BE SOFTWARE ARCHITECTURE WISE · KAPITEL 4

Architektur dokumentieren und kommunizieren

Dieses Kapitel zeigt, wie man eine Softwarearchitektur so beschreibt und kommuniziert, dass die richtigen Menschen sie verstehen und die Beschreibung zutreffend bleibt. Es behandelt, für wen Dokumentation gedacht ist und wie viel man schreiben sollte, Sichten und Blickwinkel nach ISO/IEC/IEEE 42010, 4+1 und Rozanski und Woods, das C4-Modell, die UML-Diagramme, die Architekten noch verwenden, Architecture Decision Records, arc42, Diagrams as Code und die Präsentation von Architektur für verschiedene Zielgruppen. Es ist durchgehend praxisnah, nutzt eine Essenslieferplattform als durchgängiges Beispiel und richtet sich an Entwickler, Architekten und technische Leitungen.

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

AUF EINEN BLICK

Das lernen Sie

- Erklären, wer Architekturdokumentation liest, was jede Leserin und jeder Leser braucht und wie man entscheidet, was sich festzuhalten lohnt.
- Stakeholder, Anliegen, Blickwinkel, Sichten und Modellarten nach ISO/IEC/IEEE 42010:2022 unterscheiden.
- Vergleichen Sie das 4+1-Sichtenmodell, die Blickwinkel und Perspektiven von Rozanski und Woods und die drei Sichtkategorien des SEI.
- Zeichnen Sie Systemkontext-, Container- und Komponentendiagramme, die den Notationsregeln von C4 folgen.
- Lesen und zeichnen Sie die UML-Sequenz-, Zustands-, Komponenten-, Deployment-, Aktivitäts- und Klassendiagramme, die Architekten verwenden.
- Schreiben Sie Architecture Decision Records und verwalten Sie ihren Lebenszyklus in einem Entscheidungsprotokoll.
- Wenden Sie arc42 und Diagrams-as-Code-Praktiken an, damit die Dokumentation nützlich und aktuell bleibt.
- Passen Sie eine Architekturerklärung an Führungskräfte, Entwickler, Betrieb, Sicherheitsfachleute und Prüfer an.

Bevor Sie beginnen

Kapitel 1 bis 3, insbesondere Stakeholder und Anliegen (Abschnitt 1.5) sowie Entscheidungen und Abwägungen (Abschnitt 1.6).

Dauer

Etwa 3 Stunden Lesezeit; 45 Minuten für den Test.

Schlüsselbegriffe

Architekturbeschreibung · Sicht · Blickwinkel (Viewpoint) · Anliegen · 4+1-Sichtenmodell · C4-Modell · Container (C4) · UML · Sequenzdiagramm · Zustandsdiagramm · Architecture Decision Record · arc42 · Diagrams as Code · Docs as Code

Warum das Dokumentieren von Architektur zur Arbeit gehört

Eine Architektur, die nur in den Köpfen weniger Menschen existiert, ist fragil. Niemand außerhalb des Raums kann sie prüfen, die nächste Person im Team kann sie nicht erlernen, ein Prüfer kann sie nicht kontrollieren, und sie geht mit den Menschen, die sie im Kopf haben. Kapitel 1 hat eine Architektur (was das System tatsächlich ist) von einer Architekturbeschreibung unterschieden (den Dokumenten, Diagrammen und Aufzeichnungen, über die Menschen sie teilen). In diesem Kapitel geht es um Letztere: wie man Beschreibungen erstellt, die tatsächlich gelesen werden, die zutreffend bleiben, während sich das System ändert, und die helfen, Entscheidungen zu treffen.

Das ist eher eine praktische als eine theoretische Fähigkeit, und man kann leicht in beide Richtungen danebenliegen. Teams, die zu wenig schreiben, verlieren die Gründe hinter ihrer Struktur und führen immer wieder dieselben Diskussionen. Teams, die zu viel schreiben, produzieren Wikis, denen niemand traut. Der Leitgedanke des Kapitels stammt aus *Documenting Software Architectures: Views and Beyond* des SEI: Schreiben Sie für den Leser. Entscheiden Sie, wer jedes Stück Dokumentation nutzen wird und was diese Person daraus braucht, schreiben Sie genau das knapp und präzise auf, und schreiben Sie nichts anderes.

Ein durchgängiges Beispiel zieht sich durch das ganze Kapitel: eine Essenslieferplattform mit einer Kunden-App, einer Restaurant-App und einer Kurier-App, Services für Bestellungen, Disposition und Benachrichtigungen sowie Kartenzahlungen über einen externen Anbieter. Sie sehen sie als Kontextdiagramm, als Containerdiagramm, als Komponentendiagramm, als UML-Sequenz- und Zustandsdiagramme, als Entscheidungsdokument und als einseitige arc42-Zusammenfassung, sodass Sie jede Technik am selben System vergleichen können.

Beschreiben und zeichnen: Abschnitte 4.1 bis 4.4

Die ersten vier Abschnitte behandeln, *was* zu beschreiben ist und *wie man es zeichnet*. Abschnitt 4.1 fragt, für wen Dokumentation gedacht ist, was das Agile Manifest darüber sagt und was nicht, wie man entscheidet, was es verdient, aufgeschrieben zu werden, und welche Anti-Patterns man vermeiden sollte. Die kurze Antwort: Schreiben Sie von Hand auf, was der Code Ihnen nicht sagen kann (Kontext, Qualitätsziele, Entscheidungen und ihre Gründe), und generieren Sie, was der Code bereits aussagt.

Abschnitt 4.2 führt den zentralen Gedanken ein, dass ein einzelnes Diagramm nie genügt. Eine Architektur wird durch mehrere Sichten beschrieben, von denen jede die Anliegen bestimmter Stakeholder beantwortet und jede nach einem Blickwinkel aufgebaut ist. Der Abschnitt behandelt das konzeptionelle Modell von ISO/IEC/IEEE 42010:2022, Kruchtens 4+1-Sichtenmodell, die Blickwinkel und Perspektiven von Rozanski und Woods sowie die Kategorien Modul, Komponenten und Konnektoren und Zuordnung des SEI und zeigt, wie man die minimale Menge an Sichten wählt, die ein System braucht.

Abschnitt 4.3 macht aus dieser Theorie den am weitesten verbreiteten praktischen Ansatz, Simon Browns C4-Modell: vier Zoomstufen (Systemkontext, Container, Komponente, Code), einige ergänzende Diagramme und eine kurze Reihe von Notationsregeln, die Diagramme selbsterklärend machen. Abschnitt 4.4 behandelt die Teile der UML, die Architekten noch verwenden, vor allem Sequenz-, Zustands-, Komponenten-, Deployment-, Aktivitäts- und kleine Klassendiagramme, und wann sich die Präzision der UML lohnt.

Entscheidungen, Vorlagen, Werkzeuge und Menschen: Abschnitte 4.5 bis 4.8

Diagramme zeigen Struktur; sie zeigen selten das *Warum*. Abschnitt 4.5 behandelt Architecture Decision Records: Michael Nygards Format von 2011, Alternativen wie MADR und Y-Statements, den Lebenszyklus eines Records von „vorgeschlagen“ bis „ersetzt“ und einen vollständig ausgearbeiteten Record für die Lieferplattform. Abschnitt 4.6 zeigt, wie eine Vorlage wie arc42 der gesamten Beschreibung eine Standardgliederung gibt, sodass Schreibende wissen, was sie abdecken müssen, und Lesende wissen, wo sie nachsehen müssen, und wie man sie für ein kleines System schlank einsetzt.

Abschnitt 4.7 befasst sich mit dem Problem, an dem die meiste Dokumentation scheitert: dem Veralten. Er behandelt das Schreiben von Diagrammen als Text mit PlantUML, Mermaid, Structurizr DSL und D2, den Unterschied zwischen Zeichen- und Modellierungswerkzeugen, das Veröffentlichen von Dokumentation aus der Build-Pipeline sowie die Gewohnheiten bei Verantwortung und Review, die Dokumentation zutreffend halten. Abschnitt 4.8 weitet den Blick von Dokumenten auf Menschen: die Botschaft auf Führungskräfte, Entwickler, Betriebsfachleute, Sicherheitsteams und Prüfer zuschneiden, Abwägungen erklären, einer Anfrage durch ein System folgen, Designdokumente und RFC-Prozesse, in einem Review präsentieren und Konsens herstellen, wenn Menschen unterschiedlicher Meinung sind.

Der eine Gedanke zum Mitnehmen

Jede Technik in diesem Kapitel dient einem Ziel: einem gemeinsamen, zutreffenden Verständnis des Systems unter den Menschen, die es bauen, betreiben, bezahlen und von ihm abhängen. Ein Diagramm, ein Entscheidungsdokument oder eine Vorlage lohnt sich nur, wenn ein echter Leser es braucht, und nur so lange, wie jemand es aktuell hält. Kleine, aktuelle Dokumentation mit klarer Verantwortung, die neben dem Code liegt, schlägt ein umfassendes Dokument, dem niemand traut. Wenn Sie sich eine einzige Frage aus diesem Kapitel merken, dann diese: *Wer wird das lesen, und was wird er damit tun?*

Warum und wie viel dokumentieren

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 1.1 Architektur definieren
- 1.5 Stakeholder und Anliegen

Begriffe von früher, die er verwendet: Architekturbeschreibung, Stakeholder, Versionskontrolle, Pull Request

DIE IDEE IN KÜRZE

Denken Sie an die Unterlagen zu einem Haus. Wer es kauft, will einen Grundriss, die Elektrikerin will den Schaltplan, die Versicherung will das Gutachten, und wer es in zwanzig Jahren renoviert, will wissen, warum diese Wand nie versetzt wurde. Niemand will eine Liste aller Ziegelsteine, und ein Plan, der eine längst abgerissene Wand zeigt, ist schlimmer als gar kein Plan. Bei der Dokumentation von Softwarearchitektur ist es genauso: eine kleine Zahl zutreffender Dokumente, jedes für jemanden geschrieben, der es wirklich braucht, die das erklären, was man nicht einfach durch einen Blick in den Code sieht.

WAS SIE WISSEN MÜSSEN

- Eine Architekturbeschreibung ist die Gesamtheit der Dokumente, Diagramme und Aufzeichnungen, über die Menschen eine Architektur teilen; sie dient der Wissensvermittlung, der Kommunikation zwischen Stakeholdern und der Analyse.
- Verschiedene Leser (Neulinge, Entwickler, Betrieb, Sicherheit, Prüfer, künftige Wartende und Entscheider) bringen verschiedene Fragen mit, daher sollte Dokumentation nach Lesern und ihren Aufgaben gegliedert sein.
- Das Agile Manifest schätzt funktionierende Software höher als umfassende Dokumentation, sagt aber ausdrücklich, dass die Dokumentation trotzdem Wert hat; es lehnt Dokumentation als Selbstzweck ab, nicht Dokumentation.
- Schreiben Sie von Hand auf, was sich nicht aus dem Code ableiten lässt (Kontext, Qualitätsziele, Entscheidungen und ihre Gründe, Prinzipien); generieren Sie, was der Code bereits aussagt und was sich oft ändert.
- Die sieben Regeln von Views and Beyond: für den Leser schreiben, unnötige Wiederholung vermeiden, Mehrdeutigkeit vermeiden, eine Standardgliederung verwenden, Begründungen festhalten, die Dokumentation aktuell halten, aber nicht zu aktuell, und sie

auf ihre Zweckeignung prüfen.

- Docs as Code und lebende Dokumentation halten die Dokumentation in der Versionskontrolle, prüfen sie zusammen mit Codeänderungen und leiten sie, wo möglich, aus dem System ab; das ist der zuverlässigste Schutz gegen das Veralten.
- Häufige Anti-Patterns sind das veraltete Wiki, das Big-Bang-Dokument, das niemand liest, Diagramme ohne Legende, vermischte Abstraktionsebenen und Beschreibungen, die das Warum weglassen.

Warum eine Architektur aufgeschrieben werden muss

Jedes System hat eine Architektur, ob sie nun jemand beschreibt oder nicht (siehe Abschnitt 1.1). Die Architekturbeschreibung ist etwas anderes: die Dokumente, Diagramme, Modelle und Entscheidungsdokumente, über die Menschen ein Verständnis dieser Architektur teilen. In einem Team von drei Personen, die zusammensitzen und jede Zeile selbst geschrieben haben, kann ein Großteil der Beschreibung in ihren Köpfen leben. Sobald das Team wächst, Menschen gehen, ein zweites Team hinzukommt, ein Prüfer auftaucht oder das System seine Autoren überdauert, wird Wissen, das nur in Köpfen existiert, zum Risiko. Ralph Johnsons Definition von Architektur als gemeinsames Verständnis der erfahrenen Entwickler enthält eine Warnung: Ein Verständnis, das nicht aufgeschrieben ist, lässt sich nur durch Gespräche teilen, und Gespräche skalieren nicht.

Documenting Software Architectures: Views and Beyond (Paul Clements und Kollegen am Software Engineering Institute, 2. Auflage, 2010) nennt drei Verwendungszwecke für Architekturdokumentation, und sie sind noch immer die klarste Art, darüber nachzudenken, warum sich der Aufwand lohnt. Sie dient der **Wissensvermittlung**: Sie führt neue Teammitglieder, externe Prüfer und neue Architekten in das System ein. Sie ist das wichtigste Mittel der **Kommunikation zwischen Stakeholdern**: Entwickler, Betrieb, Sicherheitsfachleute und Management diskutieren über das System anhand der Dokumentation, und Entscheidungen aus einem Meeting erreichen Menschen, die nicht dabei waren. Und sie ist die **Grundlage für Analyse und Umsetzung**: Sie sagt den Implementierenden, was sie tun dürfen und was nicht, und sie ist das, was eine Bewerterin untersucht, wenn sie fragt, ob das System seine Qualitätsziele erreichen wird (die Bewertungsmethoden aus Kapitel 11 brauchen eine Beschreibung als Arbeitsgrundlage).

Daraus folgen zwei Konsequenzen. Erstens ist Dokumentation ein Mittel, kein Zweck: Ein Dokument, das keinem dieser Zwecke dient, ist Verschwendung, wie gut es auch formatiert sein mag. Zweitens entscheidet der Leser, ob sie funktioniert hat. Eine Beschreibung, die ihr Autor vollständig findet, mit der ein Neuling aber nichts anfangen kann, ist bei der Wissensvermittlung gescheitert, und ein Diagramm, das das Betriebsteam nicht auf echte Server abbilden kann, ist bei der Analyse gescheitert.

Wer Architekturdokumentation liest und was er braucht

Leser	Fragen, die sie mitbringen	Was ihnen am besten dient
Neulinge (neue Entwickler, externe Kräfte, eine neue Architektin)	Was tut dieses System, für wen, und wo fange ich an? Was sind die Hauptteile, und welche Begriffe verwenden die Leute?	Eine einseitige Übersicht, ein Systemkontextdiagramm, ein Containerdiagramm, ein Glossar und eine kurze Lesereihenfolge
Aktuelle Entwickler	Wohin gehört dieses neue Feature? Wovon darf ich abhängen? Warum ist es so gebaut, und darf ich es ändern?	Modul- oder Komponentenstruktur, Konventionen, querschnittliche Konzepte (Fehlerbehandlung, Sicherheit, Logging) und Entscheidungsdokumente
Betrieb und SRE	Wo läuft es, wovon hängt es ab, wie fällt es aus, und wie stellen wir es wieder her?	Deployment-Sicht je Umgebung, externe Abhängigkeiten, Runbooks, Monitoring und Wiederherstellungsziele
Sicherheits- und Datenschutzfachleute	Wo liegen die Vertrauensgrenzen? Wohin fließen personenbezogene Daten, und wo werden sie gespeichert? Wer darf was aufrufen?	Kontext- und Datenflusssichten, Identitäts- und Zugriffskonzept, Liste der Drittanbieterdienste
Prüfer und Aufsichtsbehörden	Können Sie zeigen, dass die Kontrollen existieren, dass Entscheidungen bewusst getroffen wurden und dass Änderungen nachvollziehbar sind?	Entscheidungsdokumente mit Datum und Verantwortlichen, Nachverfolgbarkeit von Anforderungen zu Design und Tests, Versionshistorie
Künftige Wartende	Warum um Himmels willen wurde das so gemacht? Welche Randbedingungen gelten noch? Was kann man gefahrlos entfernen?	Vor allem Begründungen: Entscheidungen mit ihrem Kontext, den verworfenen Optionen und den getroffenen Annahmen
Entscheider (Auftraggeber, Produkt- und Engineering-Management)	Was sind die Risiken, Kosten und Optionen? Was muss ich entscheiden, und bis wann?	Eine einseitige Zusammenfassung, Qualitätsziele, die größten Risiken und technischen Schulden, Entscheidungen, die noch einen Verantwortlichen brauchen

Funktionierende Software vor umfassender Dokumentation

Das Agile Manifest (2001) nennt vier Wertepaare, eines davon lautet „Funktionierende Software mehr als umfassende Dokumentation“. Es wird oft als Erlaubnis zitiert, gar nicht zu dokumentieren. Vollständig gelesen, sagt das Manifest etwas Engeres. Seine Autoren erklären, dass sie zwar die Werte auf der rechten Seite wichtig finden, die Werte auf der linken Seite aber höher einschätzen. Der Wert ist eine Priorität für den Fall, dass beides konkurriert, keine Ablehnung von Dokumentation.

