei dem sich die Schritte häufiger

ändern als der Gesamtablauf. Es kommt auch innerhalb anderer Stile vor: ein nächtlicher Reporting-Job in einem modularen Monolithen oder ein Stream-Processing-Service in einem ereignisgesteuerten System.

Vermeiden Sie es als Hauptstruktur interaktiver Anwendungen, bei denen Nutzer sofortige, dialogartige Antworten erwarten und die Arbeit kein Fluss ist; für Logik, bei der viele Schritte denselben Zustand teilen und ändern müssen; und für Prozesse, die ein koordiniertes Rollback über Schritte hinweg brauchen, was eine Transaktion oder eine Saga erfordert.

Häufige Anti-Patterns: **der Gott-Filter**, ein Schritt, der parst, anreichert, validiert und schreibt und damit die Vorteile des Stils verliert; **verborgener gemeinsamer Zustand**, Filter, die über eine gemeinsame Datenbanktabelle oder globale Variable statt über die Pipe kommunizieren und damit die Kopplung zurückbringen, die der Stil beseitigt; **Formatdrift**, ein Producer, der still sein Ausgabeformat ändert und jeden nachgelagerten Filter bricht, verhindert durch explizite Schemas und Schemaprüfungen (siehe Abschnitt 7.7); **unbegrenzte Puffer**, die einen langsamen Consumer in einen Absturz wegen Speichermangels verwandeln; und **nicht idempotente Senken**, die nach jeder Wiederholung doppelt zählen.

Den Stil im Code durchsetzen

Die Einschränkungen des Stils lassen sich explizit machen. Geben Sie jedem Filter dieselbe einfache Signatur (ein Iterable von Datensätzen hinein, ein Iterable heraus), damit das Zusammensetzen mechanisch wird. Definieren Sie das Datensatzformat an jeder Pipe als Schema (eine Dataclass, ein JSON Schema, eine Avro- oder Protocol-Buffers-Definition) und validieren Sie es an der Grenze. Halten Sie Filter frei von gemeinsamem veränderlichem Zustand: Ein Architekturtest kann Filtermodulen verbieten, die Datenbankschicht oder einander zu importieren, sodass Daten nur über die Pipe zwischen ihnen fließen können. Testen Sie jeden Filter mit kleinen Ein- und Ausgabe-Fixtures und die zusammengesetzte Pipeline Ende-zu-Ende mit einer repräsentativen Stichprobe, einschließlich fehlerhafter Datensätze, um zu belegen, dass die Fehlerstrategie funktioniert.

Überwachen Sie in verteilten Pipelines den Durchsatz pro Stufe, die Queue-Tiefe und den Consumer-Lag: Eine wachsende Queue vor einer Stufe ist das sichtbare Zeichen eines Engpasses oder eines versagenden Filters, und sie ist die erste Metrik, auf die man einen Alarm legen sollte.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Welche drei Einschränkungen legten Garlan und Shaw Filtern auf?**

Filter müssen unabhängige Einheiten sein, dürfen keinen Zustand mit anderen Filtern teilen und dürfen die Identität der Filter vor oder hinter ihnen nicht kennen.

▼ **Warum bricht der Unix-Befehl sort die inkrementelle Eigenschaft einer Pipeline?**

Er muss seine gesamte Eingabe lesen, bevor er die erste Zeile ausgeben kann, hält also den ganzen Datenbestand und verzögert alles Nachfolgende, bis seine Eingabe endet.

▼ **Eine Pipeline hat Stufen, die 20.000, 5000 und 15.000 Datensätze pro Sekunde verarbeiten können. Wie hoch ist ihr Durchsatz, und was geschieht, wenn die mittlere Stufe verdoppelt wird?**

5000 pro Sekunde, bestimmt durch die langsamste Stufe. Eine Verdopplung auf 10.000 lässt die Pipeline mit 10.000 pro Sekunde laufen, immer noch unter den beiden anderen Stufen.

▼ **Warum sollte der letzte Consumer einer neu startbaren Pipeline idempotent sein?**

Weil ein Neustart oder eine Wiederholung manche Datensätze doppelt liefern kann; ein idempotenter Schreibvorgang wie ein Upsert auf einen natürlichen Schlüssel liefert dasselbe Ergebnis, egal wie oft ein Datensatz ankommt.

▼ **Probieren Sie es aus: Eine Pipeline für Bestellereignisse hat die Filter Parse mit 40.000/s, Enrich mit 6000/s, Validate mit 30.000/s und Load mit 10.000/s, und zur Spitzenzeit treffen 9000 Datensätze/s ein. Was passiert? Wie viele Worker für Enrich beheben das Problem, und was begrenzt dann den Durchsatz?**

Der Durchsatz wird vom langsamsten Filter bestimmt, Enrich mit 6000/s, sodass zur Spitzenzeit ein Rückstau von 3000 Datensätzen pro Sekunde wächst. Zwei Enrich-Worker schaffen 12.000/s, sofern es sauber skaliert. Load wird dann mit 10.000/s zur Grenze, was 9000/s immer noch übertrifft, aber nur etwa 10 % Reserve lässt. Halten Sie die Puffer begrenzt, mit Backpressure, damit eine langsame Stufe nicht den Speicher erschöpfen kann.

QUELLEN

- Doug McIlroy's 1964 memo on coupling programs (cat-v.org): https://doc.cat-v.org/unix/pipes/
- Enterprise Integration Patterns: Pipes and Filters: https://www.enterpriseintegrationpatterns.com/patterns/messaging/PipesAndFilters.html

- Microsoft Azure Architecture Center: Pipes and Filters pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters
- Python documentation: Functional Programming HOWTO (generators): https://docs.python.org/3/howto/functional.html

Microkernel- und Plug-in-Architekturen

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.6 APIs und Integration
- 3.3 Information Hiding
- 5.1 Stile und Muster

Begriffe von früher, die er verwendet: Deployment-Einheit, Breaking Change, Schnittstelle

DIE IDEE IN KÜRZE

Denken Sie an eine Spielkonsole. Die Konsole selbst erledigt ein paar allgemeine Dinge: Sie startet, zeigt ein Menü, liest die Controller aus und zeichnet Bilder. Jedes Spiel ist ein Modul, das über einen Standardanschluss eingesteckt wird. Sie können jahrelang neue Spiele hinzufügen, ohne die Konsole zu öffnen, und ein schlecht gemachtes Spiel kann sie nicht umverdrahten. Eine Microkernel-Architektur baut Software auf dieselbe Weise: Ein kleiner, stabiler Kern erledigt die allgemeine Arbeit, und Plug-ins fügen die veränderlichen Teile hinzu, über einen vereinbarten Anschluss, den man Vertrag nennt.

WAS SIE WISSEN MÜSSEN

- Eine Microkernel-Architektur (Plug-in-Architektur) hat ein minimales Kernsystem plus unabhängige Plug-ins, die nur über veröffentlichte Verträge angebunden und über eine Registry gefunden werden.
- Sie passt zu Produkten mit einem stabilen Ablauf und vielen Varianten bestimmter Schritte, etwa Steuerregeln je Land, Zahlungsmethoden, Datei-Importer oder Editor-Sprachen.
- Plug-ins sollten nur vom Vertrag abhängen, nie voneinander oder von Interna des Kerns, und der Kern sollte nie Sonderfälle für bestimmte Plug-ins enthalten.
- Isolation ist eine Abwägung: Prozessinterne Plug-ins sind schnell, teilen aber das Schicksal des Hosts; getrennte Prozesse, Sandboxes oder entfernte Aufrufe schützen den Kern, haben aber ihren Preis.
- Versionieren Sie den Vertrag separat, prüfen Sie die Kompatibilität beim Laden der Plug-ins und bevorzugen Sie additive Änderungen; nach Hyrums Gesetz wird alles, was Plug-ins beobachten können, Teil des Vertrags.

- Anwendungs-Microkernels zielen auf Erweiterbarkeit; Betriebssystem-Microkernels wie L4 oder QNX zielen auf Isolation und einen kleinen vertrauenswürdigen Kernel.

Definition und Ursprung

Eine Microkernel-Architektur, auch Plug-in-Architektur genannt, teilt eine Anwendung in zwei Arten von Teilen: ein Kernsystem, das die minimale, allgemeine Funktionalität enthält, die für den Betrieb nötig ist, und Plug-ins, die Verhalten hinzufügen oder ändern, ohne dass der Kern verändert wird. Plug-ins sind nur über veröffentlichte Verträge an den Kern angebunden, und der Kern findet sie über eine Registry. Mark Richards und Neal Ford führen sie in *Fundamentals of Software Architecture* als einen der klassischen Architekturstile, und sie passt zur Beschreibung eines Architekturstils in Abschnitt 5.1: eine benannte Menge von Einschränkungen für die Elemente (Kern, Plug-ins) und Konnektoren (Verträge) eines Systems.

Die Idee entstand in Produktsoftware: Werkzeugen, die von vielen Kunden installiert werden, von denen jeder etwas leicht anderes möchte. Statt für jeden Kunden eine eigene Version zu pflegen, liefert der Hersteller einen Kern aus und lässt Kunden, Partner oder die eigenen Teams Plug-ins hinzufügen. Integrierte Entwicklungsumgebungen, Webbrowser, Grafikprogramme und Buchhaltungspakete funktionieren alle so. Der Name ist von Betriebssystemen entlehnt, wo ein Microkernel ein sehr kleiner Kernel ist, außerhalb dessen alles andere läuft; die beiden Ideen sind verwandt, sollten aber nicht verwechselt werden, wie weiter unten erklärt.

Eine Microkernel-Anwendung ist normalerweise eine einzige Deployment-Einheit und gehört daher zu den monolithischen Stilen dieses Kapitels: eine Deployment-Einheit, ein Prozess, ein Satz betrieblicher Anliegen. Plug-ins können separat installiert werden, aber zur Laufzeit leben sie im Kern oder direkt daneben.

Der Aufbau eines Microkernel-Systems



Das Kernsystem besitzt den allgemeinen Ablauf und eine Registry; jedes Plug-in implementiert einen Vertrag, den der Kern veröffentlicht, und Plug-ins hängen nie voneinander ab.

So funktioniert es: Kern, Plug-ins, Registry und Verträge

Das Kernsystem enthält, was jede Installation braucht: den Hauptablauf, gemeinsame Dienste wie Logging oder Konfiguration und die Erweiterungspunkte, an denen Plug-ins andocken. Richards und Ford beschreiben es als den Happy Path durch die Anwendung, mit wenig oder keiner kundenspezifischen Verarbeitung; im Checkout einer Essenslieferplattform heißt das: einen Warenkorb bepreisen, die Zahlung entgegennehmen und die Bestellung bestätigen. Ein guter Kern ist bewusst langweilig. Jeder Sonderfall, der je nach Kunde, Land oder Produktlinie variiert, ist ein Kandidat dafür, aus ihm herausgelöst zu werden.

Plug-ins sind unabhängige, in sich geschlossene Module mit spezialisierter Verarbeitung. Sie sollten nur vom Vertrag abhängen, nicht voneinander und nicht von den internen Klassen des Kerns. Braucht ein Plug-in etwas, das ein anderes bereitstellt, sollte der Kern diesen Bedarf decken (als Dienst, den er über den Vertrag anbietet), oder die beiden Plug-ins sollten zusammengelegt werden; Abhängigkeiten zwischen Plug-ins verwandeln einen sauberen Stern unbemerkt in ein Knäuel.

Die Registry hält fest, welche Plug-ins verfügbar sind, wie jedes heißt, welche Version es hat, welchen Vertrag es implementiert und wie es erreichbar ist. Sie kann so einfach sein wie ein beim Start gefülltes Dictionary, ein Verzeichnis, das der Kern nach Manifesten durchsucht, Einträge in den Metadaten eines Pakets (wie bei Python-Entry Points), eine Datenbanktabelle

oder ein entfernter Dienst, der anderswo deployte Plug-ins auflistet. Die Registry ermöglicht es dem Kern, „Gib mir die Steuerregeln für Deutschland“ zu sagen, ohne zu wissen, dass eine Klasse namens `GermanTax` existiert.

Verträge legen fest, was ein Plug-in bereitstellen muss und worauf es sich verlassen darf: eine Schnittstelle mit Methodensignaturen, die Datenstrukturen, die sie überqueren, das Fehlerverhalten und eine Version. In Eclipse ist ein Vertrag ein Erweiterungspunkt, der in einem Plug-in-Manifest mit einem Schema deklariert wird; in Visual Studio Code ist es die Extension-API plus die Contribution Points, die in der `package.json` einer Extension aufgeführt sind; in Ihrem eigenen Code ist es meist eine Schnittstelle (oder ein Python-`Protocol`) in einem eigenen kleinen Paket. Stammt das Plug-in von einer anderen Organisation und kann es Ihrem Vertrag nicht folgen, übersetzt ein Adapter zwischen beiden, genau wie in der hexagonalen Architektur.

Plug-ins lassen sich auf drei Arten anbinden. **Punkt-zu-Punkt-Aufrufe im selben Prozess** sind der Normalfall: Der Kern ruft eine Methode des Plug-in-Objekts auf. Plug-ins in **getrennten Prozessen** laufen in einem eigenen Prozess und sprechen über Interprozessnachrichten mit dem Kern, was den Kern vor Abstürzen schützt. **Entfernte** Plug-ins sitzen hinter einer HTTP- oder Messaging-Schnittstelle; das erlaubt Plug-ins in anderen Sprachen oder im Besitz anderer Teams, bringt aber Netzwerklatenz und neue Ausfallarten mit sich, und das System ähnelt allmählich den verteilten Stilen aus Kapitel 6.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: ein Kern, der seine Plug-ins nie beim Namen nennt

Die schwierigste Idee dieses Abschnitts ist, wie ein Kern Plug-ins nutzen kann, ohne zu wissen, welche es gibt. Der Kern hängt nur vom Vertrag ab, und eine Registry teilt ihm zur Laufzeit mit, welche Plug-ins installiert sind. Verfolgen Sie, wie der Checkout der Essenslieferplattform ein Land hinzufügt.

- Heute berechnet der Checkout-Code die Steuer selbst, in Verzweigungen wie: Wenn das Land GB ist, wende die britischen Regeln an.
- Der Start in Deutschland bedeutet: den Checkout-Code öffnen, eine deutsche Verzweigung hinzufügen und dann den gesamten Checkout neu testen und ausliefern. Der Kern kennt jedes Land beim Namen und wächst mit jedem Markteintritt.
- Im Microkernel-Design veröffentlicht der Kern stattdessen einen Vertrag: eine kleine Schnittstelle, die festlegt, was jeder Satz Steuerregeln bereitstellen muss, nämlich eine Möglichkeit, die Steuer auf einen bepreisten Warenkorb zu berechnen, und genau welche Daten sie überqueren.
- Das deutsche Länderteam schreibt ein Plug-in, das den Vertrag implementiert und angibt, welches Land es bedient: DE. Die britischen Regeln wandern auf dieselbe Weise aus dem Checkout in ein GB-Plug-in, sodass der Checkout kein Land mehr beim Namen nennt.
- Beim Start findet die Registry die installierten Plug-ins und hält fest, welches Land jedes bedient. Wenn ein deutscher Kunde bezahlt, fragt der Kern die Registry nach den Steuerregeln für DE und ruft sie über den Vertrag auf, ohne zu wissen, welches Stück Code antwortet.
- Später die Vereinigten Staaten hinzuzufügen funktioniert genauso: ein neues Plug-in mit eigenen Tests, das von der Registry gefunden wird. Der Checkout-Code selbst ändert sich nicht.
- **Die Lehre:** Der Kern kennt den Vertrag und die Registry, nie die Plug-ins. Jede Zeile im Kern, die ein bestimmtes Plug-in nennt, ist ein Zeichen dafür, dass dem Vertrag etwas fehlt.

Echte Beispiele

Eclipse. Die Eclipse-IDE ist das Lehrbuchbeispiel. Ihr Kern ist kaum mehr als eine Workbench und eine Plug-in-Laufzeitumgebung (aufgebaut auf Equinox, einer Implementierung der OSGi-Framework-Spezifikation). Fast alles, was Entwicklungsteams sehen, von den Java-Werkzeugen bis zur Integration der Versionsverwaltung, ist ein Plug-in. Jedes Plug-in hat ein Manifest, das

deklariert, was es braucht, welche Erweiterungspunkte es anderen anbietet und in welche Erweiterungen es sich einklinkt. Plug-ins werden verzögert aktiviert, erst wenn tatsächlich etwas sie braucht, damit Hunderte installierter Plug-ins den Start nicht verlangsamen.

Visual Studio Code. Extensions deklarieren Aktivierungsereignisse (eine Markdown-Datei öffnen, einen Befehl ausführen), damit sie nur bei Bedarf geladen werden, und steuern über ihr Manifest Befehle, Sprachen, Themes und Ansichten bei. Extensions laufen in einem separaten Extension-Host-Prozess, sodass eine fehlerhafte Extension die Benutzeroberfläche des Editors nicht einfrieren kann. Das ist ein gutes Beispiel dafür, Prozessisolation für ein Plug-in-Ökosystem mit Tausenden externer Autoren zu wählen.

Webbrowser. Browser-Extensions nutzen einen Standardsatz von Extension-APIs und müssen die benötigten Berechtigungen (welche Websites sie lesen dürfen, ob sie Tabs oder Downloads sehen dürfen) in einem Manifest deklarieren. Der Browser setzt diese Berechtigungen durch, weil man den Autoren von Extensions nicht von vornherein vertrauen kann.

Business-Rules-Engines. Das Schadensystem eines Versicherers kann den Ablauf der Schadenbearbeitung im Kern halten und die Regeln jeder Rechtsordnung (was gedeckt ist, welche Unterlagen nötig sind, gesetzliche Fristen) in Plug-ins auslagern. Steuer-Engines, Lohnabrechnungssysteme und E-Commerce-Plattformen mit Plug-ins für Zahlungsmethoden folgen demselben Muster: ein stabiler Prozess mit Regeln oder Anbietern, die je nach Land, Produkt oder Kunde variieren.

Entwicklerwerkzeuge. Das Test-Framework pytest ist um die Plug-in-Bibliothek `pluggy` herum gebaut; viele seiner eigenen Funktionen sind als Plug-ins implementiert, mit denselben Hooks, die auch Dritte nutzen. Build-Tools, Linter und Static-Site-Generatoren machen es häufig genauso.

Anwendungs-Microkernel und Betriebssystem-Microkernel im Vergleich

Aspekt	Betriebssystem-Microkernel	Anwendungs-Microkernel (Plug-in-Architektur)
Was minimal ist	Der Kernel: Adressräume, Threads, Scheduling und Interprozesskommunikation	Das Kernsystem: der allgemeine Ablauf einer Anwendung
Was außerhalb des Kerns liegt	Gerätetreiber, Dateisysteme und Netzwerkstacks, die als separate Server im User Space laufen	Variables fachliches Verhalten oder Features, etwa Steuerregeln, Zahlungsmethoden oder Editor-Sprachen
Hauptziele	Zuverlässigkeit, Sicherheit und eine kleine Trusted Computing Base	Erweiterbarkeit, Anpassbarkeit und ein Ökosystem von Drittanbietern
Wie die Teile kommunizieren	Nachrichtenaustausch zwischen isolierten Prozessen	Meist prozessinterne Aufrufe über eine Schnittstelle; manchmal ein separater Prozess oder ein entfernter Aufruf
Beispiele	Mach, die L4-Familie (einschließlich des formal verifizierten seL4), QNX, MINIX 3	Eclipse, Visual Studio Code, Webbrowser, pytest, Regel-Engines für Versicherungen und Steuern

HINWEIS

Verwechseln Sie die beiden Microkernels nicht

Betriebssystem-Microkernels wie Mach und die L4-Familie verlagern Gerätetreiber und Dateisysteme aus dem Kernel in separate Prozesse im User Space, damit ein fehlerhafter Treiber nicht die ganze Maschine lahmlegen kann. Linux dagegen ist ein monolithischer Kernel mit ladbaren Modulen. Über die beiden Designs stritten Andrew Tanenbaum und Linus Torvalds 1992 berühmt im Usenet. Ein Anwendungs-Microkernel übernimmt die Form (kleiner Kern, alles andere außerhalb), doch sein übliches Ziel ist Erweiterbarkeit, nicht Isolation, und seine Plug-ins laufen normalerweise im selben Prozess wie der Kern.

Isolation, Versionierung und Kompatibilität von Plug-ins

Drei Fragen entscheiden, ob eine Plug-in-Architektur über Jahre gesund bleibt: wie isoliert Plug-ins sind, wie sich der Vertrag weiterentwickelt und wie Inkompatibilität erkannt wird.