Was der Wert ablehnt, ist das dokumentationsgetriebene Projekt der 1990er-Jahre: Hunderte Seiten Spezifikation, abgenommen, bevor Code geschrieben wird, als Fortschritt an sich behandelt und aufgegeben, sobald der Code davon abweicht. Das entscheidende Wort ist *umfassend*: Dokumentation, die versucht, alles zu beschreiben, als Selbstzweck, und die höher geschätzt wird als das laufende System. Nichts im Manifest spricht gegen ein Kontextdiagramm, eine kurze Aufzeichnung, warum das Team asynchrones Messaging gewählt hat, oder ein Runbook, mit dem die Person in Rufbereitschaft um drei Uhr morgens den Dienst wiederherstellen kann.

Fachleute sind sich durchaus uneins, wie weit man gehen sollte. Manche agilen Coaches argumentieren, dass Gespräche und Tests fast immer ausreichen und dass schriftliche Architekturdokumentation schneller veraltet, als sie sich auszahlt. Andere, besonders wer langlebige Systeme wartet oder in regulierten Branchen arbeitet, weisen darauf hin, dass gerade die Menschen, die die Dokumentation am dringendsten brauchen (Nachfolger, Prüfer, das Betriebsteam), bei den Gesprächen fehlen. Beide Positionen sind sich über die zugrunde liegende Ökonomie einig: Dokumentation kostet etwas beim Schreiben, beim Aktuellhalten und beim Lesen, und sie lohnt sich nur dort, wo ihr Wert für die Leser diese Kosten übersteigt.

Gerade genug Dokumentation

Scott Amblers Formulierung „just barely good enough“ (gerade eben gut genug) beschreibt das Ziel: Ein Modell oder Dokument ist ausreichend, wenn es seinen Zweck für sein Publikum erfüllt, und Aufwand über diesen Punkt hinaus kostet mehr, als er einbringt. Sein Punkt, dass das Publikum und nicht der Autor entscheidet, ob ein Artefakt ausreicht, ist dasselbe Prinzip wie das Schreiben für den Leser. George Fairbanks führt das entsprechende Argument für die Architekturarbeit insgesamt in *Just Enough Software Architecture* (2010): Tun Sie so viel, wie die Risiken rechtfertigen (siehe Abschnitt 1.8).

Just-enough-Dokumentation ist daher keine feste Menge. Sie wächst mit der Größe und Lebensdauer des Systems, der Zahl der Teams, der Fluktuation, den Kosten eines Ausfalls und den Anforderungen der Aufsichtsbehörden. Ein Prototyp von zwei Personen braucht eine README und vielleicht ein Diagramm. Eine Zahlungsplattform, die von acht Teams unter Finanzregulierung betrieben wird, braucht Kontext-, Container- und Deployment-Sichten, ein Entscheidungsprotokoll, das Jahre zurückreicht, dokumentierte Sicherheitskonzepte und Nachweise, dass die Dokumente geprüft werden.

Ein praktisches Signal ist die wiederholte Frage. Wenn Neulinge in ihrer ersten Woche immer dasselbe fragen oder dieselbe Designdiskussion alle paar Monate wiederkehrt, gehört die Antwort aufgeschrieben. Ein anderes ist der teure Fehler: Hat ein Team eine ungeschriebene Regel gebrochen („Rufe den Zahlungsanbieter nie aus der Restaurant-App auf“), weil niemand sie ihm gesagt hat, dann war die Regel Architektur und hätte aufgeschrieben werden müssen.

Was sich zu dokumentieren lohnt und was nicht

Informationen unterscheiden sich auf zwei Arten, die bestimmen, wie man sie dokumentieren sollte. Die erste ist, ob sie sich **aus dem Code ableiten** lassen. Klassennamen, Methodensignaturen und Datenbanktabellen kann man jederzeit aus dem Quellcode lesen; die Gründe für eine Entscheidung, die Qualitätsziele des Systems und die verworfenen Optionen nicht, weil sie nie im Code standen. Die zweite ist, **wie oft sie sich ändern**. Der Zweck des Systems und seine wichtigsten externen Partner ändern sich selten; die Liste der Klassen in einem Modul ändert sich wöchentlich.

Handgeschriebene Architekturdokumentation sollte sich auf das konzentrieren, was wichtig ist und sich zugleich nicht aus dem Code ableiten lässt: **Kontext und Abgrenzung** des Systems (wer es nutzt und von welchen anderen Systemen es abhängt), seine **Qualitätsziele** und Randbedingungen, die **wichtigsten Entscheidungen und ihre Gründe**, die grobe **Struktur** (die wichtigsten deploybaren Teile und ihre Verantwortlichkeiten), das **Deployment** auf die Infrastruktur, die wichtigen **Laufzeitszenarien** und die **querschnittlichen Konzepte**, denen jeder Teil folgen muss, etwa wie Fehler behandelt, wie Benutzer authentifiziert und wie personenbezogene Daten geschützt werden. Diese Liste liegt nahe an der Struktur der arc42-Vorlage, die in Abschnitt 4.6 behandelt wird.

Nicht lohnenswert ist es, von Hand aufzuschreiben, was der Code oder die Werkzeuge bereits präzise aussagen und was sich oft ändert: Klassendiagramme einer ganzen Codebasis, Listen von Endpoints, die eine OpenAPI-Datei bereits enthält, oder Beschreibungen jeder einzelnen Funktion. Wo solche Sichten nützlich sind, generieren Sie sie in der Build-Pipeline aus dem Quellcode, damit sie nicht abdriften können. Lassen Sie außerdem Wunschdenken weg (Diagramme davon, wie das System aussehen sollte, präsentiert, als sähe es schon so aus), Protokolle von Meetings, deren Ergebnis anderswo festgehalten ist, und doppelten Text, der irgendwann seiner Kopie widersprechen wird.

Entscheiden, wie eine Information dokumentiert wird



Schreiben Sie von Hand auf, was der Code nicht sagen kann; generieren Sie, was der Code bereits aussagt und was sich oft ändert.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: aufschreiben, generieren oder darauf verweisen

Die Idee, die hier die meiste Sorgfalt verlangt, ist, für jedes Stück Wissen über ein System zu entscheiden, ob ein Mensch es aufschreiben, ein Werkzeug es generieren sollte oder ob ein kurzer Verweis genügt. Zwei Fragen klären das: Lässt es sich aus dem Code lesen, und wie oft ändert es sich? Probieren Sie es an fünf Fakten über die Essenslieferplattform aus.

- Warum Bestellereignisse asynchron statt über direkte Aufrufe gesendet werden. Der Code sagt nicht, warum, oder was verworfen wurde, und die Gründe ändern sich selten. Ein Mensch schreibt sie einmal auf, als Architecture Decision Record.
- Die Regel „Rufe den Zahlungsanbieter nie aus der Restaurant-App auf“. Der Code zeigt, dass die App den Anbieter heute nicht aufruft, nicht aber, dass sie es nie tun darf. Ein Team, das die Regel nie gehört hat, könnte sie brechen, also wird sie ebenfalls aufgeschrieben.
- Die Liste der Endpoints des Bestellservice. Ihre OpenAPI-Datei gibt sie bereits präzise an, und sie ändern sich oft, also generiert die Pipeline die Referenz aus dieser Datei.
- Das Diagramm der wichtigsten Apps, Services und Datenbanken der Plattform und wie sie verbunden sind. Keine einzelne Codebasis zeigt dieses Gesamtbild, und doch ändert es sich, sobald ein Teil hinzukommt oder wegfällt. Daher wird es im Repository gehalten, idealerweise aus einem dortigen Modell gezeichnet, und der Pull Request, der einen Teil hinzufügt oder entfernt, ändert es mit.
- Die Befehle, um jeden Service zu bauen und zu starten. Sie stehen in den Build-Dateien und ändern sich selten, also genügt eine kurze README, die darauf verweist.
- **Die Lehre:** Investieren Sie handschriftlichen Aufwand dort, wo der Code schweigt: Gründe, Regeln, Ziele und das Gesamtbild. Generieren Sie, was der Code bereits aussagt, oder verweisen Sie darauf: Handgemachte Kopien driften ab.

Sieben Regeln für gute Dokumentation

Views and Beyond stellt sieben Regeln für gute Dokumentation auf. Es lohnt sich, sie auswendig zu kennen, denn fast jedes Scheitern von Dokumentation verstößt gegen eine davon.

1

1. Für den Leser schreiben

Dokumentation wird einmal geschrieben und oft gelesen, also optimieren Sie für den Leser. Entscheiden Sie, wer die Leser sind und was sie mit dem Dokument tun wollen, gliedern Sie es zum Nachschlagen statt zum linearen Lesen, und vermeiden Sie internen Jargon. Ein Zeichen, dass diese Regel verletzt ist: Das Dokument ist in der Reihenfolge gegliedert, in der dem Autor die Dinge eingefallen sind.

2

2. Unnötige Wiederholung vermeiden

Jede Information sollte einen festen Ort haben, auf den von anderswo verlinkt wird. Wiederholung macht Dokumente länger und, schlimmer noch, lässt Kopien auseinanderdriften, bis Leser nicht mehr erkennen, welche stimmt. Bewusste Wiederholung zur Bequemlichkeit des Lesers, etwa eine Zusammenfassung, ist in Ordnung, wenn sie klar gekennzeichnet ist.

3

3. Mehrdeutigkeit vermeiden

Ein Kästchen-und-Linien-Diagramm ohne Legende lässt sich auf mehrere Arten lesen: Ist eine Linie ein Aufruf, ein Datenfluss oder eine Deployment-Abhängigkeit? Definieren Sie die Notation in einer Legende, verwenden Sie einen Namen für eine Sache, und geben Sie an, was jedes Element und jede Beziehung bedeutet.

4

4. Eine Standardgliederung verwenden

Eine bekannte Struktur (eine Vorlage wie arc42 oder die Vorlage von Views and Beyond) sagt Lesern, wo sie nachsehen müssen, und zeigt Autoren, was fehlt. Ein leerer Abschnitt mit dem Vermerk „nicht zutreffend, weil …“ ist aufschlussreicher als eine stille Lücke.

5

5. Begründungen festhalten

Schreiben Sie auf, warum, einschließlich der verworfenen Optionen. Künftige Wartende können meist herausfinden, was gebaut wurde; warum, können sie nicht herausfinden, und ohne das Warum scheuen sie entweder jede Änderung oder machen eine Entscheidung rückgängig, deren Gründe noch gelten. Dafür gibt es Entscheidungsdokumente.

6

6. Aktuell halten, aber nicht zu aktuell

Veraltete Dokumentation führt in die Irre. Aber jedes Dokument für jede halb ausgegorene Idee zu aktualisieren, erzeugt ständiges Hin und Her; halten Sie Entscheidungen fest, sobald sie getroffen sind, und aktualisieren Sie die Beschreibung in einem vorhersehbaren Rhythmus, etwa mit jedem Release oder jeder bedeutenden Änderung.

7 **7. Auf Zweckeignung prüfen**
Nur die vorgesehenen Leser können sagen, ob ein Dokument funktioniert. Bitten Sie einen Neuling, das System anhand der Dokumente aufzusetzen und zu erklären, oder bitten Sie eine Person in Rufbereitschaft, die Deployment-Sicht in einer Übung zu verwenden, und beheben Sie, worüber sie stolpern.

Lebende Dokumentation und Docs as Code

Das klassische Scheitern von Architekturdokumentation ist das Veralten: Die Beschreibung wird zu Beginn geschrieben, das System ändert sich, und niemand aktualisiert das Dokument, weil es getrennt von der Arbeit liegt. Zwei verwandte Ideen packen dieses Problem an der Wurzel.

Docs as Code, eine von der Write-the-Docs-Community geförderte Praxis, bedeutet, Dokumentation wie Quellcode zu behandeln: in Klartextformaten wie Markdown oder AsciiDoc geschrieben, in der Versionskontrolle gehalten (meist im selben Repository wie der beschriebene Code), über dieselben Pull Requests geändert, von denselben Menschen geprüft und von derselben Pipeline gebaut und veröffentlicht. Von einem Pull Request, der ändert, wie Bestellungen disponiert werden, kann man dann verlangen, dass er im selben Commit die Laufzeitsicht aktualisiert, und der Reviewer sieht beides.

Lebende Dokumentation, ein Begriff, den Cyrille Martraires gleichnamiges Buch (2019) bekannt gemacht hat, geht weiter: Wo immer möglich, sollte Dokumentation aus dem System selbst abgeleitet werden, damit sie nicht abdriften kann. Aus dem Code generierte Abhängigkeitsdiagramme, aus OpenAPI-Dateien generierte API-Referenzen, durch automatisierte Tests geprüfte Architekturregeln und Diagramme, die aus einem neben dem Code gehaltenen Modell generiert werden, sind Beispiele dafür. Handgeschriebener Text bleibt dem vorbehalten, was sich nicht ableiten lässt, vor allem Absicht und Begründung.

Keine der beiden Ideen erspart das Urteil darüber, was man aufschreiben sollte. Abschnitt 4.7, Diagrams as Code, behandelt die Werkzeuge und Gewohnheiten im Detail.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: Sollen wir das aufschreiben?

Stellen Sie diese Fragen der Reihe nach zu einem Stück Architekturwissen.

Q1 **Würde jemand einen teuren Fehler machen oder immer wieder dieselbe Frage stellen, wenn das nicht aufgeschrieben wäre?**

- **Ja** Weiter: Es ist ein Kandidat für Dokumentation.
- **Nein** Überlassen Sie es vorerst Gesprächen und dem Code; greifen Sie es wieder auf, wenn die Frage öfter auftaucht.

Q2 **Lässt es sich zuverlässig aus dem Code, der Konfiguration oder einem Werkzeug lesen?**

- **Ja** Generieren Sie es, oder verlinken Sie den Ort, an dem es steht; kopieren Sie es nicht von Hand.
- **Nein** Weiter: Es muss von einem Menschen geschrieben werden.

Q3 **Ist es eine Entscheidung mit Gründen und verworfenen Optionen?**

- **Ja** Schreiben Sie ein Entscheidungsdokument (siehe Abschnitt 4.5).
- **Nein** Weiter.

Q4 **Ändert es sich mit den meisten Releases?**

- **Ja** Halten Sie es im Repository neben dem Code, den es beschreibt, und machen Sie seine Aktualisierung zum Teil des Pull Requests.
- **Nein** Nehmen Sie es in die Architekturübersicht oder den passenden Abschnitt der Vorlage auf, und prüfen Sie es in einem festen Rhythmus.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: ein Just-enough-Dokumentationssatz für die Essenslieferplattform

Ein Essenslieferunternehmen betreibt eine Plattform mit einer Kunden-App, einer Restaurant-App, einer Kurier-App, Services für Bestellungen, Disposition und Benachrichtigungen sowie einem externen Zahlungsanbieter. Es hat 30 Engineers in fünf Teams und bereitet sich auf ein Audit durch einen Prüfer für Kartenzahlungen vor. Der Architekt schlägt diesen Dokumentationssatz vor, der vollständig in einem einzigen Dokumentations-Repository oder neben dem Code liegt und an keiner Stelle länger ist als nötig.

Architekturübersicht (2 bis 3 Seiten)	Zweck, Hauptnutzer, Qualitätsziele in Prioritätsreihenfolge (zum Beispiel: nie eine Bestellung verlieren oder doppelt abbuchen; Bestellbestätigung p99 unter 1,5 Sekunden in der Spitzenzeit; Restaurant-App auch bei schlechter Verbindung nutzbar), wichtige Randbedingungen und eine Lesereihenfolge für Neulinge. Verantwortlich: der Architekt. Prüfung jedes Quartal.
Systemkontext- und Containerdiagramme	Zwei C4-Diagramme (Abschnitt 4.3) mit Legenden, generiert aus einem Modell im Repository. Aktualisiert in dem Pull Request, der einen Container oder eine externe Abhängigkeit hinzufügt oder entfernt.
Entscheidungsprotokoll (Decision Log)	Etwa 40 kurze, nummerierte Entscheidungsdokumente im Repository: asynchrone Bestellereignisse, die Wahl des Zahlungsanbieters, eine Datenbank pro Service und so weiter. Nie umgeschrieben; ersetzt, wenn sich eine Entscheidung ändert.
Deployment-Sicht	Ein Diagramm und eine Tabelle je Umgebung: Regionen, Cluster, Managed Databases, der Message Broker und die Netzwerkgrenze um die Kartendaten. Verantwortlich ist das Plattformteam.
Laufzeitszenarien	Sequenzdiagramme für die vier wichtigsten Abläufe: Bestellung aufgeben, Restaurant lehnt ab, Kurierzuweisung, Erstattung. Jedes passt auf einen Bildschirm.
Querschnittliche Konzepte	Je eine Seite zu Authentifizierung, Idempotenz und Wiederholungen, Logging und personenbezogenen Daten. Das sind die Regeln, denen jedes Team folgen muss.
Runbooks	Eines pro Alarm, der jemanden wecken kann, verlinkt aus dem Alarm selbst.
Nicht von Hand geschrieben	Klassendiagramme, Endpoint-Listen und Datenbankschemata (in der Pipeline generiert); Meeting-Notizen; die Geschichte, wie das System gewachsen ist (das Entscheidungsprotokoll und die Versionshistorie erzählen sie bereits).

Jeder Eintrag hat einen benannten Leser, einen Verantwortlichen und einen Auslöser für Aktualisierungen. Dokumentation ohne Verantwortlichen verfällt zuerst.

Das Audit wird durch das Entscheidungsprotokoll, die Deployment-Sicht und die Versionshistorie bedient, nicht durch ein separates Audit-Dokument, das mit ihnen konsistent gehalten werden müsste.

Der gesamte Satz lässt sich in etwa zwei Stunden lesen, was es plausibel macht, dass Neulinge ihn tatsächlich lesen.

ACHTUNG

Anti-Patterns der Dokumentation

Das sind die Fehler, die man in der Praxis am häufigsten sieht, und die Regel, gegen die jeder verstößt.

- **Das veraltete Wiki.** Hunderte Seiten, vor Jahren zuletzt bearbeitet, teilweise zutreffend, und unmöglich zu erkennen, welcher Teil. Leser lernen, dem Ganzen zu misstrauen. Abhilfe: weniger Seiten, jede mit einem Verantwortlichen und einem Datum „zuletzt geprüft“, und löschen, was niemand pflegt.
- **Das Big-Bang-Dokument.** Eine 120-seitige Architekturspezifikation, vorab geschrieben, einmal genehmigt und nie wieder gelesen. Sie verstößt gegen die Regel, für den Leser zu schreiben, und ist innerhalb von Monaten veraltet.
- **Diagramme ohne Legende.** Kästen in verschiedenen Farben und Linien in verschiedenen Stilen ohne Erklärung. Jeder Leser erfindet eine Bedeutung, und zwei Leser erfinden verschiedene.
- **Vermischte Abstraktionsebenen.** Ein einziges Diagramm mit einer Mobile-App, einer Java-Klasse, einem Kubernetes-Cluster und einem Geschäftsprozess. Verwenden Sie getrennte Sichten (Abschnitt 4.2).
- **Dokumentation nur in Folien.** Foliensätze werden für ein Meeting und ein Publikum erstellt, sind selten versioniert und lassen sich nicht durchsuchen. Halten Sie die maßgebliche Quelle im Repository, und erstellen Sie Folien daraus.
- **Das Was ohne das Warum.** Eine vollständige Beschreibung der Struktur ohne Begründung, sodass künftige Teams nicht erkennen können, welche Randbedingungen wesentlich sind und welche Zufälle waren.
- **Write-only-Dokumentation.** Erstellt, um ein Governance-Kästchen abzuhaken, nie von einem echten Leser geprüft und daher nie auf Zweckeignung getestet.

Häufige Fragen

Unser Code ist sauber und gut getestet. Ist das nicht Dokumentation genug?
Sauberer Code und Tests sind eine hervorragende Dokumentation dessen, *was* das System im Detail tut. Sie zeigen nicht das Gesamtbild über Services und Repositories hinweg, die Qualitätsziele, die Randbedingungen von außerhalb des Codes oder warum die Struktur so ist, wie sie ist. Genau dafür ist Architekturdokumentation da.

Wer sollte die Architekturdokumentation schreiben?
Die Menschen, die die Entscheidungen treffen und mit ihnen leben, was in den meisten modernen Teams Engineers ebenso wie Architekten bedeutet. Der Architekt verantwortet meist die Gesamtstruktur und ihre Konsistenz, und die Teams verantworten die Teile, die ihre Services beschreiben. Am wichtigsten ist, dass jedes Dokument einen benannten Verantwortlichen hat.

Wie verhindern wir, dass Dokumentation veraltet?
Halten Sie sie klein, halten Sie sie neben dem Code, machen Sie ihre Aktualisierung zum Teil der Definition of Done für Änderungen, die sie betreffen, generieren Sie, was sich generieren lässt, und prüfen Sie sie in einem festen Rhythmus. Abschnitt 4.7 behandelt diese Gewohnheiten im Detail.

Entwicklerportale: Dokumentation auffindbar machen

Dokumentation, die niemand findet, könnte ebenso gut nicht existieren. Ein Entwicklerportal gibt Engineers einen Ort zum Nachsehen. Backstage, das bekannteste (Abschnitt 10.7), dreht sich um einen Softwarekatalog, der aus Metadatendateien aufgebaut wird, die beim Code liegen, sodass Zuständigkeiten über normale Reviews festgehalten und aktualisiert werden. Seine Funktion TechDocs nimmt beim Code gepflegtes Markdown und veröffentlicht es als Dokumentationssite, erreichbar von der Katalogseite des Service. Auf der Essenslieferplattform könnte die Seite des Bestellservice das zuständige Team, seine Entscheidungsdokumente und seine Runbooks zusammen zeigen. Ein Portal ist ein Verzeichnis, kein Heilmittel: Veraltete Seiten bleiben veraltet, und der Betrieb kostet etwas. Einem kleinen Team genügt vielleicht ein gut gepflegtes Repository.

AsyncAPI und Schema Registrys als Vertragsdokumentation

Ereignisflüsse verliert man leicht aus den Augen, weil kein einzelner Service sie zeigt. Ein AsyncAPI-Dokument (Abschnitt 6.5) hält sie maschinenlesbar in JSON oder YAML fest: die Kanäle, welche Anwendungen darauf senden oder empfangen, und die Nachrichtenformate. Es funktioniert protokollübergreifend, etwa mit Kafka, MQTT und AMQP; prüfen Sie die aktuelle

Version. Eine Schema Registry ergänzt die Durchsetzung. Die von Confluent prüft zum Beispiel jede neue Version eines Schemas gegen eine Kompatibilitätsregel, standardmäßig Abwärtskompatibilität, und lehnt inkompatible Versionen ab, und die Open-Source-Lösung Apicurio Registry kann auch OpenAPI- und AsyncAPI-Dokumente speichern (Abschnitt 7.7). So kann die Dokumentation von `OrderPlaced` nicht unbemerkt von dem abweichen, was veröffentlicht wird. Keines von beiden erklärt, was ein Feld *bedeutet*; dafür braucht es weiterhin Worte.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Nennen Sie die drei Verwendungszwecke von Architekturdokumentation, die Views and Beyond identifiziert.

Wissensvermittlung (Menschen in das System einführen), Kommunikation zwischen Stakeholdern und Grundlage für Analyse und Umsetzung.

▼ Bedeutet „Funktionierende Software mehr als umfassende Dokumentation“, dass agile Teams ihre Architektur nicht dokumentieren sollten?

Nein. Das Manifest sagt, dass die Werte auf der rechten Seite trotzdem Wert haben; es setzt eine Priorität, wenn beides konkurriert, und lehnt Dokumentation als Selbstzweck ab, nicht nützliche Dokumentation.

▼ Warum sollten Begründungen von Hand dokumentiert werden, Klassendiagramme im Allgemeinen aber nicht?

Begründungen lassen sich nicht aus dem Code ableiten und ändern sich selten, also muss ein Mensch sie aufschreiben. Die Klassenstruktur lässt sich aus dem Code lesen oder generieren und ändert sich oft, sodass eine handgezeichnete Kopie schnell abdriftet.

▼ Wovor warnt „Dokumentation aktuell halten, aber nicht zu aktuell“?

Sowohl vor veralteter Dokumentation, die Leser in die Irre führt, als auch vor ständigem Hin und Her durch Aktualisierungen für jede unausgereifte Idee; aktualisieren Sie in einem vorhersehbaren Rhythmus, sobald Entscheidungen getroffen sind.

▼ Probieren Sie es aus: Ein Ticketing-Team hat ein veraltetes Wiki mit 80 Seiten, das niemand liest. Die Entwickler fragen, wodurch es ersetzt werden soll, und fürchten, Zeit zu verschwenden. Was würden Sie von Hand schreiben, generieren oder löschen, und wie bliebe es aktuell?

Schreiben Sie von Hand, was der Code Ihnen nicht sagen kann: Kontext, Qualitätsziele, Prinzipien und Entscheidungen mit ihren Gründen, als kurze ADRs. Generieren Sie, was der Code bereits aussagt und sich oft ändert, etwa API-Referenzen. Löschen Sie veraltete Seiten, statt sie zu flicken. Halten Sie alles im Repository, im selben Pull Request wie die Code-Änderung geprüft, damit es aktuell bleibt. Ordnen Sie es nach den Fragen der Leser, etwa eines Neulings oder einer Betriebsperson, nicht nach der alten Struktur.

QUELLEN

- Manifesto for Agile Software Development: https://agilemanifesto.org/
- Documenting Software Architectures: Views and Beyond, 2nd edition (SEI): https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/
- Just Barely Good Enough models and documents (Scott Ambler, Agile Modeling): https://agilemodeling.com/essays/barelygoodenough.htm
- Docs as Code (Write the Docs): https://www.writethedocs.org/guide/docs-as-code/
- Principles of technical documentation (Gernot Starke, INNOQ): https://www.innoq.com/en/blog/2022/01/principles-of-technical-documentation/
- Backstage: TechDocs: https://backstage.io/docs/features/techdocs/
- Backstage: Software Catalog: https://backstage.io/docs/features/software-catalog/
- AsyncAPI specification: https://www.asyncapi.com/docs/reference/specification/latest
- Confluent: Schema evolution and compatibility: https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html
- Apicurio Registry: https://www.apicur.io/registry/

ABSCHNITT 4.2 VON 4.8 · MITTELSTUFE · 22 MIN. LESEZEIT · ZULETZT GEPRÜFT IM NOVEMBER 2026

Sichten und Blickwinkel: ISO/IEC/IEEE 42010, 4+1 und Rozanski und Woods

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 1.1 Architektur definieren
- 1.5 Stakeholder und Anliegen

Begriffe von früher, die er verwendet: Stakeholder, Anliegen, Architekturbeschreibung

DIE IDEE IN KÜRZE

Ein Haus wird nie durch eine einzige Zeichnung beschrieben. Wer kauft, sieht sich den Grundriss an, die Elektrikerin den Schaltplan, der Installateur den Rohrleitungsplan und das Bauamt die Ansicht der Hausfront. Jede Zeichnung lässt das meiste absichtlich weg, damit sie die Fragen einer Person klar beantworten kann. Softwarearchitektur funktioniert genauso. Statt eines riesigen Diagramms zeichnen Architekten mehrere Sichten desselben Systems, jede für eine bestimmte Gruppe von Menschen, und wählen nur die Sichten, die diese Menschen tatsächlich brauchen.

WAS SIE WISSEN MÜSSEN

- Eine Sicht stellt ein System aus der Perspektive bestimmter Anliegen dar; ein Blickwinkel (Viewpoint) ist die wiederverwendbare Menge von Konventionen (abgedeckte Anliegen, Stakeholder, Modellarten) für die Erstellung einer Art von Sicht.
- In ISO/IEC/IEEE 42010:2022 haben Stakeholder Anliegen, Blickwinkel rahmen Anliegen und regeln Sichten, Sichten behandeln Anliegen und bestehen aus Sichtkomponenten, die von Modellarten geregelt werden, und Korrespondenzen setzen Elemente über Sichten hinweg in Beziehung.
- Kruchtens 4+1-Modell (IEEE Software, 1995) verwendet logische Sicht, Prozess-, Entwicklungs- und physische Sicht, die durch eine kleine Menge von Szenarien verbunden und validiert werden.
- Rozanski und Woods definieren sieben Blickwinkel (Kontext, funktional, Information, Nebenläufigkeit, Entwicklung, Deployment, Betrieb) sowie Perspektiven wie Sicherheit und Performance, die auf alle Sichten angewendet statt als eigene Sichten gezeichnet werden.

- Das SEI gruppiert Sichten in die Kategorien Modul, Komponenten und Konnektoren sowie Zuordnung; ein vollständiger Dokumentationssatz enthält mindestens eine aus jeder.
- Wählen Sie die minimale Menge an Sichten, indem Sie Stakeholder und Anliegen auf Sichten abbilden, Sichten wo möglich zusammenlegen und riskante Bereiche ausführlicher dokumentieren als Routinebereiche.

Ein System, viele Sichten

Kein einzelnes Diagramm kann ein reales System beschreiben. Eine Zeichnung, die zeigt, wie der Code der Essenslieferplattform in Module aufgeteilt ist, sagt nichts darüber, auf welchen Servern sie laufen; eine Zeichnung von Servern und Netzwerken sagt nichts darüber, was Schritt für Schritt passiert, wenn ein Kurier einen Auftrag annimmt. Wer all das auf eine Seite bringen will, erhält das bekannte Architekturposter: Dutzende Kästen, Linien mit sechs verschiedenen Bedeutungen, und niemand kann daraus eine präzise Frage beantworten.

Der Gedanke, der das löst, ist alt und einfach. Beschreiben Sie das System durch mehrere Sichten, von denen jede eine Menge von Strukturen für eine Menge von Anliegen zeigt, so wie ein Gebäude durch einen Grundriss, eine Ansicht, einen Elektroplan und einen Sanitärplan beschrieben wird. Jede ist eine Teilbeschreibung; zusammen beschreiben sie das Ganze. Dieser Gedanke liegt der internationalen Norm zugrunde, Kruchtens 4+1-Modell, dem Blickwinkel-Katalog von Rozanski und Woods, dem Ansatz Views and Beyond des SEI und, in schlanker Form, dem C4-Modell im nächsten Abschnitt. Dieser Abschnitt erklärt das gemeinsame Vokabular und wie man die Sichten wählt, die ein bestimmtes System braucht.

Erinnern Sie sich an Abschnitt 1.5: Ein Stakeholder ist jeder, der ein Interesse am System hat, und ein Anliegen ist jede Angelegenheit, die für einen Stakeholder wichtig ist. Sichten existieren, um Anliegen zu beantworten. Eine Sicht, die niemandes Anliegen beantwortet, ist Dekoration.

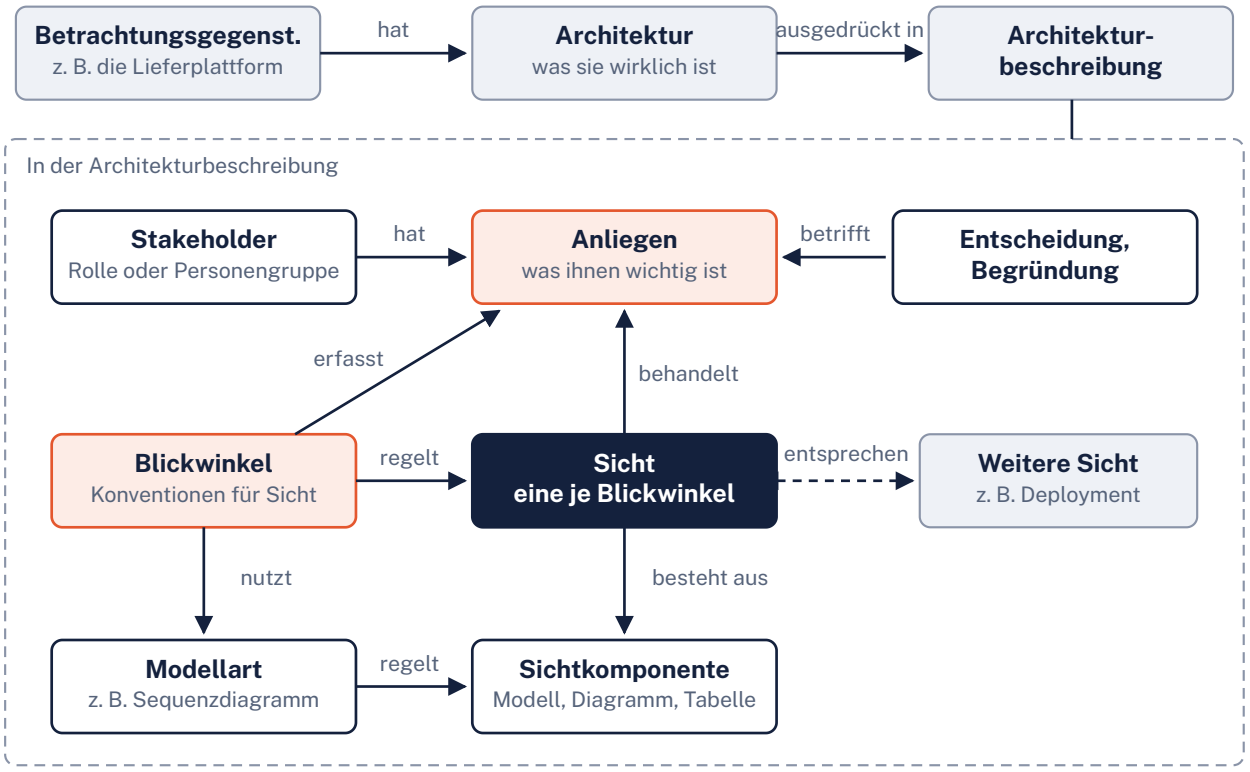
Das konzeptionelle Modell von ISO/IEC/IEEE 42010:2022

ISO/IEC/IEEE 42010 ist die internationale Norm für Architekturbeschreibung. Ihre aktuelle Ausgabe erschien 2022 und ersetzte die Ausgabe von 2011. Sie sagt Ihnen nicht, wie Sie entwerfen, welche Notation Sie verwenden oder welche Sichten Sie zeichnen sollen; sie definiert die Konzepte, die eine Architekturbeschreibung enthalten sollte, und wie sie zusammenhängen, damit sich Beschreibungen, die mit verschiedenen Methoden erstellt wurden, vergleichen und prüfen lassen. Ihr konzeptionelles Modell ist das gemeinsame Gerüst fast aller Dokumentationsansätze in diesem Kapitel.