Isolation. Prozessinterne Plug-ins teilen Speicher, Threads und Berechtigungen des Kerns. Das ist schnell und einfach, aber ein einziges Plug-in mit einer Endlosschleife, einem Speicherleck oder einer Sicherheitslücke betrifft alles. Die Optionen, nach zunehmender Isolation und zunehmenden Kosten geordnet, sind: prozessintern mit defensiven Aufrufen (Timeouts,

Abfangen von Exceptions je Plug-in, Ressourcenlimits); ein separater Prozess oder Worker je Plug-in oder Gruppe von Plug-ins; eine Sandbox wie eine WebAssembly-Laufzeitumgebung oder ein eingeschränkter Interpreter, die begrenzt, worauf Plug-in-Code zugreifen kann; und ein entfernter Dienst. Wählen Sie, indem Sie fragen, wer die Plug-ins schreibt. Die Steuerregeln Ihres eigenen Teams können prozessintern laufen; beliebiger Code aus einem öffentlichen Marktplatz sollte das nicht.

Versionierung. Der Vertrag ist eine API, und es gelten die Regeln aus Abschnitt 0.6 zu Breaking Changes. Versionieren Sie den Vertrag getrennt vom Kern, meist mit semantischer Versionierung: eine neue Major-Version für Breaking Changes, eine Minor-Version für abwärtskompatible Ergänzungen. Lassen Sie jedes Plug-in deklarieren, für welche Vertragsversion es gebaut wurde, und den Kern, welche Versionen er unterstützt. Additive Änderungen (neue optionale Felder, neue optionale Hooks) halten alte Plug-ins lauffähig; ist eine Breaking Change unvermeidlich, unterstützen Sie die alte und die neue Major-Version für eine veröffentlichte Deprecation-Frist parallel.

Kompatibilitätsprüfungen. Prüfen Sie die Kompatibilität beim Laden der Plug-ins, nicht erst, wenn ein Kunde sie zum ersten Mal auslöst. Eine Registry, die ein inkompatibles Plug-in beim Start mit einer klaren Log-Meldung ablehnt, macht aus einem Laufzeitfehler einen fehlgeschlagenen Deployment-Schritt. Die Beobachtung von Hyrum Wright, bekannt als Hyrums Gesetz, sollte man hier im Kopf behalten: Bei genügend Nutzern wird sich irgendjemand auf jedes beobachtbare Verhalten einer Schnittstelle verlassen, ganz gleich, was in der Dokumentation steht. Halten Sie den Vertrag klein, halten Sie die Interna des Kerns außer Reichweite (Eclipse kennzeichnet Nicht-API-Pakete aus diesem Grund als `internal`) und behandeln Sie alles, was für Plug-ins sichtbar ist, als Teil des Vertrags.

Merkmale des Microkernel-Stils

Merkmal	Bewertung	Begründung
Erweiterbarkeit und Änderbarkeit (in den geplanten Richtungen)	Hoch	Neues Verhalten kommt als neues Plug-in hinzu; der Kern und die anderen Plug-ins ändern sich nicht.
Einfachheit	Mittel	Die Idee ist einfach, doch Verträge, eine Registry, das Laden und die Versionsprüfungen sind echte Mechanik.
Testbarkeit	Mittel bis hoch	Jedes Plug-in lässt sich allein gegen den Vertrag testen; der Kern lässt sich mit Fake-Plug-ins testen. Kombinationen müssen trotzdem getestet werden.
Deploybarkeit	Mittel	Plug-ins lassen sich oft hinzufügen, ohne den Kern neu zu deployen; eine Vertragsänderung erzwingt koordinierte Releases.
Performance	Mittel bis hoch	Prozessinterne Plug-ins sind gewöhnliche Funktionsaufrufe; Isolation per Prozess oder Sandbox verursacht Overhead.
Skalierbarkeit und Elastizität	Niedrig	Meist ist es eine einzige Deployment-Einheit und skaliert daher als Ganzes, wie jeder Monolith.
Fehlertoleranz	Niedrig bis mittel	Ein prozessinternes Plug-in kann den Host zum Absturz bringen oder verlangsamen; Prozessisolation verbessert das zu einem Preis.
Gesamtkosten	Hoch (günstig)	Eine Codebasis und eine Laufzeitumgebung im Betrieb; die Hauptkosten liegen im Entwerfen und Weiterentwickeln der Verträge.

Wann man ihn einsetzt und wann nicht

Gute Einsatzfälle. Produktsoftware, die viele Kunden installieren und anpassen. Systeme mit einem stabilen Ablauf, aber vielen Varianten eines Schritts: Steuern und Lohnabrechnung je Land, Versicherungsregeln je Rechtsordnung, Zahlungsmethoden, Importer für Dateiformate, Versanddienstleister. Plattformen, die ein Ökosystem von Drittanbietern wollen. Systeme, in denen eine Aufsichtsbehörde oder ein Kunde häufig Regeln ändert und Sie diese Änderungen als kleine, separat getestete Releases haben wollen. Der Stil funktioniert auch innerhalb eines größeren Systems: Ein Service in einer verteilten Architektur kann selbst ein Microkernel sein.

Schlechte Einsatzfälle. Systeme, bei denen Sie noch nicht wissen, was variiert. Im Voraus für erdachte Bedürfnisse entworfene Erweiterungspunkte sind spekulative Allgemeinheit: Sie fügen Mechanik hinzu, bremsen das Team und sitzen meist an der falschen Stelle, wenn der echte Bedarf kommt (siehe die Diskussion der Umkehrbarkeit in Abschnitt 1.6). Systeme, die

sehr hohe Skalierbarkeit oder Fehlerisolation brauchen, da das Ganze als eine Einheit skaliert und ausfällt. Systeme, in denen Plug-ins eng miteinander koordiniert werden müssten, was den Hauptnutzen des Stils zunichtemacht.

Eine praktische Regel: Warten Sie, bis Sie dieselbe Art von Variation zwei- oder dreimal gebaut haben, und leiten Sie den Vertrag dann aus den echten Fällen ab. Ein Vertrag, der aus drei funktionierenden Steuerimplementierungen abgeleitet ist, wird weit besser sein als einer, der vor der ersten geraten wurde.

ACHTUNG

Häufige Anti-Patterns

- **Der wissende Kern.** Der Kern enthält `if plugin_name == "stripe"` oder Sonderfälle für bestimmte Plug-ins. Jeder davon ist ein Zeichen dafür, dass dem Vertrag etwas fehlt.
- **Fetter Kern.** Variable Regeln schleichen sich zurück in den Kern, weil das schneller geht, als den Vertrag zu erweitern, bis der Kern wieder der alte Monolith ist.
- **Plug-in-Knäuel.** Plug-ins importieren einander oder teilen versteckten Zustand, sodass sie sich nicht mehr unabhängig installieren, testen oder entfernen lassen.
- **Undichter Vertrag.** Der Vertrag reicht Domänenobjekte des Kerns oder Datenbanksitzungen an Plug-ins weiter, sodass jede interne Änderung zu einer Breaking Change für jedes Plug-in wird.
- **Unversionierter Vertrag.** Niemand kann sagen, welche Plug-ins brechen, wenn sich der Kern ändert, sodass Upgrades zu gefürchteten Alles-auf-einmal-Ereignissen werden.
- **Ein Plug-in pro Kunde.** Plug-ins werden zu einem Mittel, Hunderte von Kunden-Forks zu pflegen. Kleine Unterschiede zwischen Kunden sollten durch Konfiguration ausgedrückt werden, nicht durch Code.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: einen Plug-in-Vertrag für länderspezifische Steuerregeln entwerfen

Die Essenslieferplattform, die sich durch diesen Kurs zieht, startet nach dem Vereinigten Königreich in Deutschland und den Vereinigten Staaten. Jedes Land besteuert Speisen, Alkohol und Liefergebühren anders, und die US-Sales-Tax variiert zusätzlich nach Bundesstaat und Stadt. Der Checkout-Code enthält bereits Verzweigungen der Form `if country == "GB"`. Der Architekt schlägt einen Microkernel vor: Der Checkout ist der Kern, und die Regeln jedes Landes sind ein Plug-in. Dies sind die Entscheidungen in der Reihenfolge, in der sie getroffen wurden.

1. Den Variationspunkt benennen	Nur die Berechnung der Steuer auf einen bepreisten Warenkorb variiert je Land. Preisgestaltung, Rabatte, Zahlung und Belege bleiben im Kern. Ein schmaler Variationspunkt hält den Vertrag klein.
2. Festlegen, was den Vertrag überquert	Eingaben: Warenkorbpositionen mit einer neutralen Kategorie (`hot-food` , `alcohol` , `delivery-fee`), Nettobeträge als `Decimal` und eine optionale Region. Ausgabe: die Gesamtsteuer und eine beschriftete Aufschlüsselung für den Beleg. Es werden keine Kernobjekte (Bestellungen, Kunden, Datenbanksitzungen) übergeben, sodass Plug-ins nicht von Interna des Kerns abhängen können.
3. Den Vertrag zu einem eigenen Paket machen	`tax_contract` enthält nur Datenklassen und das Protokoll `TaxRules` und wird mit semantischer Versionierung versioniert. Kern und Plug-ins hängen beide davon ab; keiner hängt vom anderen ab.
4. Kompatibilität definieren	Ein Plug-in deklariert die Vertragsversion, die es braucht, etwa (2, 1). Der Kern akzeptiert es, wenn die Major-Version übereinstimmt und die Minor-Version des Kerns mindestens so hoch ist. Ein optionales Feld hinzuzufügen ist eine Minor-Änderung; eines umzubenennen oder zu entfernen ist eine Major-Änderung.
5. Die Erkennung wählen	Plug-ins sind gewöhnliche Python-Pakete, die sich unter der Entry-Point-Gruppe `fooddelivery.tax_rules` registrieren. Ein Paket zu installieren genügt; kein Kerncode listet die Länder auf.
6. Das Fehlerverhalten festlegen	Ist für ein Land kein Plug-in registriert, lehnt der Checkout die Bestellung mit einem klaren Fehler ab, statt null Steuern zu berechnen. Ein Plug-in, das eine Exception auslöst, lässt nur diesen einen Bezahlvorgang scheitern, und der Fehler wird mit Name und Version des Plug-ins gemeldet.
7. Die Isolation festlegen	Die Plug-ins werden von den eigenen Länderteams des Unternehmens geschrieben und gelten als vertrauenswürdig, daher laufen sie wegen der Geschwindigkeit prozessintern. Dürften externe Steueranbieter Plug-ins liefern, würde das Design sie hinter einen entfernten Aufruf mit Timeout verlagern.

8. Die Tests festlegen	Eine gemeinsame Contract-Test-Suite (Rundung, leere Warenkörbe, negative Erstattungspositionen, unbekannte Kategorien) läuft gegen jedes Plug-in in dessen eigener Pipeline. Jedes Länderteam ergänzt eigene Fälle aus veröffentlichten Steuerleitfäden.

Der Kern ändert sich nicht mehr, wenn ein Land hinzukommt; der Start in einem vierten Land ist also ein neues Paket plus Tests, keine Änderung am Checkout.

Die schwierigste Entscheidung ist Schritt 2. Eine zu grobe Kategorienliste zwingt Plug-ins zum Raten; eine zu detaillierte koppelt den Vertrag an das Recht eines einzelnen Landes. Rechnen Sie damit, sie zu überarbeiten, und planen Sie dann die Migration auf eine neue Major-Version.

Die Registry weist inkompatible Plug-ins beim Start zurück, sodass eine Versionsabweichung in einer Deployment-Prüfung auffällt und nicht als falsche Steuer auf dem Beleg eines Kunden.

```
# package: tax_contract  (version 2.1.0)
from dataclasses import dataclass
from decimal import Decimal
from typing import Protocol

CONTRACT_VERSION = (2, 1)

@dataclass(frozen=True)
class TaxLine:
    sku: str
    category: str      # "hot-food", "cold-food", "alcohol", "delivery-fee"
    net: Decimal       # never float for money

@dataclass(frozen=True)
class TaxResult:
    total_tax: Decimal
    breakdown: dict[str, Decimal]   # label shown on the receipt

class TaxRules(Protocol):
    country: str                    # ISO 3166-1 alpha-2, e.g. "DE"
    requires_contract: tuple[int, int]
    def calculate(self, lines: list[TaxLine], region: str | None) → TaxResult: ...
```

Das Vertragspaket, versioniert und veröffentlicht getrennt vom Kern und von den Plug-ins.

```
from importlib.metadata import entry_points
from tax_contract import CONTRACT_VERSION

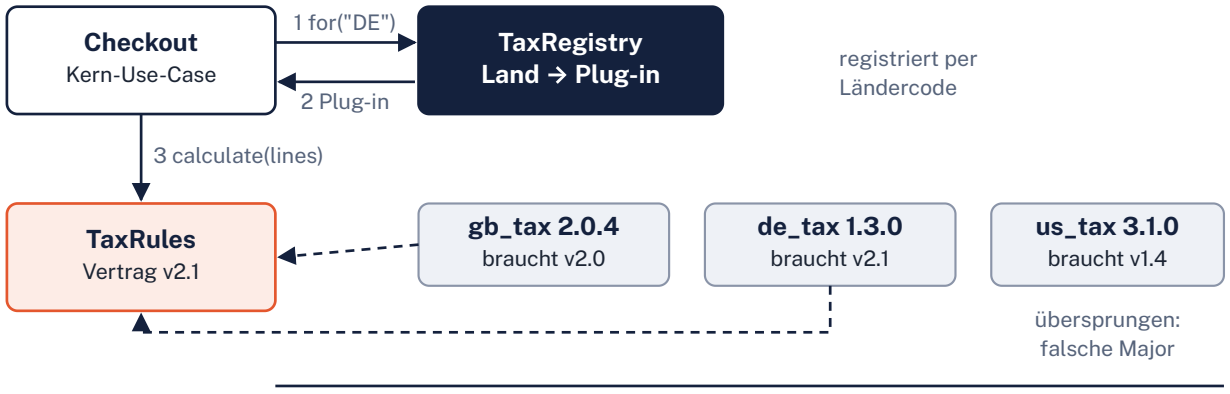
class UnsupportedCountry(Exception): pass

class TaxRegistry:
    def __init__(self, log):
        self._by_country = {}
        for ep in entry_points(group="fooddelivery.tax_rules"):
            plugin = ep.load()()
            major, minor = plugin.requires_contract
            if major ≠ CONTRACT_VERSION[0] or minor > CONTRACT_VERSION[1]:
                log.warning("skipping %s: needs contract %s", ep.name,
plugin.requires_contract)
                continue
            self._by_country[plugin.country] = plugin

    def for_country(self, country):
        if country not in self._by_country:
            raise UnsupportedCountry(country)   # fail loudly, never guess a rate
        return self._by_country[country]
```

Die Registry im Kern: erkennt installierte Plug-ins, prüft die Vertragskompatibilität und schlägt sie nach Land nach.

Die Steuer-Plug-ins zur Laufzeit



Laden: Beim Start liest die Registry installierte Pakete (Entry Points), prüft für jedes Plug-in die benötigte Vertragsversion und registriert es nach Land; inkompatible werden protokolliert und übersprungen, statt den Checkout lahmzulegen.

Der Checkout fragt die Registry nach dem für ein Land registrierten Plug-in und ruft es über den Vertrag auf; ein Plug-in, das für eine inkompatible Vertragsversion gebaut ist, wird beim Start übersprungen.

Den Stil im Code durchsetzen

Eine Plug-in-Architektur bleibt nur dann eine, wenn ihre Regeln automatisch geprüft werden. Die Struktur, die in den meisten Sprachen funktioniert, hat drei Arten von Paketen: ein kleines **Vertragspaket** nur mit Schnittstellen und Datentypen; den **Kern**, der vom Vertrag abhängt; und ein Paket pro **Plug-in**, das jeweils vom Vertrag abhängt und von nichts anderem aus Ihrem Code.

Setzen Sie die Abhängigkeitsregeln dann mit Werkzeugen der Art durch, die Abschnitt 3.4 behandelt: In Python kann import-linter dem Kern verbieten, irgendein Plug-in-Paket zu importieren, und Plug-ins verbieten, einander oder den Kern zu importieren; in Java leisten ArchUnit-Tests oder Modulsysteme (JPMS, OSGi) dasselbe. Lassen Sie den Build fehlschlagen, wenn eine Regel verletzt wird.

Ergänzen Sie eine **Contract-Test-Suite**: eine Reihe von Tests, die einmal gegen den Vertrag geschrieben werden und gegen jedes Plug-in in dessen eigener Pipeline laufen. Sie prüft das Verhalten, das der Vertrag zusagt (Rundung, leere Eingaben, Fehlertypen), nicht das Steuerrecht jedes Landes, das die eigenen Tests des Plug-ins abdecken. Fügen Sie schließlich eine Startprüfung oder einen Smoke-Test beim Deployment hinzu, der jedes installierte Plug-in lädt und seine Vertragsversion verifiziert, damit Inkompatibilitäten auffallen, bevor Nutzer sie finden.

Häufige Fragen

Ist ein Microkernel nur das Strategy-Muster?
Auf Code-Ebene ist das Strategy-Muster derselbe Mechanismus: zur Laufzeit eine Implementierung einer Schnittstelle wählen. Der Microkernel ist die architektonische Variante: Plug-ins werden separat gebaut, versioniert und oft separat installiert, eine Registry erkennt sie, und der Vertrag ist eine veröffentlichte API mit Kompatibilitätsregeln.

Können Plug-ins eigene Daten haben?
Ja, und oft sollten sie das, aber behandeln Sie sie wie die modul-eigenen Daten eines modularen Monolithen: Die Tabellen oder Dateien eines Plug-ins gehören ihm, und der Kern erreicht sie nur über den Vertrag.

Sollte Konfiguration Plug-ins ersetzen?
Für kleine, datenartige Unterschiede (ein Satz, ein Schwellenwert, eine Beschriftung) ja: Konfiguration ist günstiger und sicherer. Plug-ins sind für Unterschiede gedacht, die Logik brauchen. Viele Systeme nutzen beides: ein Plug-in pro Land plus Konfiguration für die aktuellen Sätze jedes Landes.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Welche drei Dinge muss eine Plug-in-Architektur neben dem Kern selbst festlegen?

Die Plug-ins, die Verträge, die sie implementieren, und eine Registry, mit der der Kern Plug-ins erkennen und nachschlagen kann, ohne sie im Code zu benennen.

▼ Warum sollten Plug-ins nicht voneinander abhängen?

Weil sie sich dann nicht mehr unabhängig installieren, entfernen, testen und versionieren lassen, was der Hauptnutzen des Stils ist; gemeinsame Bedürfnisse gehören in den Kern oder den Vertrag.

▼ Warum führt Visual Studio Code Extensions in einem separaten Prozess aus?

Damit eine langsame oder abstürzende Extension die Benutzeroberfläche des Editors nicht einfrieren oder abstürzen lassen kann: Es tauscht etwas Performance und Komplexität gegen Isolation von Tausenden externer Autoren.

▼ Wann ist Konfiguration besser als ein Plug-in?

Wenn der Unterschied zwischen Varianten aus Daten besteht, etwa einem Satz, einem Limit oder einer Beschriftung, statt aus Logik. Konfiguration ist einfacher und sicherer zu ändern.

▼ Probieren Sie es aus: Der Bezahlvorgang eines Ticketsystems nutzt Plug-ins für Zahlungsarten. Der Kern enthält if plugin == "voucher", und das Gutschein-Plug-in liest die Tabelle bookings des Kerns und ruft das Karten-Plug-in direkt auf. Kritisieren Sie das, und sagen Sie, was der Vertrag leisten sollte.

Das verletzt die Regeln des Stils. Der Kern darf keine Sonderfälle für benannte Plug-ins enthalten, und Plug-ins dürfen nur vom Vertrag abhängen, nie von Interna des Kerns oder voneinander. Definieren Sie einen versionierten Vertrag, etwa can_pay und take_payment, der die Buchungsdetails übergibt, die das Plug-in braucht. Verschieben Sie die Gutscheinlogik dahinter, und lassen Sie den Kern für geteilte Zahlungen Methoden verketten. Alles, was Plug-ins beobachten können, wird Teil des Vertrags, halten Sie ihn also klein.