Das Modell beginnt mit dem Betrachtungsgegenstand: dem Ding, dessen Architektur beschrieben wird; das kann ein Softwaresystem, ein System von Systemen, eine Produktlinie oder ein Unternehmen sein (die Ausgabe von 2011 nannte es „system of interest“). Der Betrachtungsgegenstand hat eine Architektur, und eine Architekturbeschreibung drückt sie aus. Innerhalb der Beschreibung haben **Stakeholder Anliegen**. Ein Blickwinkel ist eine wiederverwendbare Spezifikation von Konventionen zur Erstellung und Nutzung einer Art von Sicht: Er benennt die Anliegen, die er erfasst, die typischen Stakeholder und die zu verwendenden Modellarten (Notationen und Modellierungskonventionen). Eine **Sicht** wendet einen Blickwinkel auf einen bestimmten Betrachtungsgegenstand an: Sie behandelt diese Anliegen für dieses System und besteht aus Sichtkomponenten wie Diagrammen, Tabellen und Modellen, die jeweils von einer Modellart geregelt werden (die Ausgabe von 2011 nannte Sichtkomponenten „architecture models“).

Zwei weitere Elemente zählen in der Praxis. Korrespondenzen halten Beziehungen zwischen Elementen der Beschreibung fest, am nützlichsten zwischen Sichten: Dieser Container in der Containersicht läuft auf jenem Cluster in der Deployment-Sicht; diese Anforderung wird durch jene Entscheidung adressiert. Und **Architekturentscheidungen und ihre Begründung** sind Teil der Beschreibung, verknüpft mit den Anliegen, die sie adressieren. Die Ausgabe von 2022 definiert außerdem zwei Konzepte genauer, die schon die Ausgabe von 2011 erwähnte, Stakeholder-Perspektiven (Arten, Anliegen zu gruppieren) und Architekturaspekte (Merkmale des Betrachtungsgegenstands, die quer zu den Sichten liegen), benennt „architecture framework“ in „architecture description framework“ (Architekturbeschreibungsrahmen) um und legt fest, was konforme Architekturbeschreibungsrahmen, etwa die eigene Dokumentationsvorlage eines Unternehmens, enthalten müssen.

Das konzeptionelle Modell von 42010, vereinfacht



Anliegen sind das Scharnier: Blickwinkel erfassen sie, Sichten behandeln sie, und Entscheidungen betreffen sie.

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist: Blickwinkel und Sicht

Ein Blickwinkel ist wie eine Kartenart: „Eine Straßenkarte“ ist durch ihre Konventionen (Straßen in Rot, Autobahnen dicker, eine Maßstabsleiste) und ihre Nutzer (Autofahrer) definiert. Eine Sicht ist eine bestimmte Karte, die nach diesen Konventionen gezeichnet ist: die Straßenkarte von Schottland. Sie definieren den Blickwinkel einmal, vielleicht für das ganze Unternehmen, und zeichnen für jedes System eine Sicht danach.

Eine Modellart ist die Zeichentechnik, die ein Blickwinkel vorschreibt, etwa „UML-Sequenzdiagramm“ oder „C4-Containerdiagramm“. Die Unterscheidung erlaubt einer Organisation zu sagen: Die Deployment-Sicht jedes Systems muss Regionen, Netzwerkzonen und Managed Services zeigen, gezeichnet als C4-Deployment-Diagramm mit einer Tabelle der Umgebungen.

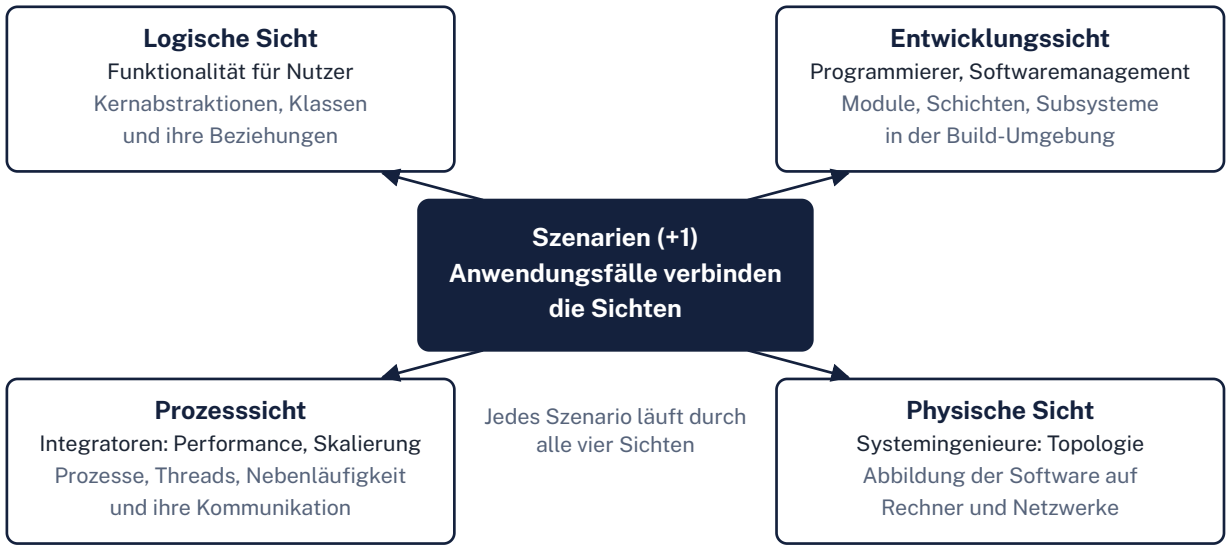
Kruchtens 4+1-Sichtenmodell

Philippe Kruchten, damals bei Rational Software, veröffentlichte *Architectural Blueprints: The „4+1“ View Model of Software Architecture* im November 1995 in *IEEE Software*. Das 4+1-Sichtenmodell war der erste weit verbreitete Ansatz mit mehreren Sichten und wurde Teil des Rational Unified Process. Kruchtens Argument war, dass die Einzeldiagramme jener Zeit zu viel auf einmal erfassen wollten und dass verschiedene Stakeholder (Endnutzer, Entwickler, Systemingenieure, Projektleiter) verschiedene Sichten brauchen.

Die vier Sichten sind diese. Die **logische Sicht** beschreibt die Funktionalität, die das System Endnutzern bietet, typischerweise als die wichtigsten Abstraktionen und ihre Beziehungen (in UML als Klassen- und Zustandsdiagramme). Die **Prozesssicht** behandelt Laufzeitanliegen: Prozesse und Threads, Nebenläufigkeit, Verteilung, Performance, Skalierbarkeit und Fehlertoleranz. Die **Entwicklungssicht** (auch Implementierungssicht genannt) beschreibt die Organisation der Software in ihrer Entwicklungsumgebung: Module, Subsysteme, Schichten und ihre Abhängigkeiten, die für Programmierer und für die Planung der Arbeit durch das Management zählen. Die **physische Sicht** (oder Deployment-Sicht) bildet die Software auf Hardware und Netzwerke ab. Das „+1“ sind die **Szenarien**: eine kleine Menge wichtiger Anwendungsfälle, jeder durch alle vier Sichten verfolgt, die die Architektur veranschaulichen und zugleich validieren, dass die Sichten zusammenpassen.

Die Szenarien sind der am häufigsten vergessene und wohl wertvollste Teil. Eine Sicht für sich allein kann ordentlich aussehen und dabei eine Lücke verbergen; ein echtes Szenario wie „Kunde gibt eine Bestellung auf“ durch die logische, die Prozess-, die Entwicklungs- und die physische Sicht durchzugehen, deckt fehlende Elemente und widersprüchliche Annahmen auf. Derselbe Gedanke taucht in der Laufzeitsicht von arc42 wieder auf, in dynamischen C4-Diagrammen und in Abschnitt 4.8 im Rat, ein System zu erklären, indem man einer Anfrage folgt.

Das 4+1-Sichtenmodell



Vier Sichten für vier Gruppen von Stakeholdern, zusammengehalten durch eine kleine Menge von Szenarien.

Rozanski und Woods: Blickwinkel und Perspektiven

Nick Rozanskis und Eóin Woods' *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives* (2. Auflage, Addison-Wesley, 2012) bietet einen Katalog von sieben Blickwinkeln für große Informationssysteme, zusammen mit einem zweiten, ergänzenden Gedanken: Perspektiven.

Ihre sieben Blickwinkel sind: **Kontext** (die Beziehungen des Systems zu Menschen, Systemen und externen Einheiten in seinem Umfeld), **funktional** (die funktionalen Laufzeitelemente, ihre Verantwortlichkeiten, Schnittstellen und wichtigsten Interaktionen), **Information** (wie Daten gespeichert, verwaltet, verantwortet und verteilt werden), **Nebenläufigkeit** (welche Teile nebenläufig laufen und wie sie koordiniert werden), **Entwicklung** (was Entwickler, Tester und Build-Verantwortliche brauchen: Modulstruktur, gemeinsame Verarbeitung, Standards und Organisation der Codeline), **Deployment** (die Laufzeitumgebung und wie die Software darauf abgebildet wird) und **Betrieb** (wie das System in der Produktion installiert, migriert, administriert, überwacht und unterstützt wird).

Eine Perspektive ist ihre Bezeichnung für ein Qualitätsanliegen, das quer durch alle Sichten verläuft, etwa Sicherheit, Performance und Skalierbarkeit, Verfügbarkeit und Resilienz oder Evolution (ihr Katalog umfasst außerdem Barrierefreiheit, Entwicklungsressourcen, Internationalisierung, Standort, Regulierung und Benutzbarkeit). Eine Perspektive ist keine weitere Sicht zum Zeichnen. Sie ist eine Menge von Fragen, Taktiken und Prüfungen, die auf jede Sicht angewendet werden: Für Sicherheit untersuchen Sie die Kontextsicht auf externe

Akteure und Vertrauensgrenzen, die Informationssicht auf sensible Daten, die Deployment-Sicht auf Netzwerkzonen und so weiter. Das erfasst etwas, was einzelne Sichten nicht können: Qualitäten entstehen aus dem ganzen System, also müssen sie überall geprüft werden (vergleichen Sie die Qualitätsmodelle aus Kapitel 2).

Der Betriebsblickwinkel verdient besondere Aufmerksamkeit, weil viele Ansätze ihn unterschätzen. Wer das System installiert, wie Daten aus dem Altsystem migriert werden, wie die Konfiguration verwaltet wird, was der Betrieb überwacht und wie er alarmiert wird: Das sind Architekturfragen, deren Antworten das System prägen, und sie bis kurz vor dem Go-live aufzuschieben, ist eine klassische Ursache für gescheiterte Starts.

Die drei Kategorien des SEI: Modul, Komponenten und Konnektoren, Zuordnung

Kapitel 1 hat die Gruppierung von Strukturen in drei Arten durch das SEI eingeführt (siehe Abschnitt 1.1). *Documenting Software Architectures: Views and Beyond* verwendet dieselbe Gruppierung, um Sichten zu ordnen, und sie ist der nützlichste Weg, um zu prüfen, ob eine Menge von Sichten vollständig ist.

Modulsichten zeigen Implementierungseinheiten (Pakete, Bibliotheken, Namensräume) und Beziehungen wie *ist Teil von*, *hängt ab von* und *ist ein*. Typische Stile sind Zerlegungs-, Nutzungs-, Generalisierungs-, Schichten- und Datenmodellsichten. Komponenten-und-Konnektoren-Sichten (C&C) zeigen die zur Laufzeit vorhandenen Elemente (Services, Clients, Datenbanken, Queues) und die Konnektoren, über die sie interagieren (HTTP-Aufrufe, Nachrichten, gemeinsame Daten); typische Stile sind Client-Server, Publish-Subscribe, Pipe-and-Filter, Shared Data und Multi-Tier. Zuordnungssichten bilden Softwareelemente auf Dinge ab, die keine Software sind: auf Hardware und Cloud-Infrastruktur (Deployment), auf das Dateisystem und Build-Artefakte (Implementierung und Installation) und auf Teams (Arbeitszuordnung).

Die drei Kategorien lassen sich natürlich auf die anderen Ansätze abbilden. Kruchtens Entwicklungssicht ist eine Modulsicht, seine Prozesssicht eine C&C-Sicht und seine physische Sicht eine Zuordnungssicht. Die funktionalen und Nebenläufigkeits-Blickwinkel von Rozanski und Woods sind überwiegend C&C, ihr Entwicklungsblickwinkel ist Modul, und ihre Deployment- und Betriebsblickwinkel sind Zuordnung. Die C4-Container- und Komponentendiagramme des nächsten Abschnitts sind C&C-Sichten mit etwas Modulcharakter, und C4-Deployment-Diagramme sind Zuordnungssichten. Wenn Sie einen Dokumentationssatz zusammenstellen, prüfen Sie, ob er mindestens eine Sicht aus jeder Kategorie enthält, denn jede beantwortet Fragen, die die anderen nicht beantworten können.

Ansätze im Vergleich

Anliegenbereich	4+1 (Kruchten)	Rozanski und Woods	SEI-Kategorie	C4-Diagramm
Systemgrenze, Nutzer, externe Systeme	(implizit in Szenarien)	Kontext	C&C (Kontext)	Systemkontext
Laufzeitelemente und Interaktionen	Prozess (und logisch)	Funktional, Nebenläufigkeit	Komponenten und Konnektoren	Container, Komponente, dynamisch
Datenhoheit und Datenfluss	(nicht separat)	Information	Modul (Datenmodell) und C&C	(nicht separat)
Codeorganisation	Entwicklung	Entwicklung	Modul	Komponente, Code
Abbildung auf Infrastruktur	Physisch	Deployment	Zuordnung	Deployment
Betrieb des Systems in der Produktion	(nicht abgedeckt)	Betrieb	Zuordnung (teilweise)	(nicht abgedeckt)
Validierung durch Anwendungsfälle	Szenarien (+1)	(auf jede Sicht angewendet)	Verhaltensdokumentation	Dynamisch

Stakeholder und Anliegen auf Sichten abbilden

Das praktischste Werkzeug aus der Norm ist eine einfache Tabelle: Listen Sie die Stakeholder und ihre wichtigsten Anliegen auf, und benennen Sie dann die Sicht oder die Sichten, die jedes Anliegen behandeln. Lücken (ein Anliegen, das keine Sicht behandelt) zeigen, wo die Beschreibung unvollständig ist; Sichten, die niemand braucht, sind Kandidaten zum Löschen. Die folgende Tabelle ist ein erster Entwurf für die Essenslieferplattform.

Stakeholder, Anliegen und Sichten für die Essenslieferplattform

Stakeholder	Wichtigste Anliegen	Sichten, die sie behandeln
Produktmanagement und Auftraggeber	Umfang, externe Partner, Zukauf oder Eigenbau, Kosten	Kontextsicht; einseitige Übersicht mit Qualitätszielen und den größten Risiken
Entwickler in jedem Team	Wohin Code gehört, Schnittstellen zwischen Services, Konventionen	Containersicht; Komponentensicht für komplexe Services; querschnittliche Konzepte; Entscheidungen
Betrieb und SRE	Wo was läuft, Failover, Kapazität, Abhängigkeiten, Wiederherstellung	Deployment-Sicht je Umgebung; Betriebssicht (Monitoring, Runbooks, Backup und Wiederherstellung)
Sicherheitsteam	Vertrauensgrenzen, Authentifizierung, Kartendaten, Secrets	Kontext- und Deployment-Sichten mit eingezeichneten Vertrauenszonen; Sicherheitskonzept (eine auf jede Sicht angewandte Perspektive)
Datenschutzbeauftragter	Personenbezogene Daten von Kunden und Kurieren: wohin sie fließen, wie lange sie aufbewahrt werden	Informationssicht mit einem Datenflussdiagramm und einer Aufbewahrungstabelle
Integrationsteam des Zahlungsanbieters	Welche Aufrufe Sie machen, Webhooks, Idempotenz, Versionen	Schnittstellenspezifikation und ein Sequenzdiagramm des Zahlungsablaufs
Prüfer für Kartenzahlungen (Auditor)	Umfang der Kartendatenumgebung, Kontrollen, Änderungshistorie	Deployment-Sicht mit der Kartendatengrenze; Entscheidungsprotokoll; Versionshistorie

Die minimale Menge an Sichten wählen

Wenn man weiß, dass es viele Blickwinkel gibt, ist die Versuchung groß, alle zu erstellen. Widerstehen Sie ihr. Jede Sicht muss mit dem System und mit den anderen Sichten konsistent gehalten werden, also erhöht jede die Wartungskosten. Das Ziel ist die kleinste Menge an Sichten, die die wichtigen Anliegen der wichtigen Stakeholder beantwortet, gemäß der Regel aus Abschnitt 4.1, für den Leser zu schreiben. Views and Beyond beschreibt ein ähnliches Vorgehen: eine Tabelle aus Stakeholdern und Sichten erstellen, Sichten zusammenlegen, wo ein Diagramm mehrere Bedürfnisse bedienen kann, und dann priorisieren.

Ein Vorgehen zur Wahl von Sichten

1. **1. Stakeholder und Anliegen auflisten**
   Verwenden Sie die Stakeholder-Analyse aus Kapitel 1. Notieren Sie zu jedem Anliegen, wie wichtig es ist und wie viel auf dem Spiel steht, wenn es missverstanden wird.

2. **2. Anliegen auf mögliche Sichten abbilden**
   Verwenden Sie einen Katalog (4+1, Rozanski und Woods oder die SEI-Kategorien) als Checkliste, damit ganze Bereiche, typischerweise Information und Betrieb, nicht vergessen werden.

3. **3. Zusammenlegen, wo möglich**
   Ein Containerdiagramm mit eingetragenen Technologien und Protokollen bedient oft Entwickler und Betrieb zugleich; ein Deployment-Diagramm mit Vertrauenszonen bedient Betrieb und Sicherheit. Das Zusammenlegen verringert die Zahl der zu pflegenden Artefakte.