QUELLEN

- Azad Bolour: Notes on the Eclipse Plug-in Architecture (eclipse.org): https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html
- Visual Studio Code: Extension Host: https://code.visualstudio.com/api/advanced-topics/extension-host

- Python Packaging User Guide: Entry points specification: https://packaging.python.org/en/latest/specifications/entry-points/
- pluggy: the plug-in system behind pytest: https://pluggy.readthedocs.io/en/latest/
- Semantic Versioning 2.0.0: https://semver.org/

Hexagonale, Onion- und Clean Architecture

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 2.6 Änderbarkeit und Deploybarkeit
- 3.5 SOLID im Großen
- 5.2 Schichten

Begriffe von früher, die er verwendet: Abhängigkeitsumkehr, Test Double, Schicht, Schnittstelle, Framework

DIE IDEE IN KÜRZE

Denken Sie an einen Reiseadapter. Ihr Laptop-Ladegerät hat eine Steckerform; jedes Land hat seine eigene Steckdose. Sie bauen den Laptop nicht für jedes Land um: Sie haben kleine Adapter dabei, die auf der einen Seite in die Steckdose und auf der anderen zu Ihrem Stecker passen. Hexagonale und Clean Architecture bauen Software auf dieselbe Weise. Der wichtige Teil, die Geschäftsregeln, hat eigene Standardsteckdosen (Ports genannt). Datenbanken, Websites, Zahlungsanbieter und Testwerkzeuge werden über Adapter angeschlossen, sodass sich jedes davon austauschen lässt, ohne die Regeln anzufassen.

WAS SIE WISSEN MÜSSEN

- Hexagonale Architektur (Ports and Adapters, Cockburn, 2005), Onion-Architektur (Palermo, 2008) und Clean Architecture (Martin, Blog 2012, Buch 2017) teilen eine Idee: Geschäftslogik im Zentrum, unabhängig von Frameworks und Infrastruktur, mit nach innen zeigenden Abhängigkeiten.
- Ein Port ist eine Schnittstelle, die der Anwendung gehört; ein Adapter übersetzt zwischen einem Port und einer Technologie. Treibende Adapter rufen die Anwendung auf; getriebene Adapter implementieren Schnittstellen, die die Anwendung aufruft.
- Die Abhängigkeitsregel besagt, dass innerer Code nie etwas aus einem äußeren Ring benennen darf; die Abhängigkeitsumkehr lässt den Kontrollfluss nach außen gehen, während Quellcode-Abhängigkeiten nach innen zeigen.
- Der Hauptnutzen sind schnelle, isolierte Tests der Geschäftsregeln und die Möglichkeit, Infrastruktur wie Zahlungsanbieter durch einen neuen Adapter zu ersetzen.
- Die Kosten sind zusätzliche Schnittstellen, Mapping-Code und Indirektion, die sich für einfache CRUD-Anwendungen oder kurzlebige Werkzeuge nicht lohnen.

- Setzen Sie die Regel automatisch durch, etwa mit import-linter, der onionArchitecture-Regel von ArchUnit oder getrennten Build-Modulen.

Eine Idee, drei Namen

Hexagonale Architektur, Onion-Architektur und Clean Architecture sind drei Formulierungen einer Idee: Die Geschäftslogik kommt ins Zentrum, wird unabhängig von Benutzeroberflächen, Datenbanken, Frameworks und externen Diensten gemacht, und alle Quellcode-Abhängigkeiten zeigen auf sie. Sie beschreiben die innere Struktur einer Anwendung und lassen sich daher mit jedem Deployment-Stil kombinieren: einem einzelnen Monolithen, einem modularen Monolithen, dessen Module jeweils so gebaut sind, oder einem Microservice.

Sie entstanden aus Frust über die klassische Schichtenarchitektur. Im Schichtenstil aus Abschnitt 5.2 hängt die Geschäftsschicht von der Persistenzschicht ab, die von der Datenbank abhängt, sodass Geschäftsregeln am Ende von Tabellen und objektrelationalem Mapping geprägt sind und das Testen einer Regel das Starten einer Datenbank bedeutet. Die Stilfamilie in diesem Abschnitt kehrt diese Abhängigkeit mit dem Prinzip der Abhängigkeitsumkehr um: Der Geschäftscode deklariert die Schnittstellen, die er braucht, und der Infrastrukturcode implementiert sie. Das Prinzip selbst behandelt Abschnitt 3.5; dieser Abschnitt zeigt die darauf aufbauenden Architekturen.

Hexagonale Architektur: Ports and Adapters

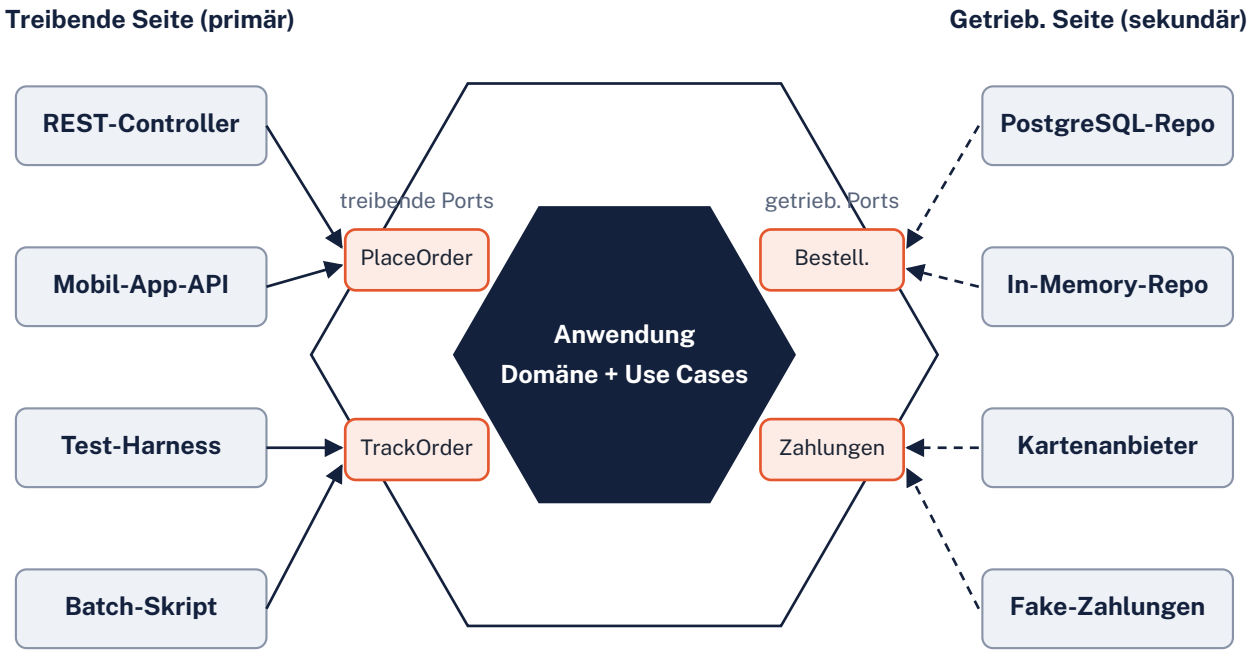
Alistair Cockburn beschrieb die Idee zuerst im Wiki des Portland Pattern Repository und verfasste 2005 einen Artikel darüber unter dem anschaulicheren Namen Ports and Adapters. Seine erklärte Absicht war, „eine Anwendung gleichermaßen von Benutzern, Programmen, automatisierten Tests oder Batch-Skripten steuern zu lassen“ und sie isoliert von ihren späteren Datenbanken und Geräten zu entwickeln und zu testen. 2024 veröffentlichten er und Juan Manuel Garrido de Paz ein kurzes Buch, *Hexagonal Architecture Explained*, das das Muster neu darlegt.

Ein Port ist ein zweckgerichtetes Gespräch zwischen der Anwendung und der Außenwelt, ausgedrückt als Schnittstelle: „eine Bestellung aufgeben“, „Bestellungen speichern“, „eine Karte belasten“. Ein Adapter ist der Code, der zwischen einem Port und einer bestimmten Technologie übersetzt: ein HTTP-Controller, ein SQL-Repository, das SDK eines Zahlungsanbieters, ein Kommandozeilenskript, ein In-Memory-Fake. Ein Port kann viele Adapter haben. Die Anwendung im Hexagon kennt nur ihre Ports.

Das Hexagon hat als Form keine Bedeutung; Cockburn wählte es, um dem Bild der Schichten von oben nach unten zu entkommen und Platz zu lassen, auf jeder Seite mehrere Ports zu zeichnen. Entscheidend sind die Grenze zwischen innen und außen und die Asymmetrie

zwischen zwei Arten von Ports.

Treibende und getriebene Ports



Durchgezogen: Adapter ruft Port auf. Gestrichelt: Adapter implementiert Port. Alle Abhängigkeiten zeigen nach innen.

Treibende Adapter rufen die Anwendung über treibende Ports auf; die Anwendung ruft getriebene Ports auf, die getriebene Adapter implementieren, sodass Datenbank und Zahlungsanbieter austauschbare Plug-ins sind.

Treibende und getriebene Seite

Treibende (primäre) Ports und Adapter sitzen im üblichen Bild links. Ein treibender Adapter nimmt eine Anfrage eines primären Akteurs entgegen (der Browser eines Nutzers, eine Mobil-App, ein anderes System, ein Test, ein geplanter Job) und ruft einen treibenden Port auf, der die eigene API der Anwendung ist: in der Praxis ein Use Case wie `PlaceOrder`. Der Adapter macht aus HTTP, einer Nachricht oder einem Kommandozeilenargument einen Aufruf mit Domänenwerten und aus dem Ergebnis wieder eine Antwort.

Getriebene (sekundäre) Ports und Adapter sitzen rechts. Die Anwendung ruft sie auf, wenn sie etwas von der Außenwelt braucht: eine Bestellung speichern, eine Karte belasten, eine Benachrichtigung senden, die Uhrzeit lesen. Der Port ist eine Schnittstelle, die der Anwendung gehört und in ihren Begriffen geschrieben ist (`OrderRepository.save(order)`), und ein getriebener Adapter implementiert sie für eine Technologie (PostgreSQL, die API eines Kartenanbieters, ein SMS-Gateway). Hier geschieht die Abhängigkeitsumkehr: Zur Laufzeit ruft die Anwendung den Adapter auf, aber im Quellcode hängt der Adapter von der Schnittstelle der Anwendung ab, nicht umgekehrt.

Cockburns Originaltext spricht von primären und sekundären Akteuren; die Begriffe treibend und getrieben sind heute verbreiteter, und er verwendet sie selbst. Die Unterscheidung ist wichtig für das Testen: Treibende Ports werden von Test-Harnesses angesprochen, die als Nutzer auftreten, und getriebene Ports werden durch Test Doubles ersetzt. Sie ist auch für Änderungen wichtig: Neue Kanäle (ein Sprachassistent, eine Partner-API) kommen als neue treibende Adapter hinzu, neue Infrastruktur als neue getriebene Adapter.

Irgendjemand muss die Adapter erzeugen und in die Anwendung einstecken. Das geschieht in einem einzigen Composition Root, typischerweise dem Startcode des Programms, dem einzigen Teil des Systems, der jeden Adapter kennt. Ein Dependency-Injection-Framework kann diese Arbeit übernehmen, aber einfache Konstruktoraufrufe in `main` sind oft klarer.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: hineinrufen und hinausrufen

Die Idee, die hier die meiste Sorgfalt verlangt, ist der Unterschied zwischen den beiden Seiten des Hexagons. Auf der treibenden Seite ruft ein Adapter die Anwendung auf; auf der getriebenen Seite ruft die Anwendung einen Adapter auf. Trotzdem zeigt im Quellcode jede Abhängigkeit nach innen. Abschnitt 3.5 hat die Abhängigkeitsumkehr eingeführt; verfolgen Sie eine Bestellung, um zu sehen, wo sie geschieht.

- Ein Kunde gibt eine Bestellung auf. Der REST-Controller, ein treibender Adapter, empfängt die HTTP-Anfrage, wandelt sie in einfache Werte wie Bestell-ID und Gesamtbetrag um und ruft den Use Case PlaceOrder auf. Der Aufruf geht nach innen.
- PlaceOrder speichert zuerst die Bestellung und muss dann die Karte autorisieren. Er ruft authorise auf PaymentGateway auf, einer Schnittstelle, die die Anwendung selbst deklariert, in ihren eigenen Worten. Er hat keine Ahnung, welcher Zahlungsanbieter dahintersteht.
- Der Adapter des Kartenanbieters außerhalb des Hexagons ist ein getriebener Adapter: Er implementiert PaymentGateway mit dem eigenen Toolkit des Anbieters. Zur Laufzeit geht der Aufruf nach außen, vom Use Case zum Adapter, aber im Quellcode verweist der Adapter auf die Schnittstelle der Anwendung, nie umgekehrt.
- Das Speichern funktioniert genauso. PlaceOrder ruft save auf OrderRepository auf; PostgresOrderRepository implementiert es mit SQL, und eine In-Memory-Version implementiert denselben Port für Tests.
- Irgendetwas muss die Adapter auswählen. Das geschieht einmal, im Composition Root beim Start, der die Adapter für PostgreSQL und den Kartenanbieter erzeugt und an PlaceOrder übergibt. Ein Test macht dasselbe mit In-Memory- und Fake-Adaptern.
- **Die Lehre:** Auf der treibenden Seite ruft das Äußere hinein; auf der getriebenen Seite ruft das Innere hinaus, über eine Schnittstelle, die ihm gehört. So oder so nennt die Anwendung nie eine Technologie, und genau das macht Adapter austauschbar und testbar.

Onion-Architektur

Jeffrey Palermo beschrieb die Onion-Architektur in einer Reihe von Blogbeiträgen ab Juli 2008, gedacht für langlebige Geschäftsanwendungen mit umfangreichem Verhalten (für kleine Websites riet er davon ab). Er zeichnete konzentrische Ringe: das Domänenmodell im Zentrum, Domain Services darum, Application Services um diese und einen äußeren Ring mit Benutzeroberfläche, Infrastruktur (einschließlich der Datenbank) und Tests. Seine zentralen Grundsätze sind, dass die Anwendung um ein unabhängiges Objektmodell herum gebaut wird,

dass innere Schichten Schnittstellen definieren, die äußere Schichten implementieren, dass alle Kopplung zum Zentrum hin verläuft und dass der Anwendungskern getrennt von der Infrastruktur kompiliert und ausgeführt werden kann.

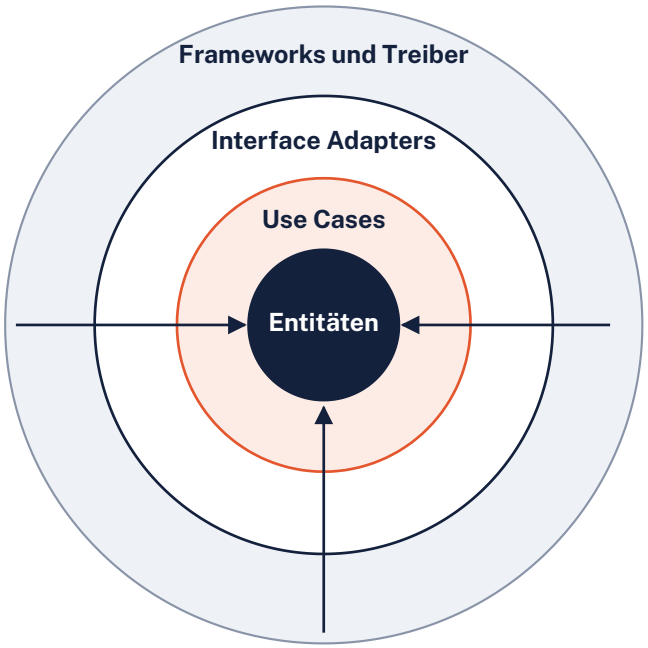
Im Vergleich zur hexagonalen Architektur fügt Onion benannte Ringe innerhalb der Anwendung hinzu und macht einen Punkt explizit, den Cockburns Bild offenlässt: Die Datenbank ist nicht das Fundament des Systems, sondern ein externer Dienst am Rand. Repository-Schnittstellen liegen im Kern; ihre Implementierungen liegen im Infrastrukturring.

Clean Architecture und die Abhängigkeitsregel

Robert C. Martin veröffentlichte „The Clean Architecture“ im August 2012 in seinem Blog als Synthese aus hexagonaler Architektur, Onion-Architektur und anderen ähnlichen Vorschlägen und baute es 2017 zum Buch *Clean Architecture* aus. Sein Bild hat vier Ringe. **Entitäten** enthalten unternehmensweite Geschäftsregeln. **Use Cases** enthalten anwendungsspezifische Regeln, die Entitäten orchestrieren. **Interface Adapters** wandeln Daten zwischen der Form, die die Use Cases brauchen, und der Form, die externe Instanzen brauchen: Controller, Presenter und Gateways. **Frameworks und Treiber** bilden den äußeren Ring: das Web-Framework, die Datenbank, die UI.

Die Abhängigkeitsregel hält alles zusammen: Quellcode-Abhängigkeiten zeigen nur nach innen, und nichts in einem inneren Ring darf etwas benennen, das in einem äußeren Ring deklariert ist, sei es eine Klasse, eine Funktion oder ein Datenformat. Muss die Steuerung nach außen fließen (ein Use Case speichert eine Bestellung), deklariert der innere Ring eine Schnittstelle, und der äußere Ring implementiert sie. Daten, die eine Grenze überqueren, sollten einfache Strukturen in den Begriffen des inneren Rings sein, nie eine Datenbankzeile oder ein HTTP-Request-Objekt. Martin argumentiert außerdem, dass die oberste Struktur den Zweck des Systems (Bestellungen, Zahlungen, Lieferungen) zeigen sollte statt seines Frameworks, eine Idee, die er Screaming Architecture nennt.

Die Ringe der Clean Architecture



Äußerer Ring: Web-Framework, ORM, Datenbank, Message Broker, UI

Interface Adapters: Controller, Presenter, Repository-Implementierungen

Use Cases: anwendungsspezifische Regeln, z. B. PlaceOrder

Entitäten: unternehmensweite Regeln, z. B. Order, Money

Die Abhängigkeitsregel: Quellcode-Abhängigkeiten nur nach innen

Jeder Ring darf nur von Ringen in seinem Inneren abhängen; Entitäten hängen von nichts ab, und Frameworks und Treiber sitzen am Rand, wo sie sich ersetzen lassen.