4. **4. Abdeckung der drei Kategorien prüfen**
   Stellen Sie sicher, dass die Menge mindestens eine Modulsicht, eine Komponenten-und-Konnektoren-Sicht und eine Zuordnungssicht enthält, dazu einige Szenarien, die sie durchlaufen.

5. **5. Die Tiefe jeder Sicht festlegen**
   Nicht jede Sicht braucht denselben Detailgrad. Dokumentieren Sie die riskanten oder komplexen Teile ausführlich und die Routineteile knapp.

6. **6. Korrespondenzen festhalten**
   Geben Sie an, wie Elemente einer Sicht mit Elementen einer anderen zusammenhängen (Container zu Deployment-Knoten, Modul zu Container), damit Leser zwischen den Sichten wechseln können. In Unstimmigkeiten zwischen Sichten verstecken sich viele Designfehler.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: die minimale Sichtenmenge für die Essenslieferplattform

Wendet man das Vorgehen auf die in Abschnitt 4.1 beschriebene Plattform an (fünf Teams, Kartenzahlungen über einen externen Anbieter, eine bevorstehende Prüfung der Zahlungssicherheit), ergibt sich folgende Menge.

Kontextsicht	Ein C4-Systemkontextdiagramm: Kunden, Restaurantpersonal, Kuriere, Support-Mitarbeitende, der Zahlungsanbieter, der Kartendienst, die Push-Benachrichtigungsdienste. Dient allen; das Erste, was ein Neuling liest.
C&C-Sicht	Ein C4-Containerdiagramm für die gesamte Plattform, dazu Komponentendiagramme nur für den Bestell- und den Dispositionsservice, die am komplexesten sind und sich am meisten ändern.
Informationssicht	Eine Tabelle der wichtigsten Datenentitäten, welcher Service jede verantwortet, wohin personenbezogene Daten fließen und welche Aufbewahrungsfristen gelten; ein Datenflussdiagramm für Kundendaten. Hinzugefügt, weil die Anliegen des Datenschutzbeauftragten sonst unbehandelt geblieben wären.
Zuordnungssicht	Ein C4-Deployment-Diagramm für die Produktion mit eingezeichneter Kartendatengrenze und Netzwerkzonen und eine kurze Tabelle der Unterschiede in der Staging-Umgebung.
Modulsicht	Nicht von Hand gezeichnet. Die interne Modulstruktur jedes Service wird aus dem Code generiert und durch Architekturtests geprüft.
Szenarien	Vier Sequenzdiagramme: Bestellung aufgeben, Restaurant lehnt Bestellung ab, Kurierzuweisung, Erstattung. Jedes wird gegen die Container- und Deployment-Sichten geprüft.
Perspektiven	Sicherheit und Verfügbarkeit werden beim Review als Checklisten auf jede Sicht angewendet, statt als eigene Sichten gezeichnet zu werden.

Sieben Artefakte, die meisten davon einzelne Diagramme, beantworten die Anliegen aller sieben Stakeholder-Gruppen aus der Tabelle oben.

Die Informationssicht wurde nur hinzugefügt, weil die Tabelle aus Stakeholdern und Sichten eine Lücke aufgedeckt hat. Ohne die Tabelle wäre sie übersehen worden, was häufig vorkommt.

Die Modulsicht wird generiert, nicht gezeichnet, weil sie sich aus dem Code ableiten lässt und sich oft ändert (Abschnitt 4.1).

VERTIEFUNG

Vertiefung: eine Sicht richtig dokumentieren

Views and Beyond empfiehlt für jede Sicht eine Standardstruktur, die sich auch gut als Review-Checkliste eignet: eine **Primärdarstellung** (meist das Diagramm mit den Elementen und Beziehungen), einen **Elementkatalog** (was jedes Element ist, seine Verantwortlichkeiten, Schnittstellen und Eigenschaften), ein **Kontextdiagramm**, das zeigt, was außerhalb des dargestellten Teils liegt, einen **Variabilitätsleitfaden** (was variieren kann, etwa optionale Komponenten oder Konfiguration) und die **Begründung** für den gezeigten Entwurf.

Es betont außerdem die Dokumentation, die *jenseits* der Sichten liegt und dem Buch seinen Namen gibt: eine Roadmap, die erklärt, wie die Dokumentation gegliedert ist, eine Systemübersicht, die Abbildung zwischen den Sichten, die Begründung für Entscheidungen, die quer zu den Sichten liegen, und ein Verzeichnis oder Glossar. Die meisten Anti-Patterns aus Abschnitt 4.1 entstehen, wenn man diese Teile auslässt.

ACHTUNG

Häufige Fehler mit Sichten

- Kategorien in einem Diagramm vermischen, zum Beispiel Codepakete, laufende Services und Server als Kästen auf derselben Fläche zeichnen, mit Linien, die Verschiedenes bedeuten.
- Sichten zeichnen, nach denen niemand gefragt hat, weil ein Framework sie auflistet, während die eine Sicht fehlt, die das Betriebsteam braucht.
- Sichten einander widersprechen lassen: Die Containersicht zeigt sechs Services, die Deployment-Sicht aber fünf. Korrespondenzen und Reviews fangen das ab.
- Qualitätsmerkmale als eine weitere Sicht behandeln statt als Perspektive, die in jeder Sicht geprüft werden muss.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Was ist der Unterschied zwischen einem Blickwinkel und einer Sicht?**

Ein Blickwinkel ist eine wiederverwendbare Spezifikation von Konventionen für eine Art von Sicht (erfasste Anliegen, Stakeholder, Modellarten). Eine Sicht wendet einen Blickwinkel auf ein bestimmtes System an und behandelt diese Anliegen dafür.

▼ **Welche Rolle spielen Szenarien im 4+1-Modell?**

Sie sind eine kleine Menge wichtiger Anwendungsfälle, die durch alle vier Sichten verfolgt werden; sie veranschaulichen die Architektur und validieren, dass die Sichten konsistent und vollständig sind.

▼ **Wie unterscheidet sich eine Perspektive bei Rozanski und Woods von einem Blickwinkel?**

Ein Blickwinkel bringt eine Sicht hervor. Eine Perspektive ist ein querschnittliches Qualitätsanliegen wie Sicherheit, das als Fragen und Taktiken auf jede Sicht angewendet wird, statt als eigene Sicht gezeichnet zu werden.

▼ **Zu welcher SEI-Kategorie gehört ein Deployment-Diagramm, und warum?**

Zur Zuordnung, weil es Softwareelemente auf Dinge abbildet, die keine Software sind, in diesem Fall Hardware und Cloud-Infrastruktur.

▼ **Probieren Sie es aus: Für eine Ticketing-Plattform zählen drei Lesergruppen: neue Entwickler, das Betriebspersonal am Vorverkaufstag und eine Prüfperson für Zahlungssicherheit. Welche Mindestmenge an Sichten würden Sie erstellen, und wie rechtfertigt das Anliegen jeder Lesergruppe diese Wahl?**

Eine Kontextsicht für alle. Eine Container-Sicht, die Entwicklung und Betrieb teilen, sodass ein Diagramm zwei Anliegen bedient. Eine Deployment-oder Betriebssicht für den Betrieb, die Zonen, Skalierung und Ausfallbehandlung zeigt. Eine Sicherheitsperspektive quer über diese Sichten, die die Vertrauensgrenzen und Kartendatenflüsse für den Prüfer zeigt. Ergänzen Sie eine Laufzeitsicht des Kaufablaufs, weil er der riskante Bereich ist. Lassen Sie detaillierte Sichten von Routineteilen weg. Weniger Sichten bleiben aktuell.

QUELLEN

- ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description (ISO): https://www.iso.org/standard/74393.html
- ISO/IEC/IEEE 42010 summary (arc42 quality model): https://quality.arc42.org/standards/iso-42010
- Philippe Kruchten, Architectural Blueprints: The 4+1 View Model of Software Architecture (arXiv copy of the 1995 IEEE Software paper): https://arxiv.org/abs/2006.04975
- Viewpoints catalogue, Rozanski and Woods: https://www.viewpoints-and-perspectives.info/home/viewpoints/

- Documenting Software Architectures: Views and Beyond, 2nd edition (SEI): https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/

Das C4-Modell: Kontext, Container, Komponenten und Code

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.4 Wie Computer kommunizieren
- 0.7 Server und die Cloud
- 4.2 Sichten und Blickwinkel

Begriffe von früher, die er verwendet: API (Programmierschnittstelle), HTTP, Blickwinkel (Viewpoint)

DIE IDEE IN KÜRZE

Denken Sie an eine Online-Karte. Herausgezoomt sehen Sie ein ganzes Land und die Autobahnen zwischen den Städten. Zoomen Sie hinein, sehen Sie die Bezirke einer Stadt, dann die Straßen eines Bezirks, dann ein einzelnes Gebäude. Jede Zoomstufe ist für andere Menschen nützlich, und Sie wissen immer, welche Stufe Sie gerade betrachten. Das C4-Modell macht dasselbe für Software: Ein Bild zeigt das ganze System und wer es nutzt, das nächste die Apps und Datenbanken darin, das nächste die Teile innerhalb einer App, und das letzte, selten benötigte, zeigt den Code selbst.

WAS SIE WISSEN MÜSSEN

- Das C4-Modell von Simon Brown beschreibt Software mit einer Hierarchie aus vier Diagrammen (Systemkontext, Container, Komponente und Code), die auf den Abstraktionen Person, Softwaresystem, Container und Komponente aufbauen.
- Ein C4-Container ist eine separat lauffähige oder deploybare Einheit, die Code ausführt oder Daten speichert, etwa eine Web-App, eine Mobile-App, ein Service, ein Datenbankschema oder ein Message-Topic; er ist nicht dasselbe wie ein Docker-Container.
- Systemkontext- und Containerdiagramme werden jedem Team empfohlen, Komponentendiagramme nur, wo sie einen Mehrwert bieten, und Codediagramme selten, vorzugsweise aus dem Code generiert.
- Ergänzende Diagramme sind die Systemlandschaft (alle Systeme einer Organisation), das dynamische Diagramm (Zusammenspiel zur Laufzeit für ein Szenario) und das Deployment-Diagramm (Container auf Infrastruktur abgebildet, eines je Umgebung).

- Jedes Diagramm braucht einen Titel mit Typ und Umfang sowie eine Legende; jedes Element einen Namen, einen Typ, eine Beschreibung und (bei Containern und Komponenten) eine Technologie; jede Beziehung eine einzige Richtung, eine konkrete Beschriftung und, zwischen Containern, ein Protokoll.
- Modellierungswerkzeuge wie Structurizr erzeugen viele konsistente Diagramme aus einem Modell, während Zeichenwerkzeuge unabhängige Bilder erzeugen, die auseinanderdriften.

Landkarten Ihres Codes

Das C4-Modell ist ein Ansatz zur Darstellung von Softwarearchitektur in Diagrammen, den Simon Brown entwickelt hat. Es ist aus den Softwarearchitektur-Schulungen entstanden, die er ab etwa 2006 gab: Laut der C4-Website reichen seine Wurzeln in die Zeit zwischen 2006 und 2009 zurück, die Diagrammtypen erhielten Anfang 2010 ihre Namen, und der Name „C4“ wurde Anfang 2011 erstmals verwendet. Brown war aufgefallen, dass Teams, die ihre Architektur zeichnen sollten, schwer verständliche Diagramme lieferten: unerklärte Formen und Farben, Kästen, die alles von einer Klasse bis zu einem Rechenzentrum sein konnten, unbeschriftete Linien und mehrere Detailebenen auf einer Seite. UML, das das Problem hätte lösen können, war im Alltag weitgehend außer Gebrauch gekommen.

Seine Antwort war keine neue Notation, sondern eine kleine Menge von *Abstraktionen* und eine darauf aufbauende Hierarchie von Diagrammen, die er mit den Zoomstufen einer Online-Karte vergleicht. Sie beginnen mit dem ganzen Land, zoomen in eine Stadt, dann in einen Bezirk, dann in eine Straße. Jede Stufe zeigt für einen anderen Zweck unterschiedlich viele Details, und Sie wissen immer, welche Stufe Sie betrachten. C4 ist bewusst unabhängig von Notation und Werkzeug: Sie können es auf einem Whiteboard, in einem Zeichenwerkzeug oder mit einem Modellierungswerkzeug zeichnen, sofern Sie ein paar Regeln befolgen.

C4 ist zu einem der am weitesten verbreiteten Ansätze für Architekturdiagramme geworden, weil es leicht zu lernen ist, sich natürlich in das Denken in mehreren Sichten aus Abschnitt 4.2 einfügt und Diagramme hervorbringt, denen Entwicklungsteams wie fachfremde Leser folgen können. In der Begriffswelt von 42010 ist jeder C4-Diagrammtyp ein Blickwinkel mit eigener Modellart; C4 hält die Menge nur klein.

Die Abstraktionen: Personen, Softwaresysteme, Container, Komponenten

C4 baut auf vier Abstraktionen auf, dazu kommt der Code. Eine **Person** ist ein menschlicher Nutzer des Systems: ein Kunde, ein Kurier, eine Supportkraft. Ein Softwaresystem ist die höchste Abstraktionsebene, etwas, das seinen Nutzern einen Wert liefert und typischerweise

von einem Team oder einer Organisation besessen und gebaut wird; die Essenslieferplattform ist eines, ebenso der Zahlungsanbieter und der Kartendienst, von denen sie abhängt.

Ein Container ist etwas, das laufen muss, damit das System funktioniert: nach Browns Worten im Wesentlichen eine Laufzeitgrenze um Code, der ausgeführt wird, oder um Daten, die gespeichert werden. Beispiele sind eine serverseitige Webanwendung, eine Single-Page-App im Browser, eine Mobile-App, eine Desktop-Anwendung, eine Serverless-Funktion, ein Datenbankschema, ein Dateispeicher oder ein Message-Topic. Jeder ist separat deploybar oder lauffähig, und Container kommunizieren über Prozessgrenzen hinweg miteinander, meist über ein Netzwerk.

Eine Komponente ist eine Gruppierung zusammengehöriger Funktionalität hinter einer klar definierten Schnittstelle; sie lebt *innerhalb* eines Containers und läuft im selben Prozess wie die anderen Komponenten dort. In einem Java-Service kann sie eine Menge von Klassen hinter einer Schnittstelle sein, in einem Node.js-Service ein Modul. Komponenten sind nicht separat deploybar; der Container ist die Deployment-Einheit. Schließlich ist **Code** die Klassen, Funktionen und Tabellen, die eine Komponente implementieren.

ACHTUNG

Ein Container ist kein Docker-Container

Das häufigste Missverständnis von C4 ist, „Container“ als „Docker-Container“ zu lesen. Der C4-Begriff ist älter als die Popularität von Docker und bezeichnet jede separat lauffähige oder deploybare Einheit: Eine Mobile-App auf einem Smartphone, eine JavaScript-Anwendung im Browser und ein Datenbankschema sind allesamt Container, obwohl keines davon in Docker läuft. Umgekehrt kann ein C4-Container als zwanzig Docker-Container hinter einem Load Balancer deployt werden; wie viele Kopien laufen und wo, gehört in ein Deployment-Diagramm, nicht in ein Containerdiagramm.

Wenn das Wort in Ihrer Organisation Verwirrung stiftet, behalten die meisten Teams es bei und erklären es in der Legende oder im Glossar. Die C4-Website räumt die Überschneidung ein, merkt an, dass beide manchmal sinnvoll übereinstimmen, und sagt, dass Teams die Terminologie bei Bedarf ändern dürfen; wenn Sie das tun, ändern Sie sie überall.

Die vier Kerndiagramme

Niveau	Umfang	Zeigt	Hauptzielgruppe	Empfohlen?
1. Systemkontext	Ein Softwaresystem	Das System als einzelner Kasten, die Personen, die es nutzen, und die anderen Softwaresysteme, mit denen es interagiert	Alle, technisch und fachfremd, innerhalb und außerhalb des Teams	Ja, für jedes Team
2. Container	Ein Softwaresystem	Die Container darin (Apps, Services, Datenspeicher, Topics), ihre Verantwortlichkeiten, Technologien und wie sie kommunizieren	Architekten, Entwicklungsteams, Betrieb und Support	Ja, für jedes Team
3. Komponente	Ein Container	Die Komponenten darin, ihre Verantwortlichkeiten und Technologien sowie ihre Beziehungen	Architekten und Entwicklungsteams	Nur wo es einen Mehrwert bietet; generieren, wenn es langfristig gepflegt werden muss
4. Code	Eine Komponente	Klassen, Schnittstellen, Funktionen oder Tabellen, typischerweise als UML-Klassendiagramm oder Entity-Relationship-Diagramm	Entwicklungsteams	Selten; bei Bedarf für die komplexesten Teile aus dem Code generieren

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: System, Container oder Komponente?

Der Punkt, an dem hier die meisten stolpern, ist die Entscheidung, welche C4-Abstraktion ein Teil eines Systems ist. Die Namen klingen ähnlich, also gehen Sie die Essenslieferplattform von außen nach innen durch und fragen bei jedem Teil, ob es für sich allein deployt werden und laufen kann.