Hexagonale, Onion- und Clean Architecture im Vergleich

Aspekt	Hexagonal (Ports and Adapters)	Onion	Clean
Urheber und Datum	Alistair Cockburn; 2005 als Artikel veröffentlicht	Jeffrey Palermo; Blogreihe ab Juli 2008	Robert C. Martin; Blogbeitrag August 2012, Buch 2017
Bild	Ein Hexagon mit einem Innen (Anwendung) und einem Außen (Adapter), links treibend, rechts getrieben	Konzentrische Ringe: Domänenmodell, Domain Services, Application Services, dann UI, Infrastruktur und Tests	Konzentrische Ringe: Entitäten, Use Cases, Interface Adapters, Frameworks und Treiber
Zentrale Regel	Die Anwendung spricht mit der Welt nur über Ports; Adapter übersetzen	Alle Kopplung verläuft zum Zentrum hin; der Kern definiert Schnittstellen, die der äußere Ring implementiert	Die Abhängigkeitsregel: Quellcode-Abhängigkeiten zeigen nur nach innen
Vorgegebene innere Struktur	Keine: nur innen gegenüber außen	Ja: benannte innere Ringe für die Domäne	Ja: Entitäten und Use Cases, dazu Hinweise zu Presentern und zu Daten, die Grenzen überqueren
Hauptschwerpunkt	Symmetrie: Nutzer, Tests, Skripte und andere Programme können die Anwendung gleichermaßen steuern	Das Domänenmodell statt der Datenbank ins Zentrum einer Geschäftsanwendung stellen	Unabhängigkeit von Frameworks, UI, Datenbank und externen Instanzen; testbare Geschäftsregeln
Typisches Vokabular	Port, Adapter, treibend (primär), getrieben (sekundär), Akteur	Domänenmodell, Domain Services, Application Services, Infrastruktur	Entität, Use Case (Interactor), Gateway, Presenter, Controller

Ein Use Case mit einem Port und zwei Adaptern

Das folgende Python-Beispiel implementiert den Use Case `PlaceOrder` der Essenslieferplattform auf hexagonale Weise. Das erste Snippet ist das Innere: eine kleine Domänenklasse, zwei getriebene Ports, geschrieben als `typing.Protocol`-Klassen, und der Use Case, der als treibender Port dient. Das zweite Snippet ist das Äußere: ein PostgreSQL-Adapter und ein In-Memory-Test-Double, die denselben Port implementieren. Das dritte zeigt, warum das wichtig ist: ein schneller Unit-Test und der Composition Root, in dem die Produktionsadapter eingesteckt werden.

```
from dataclasses import dataclass
from decimal import Decimal
from typing import Protocol

@dataclass
class Order:
    id: str
    total: Decimal
    status: str = "pending-payment"

class OrderRepository(Protocol):        # driven port
    def save(self, order: Order) → None: ...

class PaymentGateway(Protocol):         # driven port
    def authorise(self, order_id: str, amount: Decimal,
                  idempotency_key: str) → bool: ...

class PlaceOrder:                       # driving port + use case
    def __init__(self, orders: OrderRepository, payments: PaymentGateway):
        self.orders, self.payments = orders, payments

    def __call__(self, order_id: str, total: Decimal) → Order:
        if total ⩽ 0:
            raise ValueError("empty basket")
        order = Order(order_id, total)
        self.orders.save(order)                # save first: a crash cannot lose a
charged order
        approved = self.payments.authorise(order_id, total, idempotency_key=order_id)
        order.status = "placed" if approved else "payment-failed"
        self.orders.save(order)
        return order
```

Im Hexagon: eine Domänenregel, zwei getriebene Ports und der Use Case PlaceOrder. Nichts hier importiert ein Web-Framework, einen Datenbanktreiber oder ein Zahlungs-SDK. Die Bestellung wird gespeichert, bevor die Karte autorisiert wird, und die Bestell-ID dient zugleich als Idempotenzschlüssel, sodass ein Absturz oder eine Wiederholung einen Kunden weder ohne Bestellung noch doppelt belasten kann.

```
class PostgresOrderRepository:          # real driven adapter
    def __init__(self, conn):
        self.conn = conn
    def save(self, order):
        with self.conn.cursor() as cur:
            cur.execute(
                "INSERT INTO orders (id, total, status) VALUES (%s, %s, %s) "
                "ON CONFLICT (id) DO UPDATE SET status = EXCLUDED.status",
                (order.id, order.total, order.status))
        self.conn.commit()

class InMemoryOrderRepository:          # test double, same port
    def __init__(self):
        self.saved = {}
    def save(self, order):
        self.saved[order.id] = order
```

Außerhalb des Hexagons: ein echter Adapter und ein Test Double für denselben Port. Der Use Case kann sie nicht unterscheiden.

```
class DecliningPayments:
    def authorise(self, order_id, amount, idempotency_key): return False

def test_failed_payment_is_recorded():
    orders = InMemoryOrderRepository()
    place = PlaceOrder(orders, DecliningPayments())
    place("A-17", Decimal("23.50"))
    assert orders.saved["A-17"].status == "payment-failed"

# composition root (main.py): the only place that knows every adapter
# place_order = PlaceOrder(PostgresOrderRepository(conn), CardProviderGateway(key))
# app.add_route("/orders", make_rest_controller(place_order))   # driving adapter
```

Ein Unit-Test treibt den Use Case direkt an, ohne Webserver, Datenbank oder Netzwerk; die Verdrahtung für die Produktion geschieht an einer Stelle beim Start.

Beachten Sie, was der Use Case nicht enthält: kein SQL, keine HTTP-Statuscodes, keine anbieterspezifischen Exception-Typen, keine Framework-Decorators. Der Test läuft in Millisekunden und deckt einen Fall ab (eine abgelehnte Zahlung), der sich gegen die Sandbox eines echten Anbieters nur umständlich herstellen ließe. Der PostgreSQL-Adapter braucht eigene Tests, doch das sind fokussierte Integrationstests einer Klasse gegen eine echte Datenbank, keine Tests von Geschäftsregeln. Das ist der Testbarkeitsnutzen, der in Abschnitt 2.6 beschrieben ist: Steuerbarkeit und Beobachtbarkeit durch injizierte Abhängigkeiten.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: den Zahlungsanbieter wechseln

Die Essenslieferplattform muss wegen einer Preisänderung innerhalb von sechs Wochen von einem Kartenanbieter zu einem anderen wechseln. Vergleichen Sie den Aufwand in zwei Versionen des Systems: einer, in der der Checkout-Code das SDK des alten Anbieters an 23 Stellen direkt aufruft, und einer, die mit einem getriebenen Port `PaymentGateway` und einem Adapter gebaut ist.

Direkte Aufrufe: die Auswirkungen ermitteln	Die Suche findet das SDK im Checkout, in den Erstattungen, den Abonnements und drei Admin-Ansichten. Die eigenen Exception-Typen des SDK werden an 11 Stellen abgefangen, und sein `Charge`-Objekt ist in einer Datenbankspalte gespeichert.
Direkte Aufrufe: die Änderung	Jede Aufrufstelle ändert sich, dazu kommt eine Datenmigration für die gespeicherten Objekte. Jeder Test, der Zahlungen berührte, wurde gegen die Sandbox des alten SDK geschrieben und muss neu geschrieben werden. Schätzung: 5 bis 6 Wochen, mit einer riskanten Umstellung auf einen Schlag.
Ports and Adapters: die Auswirkungen ermitteln	Eine Adapterklasse (etwa 200 Zeilen) implementiert `PaymentGateway`. Der Kern kennt nur `charge`, `refund` und den eigenen Typ `PaymentResult` des Kerns.
Ports and Adapters: die Änderung	Einen zweiten Adapter für den neuen Anbieter schreiben; die vorhandenen Contract-Tests des Ports dagegen laufen lassen; den Adapter je Bestellung mit einem Feature-Flag wählen, sodass 1 %, dann 10 %, dann 100 % der Zahlungen den neuen nutzen. Schätzung: 2 Wochen, bei jedem Schritt umkehrbar.
Was der Port gekostet hat	Port und Adapter zu Beginn zu entwerfen kostete etwa 3 zusätzliche Tage, und jedes neue Zahlungsfeature braucht eine kleine Änderung am Port wie auch am Adapter.

Der Wert eines Ports ist proportional zur Wahrscheinlichkeit und zu den Kosten der Änderung, die er isoliert. Zahlungsanbieter, Message Broker und APIs von Drittanbietern ändern sich oft; Ihre eigene zentrale Datenbank-Engine ändert sich vielleicht nie, und sie in aufwendige Ports zu verpacken, ist schwerer zu rechtfertigen.

Der Port muss die Sprache der Domäne sprechen (`charge`, `PaymentResult`), nicht die des alten Anbieters. Ein Port, der die API eines einzelnen Anbieters spiegelt, schützt nichts.

Merkmale der Ports-and-Adapters-Familie

Merkmal	Bewertung	Begründung
Testbarkeit	Hoch	Geschäftsregeln lassen sich im Speicher testen, schnell und deterministisch, indem man Test Doubles in die getriebenen Ports einsteckt.
Änderbarkeit der Infrastruktur	Hoch	Eine Datenbank, einen Zahlungsanbieter oder ein Web-Framework zu ersetzen heißt, einen neuen Adapter zu schreiben; der Kern ändert sich nicht.
Änderbarkeit der Geschäftsregeln	Mittel bis hoch	Die Regeln liegen an einer Stelle, frei von Framework-Code, doch ein Feature durchquert weiterhin Port, Use Case und Adapter.
Einfachheit	Niedrig bis mittel	Mehr Typen, Schnittstellen und Mapping-Code als in einer schlichten Schichtenanwendung; anfangs schwieriger für Neue im Team.
Entwicklungsgeschwindigkeit (früh)	Mittel	Die zusätzliche Struktur bremst die ersten Features einer einfachen CRUD-Anwendung; sie zahlt sich aus, wenn die Domäne wächst.
Gesamtkosten	Günstig bis mittel	Keine zusätzliche Infrastruktur; die Kosten sind die Entwicklungszeit für das Schreiben und Pflegen von Ports, Adaptern und Mapping-Code.
Performance	Mittel bis hoch	Schnittstellenaufrufe kosten fast nichts; der echte Overhead ist das Objekt-Mapping zwischen Schichten und der Verlust mancher ORM-Annehmlichkeiten.
Skalierbarkeit, Deploybarkeit, Fehlertoleranz	Nicht durch diesen Stil bestimmt	Diese hängen vom Deployment-Stil ab (Monolith oder Services), in dem das Hexagon sitzt.

Kosten, Kritik und wann man sie einsetzt

Der Nutzen ist real, die Kosten sind es auch. Jeder Port ist eine zusätzliche Schnittstelle; jede Grenze braucht unter Umständen Mapping-Code, um etwa zwischen einer ORM-Entität, einem Domänenobjekt und einer API-Antwort umzuwandeln. Für eine Anwendung, die überwiegend Formularfelder in Tabellen schiebt, mit wenig Logik dazwischen, fügt diese Struktur Dateien hinzu, ohne Schutz hinzuzufügen, und die Konventionen eines Frameworks (ein Active-Record-Modell, eine generierte Admin-Oberfläche) liefern womöglich schneller ein besseres Ergebnis. Manche erfahrenen Praktiker argumentieren, dass strenge Varianten der Clean Architecture zu viel Indirektion erzeugen und die meisten Teams ihre Datenbank nie austauschen; andere

entgegnen, dass der Hauptnutzen nicht im Austausch von Technologie liegt, sondern in schnellen, verlässlichen Tests und in Geschäftscode, den man lesen kann, ohne das Framework zu kennen. Beide haben in unterschiedlichen Kontexten recht.

Setzen Sie sie ein, wenn die Geschäftslogik umfangreich ist und die aktuellen Frameworks überdauern wird; wenn das System mit externen Diensten integriert ist, die sich wahrscheinlich ändern; wenn schnelle, isolierte Tests der Geschäftsregeln wichtig sind; und als innere Struktur der Module eines modularen Monolithen oder von Services, die mit Domain-Driven Design gebaut sind (Abschnitt 5.7). **Seien Sie vorsichtig** bei einfachen CRUD-Anwendungen, Prototypen und kurzlebigen Werkzeugen, wo die Standardstruktur des Frameworks meist ausreicht. Ein pragmatischer Mittelweg ist verbreitet: Ports für externe Dienste und alles, was sich wahrscheinlich ändert, und anderswo die Annehmlichkeiten des Frameworks.

ACHTUNG

Häufige Anti-Patterns

- **Technologieförmige Ports.** Ein Port namens `SqlQueryExecutor` oder einer, der das SDK eines Anbieters spiegelt, behält die Kopplung, die er beseitigen sollte. Ports sollten die Sprache der Domäne sprechen.
- **Framework-Leckage.** Use Cases, die HTTP-Request-Objekte annehmen oder ORM-Entitäten zurückgeben, haben getarnte Abhängigkeiten nach außen.
- **Logik in Adaptern.** Geschäftsregeln in Controllern oder Repository-Klassen sind für die Use-Case-Tests unsichtbar und über Adapter hinweg dupliziert.
- **Eine Schnittstelle für jede Klasse.** Ports gehören an die Grenze zur Außenwelt, nicht zwischen jedes Paar interner Klassen.
- **Anämische Durchreich-Use-Cases.** Use Cases, die Aufrufe nur an ein Repository weiterleiten, sind das Architecture Sinkhole aus Abschnitt 5.2 in neuer Form; akzeptieren Sie sie für einfache Lesezugriffe, aber achten Sie darauf, ob sie die Mehrheit bilden.
- **Ein Composition Root überall.** Adapter, die tief im Geschäftscode erzeugt werden (`PostgresOrderRepository()` in einem Use Case), vereiteln die Austauschbarkeit, auf der der Stil beruht.

Die Abhängigkeitsregel durchsetzen

Wie bei Schichten erodiert die Regel, wenn keine Maschine sie prüft. In Python kann import-linter die Ringe als *layers*-Vertrag deklarieren (Adapter über Anwendung über Domäne) und *forbidden*-Verträge hinzufügen, die verhindern, dass die Domänen- und Anwendungspakete Web-Frameworks, Datenbanktreiber oder SDKs importieren. In Java hat ArchUnit eine fertige

Regel `onionArchitecture()`, in der Sie die Pakete für Domänenmodell, Domain Service, Application Service und Adapter benennen, und sie lässt einen Test fehlschlagen, wenn eine Abhängigkeit nach außen zeigt. Den Code in getrennte Build-Module aufzuteilen (ein Domänenmodul ganz ohne Framework-Abhängigkeiten) macht Abhängigkeiten nach außen unkompilierbar, und das ist die stärkste Prüfung von allen. Diese Prüfungen sind Beispiele für die Fitness Functions, die Kapitel 11 behandelt.

Eine Paketstruktur, die die Regel sichtbar macht, hilft beim Review: zum Beispiel `orders/domain`, `orders/application` (Use Cases und Ports) und `orders/adapters/{rest,postgres,payments}`. Zuerst nach Geschäftsbereich und dann nach Ring zu gliedern, hält die Struktur an den Domänengrenzen der Abschnitte 5.3 und 5.7 ausgerichtet.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Ist ein REST-Controller ein treibender oder ein getriebener Adapter? Und ein SQL-Repository?

Ein REST-Controller ist ein treibender Adapter: Er nimmt Anfragen entgegen und ruft die Anwendung auf. Ein SQL-Repository ist ein getriebener Adapter: Es implementiert eine Schnittstelle, die die Anwendung aufruft.

▼ Darf in der Clean Architecture ein Use Case eine ORM-Entität an einen Controller zurückgeben?

Nein. Das ORM gehört zum äußeren Ring, also hat ein Use Case, der es benennt, eine Abhängigkeit nach außen. Er sollte eine einfache Struktur oder ein Domänenobjekt zurückgeben, das in einem inneren Ring definiert ist.

▼ Wo werden Adapter erzeugt und mit der Anwendung verbunden?

In einem Composition Root, meist dem Startcode, der einzigen Stelle, die jeden Adapter kennt.

▼ Was fügt die Onion-Architektur dem hexagonalen Bild hinzu?

Benannte Ringe innerhalb der Anwendung (Domänenmodell, Domain Services, Application Services) und die ausdrückliche Aussage, dass die Datenbank ein externes Anliegen am Rand ist, nicht das Fundament.

▼ **Probieren Sie es aus: Ein Ticketing-Produkt hat komplexe Regeln für die Platzzuteilung und eine einfache Admin-Ansicht zum Bearbeiten der Kontaktdaten von Veranstaltungsorten. Würden Sie für beides die hexagonale Architektur verwenden? Erklären Sie, was Sie jeweils gewinnen oder verlieren würden.**

Verwenden Sie es nur für die Platzvergabe. Diese Regeln bekommen schnelle, isolierte Tests, und Datenbank oder Broker lassen sich durch Austausch eines Adapters ersetzen, was zu Logik passt, die wichtig ist und sich ändert. Der Admin-Bildschirm ist einfaches CRUD, Ports und Mapping-Code brächten dort Indirektion bei wenig zu Schützendem. Verwenden Sie dort einen schlichten Ansatz, getrennt vom Kern. Wenden Sie das Muster dort an, wo die Komplexität seine Kosten rechtfertigt, und setzen Sie die Abhängigkeitsregel automatisch durch.

QUELLEN

- Alistair Cockburn: Hexagonal architecture: https://alistair.cockburn.us/hexagonal-architecture/
- Jeffrey Palermo: The Onion Architecture, part 1 (2008): https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
- Robert C. Martin: The Clean Architecture (2012): https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- ArchUnit user guide (layered and onion architecture rules): https://www.archunit.org/userguide/html/000_Index.html
- import-linter documentation: https://import-linter.readthedocs.io/en/stable/

Strategisches Domain-Driven Design: Bounded Contexts und Context Maps

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 3.1 Modularität
- 3.7 Gesetz von Conway
- 5.6 Hexagonal und Clean

Begriffe von früher, die er verwendet: Adapter, Modularer Monolith, Big Ball of Mud, Gesetz von Conway

DIE IDEE IN KÜRZE

In einem Krankenhaus bedeutet das Wort „Patient“ für die Aufnahme (Name, Adresse, Termin), die Station (Bett, Medikamente, Beobachtungen) und die Abrechnungsstelle (Versicherer, Rechnung) jeweils etwas anderes. Wollte ein einziges riesiges Formular allen dienen, wäre es gewaltig, und alle würden darum streiten. Besser funktioniert es, wenn jede Abteilung ihre eigene einfache Akte führt und man sich darauf einigt, wie Informationen zwischen ihnen weitergegeben werden. Domain-Driven Design überträgt diese Idee auf Software: Es findet die natürlichen Abteilungen eines Unternehmens, gibt jeder ihr eigenes Modell und Vokabular und macht die Verbindungen zwischen ihnen explizit.

WAS SIE WISSEN MÜSSEN

- Domain-Driven Design (Eric Evans, 2003) baut Software um ein Modell der Fachdomäne herum, das gemeinsam mit Fachexperten entwickelt und in einer Ubiquitous Language ausgedrückt wird, die im Gespräch wie im Code verwendet wird.
- Subdomänen sind Core (wo das Unternehmen im Wettbewerb steht), Supporting (spezifisch, aber nicht differenzierend) oder generisch (gelöste Probleme); die Einordnung entscheidet, wohin Designaufwand fließt und was man zukauft.
- Ein Bounded Context ist eine Grenze, innerhalb derer ein Modell und eine Sprache konsistent gelten; dasselbe Wort kann in verschiedenen Kontexten gefahrlos Verschiedenes bedeuten.
- Eine Context Map zeigt die Beziehungen zwischen Kontexten: Partnership, Shared Kernel, Customer-Supplier, Conformist, Anticorruption Layer, Open Host Service, Published Language und Separate Ways.

- Event Storming (Alberto Brandolini) entdeckt Kontexte, indem es Domain Events gemeinsam mit den Menschen, die das Geschäft kennen, auf eine Zeitleiste setzt.
- Aggregate sind Konsistenzgrenzen, die über eine Wurzel geändert werden, die Invarianten durchsetzt; Bounded Contexts sind der beste Leitfaden für Modul- und Servicegrenzen.

Warum Domain-Driven Design in ein Kapitel über Struktur gehört

Jeder Stil in diesem Kapitel braucht Grenzen. Ein modularer Monolith braucht Modulgrenzen; die Microservices aus Kapitel 6 brauchen Servicegrenzen; selbst eine hexagonale Anwendung muss wissen, was in das Hexagon gehört. Domain-Driven Design (DDD) ist die am weitesten verbreitete Methode, diese Grenzen in der Fachdomäne selbst zu finden statt in technischen Schichten.

Eric Evans führte es in *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley, 2003) ein. Sein Argument: In den meisten Geschäftsanwendungen liegen die schwierigsten Probleme nicht in der Technik, sondern im Verständnis der Domäne, also darin, was eine Police, ein Schadensfall, eine Lieferung oder eine Erstattung wirklich ist und welche Regeln dafür gelten. Software sollte daher um ein bewusst gestaltetes Modell der Domäne herum gebaut werden, das Entwicklungsteams und Fachexperten gemeinsam entwickeln, und dieses Modell sollte vor der Vermischung mit anderen Modellen geschützt werden. 2015 veröffentlichte Evans eine kurze *DDD Reference*, die die Muster zusammenfasst und einige ergänzt, etwa Partnership und Big Ball of Mud als Context-Map-Muster sowie Domain Events als Baustein.

DDD hat zwei Hälften. **Strategisches Design** teilt eine große Domäne in Subdomänen und Bounded Contexts auf und bildet die Beziehungen zwischen ihnen ab; das ist Architektur und das Hauptthema hier. **Taktisches Design** liefert Bausteine (Entitäten, Value Objects, Aggregate und andere), um das Modell innerhalb eines Kontexts zu schreiben; das ist Design und wird kurz behandelt.

Ubiquitous Language

Eine Ubiquitous Language ist ein gemeinsames Vokabular, das Entwicklungsteams und Fachexperten zusammen aufbauen und das in Gesprächen, Dokumenten, Tests und Code konsistent verwendet wird. Wenn die Disponenten sagen, ein Kurier sei „gestapelt“, wenn er zwei Bestellungen aus demselben Restaurant fährt, dann heißt es im Code `stack`, nicht `multiOrderAssignment`. Erweist sich ein Begriff als mehrdeutig oder sperrig, ändert das Team Sprache und Code gemeinsam.

Das klingt nach einer Frage der Benennung, hat aber architektonische Folgen. Die Übersetzung zwischen den Worten des Geschäfts und den Worten des Codes ist der Ort, an dem Missverständnisse und damit Fehler entstehen. Und sobald zwei Gruppen dasselbe Wort mit unterschiedlicher Bedeutung verwenden, haben Sie einen Hinweis auf zwei verschiedene Modelle gefunden, und genau das ist der Ausgangspunkt für Bounded Contexts.

Subdomänen: Core, Supporting und generisch

Die Fachdomäne als Ganzes gliedert sich in Subdomänen: Tätigkeitsbereiche wie Speisekarten, Bestellungen, Zahlungen und Disposition. Fachleute ordnen sie drei Arten zu und bauen dabei auf Evans' Unterscheidung zwischen Core Domain und generischen Subdomänen auf; die Dreiteilung mit Supporting Subdomains wurde von späteren Autoren populär gemacht, insbesondere von Vaughn Vernon in *Implementing Domain-Driven Design* (2013).

Die Core Domain ist das, was das Unternehmen von seinen Wettbewerbern unterscheidet, und der Grund, warum sich die Software zu bauen lohnt. Für ein Essenslieferunternehmen ist das wahrscheinlich die Disposition: Bestellungen Kurieren zuordnen und Ankunftszeiten vorhersagen. Eine Supporting Subdomain ist spezifisch für das Unternehmen und notwendig, aber keine Quelle von Vorteilen: Speisekartenverwaltung, Kurier-Onboarding. Eine generische Subdomäne ist ein Problem, das viele Unternehmen teilen und für das es gute vorhandene Lösungen gibt: Zahlungen, Identität, E-Mail und SMS.

Die Einordnung ist eine Investitionsentscheidung. Die Core Domain verdient die erfahrensten Leute, die sorgfältigste Modellierung und Code, der Ihnen vollständig gehört, denn dort wird bessere Software direkt zum Geschäftsvorteil. Supporting Subdomains verdienen solide, aber einfachere Designs, oft ein schlichtes Modul in Schichten. Generische Subdomänen sollten meist zugekauft, als Service genutzt oder als dünne Adapter über Open-Source-Komponenten gebaut werden: Einen eigenen Identity Provider zu bauen ist Aufwand an einer Stelle, an der Sie nicht gewinnen können. Die Einordnung ändert sich zudem mit der Zeit; eine unterstützende Fähigkeit kann zur Core Domain werden, wenn sich die Geschäftsstrategie ändert.

Bounded Contexts

Ein Bounded Context ist eine explizite Grenze, innerhalb derer ein Modell und eine Ubiquitous Language konsistent gelten. Innerhalb davon hat „Bestellung" genau eine Bedeutung. Außerhalb kann dasselbe Wort etwas anderes bedeuten, und das ist in Ordnung, denn jedes Modell ist für einen Zweck gestaltet. Evans stellte fest, dass der Versuch, das Modell eines großen Systems zu einem einzigen zu vereinheitlichen, weder machbar noch wirtschaftlich ist; er bringt riesige gemeinsame Klassen hervor, die jedem Zweck schlecht dienen und die jedes Team ändern muss.

Subdomänen und Bounded Contexts sind verschiedene Dinge, auch wenn sie oft zusammenfallen. Eine Subdomäne ist Teil des Problems: Sie existiert im Unternehmen, ob es Software gibt oder nicht. Ein Bounded Context ist Teil der Lösung: eine Grenze, die Sie für ein Modell wählen, und meist auch für Code und Daten, die einem Team gehören. Ideal ist ein Bounded Context pro Subdomäne, doch in Altsystemen erstreckt sich oft ein Kontext über mehrere Subdomänen, und eine große Subdomäne kann mehrere Kontexte brauchen.

Ein Wort, drei Modelle



„Bestellung“ bedeutet für die Bestellungen einen Warenkorb mit Preisen, für die Disposition einen Auftrag mit Abholung und Übergabe und im Küchenkontext eine Liste zu kochender Gerichte; jedes Modell bleibt klein, weil es einem Zweck dient.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: die Teile des Geschäfts und die Grenzen der Software

Der Punkt, an dem hier die meisten stolpern, ist der Unterschied zwischen einer Subdomäne und einem Bounded Context. Sie decken sich oft und werden deshalb leicht verwechselt, doch das eine ist Teil des fachlichen Problems, das andere eine Grenze, die das Team in der Software wählt.

- Die Disposition, also Bestellungen Kurieren zuzuordnen und Ankunftszeiten vorherzusagen, ist eine Subdomäne des Essenslieferungsgeschäfts. Sie existiert, ob es Software gibt oder nicht: Auch ein Disponent mit Telefon und Stadtplan würde sie erledigen.
- Ein Bounded Context ist eine Entscheidung über die Software: eine Grenze, innerhalb derer ein Modell und eine Sprache gelten. Das Team legt fest, dass im Dispo-Kontext ein „Lieferauftrag“ mit Abholung, Übergabe, Fertigzeit und Kurier bearbeitet wird.
- Idealerweise bekommt jede Subdomäne genau einen Bounded Context, der meist einem Team gehört, und die Disposition und ihr Kontext decken sich genau so.
- Nicht immer passt es so sauber. In einem älteren System kann eine einzige gemeinsame Order-Klasse zugleich Bestellungen, Disposition und die Küchenansicht der Restaurants bedienen: ein Modell und damit ein Kontext, über mehrere Subdomänen gespannt, das jedes Team ändern muss und niemand einfach halten kann.
- Auch das Gegenteil kommt vor: Eine große Subdomäne kann mehrere Bounded Contexts brauchen, jeden mit einem kleineren Modell.
- Die Subdomänen können bleiben, wie sie sind, während das Team seine Kontexte neu zieht, so wie das ausgearbeitete Beispiel weiter unten in diesem Abschnitt jene gemeinsame Order-Klasse in drei Modelle aufteilt.
- **Die Lehre:** Finden Sie Subdomänen, indem Sie das Geschäft untersuchen; wählen Sie Bounded Contexts, indem Sie entscheiden, wo ein Modell und eine Sprache gelten sollen. Streben Sie einen Kontext pro Subdomäne an und achten Sie darauf, wo beides nicht übereinstimmt.

Context Maps

Eine Context Map zeigt die Bounded Contexts eines Systems und die Beziehungen zwischen ihnen. Die Beziehungen beschreiben sowohl die technische Integration als auch Macht und Kommunikation zwischen den Teams, weshalb es beim Context Mapping ebenso um Organisationen wie um Code geht. Es ist eine direkte Anwendung des Gesetzes von Conway: Die Map zeigt, wo die Kommunikationswege der Organisation und die Integrationswege des Systems übereinstimmen müssen.

Viele Beziehungen sind **Upstream/Downstream**: Die Entscheidungen des Upstream-Kontexts wirken sich auf den Downstream-Kontext aus, aber nicht umgekehrt. Das Downstream-Team hat dann die Wahl, wie es damit umgeht, und genau diese Wahl benennen die meisten Muster.

Beziehungsmuster der Context Map

Muster	Was es bedeutet	Wer sich wem anpasst	Beispiel Essenslieferung
Partnership	Zwei Teams haben gemeinsam Erfolg oder scheitern gemeinsam, also planen sie zusammen und entwickeln ihre Schnittstelle gemeinsam weiter.	Beide, nach Absprache	Bestellungen und Disposition releasen die Übergabe von der Bestellung zur Lieferung gemeinsam.
Shared Kernel	Zwei Kontexte teilen einen kleinen, ausdrücklich vereinbarten Teil des Modells und seines Codes; Änderungen brauchen die Zustimmung beider Teams.	Beide, mit Vorsicht	Disposition und Kurier-Onboarding teilen eine winzige Kurierprofil-Bibliothek (Fahrzeugtyp, Kurier-ID).
Customer-Supplier	Das Upstream-Team (Supplier) verpflichtet sich, die Bedürfnisse des Downstream-Teams (Customer) zu erfüllen, die Teil seiner Planung sind.	Upstream bedient Downstream	Die Speisekarte liefert Speisekarten und Preise an die Bestellungen, und die Bedürfnisse der Bestellungen kommen in das Backlog der Speisekarte.
Conformist	Das Downstream-Team übernimmt das Upstream-Modell unverändert, weil es keinen Einfluss darauf hat und sich eine Übersetzung nicht lohnt.	Downstream passt sich an	Die Disposition verwendet das Routen- und Standortmodell des Kartenanbieters direkt.
Anticorruption Layer	Das Downstream-Team baut eine Übersetzungsschicht, damit das Upstream-Modell nicht in sein eigenes durchsickern kann.	Downstream übersetzt	Die Speisekarte übersetzt Dutzende Kassensystemformate der Restaurants in ihr eigenes Speisekartenmodell; die Zahlungen kapseln die API des Anbieters ab.
Open Host Service	Der Upstream-Kontext bietet allen Konsumenten ein klar definiertes Protokoll oder API an, statt für jeden eine eigene Integration.	Upstream veröffentlicht einmal	Die Zahlungen bieten Bestellungen, Erstattungen und Abonnements eine einzige dokumentierte Zahlungs-API an.
Published Language	Ein dokumentiertes, gemeinsames Austauschformat, oft zusammen mit einem Open Host Service verwendet.	Beide nutzen die Sprache	Bestell-Events (OrderPlaced, OrderCancelled) haben ein versioniertes, dokumentiertes Schema.

Muster	Was es bedeutet	Wer sich wem an-passt	Beispiel Essenslieferung
Separate Ways	Die Kontexte sind überhaupt nicht integriert, weil der Nutzen geringer ist als die Kosten.	Keiner	Die Lohnabrechnungsberichte für Kuriere laufen in einem separaten Finanzwerkzeug, das per monatlichem Export statt über eine Live-Integration befüllt wird.

Zwei dieser Muster verdienen besondere Aufmerksamkeit. Ein Anticorruption Layer (ACL) ist eine Reihe von Adaptern und Übersetzern, die ein Downstream-Kontext baut, damit ein fremdes Modell, oft ein Altsystem oder ein externer Anbieter, nicht in sein eigenes Modell durchsickern kann. Es ist dieselbe Idee wie der getriebene Adapter aus Abschnitt 5.6, angewandt zwischen Kontexten, und die Standardwahl für die Integration mit allem, was Sie nicht kontrollieren. Ein Open Host Service zusammen mit einer Published Language ist der Weg des Upstream-Kontexts, viele Konsumenten gut zu bedienen: ein dokumentiertes, versioniertes Protokoll statt einer eigenen Integration für jeden. Ein Shared Kernel sollte klein bleiben; ein großer macht aus zwei Kontexten wieder einen, über den zwei Teams streiten, und Evans' Referenz von 2015 ergänzt Big Ball of Mud als ehrliche Bezeichnung für einen Bereich ganz ohne klare Grenzen, der eingedämmt statt erweitert werden sollte.

Event Storming: die Kontexte entdecken

Grenzen werden mit Fachexperten entdeckt, nicht im Alleingang entworfen. Event Storming, von Alberto Brandolini entwickelt und 2013 erstmals in seinem Blog beschrieben, ist das beliebteste Workshopformat dafür. Die Menschen, die das Geschäft kennen, und die Menschen, die die Software bauen, stehen an einer langen Papierwand und schreiben Domain Events (Dinge, die passiert sind, in der Vergangenheitsform: *Bestellung aufgegeben*, *Kurier zugewiesen*, *Essen abgeholt*) auf orange Haftnotizen und ordnen sie auf einer Zeitleiste an.

Danach ergänzt die Gruppe die Commands, die Events auslösen, die beteiligten Personen und externen Systeme, die Policies, die auf Events reagieren („immer wenn eine Bestellung aufgegeben wird, einen Kurier finden"), und vor allem die Hotspots, an denen Uneinigkeit herrscht oder niemand Bescheid weiß. Gruppen von Events, die eine Sprache und einen Zweck teilen, deuten auf Bounded Contexts hin; Stellen, an denen dasselbe Wort unterschiedlich verwendet wird, deuten auf Grenzen hin; Stellen, an denen eine Übergabe stattfindet, deuten auf Integrationspunkte hin. Eine Big-Picture-Session für ein ganzes Unternehmen dauert ein oder zwei Tage und braucht kaum Vorbereitung außer den richtigen Leuten und viel Wandfläche.

Taktische Muster in Kürze

Innerhalb eines Bounded Context bietet DDD Bausteine für das Modell. Eine Entität hat eine Identität, die bestehen bleibt, während sich ihre Attribute ändern: Eine Bestellung ist dieselbe Bestellung, nachdem sich ihr Status geändert hat. Ein Value Object hat keine Identität und ist allein durch seine Werte definiert: Zwei `Money(12.50, "GBP")`-Objekte sind austauschbar, und Value Objects sind meist unveränderlich. Ein Domain Event hält etwas Bedeutsames fest, das in der Domäne passiert ist.

Ein Aggregat ist eine Gruppe von Entitäten und Value Objects, die für Änderungen als eine Einheit behandelt wird, mit einer Entität als Wurzel. Code von außen darf nur Referenzen auf die Wurzel halten und das Aggregat nur über sie ändern, damit die Wurzel die Invarianten des Aggregats durchsetzen kann (Regeln, die immer gelten müssen, etwa „nach dem Aufgeben einer Bestellung können keine Positionen mehr hinzugefügt werden“). Eine Transaktion sollte ein Aggregat ändern; andere Aggregate werden per ID referenziert und über Domain Events aktualisiert, was Aggregate zur natürlichen Einheit der Konsistenz und später der Verteilung macht. Ein Repository lädt und speichert ganze Aggregate und wirkt für die Domäne wie eine Collection; seine Schnittstelle ist in hexagonaler Sprache ein getriebener Port.

```
from dataclasses import dataclass, field
from decimal import Decimal

@dataclass(frozen=True)
class Money:                                # value object: equal if values equal
    amount: Decimal
    currency: str
    def __add__(self, other):
        if other.currency ≠ self.currency:
            raise ValueError("currency mismatch")
        return Money(self.amount + other.amount, self.currency)
    def times(self, n: int):
        return Money(self.amount * n, self.currency)

@dataclass
class OrderLine:                            # entity inside the aggregate
    dish_id: str
    qty: int
    price: Money

@dataclass
class Order:                                # aggregate root
    id: str
    restaurant_id: str                      # other aggregates referenced by ID only
    lines: list[OrderLine] = field(default_factory=list)
    status: str = "draft"
    events: list = field(default_factory=list)

    def add(self, dish_id, qty, price):
        if self.status ≠ "draft":
            raise ValueError("order already placed")    # invariant
        self.lines.append(OrderLine(dish_id, qty, price))

    def place(self, minimum: Money):
        total = Money(Decimal(0), minimum.currency)
        for line in self.lines:
            total = total + line.price.times(line.qty)
        if total.amount < minimum.amount:
            raise ValueError("below restaurant minimum")  # invariant
        self.status = "placed"
        self.events.append({"type": "OrderPlaced", "order_id": self.id, "total":
str(total.amount)})
```

Ein Order-Aggregat: ein Value Object (Money), eine Entität (OrderLine), die Aggregatwurzel, die Invarianten durchsetzt, und ein Domain Event, das beim Aufgeben der Bestellung festgehalten wird.

Das Beispiel hält das Aggregat klein, referenziert das Restaurant per ID, statt ein Restaurant-Objekt zu halten, und setzt seine Regeln in Methoden durch, statt sie den Aufrufern zu überlassen. Ein häufiger Fehler ist das Gegenteil: Aggregate, die so groß sind, dass jede

Änderung eines Benutzers dieselben Zeilen sperrt, oder Domänenobjekte, die nur Sammlungen von Gettern und Settern sind, während die Regeln in Service-Klassen liegen (Martin Fowler nennt das das anämische Domänenmodell).

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine Context Map für die Essenslieferplattform

Die Plattform ist auf etwa 60 Engineers in acht Teams gewachsen, und jede Änderung scheint die gemeinsame Klasse `Order` zu berühren, die inzwischen 94 Felder hat. Die Architektin führt einen zweitägigen Event-Storming-Workshop mit Produktmanagern, Betriebspersonal, Restaurant-Support und Engineers durch und zeichnet danach die unten gezeigte Context Map.

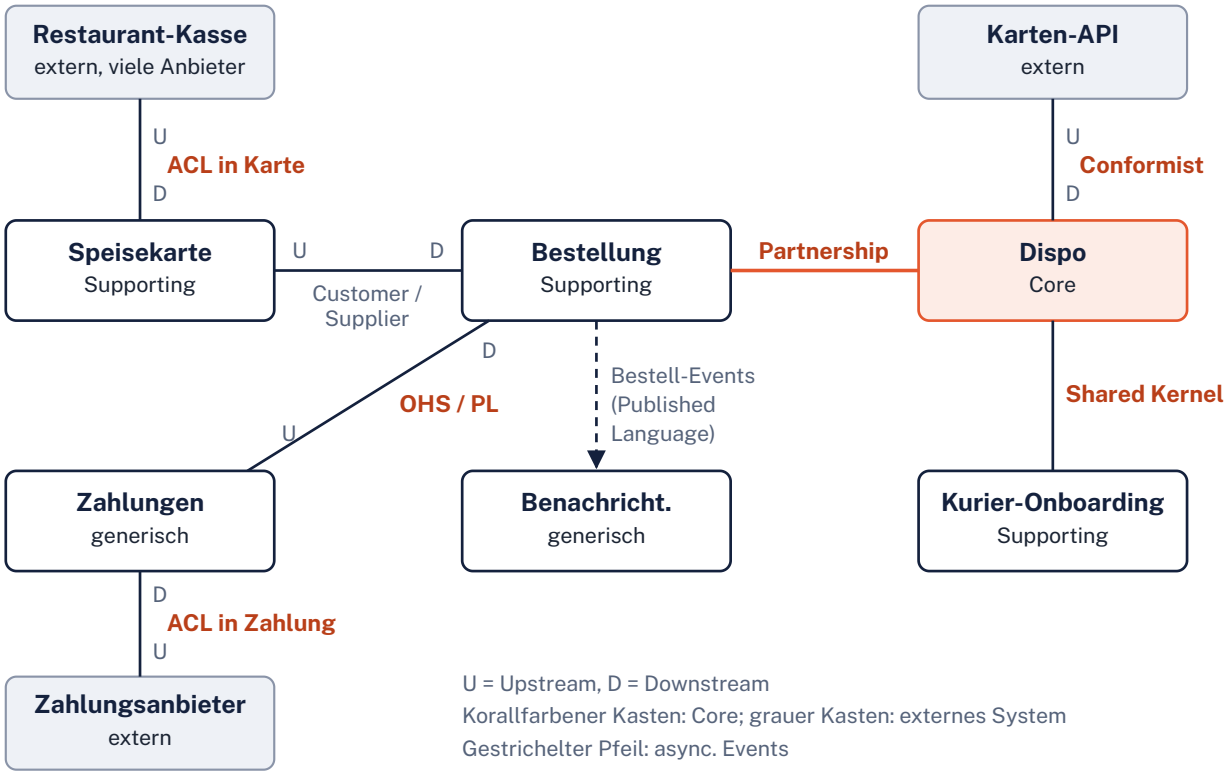
1. Die Subdomänen finden	Aus der Event-Zeitleiste: Speisekartenverwaltung, Bestellung und Bezahlvorgang, Zahlungen, Disposition (Kuriere Bestellungen zuordnen und Ankunftszeiten vorhersagen), Kurier-Onboarding, Kundenbenachrichtigungen, Restaurantintegration.
2. Sie einordnen	Die Disposition ist die **Core Domain**: Schnellere, günstigere Lieferungen gewinnen Kunden und Restaurants. Speisekarte, Bestellungen und Kurier-Onboarding sind **Supporting**: notwendig und spezifisch für dieses Geschäft, aber nicht dort, wo es im Wettbewerb steht. Zahlungen und Benachrichtigungen sind **generisch**: gelöste Probleme, die man am besten zukauft oder dünn über einem Anbieter aufbaut.
3. Bounded Contexts ziehen	Jede Subdomäne bekommt ihr eigenes Modell. „Bestellung“ in den Bestellungen (Warenkorb, Preise, Zahlstatus) wird getrennt vom „Lieferauftrag“ in der Disposition (Abholung, Übergabe, Fertigzeit) und vom „Ticket“ in der Küchenansicht des Restaurants. Aus der Klasse mit 94 Feldern werden drei fokussierte Modelle, verknüpft über die Bestell-ID.
4. Die Beziehungen benennen	Bestellungen und Disposition: Partnership. Speisekarte zu Bestellungen: Customer-Supplier. Zahlungen: Open Host Service mit einer Published Language und ein Anticorruption Layer zum externen Anbieter. Speisekarte: Anticorruption Layer zu den Kassensystemen der Restaurants. Disposition: Conformist zur Karten-API, eine bewusste Wahl, weil die Übersetzung ihres reichhaltigen Modells mehr kosten würde, als sie einspart.
5. Über die Investition entscheiden	Die besten Engineers und die sorgfältigste Modellierung gehen an die Disposition. Die Zahlungen nutzen einen Anbieter über einen dünnen Adapter. Die Benachrichtigungen nutzen einen zugekauften Messaging-Service.
6. Über das Deployment entscheiden	Die Kontexte werden zunächst Module der bestehenden Anwendung und machen sie so zu einem modularen Monolithen (Abschnitt 5.3), jedes mit eigenem Schema und eigener öffentlicher Schnittstelle. Die Disposition mit ihren ganz anderen Skalierungs- und Release-Anforderungen ist der erste Kandidat, später ein eigener Service zu werden.