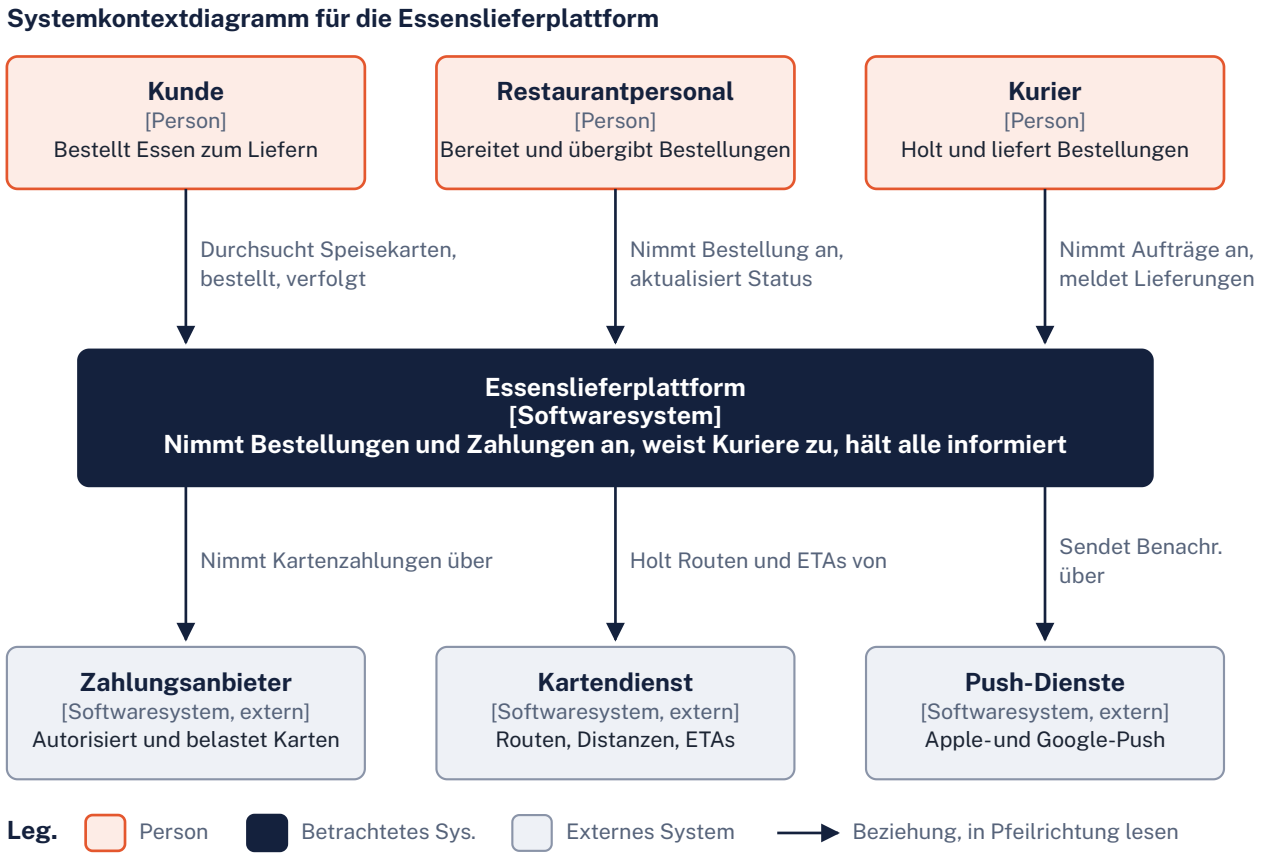
- Beginnen Sie oben. Die Essenslieferplattform als Ganzes ist ein Softwaresystem: Sie liefert Kunden, Restaurantpersonal und Kurieren einen Wert. Der Zahlungsanbieter ist ein weiteres Softwaresystem, das von einer anderen Organisation gebaut wird und ihr gehört.
- Schauen Sie nun in die Plattform hinein. Der Bestellservice läuft als eigenes Programm und spricht über das Netzwerk mit den anderen Teilen, also ist er ein Container. Das gilt auch für die Kunden-App auf dem Smartphone und die Restaurant-App, die im Browser eines Tablets läuft.
- Ein Container muss nicht Ihren eigenen Code ausführen. Die Bestelldatenbank speichert Daten, und die Plattform funktioniert nur, wenn sie läuft, also ist auch sie ein Container. Die Diagramme dieses Abschnitts behandeln das Topic für Bestellereignisse, das Nachrichten enthält, genauso.
- Zoomen Sie in den Bestellservice. Sein Bestell-Workflow, der die Bestellregeln anwendet, und sein Zahlungsclient, der den Zahlungsanbieter aufruft, sind Komponenten. Sie laufen im selben Prozess wie der Rest des Service und werden mit ihm deployt; der Zahlungsclient kann nicht für sich allein deployt werden.
- **Die Lehre:** Ein Teil, das für sich allein deployt werden und laufen kann, ist ein Container; ein Teil innerhalb eines solchen, das seinen Prozess teilt, ist eine Komponente; das Ganze, das seinen Nutzern einen Wert liefert und meist einem Team oder einer Organisation gehört, ist ein Softwaresystem. Klären Sie das vor dem Zeichnen, und jeder Kasten landet auf der richtigen Zoomstufe.

Ebene 1: das Systemkontextdiagramm

Das Systemkontextdiagramm ist der Ausgangspunkt. Es stellt das beschriebene System als einzelnen Kasten in die Mitte und umgibt es mit den Personen, die es nutzen, und den Softwaresystemen, von denen es abhängt oder die von ihm abhängen. Es verbirgt bewusst alle internen Details, einschließlich Technologien, Protokollen und interner Services. Es beantwortet die Fragen, die jeder Neuling, jedes Produktmanagement und jede Sicherheitsprüfung zuerst stellt: Was ist dieses System, wer nutzt es, und womit ist es verbunden?

Weil es das Innenleben verbirgt, ist das Kontextdiagramm auch das richtige Diagramm für Gespräche mit fachfremden Stakeholdern und für die Abgrenzung: Wo endet unsere Verantwortung? Im folgenden Essenslieferbeispiel zeigt es sofort, dass Kartenzahlungen, Routenplanung und Push-Zustellung zugekauft sind, was architekturrelevant ist (jedes davon ist eine externe Abhängigkeit mit eigener Verfügbarkeit, eigenen Kosten und eigenem Vertrag) und in den meisten Sichten auf Codeebene unsichtbar bleibt.

Essenslieferplattform: Systemkontext



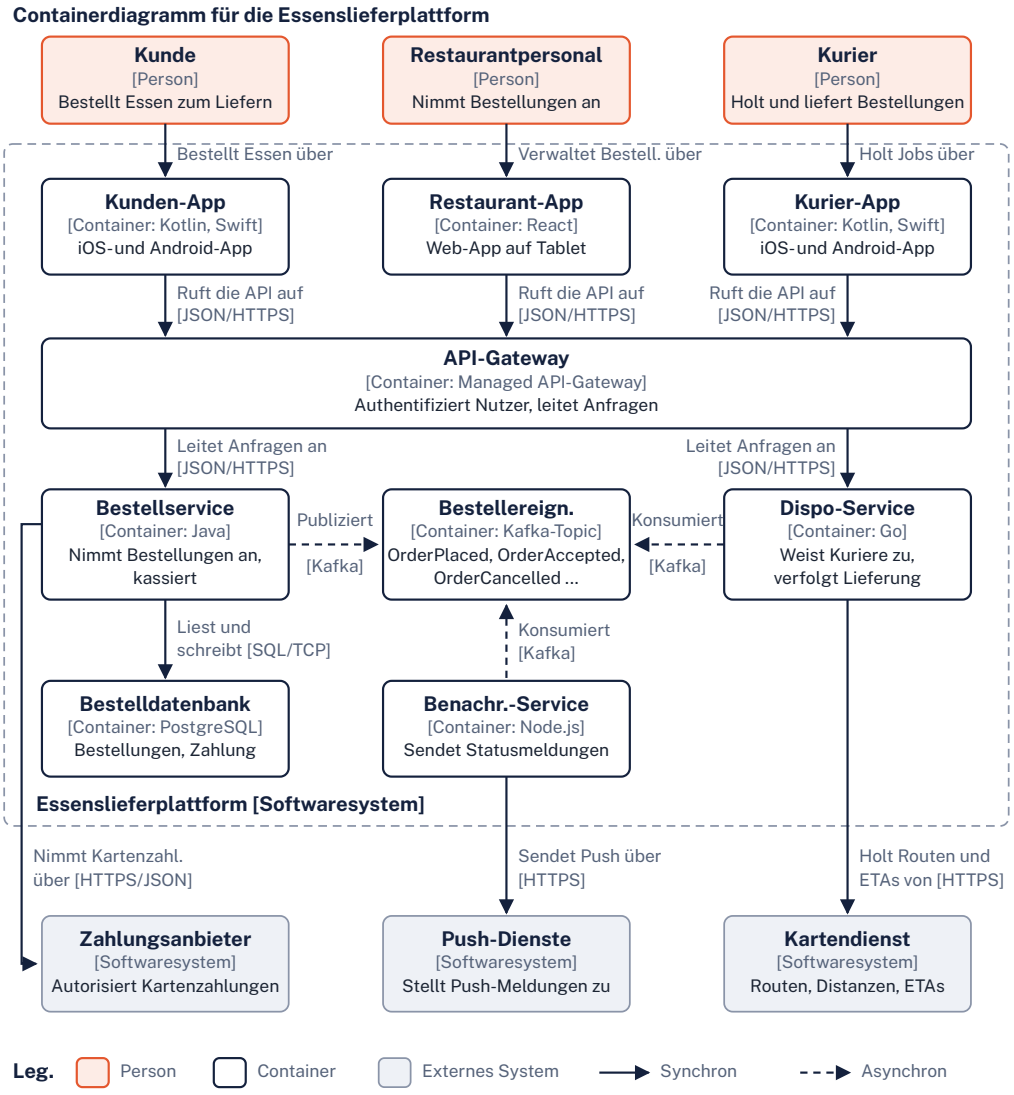
Ebene 1: die Plattform als ein Kasten, mit ihren drei Arten von Nutzern und drei externen Systemen; jedes Element hat einen Namen, einen Typ und eine Beschreibung, und jeder Pfeil ist beschriftet.

Ebene 2: das Containerdiagramm

Das Containerdiagramm zoomt in den Systemkasten und zeigt die Container darin: die Anwendungen und Datenspeicher, aus denen das System besteht, wofür jeder verantwortlich ist, welche Technologie jeder nutzt und wie sie kommunizieren. Für die meisten Teams ist das das wertvollste Diagramm, das sie besitzen. Es zeigt die wichtigsten Strukturentscheidungen (wie viele deploybare Einheiten, welche synchron und welche über Messaging kommunizieren, welche Daten besitzen) und ist konkret genug für Entwicklung und Betrieb, passt aber immer noch auf eine Seite.

Beachten Sie zwei Entscheidungen im folgenden Diagramm. Das Message-Topic für Bestellereignisse ist als eigener Container gezeichnet, weil es ein Speicher ist, in den Produzenten schreiben und aus dem Konsumenten lesen; die C4-Anleitung bietet diesen expliziten Stil an oder alternativ, das Topic wegzulassen und „über Topic für Bestellereignisse“ an die Pfeile zu schreiben. Das Zeichnen macht die Kopplung über das Topic deutlich sichtbar. Und die asynchronen Beziehungen verwenden eine gestrichelte Linie, die die Legende erklärt. Das Containerdiagramm sagt nichts darüber, wie viele Kopien jedes Service laufen, in welcher Region oder hinter welchem Load Balancer; das gehört in ein Deployment-Diagramm.

Essenslieferplattform: Container



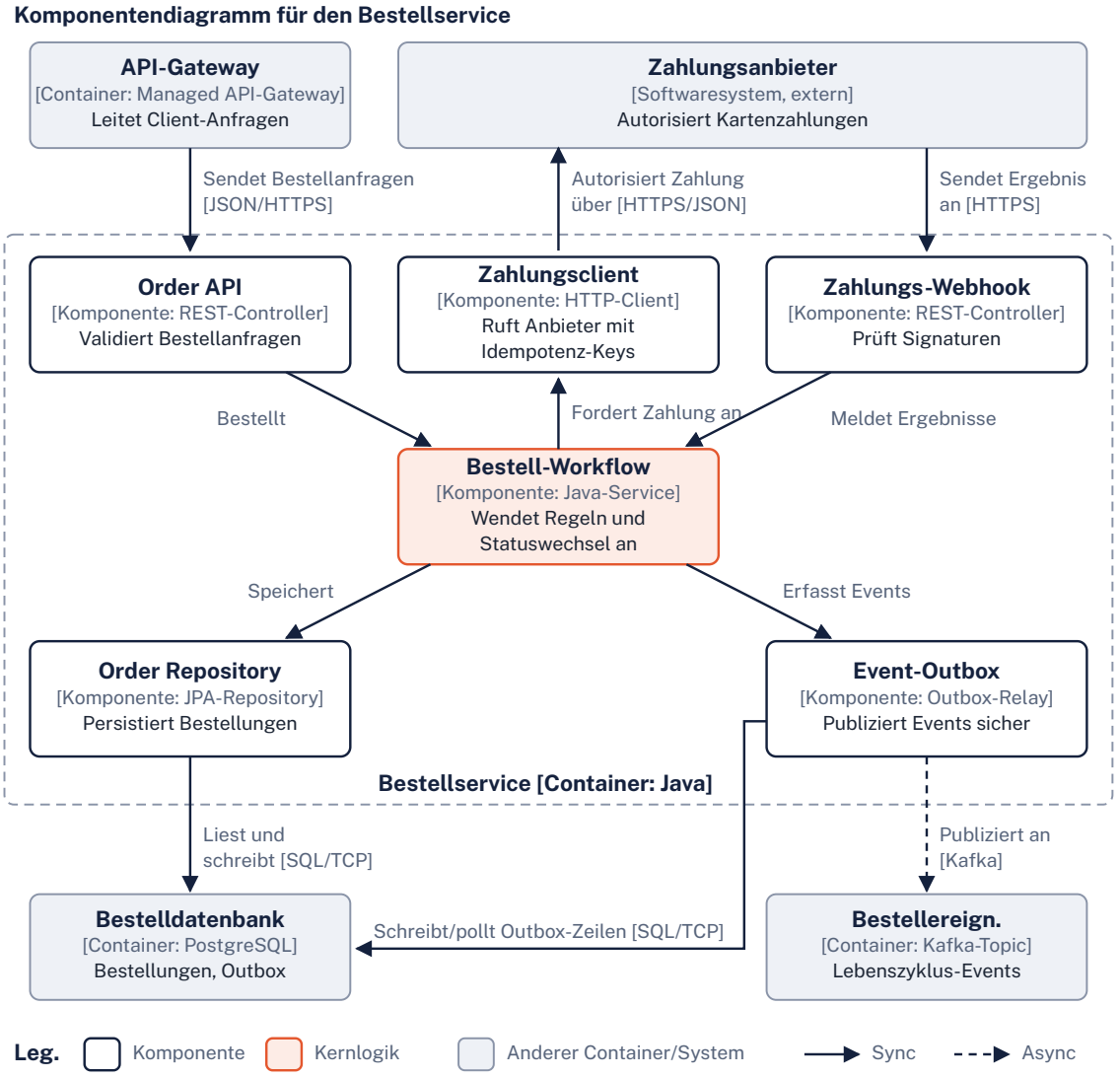
Ebene 2: die Apps, Services und Datenspeicher innerhalb der Plattformgrenze, mit Technologien an jedem Element und Protokollen an jedem Pfeil.

Ebene 3: das Komponentendiagramm

Das Komponentendiagramm zoomt in einen Container und zeigt die Komponenten darin. Es richtet sich an Architekten und Entwicklungsteams, die an diesem Container arbeiten, und hier wird die innere Struktur eines Service sichtbar: welcher Teil eingehende Anfragen bearbeitet, wo die Geschäftsregeln liegen, welcher Teil mit der Datenbank spricht, wie Ereignisse den Service verlassen.

Die C4-Anleitung empfiehlt, Komponentendiagramme nur dort zu zeichnen, wo sie einen Mehrwert bieten, typischerweise für Container, die komplex oder riskant sind oder von vielen Personen geändert werden, und sie automatisch zu generieren, wenn sie lange erhalten bleiben sollen, weil sich die Komponentenstruktur so oft ändert wie der Code. Bei der Essenslieferplattform erfüllt der Bestellservice diese Kriterien: Er verarbeitet Geld, spricht mit einem externen Anbieter und muss Ereignisse zuverlässig publizieren. Das folgende Diagramm zeigt eine typische Struktur, einschließlich einer Outbox-Komponente, die Ereignisse in derselben Transaktion wie die Bestellung in die Datenbank schreibt und sie anschließend an Kafka weiterleitet (das Outbox-Muster wird in Kapitel 6 behandelt).

Bestellservice: Komponenten



Ebene 3: innerhalb eines Containers, mit benachbarten Containern und Systemen außerhalb der gestrichelten Grenze, sodass jede Beziehung beide Enden hat.

Ebene 4: Code und die ergänzenden Diagramme

Die Codeebene zoomt in eine Komponente und zeigt ihre Klassen, Schnittstellen oder Tabellen, meist als UML-Klassendiagramm oder Entity-Relationship-Diagramm. Die C4-Anleitung behandelt sie als optional und hält es in der Regel nicht für lohnend, sie von Hand zu zeichnen: Die meisten Entwicklungsumgebungen können sie bei Bedarf generieren, und eine handgezeichnete Kopie ist veraltet, sobald jemand refaktoriert. Zeichnen Sie sie nur für die wichtigsten oder verwickeltsten Teile, etwa ein Domänenmodell im Herzen des Geschäfts (Abschnitt 4.4 zeigt ein solches Klassendiagramm).