Dass die Bestellungen als Supporting eingeordnet wurden, überraschte einige Stakeholder. Der Maßstab ist nicht die Wichtigkeit (ohne Bestellungen kann das Unternehmen nicht arbeiten), sondern die Differenzierung: Würde ein besseres

Bestellmodell dem Unternehmen mehr Kunden bringen als ein besseres Dispositionsmodell?

Die Conformist-Beziehung zum Kartenanbieter ist eine bewusste Abwägung, festgehalten in einem Entscheidungsprotokoll, kein Zufall. Ändert sich der Anbieter oder wird er ersetzt, wird die Disposition das zu spüren bekommen.

Die Context Map



Jede Linie benennt das Beziehungsmuster; U und D markieren Upstream und Downstream, und die Anticorruption Layers liegen auf der Downstream-Seite zu Systemen, die die Plattform nicht kontrolliert.

Bounded Contexts als Modul- und Servicegrenzen

Bounded Contexts sind der beste verfügbare Leitfaden dafür, wo Modulgrenzen in einem Monolithen und Servicegrenzen in einem verteilten System zu ziehen sind. Ein Kontext hat sein eigenes Modell, seine eigene Sprache, seine eigenen Daten und idealerweise ein verantwortliches Team, und genau diese Eigenschaften machen ein Modul kohäsiv und lose gekoppelt. In einem modularen Monolithen wird jeder Kontext zu einem Modul mit eigenem Schema; darin hält die hexagonale Struktur aus Abschnitt 5.6 das Modell unabhängig von der Infrastruktur.

Muss ein Kontext später separat deployt werden, hat seine Grenze bereits die richtige Form für einen Service, und seine Beziehungen auf der Context Map sagen Ihnen, wie die Integration aussehen muss. Deshalb empfehlen viele Fachleute, zuerst die Kontexte zu finden und erst danach über das Deployment zu entscheiden. Auch das Umgekehrte gilt: Services, die quer zu Bounded Contexts geschnitten sind (ein „Order-Service“, in den jeder Kontext schreibt), ergeben den eng gekoppelten verteilten Monolithen, der in Abschnitt 6.3 behandelt wird. Beachten Sie auch, dass ein Bounded Context nicht automatisch ein Microservice ist; ein Kontext kann als mehrere Services deployt werden, und mehrere kleine Kontexte können sich ein Deployment teilen.

Was strategisches DDD bringt und kostet

Aspekt	Wirkung	Begründung
Änderbarkeit innerhalb eines Kontexts	Hoch	Ein Modell, eine Sprache und ein Verantwortlicher: Änderungen bleiben lokal und sind leicht zu durchdenken.
Teamautonomie	Hoch	Kontextgrenzen decken sich mit Teamgrenzen, sodass Teams ihre Modelle ändern können, ohne jede Änderung auszuhandeln.
Passung zwischen Software und Geschäft	Hoch	Ubiquitous Language und gemeinsame Modellierung verringern Übersetzungsfehler zwischen Fachexperten und Code.
Anfänglicher und laufender Aufwand	Mittel bis hoch	Modellierungssessions, die Zeit der Fachexperten, Übersetzungsschichten und die Pflege von Verträgen kosten alle Aufwand.
Datenduplizierung und Integration	Mittlere Kosten	Jeder Kontext hält seine eigene Kopie dessen, was er braucht, und diese muss über Events oder APIs synchron gehalten werden.
Nutzen bei einfachen Domänen	Niedrig	Bei CRUD-lastigen Systemen mit wenig Logik bringt vor allem taktisches DDD viel Zeremonie mit wenig Ertrag.

ACHTUNG

Häufige Fehler

- **Taktisches DDD ohne Strategie.** Teams setzen überall Entitäten, Repositories und Aggregate ein, ziehen aber nie Bounded Contexts, und bekommen so die Zeremonie ohne die Grenzen.
- **Ein Kontext pro Datenbanktabelle oder pro Substantiv.** Kontexte folgen Sprache und Zweck, nicht Datenentitäten; ein „Customer-Service“ und ein „Order-Service“ für jedes Substantiv ergeben gesprächige, gekoppelte Teile.
- **Modellieren ohne Fachexperten.** Eine Ubiquitous Language, die Entwickler allein erfunden haben, ist nur eine Namenskonvention.
- **Die Map als dauerhaft behandeln.** Kontexte und Beziehungen ändern sich, wenn sich Geschäft und Organisation ändern; überprüfen Sie die Map mindestens jährlich.
- **Generische Subdomänen vergolden.** Engineers finden Zahlungen oder Identität oft interessanter als die Core Domain; die Einordnung existiert, um dem zu widerstehen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Was ist der Unterschied zwischen einer Subdomäne und einem Bounded Context?

Eine Subdomäne ist Teil des fachlichen Problems und existiert, ob es Software gibt oder nicht; ein Bounded Context ist eine in der Lösung gewählte Grenze, innerhalb derer ein Modell gilt. Idealerweise decken sie sich, aber das muss nicht so sein.

▼ Ihr Team integriert ein Abrechnungs-Altsystem, dessen Modell verwirrend ist und das Sie nicht ändern können. Welches Context-Map-Muster schützt Ihr Modell?

Ein Anticorruption Layer: Übersetzer auf Ihrer Seite, die das Modell des Altsystems in Ihre eigenen Begriffe umwandeln, damit es nicht in Ihren Kontext durchsickern kann.

▼ Warum sollte eine Transaktion in der Regel nur ein Aggregat ändern?

Weil das Aggregat die Einheit ist, deren Invarianten zusammen gelten müssen; mehrere auf einmal zu ändern erzeugt große Sperren und Kopplung, während andere Aggregate über Domain Events aktualisiert werden können, was es auch erlaubt, sie später zu verteilen.

▼ **Warum sind Zahlungen für ein Essenslieferunternehmen meist eine generische Subdomäne?**

Weil das Annehmen von Zahlungen ein gelöstes Problem ist, das viele Unternehmen teilen, und nicht dort liegt, wo das Unternehmen im Wettbewerb steht; es wird daher besser zugekauft oder dünn über einem Anbieter gebaut als tiefgehend modelliert.

▼ **Probieren Sie es aus: In einer Ticketing-Firma bedeutet „Ticket“ für den Vertrieb einen verkauften Platz und für den Support eine Kundenbeschwerde, und eine gemeinsame Klasse Ticket ist auf 80 Felder angewachsen. Payments kommt von einem externen Anbieter. Welche Kontexte und Beziehungen würden Sie vorschlagen?**

Teilen Sie die Klasse in die Bounded Contexts Sales und Support auf, jeder mit eigenem Modell und eigener Sprache, sodass „Ticket“ gefahrlos etwas anderes bedeuten kann und Support vielleicht „Fall“ sagt. Die Platzvergabe ist wohl die Core-Subdomäne, Support ist unterstützend und E-Mail generisch, investieren und kaufen Sie also entsprechend. Schützen Sie Sales mit einem Anticorruption Layer vor dem externen Zahlungsanbieter. Zwischen Sales und Support kann eine Customer-Supplier-Beziehung funktionieren, die nur eine Buchungsreferenz teilt. Prüfen Sie das mit Fachleuten der Domäne durch Event Storming.

QUELLEN

- Eric Evans: Domain-Driven Design Reference (2015, PDF): https://www.domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf
- Martin Fowler: Bounded Context: https://martinfowler.com/bliki/BoundedContext.html
- Martin Fowler: DDD Aggregate: https://martinfowler.com/bliki/DDD_Aggregate.html
- EventStorming (Alberto Brandolini): https://www.eventstorming.com/

Wann ein Monolith die richtige Antwort ist

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 5.1 Stile und Muster
- 5.3 Modularer Monolith
- 5.7 Domain-Driven Design

Begriffe von früher, die er verwendet: Modularer Monolith, Bounded Context, Deployment-Einheit, Big Ball of Mud, Load Balancer

DIE IDEE IN KÜRZE

Ein neues Restaurant beginnt meist mit einer Küche, in der jeder Koch jeden anderen sehen und ihm helfen kann. Am ersten Tag fünf getrennte Küchen in der ganzen Stadt zu eröffnen, jede mit eigenem Personal, eigenen Lieferungen und eigener Leitung, würde weit mehr kosten und jede Änderung an der Speisekarte zu einer Verhandlung machen. Erst wenn die eine Küche wirklich überlastet ist oder ein Gericht eine besondere Ausstattung braucht, ist es sinnvoll, eine weitere zu eröffnen. Bei Software ist es ähnlich: Ein einziges, gut organisiertes Programm ist meist der günstigste und schnellste Start, und eine Aufteilung lohnt sich erst, wenn es klare Belege dafür gibt, dass die eine Einheit zum Engpass geworden ist.

WAS SIE WISSEN MÜSSEN

- Ein Monolith ist eine Deployment-Entscheidung (eine Einheit, die zusammen gebaut und ausgeliefert wird), kein Urteil über die innere Qualität; ein gut modularisierter Monolith ist eine respektable, oft überlegene Architektur.
- Martin Fowlers Microservice Premium und MonolithFirst (2015) argumentieren, dass sich Microservices erst oberhalb eines gewissen Komplexitätsniveaus auszahlen und dass die meisten erfolgreichen Microservice-Systeme als Monolithen begonnen haben.
- David Heinemeier Hanssons Majestic Monolith (2016) spricht sich für kleine Teams aus; Stefan Tilkov (2015) warnt, dass nicht modularisierte Monolithen schwer aufzuteilen sind, setzen Sie Modulgrenzen daher von Anfang an durch.
- Ein Monolith ist meist richtig bei kleinen Teams, unsicheren Domänen, frühen Produkten, hohen Konsistenzanforderungen und begrenzter Betriebskapazität.

- Monolithen skalieren über zustandslose Replikate hinter einem Load Balancer, Read Replicas, Caching, Background Worker und bei Bedarf eine Partitionierung nach Mandanten.
- Teilen Sie auf, wenn es Belege gibt: Konkurrenz zwischen Teams, Deployment-Warteschlangen, stark abweichende Skalierungs- oder Compliance-Anforderungen; teilen Sie schrittweise entlang von Bounded Contexts auf.

Die Frage hinter dem Kapitel

Die Stile in diesem Kapitel werden alle als eine Einheit deployt. Kapitel 6 trägt dieselben Fragen über ein Netzwerk, wo jeder Aufruf fehlschlagen, langsam sein oder doppelt ankommen kann. Bevor wir diese Linie überschreiten, lohnt es sich, direkt zu fragen: Wann ist ein Monolith die richtige Antwort? Einen großen Teil der 2010er-Jahre lautete die Standardantwort der Branche „selten“, und viele Teams teilten junge Systeme in Microservices auf, weil große, berühmte Unternehmen es so gemacht hatten. Ein Jahrzehnt Erfahrung hat zu einer ausgewogeneren Sicht geführt, und dieser Abschnitt legt sie dar.

Das Wort Monolith erfordert Sorgfalt. Wie Abschnitt 5.1 erklärt hat, beschreibt es das Deployment (eine Einheit, die zusammen gebaut, getestet und ausgeliefert wird), nicht die innere Qualität. Ein Monolith kann ein gut strukturiertes modulares System oder ein Big Ball of Mud sein, und dasselbe gilt für eine Menge von Services. Viele der Probleme, die man Monolithen anlastet, sind in Wahrheit Probleme mangelnder Modularität, und wer ein schlecht modularisiertes System in Services aufteilt, erhält einen verteilten Big Ball of Mud, und der ist schlimmer.

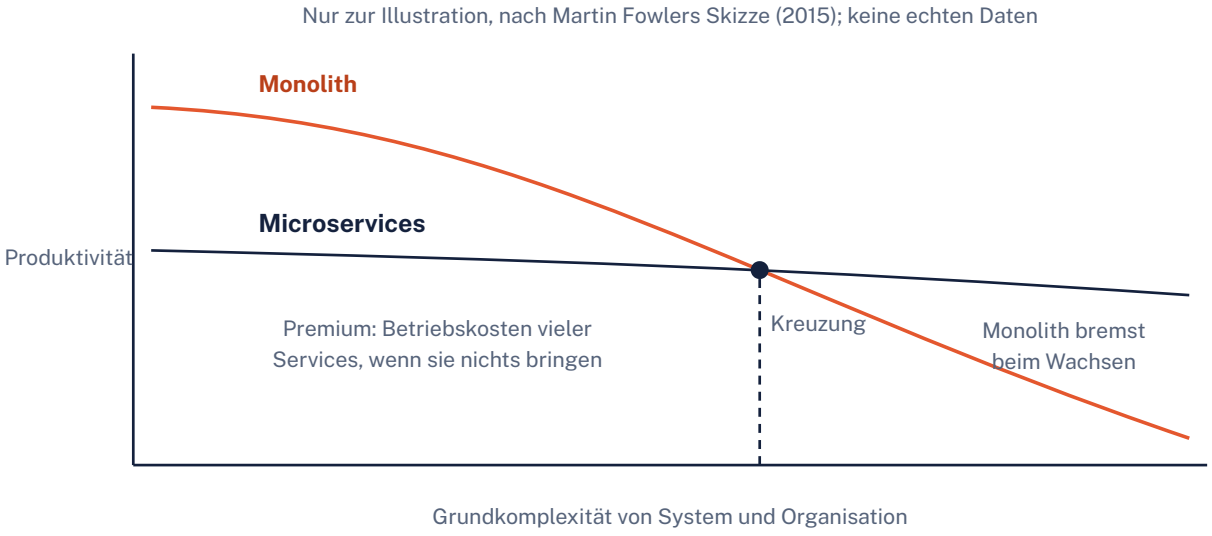
MonolithFirst und die Microservice Premium

Im Mai 2015 beschrieb Martin Fowler die Microservice Premium: Microservices bringen eigene, reale Kosten mit sich (automatisiertes Deployment vieler Einheiten, Monitoring, Umgang mit Teilausfällen, Eventual Consistency, versionierte APIs zwischen Teams), und diese Kosten bremsen ein Team, solange das System nicht komplex genug ist, damit Microservices sie wieder einspielen. Sein Rat lautete, standardmäßig auf einen einzigen, gut modularisierten Monolithen zu setzen und erst dann zu Microservices zu wechseln, wenn die Komplexität es verlangt.

Einen Monat später, in einem Artikel mit dem Titel MonolithFirst (Juni 2015), ergänzte er zwei Beobachtungen aus den Projekten, die er gesehen hatte: Fast alle erfolgreichen Microservice-Systeme hatten als Monolith begonnen, der zu groß wurde und dann aufgeteilt wurde, und fast alle von Grund auf als Microservices gebauten Systeme waren in ernste Schwierigkeiten geraten. Ein Grund ist, dass gute Servicegrenzen schwer zu finden sind, bevor man die Domäne

versteht, und eine falsche Grenze zwischen Services weit teurer zu verschieben ist als eine falsche Grenze zwischen Modulen. Er beschrieb mehrere Wege: einen Monolithen bauen und später Services an seinen Rändern abtrennen; einen bewusst wegwerfbaren Monolithen bauen; oder mit wenigen grobgranularen Services beginnen und sie weiter aufteilen, sobald sich die Grenzen gefestigt haben.

Die Microservice Premium



Eine Konzeptskizze: Microservices zahlen sich erst ab einem gewissen Komplexitätsniveau aus; darunter sind ihre Betriebskosten eine Prämie, die nichts bringt.

Der Majestic Monolith und die Gegenposition

Im Februar 2016 veröffentlichte David Heinemeier Hansson, der Schöpfer von Ruby on Rails, „The Majestic Monolith“ im Blog Signal v. Noise. Er beschrieb Basecamp, damals von einem Team aus etwa einem Dutzend Programmierern gebaut, als eine integrierte Anwendung, die Web-, Mobil-, Desktop- und E-Mail-Clients bedient, und argumentierte, kleine Teams sollten auf einen gut gemachten Monolithen setzen, statt die Architektur von Organisationen mit Tausenden Engineers zu kopieren. Sein Kernsatz lautete „Don't distribute your computing!“, es sei denn, man muss es wirklich.

Nicht alle Fachleute sind sich einig. Ebenfalls im Juni 2015 argumentierte Stefan Tilkov auf Fowlers Website in „Don't start with a monolith“, dass es oft schwieriger als erwartet ist, einen Monolithen später aufzuteilen, weil seine Teile über gemeinsame Bibliotheken, Domänenobjekte und die Datenbank eng zusammenwachsen. Ist das System eindeutig groß genug, um getrennte Teile zu rechtfertigen, und ist die Domäne gut verstanden, empfahl er, von Anfang an getrennte Subsysteme zu entwerfen. Die beiden Positionen liegen weniger weit

auseinander, als es klingt: Beide sagen, dass die Grenzen das Schwierige sind, und ein Monolith, dessen Module vom ersten Tag an durchgesetzt werden (Abschnitt 5.3), beantwortet den Großteil von Tilkovs Bedenken.

Reale Fälle weisen in beide Richtungen, haben aber eine gemeinsame Lehre. Segment beschrieb 2018, wie es mehr als 140 kleine Services wieder zu einem einzigen Service zusammengelegt hatte, nachdem Betriebsaufwand und auseinanderlaufende gemeinsame Bibliotheken die Auslieferung gebremst hatten; Amazon Prime Video verlagerte 2023 eine Monitoring-Workload von verteilten Serverless-Komponenten in einen einzigen Prozess (siehe Abschnitt 1.3). Ebenso gewinnen Organisationen mit Hunderten von Teams, wie die Großfälle aus Abschnitt 6.3, durch Services echte Unabhängigkeit. Die Lehre heißt Passung: Der Deployment-Stil muss zur Organisation, zur Reife der Domäne und zu den geforderten Qualitäten passen.

Wann ein Monolith die richtige Antwort ist

Ein kleines Team. Bei einem bis wenigen Teams sind die Kommunikationskosten, die Microservices beseitigen, gering, ihre Betriebskosten aber dieselben wie für eine große Organisation. Ein Monolith erlaubt es allen, das ganze System zu verstehen und zu ändern.

Eine unsichere Domäne oder ein frühes Produkt. Solange Sie noch herausfinden, was das Produkt ist, werden sich Grenzen verschieben. Eine Grenze zwischen Modulen zu verschieben ist ein Refactoring; sie zwischen Services zu verschieben ist eine Migration mit Daten, APIs und Deployments.

Hohe Konsistenzanforderungen. Eine einzige Datenbanktransaktion über Bestellungen, Zahlungen und Lagerbestand ist in einem Monolithen einfach und korrekt. Über Services hinweg braucht sie Sagas, Outboxes und Kompensation (Abschnitt 6.7).

Begrenzte Betriebskapazität. Jeder Service braucht seine eigene Pipeline, sein Monitoring, sein Alerting, seine Skalierungsregeln und Bereitschaftswissen. Ein Team ohne Plattform dafür verbringt seine Zeit mit Klempnerarbeit.

Performancekritische Interaktionen. Prozessinterne Aufrufe dauern Nanosekunden bis Mikrosekunden; Netzwerkaufrufe dauern Millisekunden und können fehlschlagen. Gesprächige Interaktionen zwischen Teilen der Domäne sind innerhalb eines Prozesses weit günstiger.

In all diesen Fällen ist die empfohlene Form ein modularer Monolith, kein unstrukturierter. Er hält die Option offen, Services herauszulösen, wie Abschnitt 5.3 erklärt.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: eine zu früh gezogene Grenze verschieben

Die schwierigste Idee dieses Abschnitts ist, warum eine falsche Grenze zwischen Services so viel mehr kostet als zwischen Modulen, einer der Hauptgründe hinter MonolithFirst. Anfangs lernt ein Team die Domäne noch kennen, also werden manche Grenzen an der falschen Stelle liegen. Verfolgen Sie ein Datum, das umziehen muss.