Drei **ergänzende Diagramme** füllen die Lücken, die die Hierarchie lässt. Ein **Systemlandschaftsdiagramm** ist ein Kontextdiagramm ohne ein einzelnes System im Fokus: Es zeigt alle Softwaresysteme und Personen einer Organisation oder Domäne, was für Enterprise- und Plattformarchitekten nützlich ist. Ein dynamisches Diagramm zeigt, wie Elemente zur Laufzeit für ein Szenario zusammenarbeiten, als nummerierte Interaktionen auf einem Container- oder Komponentendiagramm (oder als UML-Sequenzdiagramm); das ist das C4-Gegenstück zu den Szenarien in 4+1. Ein Deployment-Diagramm zeigt, wie Container in einer Umgebung auf Infrastruktur abgebildet werden: Deployment-Knoten wie Cloud-Regionen, Kubernetes-Cluster, virtuelle Maschinen, Managed-Datenbankdienste und Geräte, mit der Anzahl der Instanzen. C4 empfiehlt ein Deployment-Diagramm je relevanter Umgebung, typischerweise Produktion und vielleicht Staging.

Notationsregeln, die Diagramme lesbar machen

C4 schreibt keine Formen oder Farben vor, wohl aber, was jedes Diagramm aussagen muss. Diese Regeln, verdichtet aus der C4-Notationsanleitung, machen die obigen Diagramme lesbar, ohne dass es eines Meetings zu ihrer Erklärung bedarf.

Checkliste: ein C4-Diagramm, das sich selbst erklärt

- [ ] **Ein Titel, der Typ und Umfang nennt**
  Zum Beispiel „Containerdiagramm für die Essenslieferplattform“, damit Leser die Zoomstufe sofort erkennen.
- [ ] **Eine Legende**
  Erklären Sie jede verwendete Form, Farbe, Rahmenart, Linienart und Pfeilspitze, auch wenn die Notation offensichtlich scheint. Leser aus anderen Teams kennen Ihre Konventionen nicht.
- [ ] **Jedes Element hat einen Namen, einen Typ und eine Beschreibung**
  Der Typ in Klammern (Person, Softwaresystem, Container, Komponente) beseitigt Mehrdeutigkeit; ein kurzer Satz zur Verantwortung sagt, wofür das Element da ist.
- [ ] **Technologie an jedem Container und jeder Komponente**
  „Container: Java“, „Container: PostgreSQL“. Technologieentscheidungen sind Architektur, und Leser brauchen sie.
- [ ] **Jede Beziehung ist gerichtet und beschriftet**
  Verwenden Sie einen Pfeil pro Beziehung, beschriftet mit dem, was sie in Pfeilrichtung tut („Publiziert Bestellereignisse an“), niemals nur „nutzt“ oder ein Doppelpfeil.
- [ ] **Protokolle an Beziehungen zwischen Containern**
  JSON/HTTPS, SQL/TCP, Kafka. Synchron oder asynchron ist eine der wichtigsten Tatsachen im Diagramm.
- [ ] **Abkürzungen erklärt**
  Schreiben Sie sie aus oder definieren Sie sie in der Legende oder im Glossar.
- [ ] **Eine Abstraktionsebene pro Diagramm**
  Keine Klassen in einem Containerdiagramm, keine Server in einem Komponentendiagramm.
- [ ] **Farbe als Ergänzung, nicht als einziges Signal**
  Diagramme werden schwarz-weiß gedruckt und von farbenblinden Menschen gelesen; nutzen Sie auch Linienarten und Beschriftungen, und halten Sie Farbbedeutungen über Diagramme hinweg einheitlich.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: ein typisches Teamdiagramm verbessern

Ein Team im Essenslieferunternehmen teilt diese Beschreibung seines Whiteboard-Diagramms: fünf Kästen mit den Beschriftungen „App“, „Backend“, „Orders“, „DB“ und „Stripe“; Linien dazwischen ohne Pfeilspitzen; der Kasten Orders ist grün, die anderen blau; es gibt keinen Titel und keine Legende. Die C4-Regeln Schritt für Schritt angewendet:

Titel und Ebene	Entscheiden Sie, was das Diagramm ist. Es mischt eine Mobile-App, einen Service und eine Datenbank, also ist es ein Containerdiagramm. Geben Sie ihm den Titel „Containerdiagramm für die Essenslieferplattform“.
Name, Typ, Beschreibung	Aus „App“ wird „Kunden-App [Container: Kotlin, Swift] iOS- und Android-App zum Bestellen von Essen“. „Backend“ entpuppt sich als das API-Gateway; aus „DB“ wird „Bestelldatenbank [Container: PostgreSQL]“.
Externe Systeme	„Stripe“ ist ein Produktname für ein externes System. Zeigen Sie es als „Zahlungsanbieter [Softwaresystem]“ außerhalb der Systemgrenze und nennen Sie den Anbieter in der Beschreibung, wenn das für die Leser wichtig ist.
Beziehungen	Ersetzen Sie jede einfache Linie durch einen beschrifteten Pfeil: „Ruft auf [JSON/HTTPS]“, „Liest und schreibt [SQL/TCP]“, „Autorisiert Kartenzahlungen über [HTTPS]“. Das Team stellt fest, dass eine Linie zwei verschiedene Dinge bedeutete (einen Aufruf in die eine Richtung und einen Webhook in die andere), also werden daraus zwei Pfeile.
Fehlende Elemente	Das Beschriften der Pfeile zeigt, dass Bestellereignisse den Dispositionsservice über ein Kafka-Topic erreichen, das nicht gezeichnet war. Fügen Sie das Topic als Container hinzu, mit gestrichelten asynchronen Pfeilen.
Legende und Farbe	Der grüne Kasten bedeutet, wie sich herausstellt, „gehört unserem Team“. Nehmen Sie das in eine Legende auf, oder ersetzen Sie die Farbe durch eine gestrichelte Grenze, die mit dem Teamnamen beschriftet ist.
Ergebnis	Das Diagramm hat einen Titel, eine Legende, drei Elemente und doppelt so viele Pfeile hinzugewonnen und braucht dennoch weniger Erklärung, weil jede Frage, die ein Leser stellen würde, auf der Seite beantwortet ist.

Der größte Nutzen kam vom Beschriften der Pfeile: Unbeschriftete Linien verbergen die wichtigsten architektonischen Tatsachen (Richtung, Protokoll, synchron oder asynchron).

Das verbesserte Diagramm hat eine Abhängigkeit (das Topic) offengelegt, die das Team selbst vergessen hatte, ein typischer Nebeneffekt der Regeln.

ACHTUNG

Häufige C4-Fehler

- Jeden Microservice als eigenes Softwaresystem zeichnen, obwohl ein Team sie alle besitzt. Laut C4-Anleitung sind Services, die einem Team gehören, Container innerhalb des Systems dieses Teams; Services, die verschiedenen Teams gehören, können jeweils als Softwaresystem modelliert werden.
- Einen gemeinsamen Message-Bus oder ein Service Mesh als einen großen Container darstellen, durch den alles fließt, was verbirgt, wer tatsächlich von wem abhängt. Modellieren Sie die einzelnen Topics oder Queues, oder schreiben Sie sie an die Pfeile.
- Komponentendiagramme für jeden Container von Hand zeichnen. Sie veralten schnell; zeichnen Sie sie gezielt oder generieren Sie sie.
- Deployment-Details (Replikate, Load Balancer, Availability Zones) ins Containerdiagramm statt in ein Deployment-Diagramm aufnehmen.
- Pfeile, die nur mit „nutzt“ oder „Daten“ beschriftet sind, oder Pfeile in beide Richtungen.
- Personen im Kontextdiagramm weglassen, wodurch der Grund verschwindet, warum das System existiert.

Werkzeuge: Zeichnen oder Modellieren

C4-Diagramme lassen sich mit jedem Zeichenwerkzeug erstellen, und viele Teams beginnen mit einem Whiteboard oder einer allgemeinen Diagrammanwendung. Die C4-Website unterscheidet zwei Familien von Werkzeugen. **Zeichenwerkzeuge** zeichnen Kästen und Linien; sie sind flexibel und vertraut, aber die Diagramme sind unabhängige Bilder, sodass das Umbenennen eines Elements bedeutet, jedes Diagramm zu bearbeiten, das es zeigt, und nichts prüft, ob die Diagramme übereinstimmen. **Modellierungswerkzeuge** halten ein einziges Modell aus Personen, Systemen, Containern, Komponenten und Beziehungen und erzeugen jedes Diagramm als Sicht auf dieses Modell; umbenannt wird einmal, die Diagramme bleiben konsistent, und das Modell lässt sich abfragen und validieren.

Structurizr, von Simon Brown entwickelt, ist das Referenz-Modellierungswerkzeug für C4. Seine textbasierte Structurizr DSL beschreibt das Modell und die Sichten in einer Datei, die neben dem Code in der Versionsverwaltung liegt. Weitere Optionen sind C4-PlantUML (C4-Formen für PlantUML), die C4-Diagrammunterstützung von Mermaid und verschiedene kommerzielle Modellierungswerkzeuge; Abschnitt 4.7, Diagrams as Code, zeigt Beispiele. Die Wahl ist weniger wichtig als die Gewohnheit: Halten Sie die Quelle der Diagramme dort, wo das Team arbeitet, und aktualisieren Sie sie im selben Pull Request wie die Änderung, die sie beschreibt.

Häufige Fragen

Ersetzt C4 die UML?
Nicht ganz. C4 definiert Abstraktionen und Diagrammtypen, keine Notation, und sagt nichts über vieles, was UML präzise ausdrücken kann, etwa Zustandsautomaten oder detaillierte Nachrichtenabfolgen. Viele Teams nutzen C4 für die Struktur und entlehnen UML-Sequenz-, Zustands- und Deployment-Diagramme, wo Präzision gebraucht wird (Abschnitt 4.4).

Wie passen Serverless-Funktionen in C4?
Jede Funktion oder jede Gruppe gemeinsam deployter Funktionen ist ein Container: eine separat deploybare Einheit, die Code ausführt. Gemeinsame Datenspeicher und Queues, die sie nutzen, sind ebenfalls Container.

Unser System ist ein einziger deploybarer Monolith. Ist das Containerdiagramm sinnlos?
Nein. Ein Monolith hat typischerweise trotzdem einen Web- oder Mobile-Client, die Serveranwendung, eine Datenbank und vielleicht einen Dateispeicher und einen zeitgesteuerten Job, jeweils ein Container. Das Komponentendiagramm des Monolithen wird dann zur interessantesten Sicht auf seine internen Module.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Was ist ein Container im C4-Modell? Nennen Sie zwei Beispiele, die nichts mit Docker zu tun haben.

Eine separat lauffähige oder deploybare Einheit, die Code ausführt oder Daten speichert. Beispiele: eine Mobile-App auf einem Smartphone, eine Single-Page-App im Browser, ein Datenbankschema, eine Serverless-Funktion.

▼ Welche beiden C4-Diagramme empfiehlt die C4-Anleitung jedem Team?

Das Systemkontextdiagramm und das Containerdiagramm.

▼ Wo sollten Replikatzahlen, Regionen und Load Balancer dargestellt werden?

In einem Deployment-Diagramm, eines je Umgebung, nicht im Containerdiagramm.

▼ Warum besteht C4 auf gerichteten, beschrifteten Pfeilen?

Unbeschriftete oder beidseitige Linien verbergen, was die Beziehung ist (Aufruf, Datenfluss, Ereignis), in welche Richtung sie geht und welches Protokoll sie nutzt, und das gehört zu den wichtigsten architektonischen Tatsachen im Diagramm.

▼ **Probieren Sie es aus: Ein Diagramm zeigt die Kästen „Web“, „API“, „Kafka“, „Postgres“ und „Payment“, verbunden durch Linien ohne Pfeile, ohne Titel oder Legende. Ein Kasten ist rot. Wenden Sie die C4-Regeln an: Was ist falsch, und was würden Sie ändern?**

Es hat weder einen Titel, der Typ und Umfang nennt, noch eine Legende, sodass niemand weiß, was Rot bedeutet. Boxen brauchen Name, Typ, Beschreibung und bei Containern die Technologie. Payment ist ein externes Softwaresystem und sollte anders aussehen. Linien brauchen eine einzige Richtung, eine konkrete Beschriftung und ein Protokoll, etwa „veröffentlicht OrderPlaced [Kafka]“. Dies ist ein Container-Diagramm, nennen Sie es also so. Behalten Sie Rot nur, wenn die Legende es erklärt, sonst entfernen Sie es.

QUELLEN

- The C4 model for visualising software architecture (Simon Brown): https://c4model.com/
- C4 model: Container (not Docker!): https://c4model.com/abstractions/container
- C4 model: Notation: https://c4model.com/diagrams/notation
- C4 model: System context diagram: https://c4model.com/diagrams/system-context
- Structurizr DSL documentation: https://docs.structurizr.com/dsl

Die UML-Diagramme, die Architekten noch verwenden

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.6 APIs und Integration
- 4.2 Sichten und Blickwinkel
- 4.3 Das C4-Modell

Begriffe von früher, die er verwendet: Schnittstelle, Synchroner Aufruf, Zuordnungssicht, Abhängigkeitsumkehr

DIE IDEE IN KÜRZE

Musiker auf der ganzen Welt können dieselben Noten lesen, weil Noten und Zeichen überall dasselbe bedeuten. UML ist ein ähnlicher gemeinsamer Satz von Symbolen zum Zeichnen von Software: Ein bestimmter Pfeil bedeutet „wartet auf eine Antwort“, ein anderer „sendet und macht weiter“, eine gefüllte Raute „besteht aus“. Sie müssen nicht alles davon kennen. Architekten nutzen meist eine Handvoll ihrer Zeichnungen: eine, die zeigt, wer mit wem in welcher Reihenfolge spricht, eine, die die Stadien zeigt, die etwas durchläuft, eine, die zeigt, wo alles läuft, und ein paar weitere, jede nur dann eingesetzt, wenn ihre Präzision wirklich hilft.

WAS SIE WISSEN MÜSSEN

- UML ist eine von der OMG standardisierte universelle Modellierungssprache; die aktuelle Version ist UML 2.5.1 (Dezember 2017), die vierzehn Diagrammtypen in den Familien Struktur und Verhalten definiert.
- Fowler unterscheidet UML als Skizze (selektive, informelle Kommunikation), als Bauplan (vollständige, präzise Modelle) und als Programmiersprache (ausführbare Modelle); Architekten skizzieren meistens.
- Komponentendiagramme zeigen bereitgestellte (Ball) und benötigte (Socket) Schnittstellen; Deployment-Diagramme zeigen Knoten, Artefakte und Kommunikationspfade.
- Sequenzdiagramme zeigen Lebenslinien und Aktivierungen; eine gefüllte Pfeilspitze ist ein synchroner Aufruf, eine offene Pfeilspitze an einer durchgezogenen Linie eine asynchrone Nachricht und eine gestrichelte Linie eine Antwort; Fragmente wie alt, opt und loop ergänzen den Kontrollfluss.

- Zustandsdiagramme beschreiben Lebenszyklen mit Zuständen und Transitionen, beschriftet mit Ereignis [Guard] / Aktion; Aktivitätsdiagramme beschreiben Workflows mit Entscheidungen, Gabelungen, Zusammenführungen und Swimlanes.
- Klassendiagramme bleiben am besten kleinen Domänenmodellen mit Assoziationen, Multiplizitäten und Komposition vorbehalten; vollständige Klassendiagramme sollten generiert, nicht von Hand gezeichnet werden.
- Präzision lohnt sich an Schnittstellen zwischen Organisationen, in regulierten und sicherheitskritischen Bereichen, bei Lebenszyklen, die sich mehrere Teams teilen, und bei verteilter Entwicklung.

Was UML ist und wo sie heute steht

Die Unified Modeling Language (UML) ist eine universelle grafische Sprache zur Beschreibung von Struktur und Verhalten von Software. Sie entstand Mitte der 1990er-Jahre bei Rational Software aus der Zusammenführung der Methoden von Grady Booch, James Rumbaugh und Ivar Jacobson und wurde 1997 von der Object Management Group (OMG), einem Industriekonsortium für Standards, als Standard übernommen. UML 2.0 (2005) war eine größere Überarbeitung. Die aktuelle Version ist UML 2.5.1, im Dezember 2017 von der OMG veröffentlicht; die frühere UML 2.4.1 wurde zudem von der ISO als ISO/IEC 19505 (2012) veröffentlicht. Die Spezifikation definiert vierzehn Diagrammtypen, unterteilt in Strukturdiagramme (etwa Klassen-, Komponenten- und Deployment-Diagramme) und Verhaltensdiagramme (etwa Aktivitäts-, Zustands- und Sequenzdiagramme).

Der Ruf der UML hat sich erheblich gewandelt. In den späten 1990er- und frühen 2000er-Jahren wurde sie als Grundlage der modellgetriebenen Entwicklung propagiert, bei der detaillierte Modelle in Code umgewandelt werden sollten. Dieser Anspruch ist außerhalb spezialisierter Domänen weitgehend gescheitert, und viele Teams verbinden UML mit schwergewichtigen Prozessen und Diagrammen, die niemand las. Als Marian Petre für eine auf der ICSE 2013 vorgestellte Studie 50 professionelle Softwareentwickler befragte, gaben 35 an, UML überhaupt nicht zu verwenden, und die meisten anderen nutzten sie selektiv und informell; die meisten Teams zeichnen informelle Kästen und Linien statt strikter UML. Dennoch bleibt eine Handvoll UML-Diagrammtypen das beste verfügbare Werkzeug für bestimmte Fragen, besonders für Fragen zum Verhalten über die Zeit, und ihre Notation ist weithin verständlich. Ein Architekt braucht nicht alle vierzehn Diagrammtypen, sondern vielleicht sechs, bewusst eingesetzt.