- Ein junges Essenslieferprodukt hat einen Teil für Bestellungen und einen für die Disposition. Zunächst hält das Team die Fertigzeit jeder Bestellung, also wann die Küche das Essen fertig erwartet, in den Bestellungen.
- Monate später stellt sich heraus, dass die Disposition die Fertigzeit viel stärker nutzt, um Kuriere zu planen. Sie gehört in die Disposition, wie im Modell aus Abschnitt 5.7.
- In einem modularen Monolithen ist der Umzug ein Refactoring: Code und Spalte in das andere Modul verschieben, die Aufrufer anpassen, die Tests alles Kaputte finden lassen und alles in einem Release ausliefern.
- Wären Bestellungen und Disposition getrennte Services, wäre derselbe Umzug eine Migration: zwei APIs ändern, Live-Daten zwischen zwei Datenbanken kopieren, während weiter Bestellungen eingehen, beide Services in sicherer Reihenfolge releasen und den alten Weg funktionsfähig halten, bis jeder Aufrufer umgestellt hat.
- Früh in einem Produkt passieren solche Umzüge immer wieder, weil das Team noch lernt, wie das Geschäft funktioniert. Jeder ist innerhalb eines Deployments günstig und über ein Netzwerk teuer, und deshalb rät Fowler, mit einem Monolithen zu beginnen.
- **Die Lehre:** Solange Sie noch herausfinden, wohin die Grenzen gehören, halten Sie sie innerhalb eines Deployments, wo das Verschieben günstig ist. Setzen Sie sie von Anfang an als Module durch, damit jede spätere Aufteilung, wenn es Belege dafür gibt, Grenzen folgt, die sich bereits bewährt haben.

Einen Monolithen skalieren

Eine verbreitete Annahme lautet, Monolithen könnten nicht skalieren. Sie skalieren weiter, als die meisten Systeme es je brauchen, mithilfe der X-Achse des von Martin Abbott und Michael Fisher beschriebenen Scale Cube (viele identische Kopien betreiben) zusammen mit einigen Standardtaktiken. Kapitel 9 behandelt jede davon ausführlich (Abschnitt 9.1).

Horizontale Replikate hinter einem Load Balancer. Betreiben Sie viele identische Kopien der Anwendung und verteilen Sie die Anfragen auf sie. Das funktioniert nur, wenn die Anwendung zustandslos ist: Sessions, hochgeladene Dateien und Caches müssen in gemeinsamen Speichern liegen, damit jedes Replikat jede Anfrage bedienen kann (Abschnitt 9.2). Es bringt außerdem Redundanz: Fällt ein Replikat aus, steht der Dienst nicht mehr still.

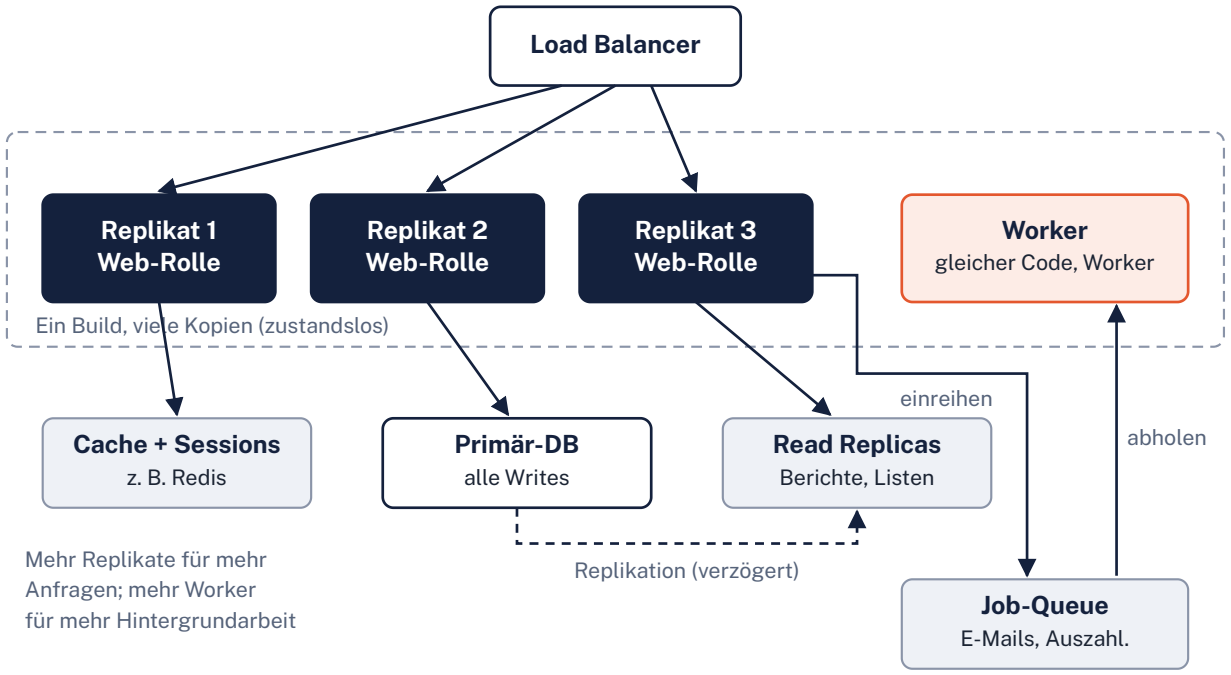
Read Replicas. Die meisten Systeme lesen weit mehr, als sie schreiben. Datenbank-Read Replicas kopieren die Änderungen des Primärsystems und bedienen Lesezugriffe wie Bestellhistorie und Berichte, um den Preis der Replikationsverzögerung: Ein Replikat kann einen Sekundenbruchteil hinterherhinken (Abschnitt 7.4).

Caching. Speisekarten, Produktseiten und Konfiguration ändern sich selten und werden ständig gelesen; sie im Arbeitsspeicher oder in einem gemeinsamen Cache vorzuhalten nimmt der Datenbank den Großteil der Last ab (Abschnitt 7.6). Statische Assets kommen in ein CDN.

Background Worker. Langsame oder stoßweise Arbeit (E-Mails, PDF-Rechnungen, Importe, Auszahlungen) kommt in eine Queue und wird von Worker-Prozessen abgearbeitet. Worker laufen meist mit derselben Codebasis und einem anderen Startbefehl, sodass das System eine Codebasis bleibt, die Web- und die Worker-Rolle aber getrennt skalieren (Abschnitt 9.3).

Größere Maschinen und Partitionierung. Vertikale Skalierung der Datenbank ist oft der einfachste nächste Schritt. Bei sehr großen Systemen lassen sich die Daten nach Mandant oder Region partitionieren, mit einer vollständigen Kopie des Monolithen pro Partition (die Z-Achse des Scale Cube). Stack Overflow ist ein bekanntes Beispiel dafür, wie weit ein weitgehend monolithisches Design reichen kann: Laut dem eigenen Bericht seiner Engineers verarbeiteten seine Load Balancer an einem Tag im Februar 2016 etwa 209 Millionen HTTP-Anfragen, wobei die zentrale Q&A-Anwendung (eine mandantenfähige Anwendung für alle Q&A-Sites) auf neun primären Webservern lief und vier SQL-Server-Maschinen das gesamte Netzwerk bedienten.

Ein horizontal skalierter Monolith



Eine Codebasis, viele Kopien: zustandslose Web-Replikate hinter einem Load Balancer, gemeinsamer Cache und Sessions, eine Primärdatenbank mit Read Replicas und Worker, gespeist von einer Queue.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Wie weit trägt ein Monolith?

Ein neues Essenslieferungs-Start-up hat 7 Engineers und erwartet bis Ende des zweiten Jahres eine Spitze von 1200 HTTP-Anfragen pro Sekunde zur Abendessenszeit am Freitag. Ein Lasttest des modularen Monolithen zeigt, dass ein Replikat (4 vCPU) 180 Anfragen pro Sekunde innerhalb des p95-Ziels von 300 ms bewältigt. Etwa 85 % der Anfragen sind Lesezugriffe (Speisekarten, Bestellverfolgung). Die Zahlen dienen der Veranschaulichung.

Replikate für die Spitze	1200 / 180 = 6,7, also 7 Replikate zur Spitzenzeit
Kapazitätsreserve (Headroom)	Die Auslastung deutlich unter der Sättigung halten (Abschnitt 2.4) und den Ausfall einer von drei Availability Zones überstehen: etwa 50 % Kapazitätsreserve einplanen (bei nur zwei Zonen wären es etwa 100 %), also 7 x 1,5 = etwa 11 Replikate zur Spitzenzeit, außerhalb der Spitze herunterskaliert
Schreibzugriffe auf die Datenbank	15 % von 1200 = 180 Anfragen pro Sekunde mit je etwa 3 Schreibvorgängen: etwa 540 Schreibvorgänge pro Sekunde, gut machbar für ein einzelnes verwaltetes PostgreSQL-Primärsystem auf einer mittelgroßen Instanz
Lesezugriffe auf die Datenbank	1020 Leseanfragen pro Sekunde; ein Cache für Speisekarten (die sich selten ändern) fängt die meisten ab, und zwei Read Replicas übernehmen Bestellhistorie und Berichte, wobei etwa eine Sekunde Replikationsverzögerung akzeptiert wird
Hintergrundarbeit	Belegmails, Kurierauszahlungen und Restaurantberichte wandern in einen Worker-Pool, gespeist von einer Queue, damit ein langsamer E-Mail-Anbieter den Bezahlvorgang nicht bremsen kann
Fazit	Eine Codebasis, als eine Einheit deployt und als etwa 11 Web-Replikate plus Worker betrieben, erreicht das Zweijahresziel. Das Architekturbudget fließt in Modulgrenzen und Zustandslosigkeit, nicht in die Aufteilung.

Die Grenze, die zuerst erreicht wird, ist selten die CPU. Häufiger ist es die Teamkoordination, wenn die Engineering-Gruppe wächst, oder ein einzelnes Modul (etwa die Disposition) mit ganz anderen Skalierungs- oder Release-Anforderungen.

Jede Zahl hier sollte durch Messungen aus Ihren eigenen Lasttests ersetzt werden; die Kapazitätsplanung behandelt Abschnitt 9.7.

Anzeichen, dass es Zeit für eine Aufteilung ist

Eine Aufteilung sollte eine Reaktion auf Belege sein, nicht auf Moden. Die Anzeichen, dass eine Deployment-Einheit zur Einschränkung geworden ist, sind überwiegend organisatorischer und betrieblicher Art:

Konkurrenz zwischen Teams. Mehrere Teams ändern denselben Code und warten aufeinander; Merge-Konflikte, Abstimmungsmeetings und von anderen Teams verursachte kaputte Builds nehmen zu, wenn die Organisation wächst. Sind die Module bereits gut abgegrenzt und hält die Konkurrenz an, fehlt womöglich die Unabhängigkeit im Deployment.

Deployment-Warteschlangen. Releases werden zu Release-Zügen gebündelt, weil jede Änderung einen erneuten Test des ganzen Systems erfordert; eine kleine Korrektur wartet tagelang hinter fremder Arbeit; die fehlgeschlagene Änderung eines Teams blockiert das Release aller.

Stark abweichende Skalierungs- oder Verfügbarkeitsanforderungen. Ein Modul (etwa die Echtzeit-Kurierverfolgung) braucht zur Spitzenzeit das Fünfzigfache der Kapazität des Rests oder muss laufen, wenn alles andere ausgefallen ist. Den ganzen Monolithen für die Anforderungen eines Moduls zu skalieren wird teuer.

Anderer Sicherheits- oder Compliance-Umfang. Kartendaten oder Gesundheitsakten können eine separate, streng kontrollierte Einheit rechtfertigen, um den Rest des Systems aus dem Prüfungsumfang herauszuhalten.

Andere Technologieanforderungen. Etwa ein Machine-Learning-Modell, das sich am besten aus Python bereitstellen lässt, innerhalb eines Java-Monolithen.

Manche Gründe sind für sich genommen keine guten Gründe: „Monolithen sind Legacy“, der Wunsch, eine neue Technologie einzusetzen, oder langsamer Code (ein Designproblem, das eine Aufteilung nicht behebt und durch zusätzliche Netzwerkaufrufe sogar verschlimmern kann).

Wie man aufteilt, wenn es so weit ist

Teilen Sie schrittweise auf, ein Modul nach dem anderen, beginnend mit dem Modul, das sowohl die klarste Grenze als auch den stärksten Grund zum Herauslösen hat. Modularisieren Sie zuerst, wenn der Code noch nicht modular ist: Wer einen Service aus einem Knäuel herauslöst, zieht das Knäuel mit. Lösen Sie dann entlang von Bounded Contexts heraus (Abschnitt 5.7): Setzen Sie eine Netzwerk-API vor die bestehende API des Moduls, verlagern Sie seine Daten in eine eigene Datenbank, ersetzen Sie prozessinterne Ereignisse durch einen

Message Broker und gemeinsame Transaktionen durch explizite Sagas. Leiten Sie den Verkehr schrittweise auf den neuen Service um und behalten Sie den alten Weg, bis sich der neue bewährt hat.

Das ist der Strangler-Fig-Ansatz samt seinen Verwandten, behandelt in Abschnitt 11.6. Das Ziel ist oft nicht „alles Microservices“, sondern ein Monolith mit einigen herausgelösten Services dort, wo die Belege es verlangt haben; die servicebasierte Architektur aus Abschnitt 6.2 ist ein verbreiteter, pragmatischer Endzustand.

Die Stile aus Kapitel 5 im Vergleich

Merkmal	Schichten	Modularer Monolith	Pipes and Filters	Microker-nel	Hexagonal / Clean
Hauptpartitionierung	Technisch	Fachlich	Verarbei-tungs-schritte	Kern plus Plug-ins	Innen gegen-über außen
Einfachheit	Hoch	Mittel bis hoch	Hoch	Mittel	Niedrig bis mittel
Gesamtkosten	Günstig	Günstig	Günstig	Günstig	Günstig bis mittel
Änderbarkeit bei fachli-chen Änderungen	Niedrig bis mittel	Hoch	Hoch in-nerhalb des Flusses	Hoch in ge-planten Richtungen	Hoch für Infrastruktur
Testbarkeit	Mittel	Mittel bis hoch	Hoch	Mittel bis hoch	Hoch
Deploybarkeit	Niedrig	Niedrig bis mittel	Niedrig bis mittel	Mittel	Durch den De-ployment-Stil bestimmt
Skalierbarkeit	Niedrig	Niedrig bis mittel	Niedrig bis mittel	Niedrig	Durch den De-ployment-Stil bestimmt
Fehlertoleranz	Niedrig	Niedrig	Niedrig bis mittel	Niedrig bis mittel	Durch den De-ployment-Stil bestimmt
Typischer Einsatz	Kleine bis mittlere Apps, ein Team	Wachsende Produkte, mehrere Teams	ETL, Me-dien, Logs, Compiler	Produkte mit vielen Varianten	Logik, die Frameworks überdauern muss

Die Bewertungen fassen die Urteile aus den einzelnen Abschnitten zusammen und gehen von typischen Implementierungen aus. Domain-Driven Design erscheint nicht als Spalte, weil es eine Methode zum Finden von Grenzen ist, die jeder dieser Stile nutzen kann. Reale Systeme kombinieren Stile: Ein modularer Monolith, dessen Module innen hexagonal sind, mit einem Pipes-and-Filters-Importjob und einem Microkernel für Steuerregeln, ist völlig normal.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: zwischen den Stilen aus Kapitel 5 wählen

Nutzen Sie sie für ein neues System oder eine größere Umstrukturierung, die als eine Deployment-Einheit laufen wird. Gehen Sie die Fragen von oben nach unten durch; mehrere Stile können auf verschiedenen Ebenen zutreffen.

Q1 **Ist das System hauptsächlich ein Datenfluss durch eine Reihe von Transformationen (importieren, bereinigen, anreichern, exportieren) statt Benutzer, die mit gemeinsamem Zustand interagieren?**

Ja Verwenden Sie Pipes and Filters für diesen Fluss (Abschnitt 5.4). Halten Sie jeden Filter zustandslos und das Datenformat zwischen den Filtern explizit.

Nein Weiter zur nächsten Frage.

Q2 **Hat das Produkt einen stabilen Kernablauf, aber viele Varianten bestimmter Schritte (pro Land, Kunde, Anbieter oder Dateiformat), oder müssen Dritte es erweitern?**

Ja Verwenden Sie einen Microkernel (Abschnitt 5.5) für diese Variationspunkte, idealerweise nachdem zwei oder drei Varianten gebaut sind, damit der Vertrag auf echten Fällen beruht.

Nein Weiter zur nächsten Frage.

Q3 **Ist es klein (ein Team, ein paar Dutzend Bildschirmmasken), mit einfachen Regeln, und wird es voraussichtlich kaum wachsen?**

Ja Verwenden Sie eine Schichtenarchitektur (Abschnitt 5.2) mit geschlossenen Schichten, durchgesetzt durch einen Architekturtest. Fügen Sie nicht mehr Struktur hinzu, als das Problem braucht.

Nein Weiter zur nächsten Frage.

Q4 **Werden mehrere Teams daran arbeiten, oder umfasst die Domäne mehrere Bereiche mit eigener Sprache (Speisekarten, Bestellungen, Disposition)?**

Ja Verwenden Sie einen modularen Monolithen (Abschnitt 5.3) mit Modulen entlang von Bounded Contexts (Abschnitt 5.7), von denen jedes seine Daten und seine API besitzt.

Nein Eine Schichten- oder eine modulare Struktur funktionieren beide; bevorzugen Sie Module nach Domäne, wenn Wachstum wahrscheinlich ist, denn eine spätere Umstrukturierung ist teuer.

Q5 **Ist die Geschäftslogik in jedem Modul oder jeder Anwendung umfangreich, oder hängt sie von externen Diensten ab, die sich wahrscheinlich ändern?**

Ja Strukturieren Sie das Innere mit Ports and Adapters (Abschnitt 5.6): die Domäne im Zentrum, Adapter für Web, Datenbank und Anbieter.

Nein Behalten Sie im Modul die Standardstruktur des Frameworks bei; fügen Sie Ports nur an Grenzen hinzu, die sich wahrscheinlich ändern.

Checkliste: ein Monolith, der gesund und aufteilbar bleibt

Das meiste davon kostet wenig, wenn es früh geschieht, und sehr viel, wenn es nachgerüstet wird.

- [ ] **Module folgen fachlichen Domänen**
  Die Pakete auf oberster Ebene sind nach Bounded Contexts benannt (orders, dispatch, payments), nicht nach technischen Schichten.
- [ ] **Jedes Modul hat eine kleine öffentliche API**
  Alles andere ist intern; ein Architekturtest lässt den Build bei einem verbotenen Import fehlschlagen.
- [ ] **Jede Tabelle gehört genau einem Modul**
  Getrennte Schemas oder Namenskonventionen machen die Zuständigkeit sichtbar; kein Modul liest die Tabellen eines anderen.
- [ ] **Die Anwendung ist zustandslos**
  Sessions, Uploads und Caches liegen in gemeinsamen Speichern, sodass jedes Replikat jede Anfrage bedienen kann und sich Replikate beliebig hinzufügen lassen.
- [ ] **Langsame Arbeit läuft in Background Workern**
  E-Mails, Berichte, Auszahlungen und Importe laufen über eine Queue, damit die Web-Replikate schnell bleiben und beide getrennt skaliert werden können.
- [ ] **Build und Tests bleiben schnell**
  Verfolgen Sie Build- und Testzeiten; halten Sie die Haupt-Pipeline unter etwa 10 bis 15 Minuten, indem Sie Module getrennt und parallel testen.
- [ ] **Das Deployment ist automatisiert und häufig**
  Kleine, häufige Releases mit automatisiertem Rollback; Feature-Flags entkoppeln das Deployment vom Release.
- [ ] **Observability pro Modul**
  Logs, Metriken und Traces sind mit dem Modul gekennzeichnet, sodass Sie sehen, welches Modul langsam ist, fehlschlägt oder teuer ist.
- [ ] **Anzeichen von Überlastung werden gemessen**
  Durchlaufzeit, Länge der Deployment-Warteschlange, Merge-Konflikte pro Team und Ressourcenverbrauch pro Modul werden regelmäßig überprüft, damit die Entscheidung zur Aufteilung auf Belegen beruht.

ACHTUNG

Häufige Fehler

- **„Monolith“ als Struktur behandeln.** Es ist eine Deployment-Entscheidung. Die innere Qualität ist eine separate Entscheidung, und sie bestimmt, ob Sie das System später ändern oder aufteilen können.
- **Aufteilen, um die Codequalität zu verbessern.** Services, die um eine verknäuelte Codebasis gezogen werden, werden zu einem verteilten Monolithen: die ganze Kopplung, plus das Netzwerk.
- **Das Falsche skalieren.** Replikate hinzuzufügen, während Sessions im Arbeitsspeicher liegen oder die Datenbank der Engpass ist, bringt nichts.
- **Die Überlastung nie messen.** Ohne Daten zu Durchlaufzeit, Deployments und Konkurrenz wird die Entscheidung zur Aufteilung zu einer Frage von Meinung und Mode.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum ist eine falsche Grenze zwischen Services teurer als eine falsche Grenze zwischen Modulen?

Eine Modulgrenze zu verschieben ist ein Refactoring innerhalb einer Codebasis und einer Datenbanktransaktion; eine Servicegrenze zu verschieben ändert Netzwerk-APIs, Datenspeicher, Deployments und oft die Teamzuständigkeit und muss ohne Ausfallzeit geschehen.

▼ Was muss für einen Monolithen gelten, bevor Sie ihn durch zusätzliche Replikate skalieren können?

Er muss zustandslos sein: Sessions, Dateien und Caches müssen in gemeinsamen Speichern liegen, damit jedes Replikat jede Anfrage bedienen kann.

▼ Nennen Sie zwei Anzeichen, dass ein Monolith zur Einschränkung geworden ist, und einen Grund für eine Aufteilung, der für sich allein nicht ausreicht.

Anzeichen: anhaltende Konkurrenz zwischen Teams und Deployment-Warteschlangen oder ein Modul mit ganz anderen Skalierungs- oder Compliance-Anforderungen. Für sich allein nicht ausreichend: Mode, der Wunsch nach einer neuen Technologie oder langsamer Code.

▼ **Worin überschneiden sich die Positionen MonolithFirst und „Don't start with a monolith“?**

Beide sagen, dass das schwierige Problem darin liegt, gute Grenzen zu finden und zu halten; ein Monolith mit durchgesetzten Modulgrenzen geht auf Tilkovs Bedenken ein und behält Fowlers geringere Kosten bei.

▼ **Probieren Sie es aus: Sechs Ingenieure erwarten eine Spitze von 900 Anfragen pro Sekunde, davon 80 % Lesezugriffe. Ein Replikat ihres modularen Monolithen verarbeitet 150 innerhalb des Zielwerts. Wie viele Replikate werden bei 30 % Reserve gebraucht? Sie möchten Microservices: Was raten Sie ihnen?**

900 / 150 = 6 Replikate am Limit. Ein Betrieb bei 70 % bedeutet 900 / (150 x 0,7) = 8,6, also 9 Replikate. Das lässt sich leicht hinter einem Load Balancer betreiben, und bei 80 % Lesevorgängen schützen Caching und Read Replicas die Datenbank. Es gibt keine Belege für Team-Konflikte, Deployment-Warteschlangen oder sehr unterschiedlichen Skalierungsbedarf, und sechs Fachleute können die Betriebskosten nicht tragen. Bleiben Sie beim Monolithen, und teilen Sie später entlang der Bounded Contexts, falls sich Belege zeigen.

QUELLEN

- Martin Fowler: MonolithFirst (2015): https://martinfowler.com/bliki/MonolithFirst.html
- Martin Fowler: Microservice Premium (2015): https://martinfowler.com/bliki/MicroservicePremium.html
- David Heinemeier Hansson: The Majestic Monolith (2016): https://signalvnoise.com/svn3/the-majestic-monolith/
- Stefan Tilkov: Don't start with a monolith (2015): https://martinfowler.com/articles/dont-start-monolith.html
- Nick Craver: Stack Overflow, the architecture (2016 edition): https://nickcraver.com/blog/2016/02/17/stack-overflow-the-architecture-2016-edition/

Weiterführende Literatur

- *Fundamentals of Software Architecture (2nd edition)*, Mark Richards and Neal Ford (2025), O'Reilly Media. Ein Kapitel pro Stil, darunter Schichten, modularer Monolith, Pipeline und Microkernel, jeweils mit einer Bewertung der Merkmale; das Vorbild für die Vergleiche in diesem Kapitel.
- *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal (1996), John Wiley and Sons. Der klassische Katalog von Architekturmustern, darunter Layers, Pipes and Filters und Microkernel, jeweils mit Kontext, Kräften und Konsequenzen.
- *An Introduction to Software Architecture*, David Garlan and Mary Shaw (1994), Carnegie Mellon University, Software Engineering Institute (CMU/SEI-94-TR-021). Der kurze Aufsatz, der Architekturstile als Komponenten, Konnektoren und Randbedingungen definierte, mit Pipes and Filters und Schichtensystemen unter seinen Beispielen.
  https://www.sei.cmu.edu/library/an-introduction-to-software-architecture/
- *Software Architecture: The Hard Parts*, Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani (2021), O'Reilly Media. Entwickelt das Architekturquantum und die Analyse von Kopplung und Datenhoheit, die entscheidet, wo sich ein System aufteilen lässt.
- *Monolith to Microservices*, Sam Newman (2019), O'Reilly Media. Ein ausgewogener Leitfaden dazu, wann, ob und wie man einen Monolithen zerlegt, mit einem starken Plädoyer für den modularen Monolithen.
- *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Eric Evans (2003), Addison-Wesley. Der Ursprung von Bounded Contexts, Context Maps und der Ubiquitous Language, mit denen man Modul- und Servicegrenzen findet.
- *Hexagonal Architecture*, Alistair Cockburn (2005), alistair.cockburn.us. Die ursprüngliche Beschreibung von Ports and Adapters, kurz und noch immer die klarste Darstellung der Idee.
  https://alistair.cockburn.us/hexagonal-architecture/
- *Deconstructing the Monolith: Designing Software that Maximizes Developer Productivity*, Kirsten Westeinde (2019), Shopify Engineering. Ein Bericht aus erster Hand darüber, wie eine sehr große Rails-Anwendung zu einem modularen Monolithen statt zu Microservices umgebaut wurde.
  https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity

Glossar

Abhängigkeitsregel (Dependency Rule)
Die Regel von Robert C. Martin für Clean Architecture: Quellcode-Abhängigkeiten zeigen nur nach innen, sodass innere Ringe nie etwas benennen, das in äußeren Ringen deklariert ist.

Adapter
Code, der zwischen einem Port und einer bestimmten Technologie übersetzt, etwa ein HTTP-Controller, ein SQL-Repository oder das SDK eines Zahlungsanbieters. Ein Port kann viele Adapter haben.

Aggregat
Eine Gruppe von Domänenobjekten, die als eine Einheit über eine einzige Wurzelentität geändert wird; diese setzt die Regeln durch, die für die Gruppe immer gelten müssen.

Anti-Pattern
Eine häufig verwendete Lösung, die vernünftig aussieht, aber zuverlässig mehr Probleme verursacht, als sie löst.

Anticorruption Layer
Eine Übersetzungsschicht, die ein nachgelagerter Kontext baut, damit das Modell eines anderen Kontexts oder Systems nicht in sein eigenes durchsickern kann.

Architecture Sinkhole
Ein Anti-Pattern in Schichtensystemen, bei dem Anfragen durch Schichten laufen, die sie nur weiterreichen, was Code und Verzögerung hinzufügt, ohne Verhalten hinzuzufügen.

Architekturmerkmal
Der Begriff von Richards und Ford für ein Qualitätsmerkmal, das die Architektur unterstützen muss, etwa Skalierbarkeit, Deploybarkeit oder Fehlertoleranz.

Architekturmuster
Eine dokumentierte, wiederverwendbare Lösung für ein wiederkehrendes strukturelles Problem in einem benannten Kontext, die die Teile, ihre Verantwortlichkeiten und ihre Beziehungen benennt. Wird oft gleichbedeutend mit Stil verwendet.

Architekturquantum
Ein unabhängig deploybarer Teil eines Systems mit hoher funktionaler Kohäsion, der alles umfasst, was er zum Laufen braucht, etwa seine Datenbank; der Bereich, in dem ein bestimmter Satz von Architekturmerkmalen gilt.

Architekturstil
Eine benannte Familie von Architekturen, definiert durch die Arten von Elementen und Konnektoren, die sie verwendet, und durch die Regeln, wie diese kombiniert werden dürfen, etwa Schichtenarchitektur oder Pipes and Filters.

Architekturtest
Ein automatisierter Test, der den Build scheitern lässt, wenn Code eine Strukturregel verletzt, etwa eine verbotene Abhängigkeit zwischen Schichten oder Modulen; Werkzeuge dafür sind unter anderem ArchUnit und import-linter.

Background Worker
Ein Prozess, der langsame oder schubweise anfallende Aufträge wie E-Mails oder Berichte aus einer Queue nimmt und sie außerhalb der Anfrage verarbeitet, die sie erzeugt hat.

Backpressure
Ein Mechanismus, mit dem ein langsamer Consumer einen schnellen Producer bremsen kann, statt Puffer und Queues unbegrenzt wachsen zu lassen.

Big Ball of Mud
Ein System ohne erkennbare Struktur, in dem fast alles von allem anderen abhängt; der Name stammt von Brian Foote und Joseph Yoder (1997).

Bounded Context
Eine ausdrückliche Grenze, innerhalb derer ein Domänenmodell und seine Ubiquitous Language einheitlich gelten; außerhalb davon können dieselben Wörter etwas anderes bedeuten.

Composition Root
Die einzige Stelle, meist der Startcode des Programms, an der konkrete Adapter erzeugt und in die Anwendung eingesteckt werden.

Context Map
Eine Darstellung der Bounded Contexts eines Systems und ihrer Beziehungen, etwa Customer-Supplier oder Anticorruption Layer, die sowohl Integrations- als auch Teambeziehungen abdeckt.

Core Domain
Die Subdomäne, die dem Unternehmen seinen Wettbewerbsvorteil verschafft und deshalb die besten Leute, die sorgfältigste Modellierung und Code verdient, der dem Unternehmen selbst gehört.

Dead Letter Queue
Ein separater Ort, an dem Nachrichten, deren Verarbeitung wiederholt fehlschlägt, zur Untersuchung abgestellt werden, damit sie den übrigen Fluss nicht blockieren.

Deployment-Einheit
Das, was als ein Stück gebaut, released und gestartet wird: zum Beispiel eine ausführbare Datei, ein Container-Image oder ein Webarchiv.

Domain Event
Eine Aufzeichnung über etwas Bedeutsames, das in der Geschäftsdomäne geschehen ist, in der Vergangenheitsform benannt, etwa OrderPlaced oder CourierAssigned.

Domain-Driven Design (DDD)
Ein 2003 von Eric Evans eingeführter Ansatz, der Software um ein gemeinsam mit Fachleuten entwickeltes Modell der Geschäftsdomäne herum baut und große Domänen in ausdrücklich abgegrenzte Modelle aufteilt.

Entität
Ein Domänenobjekt, das durch eine Identität bestimmt ist, die über die Zeit bestehen bleibt, während sich seine Attribute ändern, etwa eine Bestellung oder ein Kunde.

Entry Point (Python-Packaging)
Metadaten in einem installierten Python-Paket, die ein Objekt, etwa eine Plug-in-Klasse, unter einer benannten Gruppe bekannt machen, damit anderer Code es zur Laufzeit finden kann.

Entwurfsmuster
Eine wiederverwendbare Lösung für ein wiederkehrendes Problem im Maßstab weniger Klassen oder Objekte, etwa Strategy oder Observer; kleiner im Umfang als ein Architekturmuster.

Erweiterungspunkt
Ein benannter Steckplatz, den eine Host-Anwendung oder ein Plug-in deklariert, damit andere Plug-ins Verhalten dazu beitragen können, wie in der Eclipse-Plattform.

ETL
Extract, Transform, Load: eine Pipeline, die Daten aus Quellsystemen entnimmt, bereinigt und umformt und in einen Zielspeicher wie ein Data Warehouse lädt.

Event Storming
Ein von Alberto Brandolini entwickeltes Workshop-Format, bei dem Fachleute der Domäne und Entwicklungsteams Domain Events auf einer Zeitachse anordnen, um einen Geschäftsprozess zu erkunden und Grenzen zu finden.

Fachliche Partitionierung
Die Aufteilung der obersten Struktur eines Systems nach Geschäftsfähigkeit oder Subdomäne, etwa Bestellungen, Zahlungen und Lieferung.

Filter
Ein in sich geschlossener Verarbeitungsschritt, der Eingaben liest und sie dann erzeugt, transformiert, prüft oder verbraucht, ohne zu wissen, welche Schritte vor oder nach ihm kommen.

Generische Subdomäne
Eine Subdomäne, deren Probleme viele Unternehmen teilen und für die es gute Lösungen gibt, etwa Zahlungen oder Identität; meist zugekauft oder schlank selbst gebaut.

Geschlossene Schicht
Eine Schicht, die jede Anfrage durchlaufen muss: Die darüberliegende Schicht darf sie nicht überspringen, um direkt eine tiefere Schicht aufzurufen.

Getriebener (sekundärer) Adapter
Ein Adapter, der eine Schnittstelle implementiert, die die Anwendung aufruft, wenn sie etwas von außen braucht, etwa Daten speichern, eine Karte belasten oder eine Nachricht senden.

Idempotent
Bezeichnet eine Operation, die dieselbe Wirkung hat, ob sie einmal oder mehrmals angewendet wird, sodass sie sicher wiederholt werden kann.

Isolationsschichten (Layers of Isolation)
Die Eigenschaft, dass sich eine Änderung innerhalb einer Schicht nicht auf andere Schichten auswirkt, solange ihre Schnittstelle gleich bleibt; sie setzt geschlossene Schichten voraus.

Kernsystem
In einer Microkernel-Architektur der minimale Teil der Anwendung, den jede Installation braucht: der Hauptablauf, gemeinsame Dienste und die Stellen, an denen Plug-ins andocken.

Microkernel-Architektur
Ein Architekturstil, auch Plug-in-Architektur genannt, bei dem ein kleines Kernsystem den allgemeinen Ablauf bereitstellt und unabhängige Plug-ins über veröffentlichte Verträge Verhalten hinzufügen oder ändern.

Microservice Premium
Martin Fowlers Begriff (2015) für die zusätzlichen Kosten und Risiken, die Microservices mit sich bringen und die sich erst lohnen, wenn System und Organisation komplex genug sind.

Modul-API
Der kleine, bewusst öffentliche Teil eines Moduls, den andere Module nutzen dürfen; alles andere ist intern und darf sich frei ändern.

Modularer Monolith
Eine einzige deploybare Anwendung, deren Code entlang fachlicher Grenzen in Module aufgeteilt ist, jedes mit einer expliziten öffentlichen Schnittstelle und eigenen Daten.

MonolithFirst
Martin Fowlers Rat von 2015, ein neues System als gut modularisierten Monolithen zu beginnen und Services erst später herauszulösen, wenn die Grenzen verstanden und Services nötig sind.

Monolithische Architektur
Eine Architektur, bei der die gesamte Anwendung als eine einzige Einheit gebaut und deployt wird, unabhängig von ihrer inneren Struktur.

Offene Schicht
Eine Schicht, die die darüberliegende Schicht umgehen darf, typischerweise eine Schicht gemeinsamer Services, die nur manche Anfragen benötigen.

Open Host Service
Ein Context-Map-Muster, bei dem ein Upstream-Kontext allen seinen Konsumenten ein einziges klar definiertes, dokumentiertes Protokoll oder API anbietet.

Pipe
Ein Einwegkanal, der Daten von einem Filter zum nächsten transportiert, meist mit einem Puffer dazwischen.

Pipes and Filters
Ein Stil, bei dem Daten durch eine Kette unabhängiger Verarbeitungsschritte (Filter) fließen, die über Einwegkanäle (Pipes) verbunden sind.

Plug-in
Ein separat gebautes Modul, das einer Host-Anwendung über einen vom Host veröffentlichten Vertrag Verhalten hinzufügt, ohne dass der Code des Hosts geändert wird.

Plug-in-Registry
Der Teil eines Kernsystems, der festhält, welche Plug-ins verfügbar sind, mit ihren Namen, Versionen und Verträgen, damit der Kern sie findet, ohne sie im Code zu benennen.

Plug-in-Vertrag
Die veröffentlichte Schnittstelle zwischen einem Kernsystem und seinen Plug-ins: die Operationen, Datentypen, das Fehlerverhalten und die Version, auf die sich beide Seiten einigen.

Ports and Adapters (hexagonale Architektur)
Alistair Cockburns Stil, bei dem eine Anwendung nur über Ports (Schnittstellen, die ihr gehören) mit der Außenwelt kommuniziert und Adapter zwischen jedem Port und einer bestimmten Technologie übersetzen.

Prozessinternes Ereignis (In-Process Event)
Eine Nachricht, die innerhalb einer laufenden Anwendung veröffentlicht wird, damit andere Module auf ein Geschehen reagieren können, ohne dass der Herausgeber weiß, wer zuhört.

Published Language
Ein dokumentiertes, gemeinsames Format für den Informationsaustausch zwischen Kontexten, etwa ein versioniertes Event-Schema, oft über einen Open Host Service angeboten.

Read Replica
Eine Kopie einer Datenbank, die die Änderungen des Primärsystems empfängt und Leseabfragen bedient, was den Primärknoten entlastet, um den Preis leicht veralteter Daten.

Referenzarchitektur
Eine Musterarchitektur für eine Domäne oder Plattform, die typische Funktionen auf einen Standardsatz von Bausteinen abbildet und die Projekte anpassen, statt von Grund auf neu zu entwerfen.

Repository
Ein Objekt, das ganze Aggregate lädt und speichert und für die Domäne wie eine Sammlung aussieht, wobei es verbirgt, wie und wo sie gespeichert werden.

Scale Cube
Das Modell von Abbott und Fisher mit drei Arten der Skalierung: X-Achse (identische Kopien hinter einem Load Balancer), Y-Achse (Aufteilung nach Funktion) und Z-Achse (Aufteilung nach Daten, etwa nach Kunde).

Schicht
Eine Gruppe von Komponenten mit derselben technischen Verantwortung, die der darüberliegenden Schicht Dienste anbietet und die darunterliegende Schicht nutzt.

Schichtenarchitektur
Ein Stil, der Komponenten in horizontalen Schichten mit jeweils einer technischen Rolle anordnet, wobei jede Schicht nur die Schichten unter ihr nutzt.

Semantische Versionierung
Eine Konvention zur Versionsnummerierung, MAJOR.MINOR.PATCH, bei der sich die Major-Nummer bei Breaking Changes ändert, die Minor-Nummer bei abwärtskompatiblen Ergänzungen und die Patch-Nummer bei Fehlerbehebungen.

Shared Kernel
Ein kleiner Teil eines Modells samt Code, den zwei Kontexte ausdrücklich gemeinsam nutzen und der nur mit Zustimmung beider Teams geändert werden darf.

Stapelverarbeitung (Batch Processing)
Die Verarbeitung einer abgegrenzten Datenmenge in einem Durchlauf, etwa aller Bestellungen von gestern, mit anschließendem Abschluss.

Stream Processing
Die kontinuierliche Verarbeitung eines unbegrenzten Stroms von Datensätzen, jeder kurz nach seinem Eintreffen.

Subdomäne
Ein abgegrenzter Bereich der Geschäftstätigkeit eines Unternehmens, etwa Speisekarten, Bestellungen, Zahlungen oder Disposition. Subdomänen gehören zum Problem, unabhängig davon, ob es Software gibt.

Supporting Subdomain
Eine Subdomäne, die spezifisch für das Unternehmen und für es notwendig ist, in der es aber nicht im Wettbewerb steht; sie verdient einen soliden, aber einfacheren Entwurf.

Technische Partitionierung
Die Aufteilung der obersten Struktur eines Systems nach technischer Rolle, etwa Benutzeroberfläche, Geschäftslogik und Datenzugriff.

Tier
Eine physische Grenze zur Laufzeit, etwa ein separater Prozess oder Server. Eine Schicht ist eine logische Grenze im Code; mehrere Schichten können in einem Tier laufen.

Treibender (primärer) Adapter
Ein Adapter, der Anfragen von außen entgegennimmt, etwa ein Web-Controller, ein Test-Harness oder ein geplanter Job, und die Anwendung über einen treibenden Port aufruft.

Ubiquitous Language
Ein Vokabular, das Entwickler und Fachexperten teilen und innerhalb eines Bounded Context einheitlich in Gesprächen, Dokumenten, Tests und Code verwenden.

Value Object
Ein Domänenobjekt, das allein durch seine Werte definiert ist, keine eigene Identität hat und meist unveränderlich ist, etwa ein Geldbetrag oder eine Adresse.

Verteilte Architektur
Eine Architektur aus mehreren separat deployten Einheiten, die zusammenarbeiten, indem sie Nachrichten über ein Netzwerk senden.

Verteilter Monolith
Ein als separate Services deployoptes System, deren Kopplung so eng ist, dass sie gemeinsam geändert und ausgeliefert werden müssen; es hat die Kosten der Verteilung ohne ihre Vorteile.

Zustandslose Anwendung (Stateless)
Eine Anwendung, die zwischen Anfragen keine benutzerbezogenen Daten im eigenen Speicher hält, sodass jede Kopie jede Anfrage bearbeiten kann und Kopien beliebig hinzugefügt oder entfernt werden können.