Dieser Abschnitt behandelt diese sechs, jeweils mit einem kleinen Beispiel aus der Essenslieferplattform, die das ganze Kapitel begleitet. Er setzt das C4-Vokabular aus Abschnitt 4.3 voraus; beide lassen sich gut kombinieren, mit C4 für den strukturellen Überblick und UML dort, wo sich Präzision bei Schnittstellen, Interaktionen oder Zuständen bezahlt macht.

Skizze, Bauplan oder Programmiersprache

Martin Fowlers *UML Distilled* (3. Auflage, Addison-Wesley, 2003) und seine Essays aus demselben Jahr unterscheiden drei Arten, UML zu verwenden, eine Einteilung, zu der Steve Mellor unabhängig gelangte. Als **Skizze** wird UML informell und selektiv verwendet, um Ideen zu vermitteln: auf einem Whiteboard, bevor Code geschrieben wird (Vorwärtsskizze), oder um einen Teil eines bestehenden Systems zu erklären (Rückwärtsskizze). Nur die wichtigen Elemente erscheinen, die Notation wird locker befolgt, und das Diagramm wird oft weggeworfen. Fowlers eigene Zusammenfassung lautet „comprehensiveness is the enemy of comprehensibility“ (Vollständigkeit ist der Feind der Verständlichkeit).

Als **Bauplan** dient UML dazu, vollständige, präzise Modelle zu erstellen, die jemand anderes implementiert, oder die aus bestehendem Code generiert werden, um ihn vollständig zu dokumentieren; es geht um Vollständigkeit, und meist braucht man Werkzeuge. Als **Programmiersprache** ist das UML-Modell detailliert genug, um in ausführbaren Code kompiliert zu werden, das Ziel der Model-Driven Architecture (MDA). Dieser Modus überlebt in einigen Nischen im Embedded- und sicherheitskritischen Bereich, oft über spezialisierte Profile und Werkzeuge, hat sich aber nicht durchgesetzt.

Für Architekten ist der Skizzenmodus der Normalfall und der Bauplanmodus die Ausnahme. Die nützliche Frage zu jedem Diagramm lautet: Wer wird es lesen, und welche Entscheidung oder welches Verständnis unterstützt es? Ein Sequenzdiagramm, skizziert, um sich darauf zu einigen, wie Zahlungsfehler behandelt werden, ist wertvoll, auch wenn es die Hälfte der Notation ignoriert. Ein vollständiges, von Hand gepflegtes Klassenmodell eines Service mit 200 Klassen ist es meist nicht.

Komponentendiagramme: Teile und ihre Schnittstellen

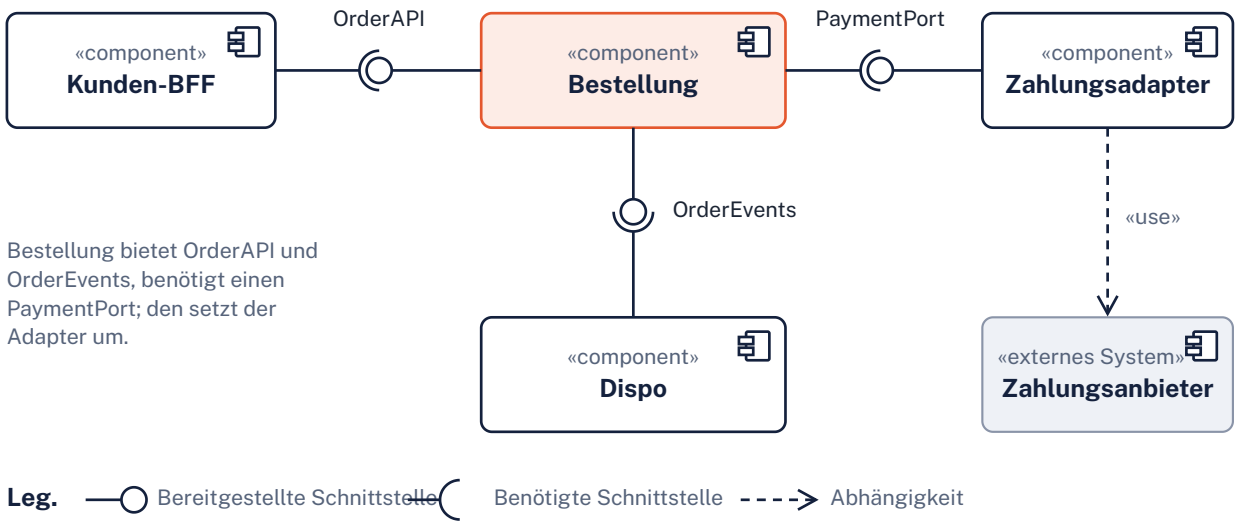
Ein UML-Komponentendiagramm zeigt modulare Teile eines Systems und vor allem ihre **Schnittstellen**. Seine charakteristische Notation ist „Ball und Socket“: ein kleiner Kreis an einem Stiel (eine bereitgestellte Schnittstelle, die die Komponente anbietet) und ein Halbkreis (eine benötigte Schnittstelle, die jemand anderes bereitstellen muss). Sitzt ein Ball in einem Socket, sind die beiden Komponenten über diese Schnittstelle verbunden. Abhängigkeiten sind gestrichelte Pfeile, optional beschriftet mit einem Stereotyp wie «use».

Genau das interessiert Architekten, wenn sie Abhängigkeitsrichtung und Austauschbarkeit zeigen wollen. Im Beispiel hängt die Bestellkomponente nicht direkt vom Zahlungsanbieter ab; sie benötigt eine Schnittstelle `PaymentPort`, die ein separater Zahlungsadapter implementiert. Das ist das Denken in Abhängigkeitsumkehr und Ports and Adapters aus Kapitel 3, sichtbar gemacht: Einen anderen Zahlungsanbieter einzusetzen bedeutet, einen neuen

Adapter zu schreiben, nicht die Bestellkomponente zu ändern. Ein C4-Komponentendiagramm kann dieselben Teile zeigen, unterscheidet aber nicht zwischen bereitgestellten und benötigten Schnittstellen.

Komponentendiagramm

Komponentendiagramm: Bestellung und Schnittstellen



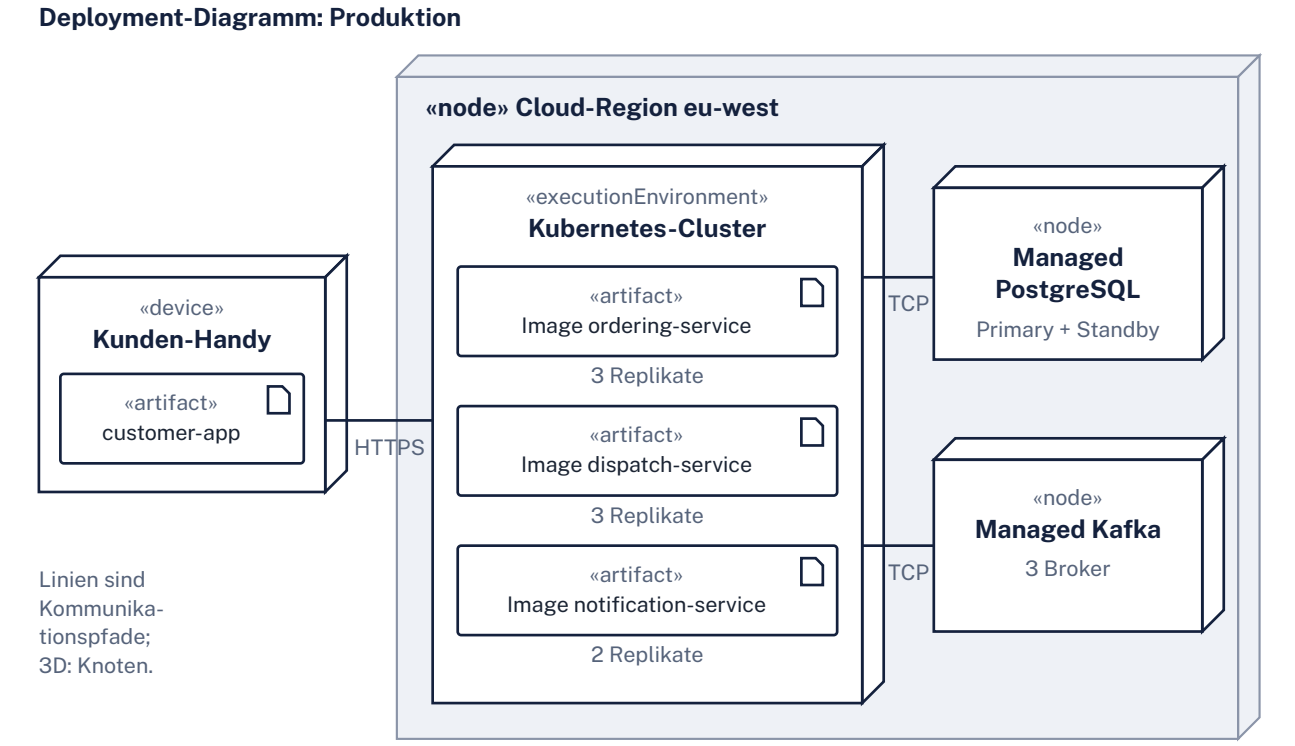
Bälle sind Schnittstellen, die eine Komponente bereitstellt; Sockets sind Schnittstellen, die sie benötigt. Die Bestellung hängt von einem Port ab, nicht vom Anbieter. (BFF: Backend for Frontend, eine serverseitige Komponente, die einer einzigen Client-App dient.)

Deployment-Diagramme: Knoten und Artefakte

Ein UML-Deployment-Diagramm zeigt die physische oder virtuelle Infrastruktur und was darauf läuft. Seine Elemente sind **Knoten**, als dreidimensionale Kästen gezeichnet, also Rechenressourcen: ein «device» wie ein Smartphone oder Server oder eine «executionEnvironment» wie ein Betriebssystem, eine Container-Laufzeit oder ein Kubernetes-Cluster; Knoten können verschachtelt sein. **Artefakte**, gezeichnet als Rechtecke mit einem Dokumentsymbol und dem Stereotyp «artifact», sind die physischen Dateien, die deployt werden: ein JAR, ein Container-Image, ein App-Bundle, eine Konfigurationsdatei. Linien zwischen Knoten sind **Kommunikationspfade**, beschriftet mit Protokollen.

Deployment-Diagramme sind Zuordnungssichten im Sinne des SEI (Abschnitt 4.2). Sie beantworten die Fragen des Betriebsteams: Wo läuft jeder Teil, wie viele Kopien, in welcher Region, welche Netzwerkpfade gibt es? Das Deployment-Diagramm von C4 deckt dasselbe Gebiet mit weniger formaler Notation ab; beides ist in Ordnung, solange es eines je wichtiger Umgebung gibt und es Replikate, Managed Services und Netzwerkgrenzen zeigt.

Deployment-Diagramm



Knoten sind, wo etwas läuft; Artefakte sind, was deployt wird; Linien sind Kommunikationspfade.

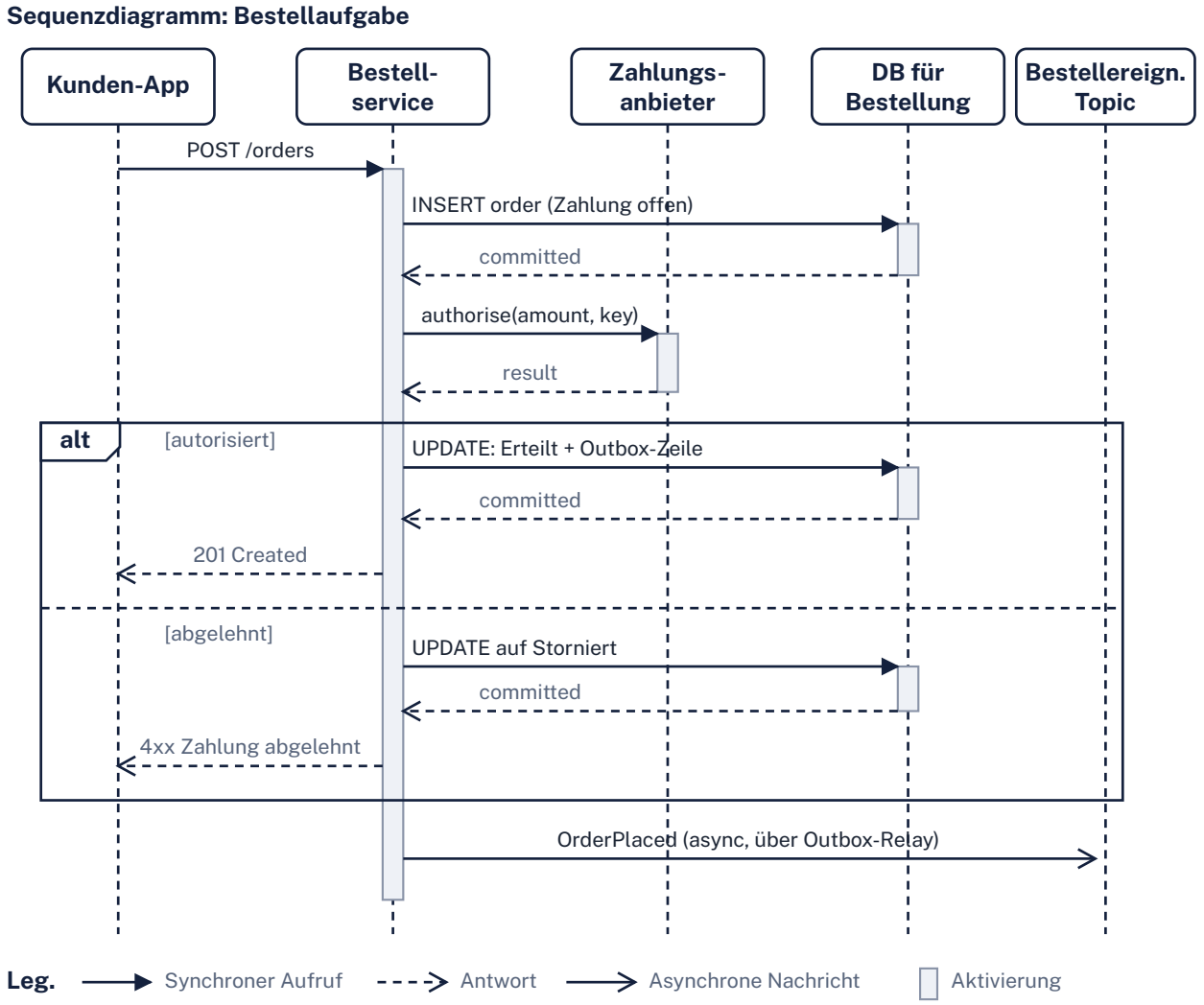
Sequenzdiagramme: wer wen in welcher Reihenfolge aufruft

Wenn Architekten nur ein einziges UML-Diagramm behalten, ist es meist das Sequenzdiagramm. Es zeigt, wie Beteiligte in einem Szenario über die Zeit interagieren. Die Beteiligten stehen oben nebeneinander, jeder mit einer gestrichelten senkrechten Lebenslinie; die Zeit läuft nach unten. Ein schmales Rechteck auf einer Lebenslinie, die **Aktivierung** (Ausführungsspezifikation), zeigt, wann dieser Beteiligte mit der Bearbeitung einer Nachricht beschäftigt ist.

Die Pfeilspitzen tragen Bedeutung. Eine **synchrone Nachricht**, bei der der Sender auf eine Antwort wartet, hat eine durchgezogene Linie und eine gefüllte Pfeilspitze. Eine **Antwort** hat eine gestrichelte Linie und eine offene Pfeilspitze. Eine **asynchrone Nachricht**, bei der der Sender weitermacht, ohne zu warten, hat eine durchgezogene Linie und eine offene Pfeilspitze. Das richtig darzustellen ist architektonisch wichtig: Ob der Bestellservice auf den Zahlungsanbieter wartet oder veröffentlicht und weitermacht, entscheidet über seine Latenz, seine Fehlerbilder und seine Kopplung (siehe Abschnitt 0.6). Auch die Reihenfolge der Schritte zählt: Das folgende Diagramm speichert die Bestellung, bevor es den Anbieter aufruft, sodass ein Absturz zwischen den beiden Schritten nie dazu führen kann, dass ein Kunde belastet wird, ohne dass die Bestellung erfasst ist, und der Idempotenzschlüssel macht eine wiederholte Autorisierung sicher.

Kombinierte Fragmente ergänzen den Kontrollfluss: Ein Kasten mit der Beschriftung `alt` und mit Guards versehenen Abschnitten, getrennt durch eine gestrichelte Linie, zeigt Alternativen (if/else), `opt` zeigt einen optionalen Teil, `loop` eine Wiederholung und `par` parallele Abschnitte. Setzen Sie sie sparsam ein; ein Sequenzdiagramm mit verschachtelten Schleifen und Alternativen ist schlecht gezeichneter Code. Zeichnen Sie ein Diagramm je wichtigem Szenario, einschließlich der Fehlerszenarien, in denen das meiste architektonische Denken stattfindet.

Sequenzdiagramm



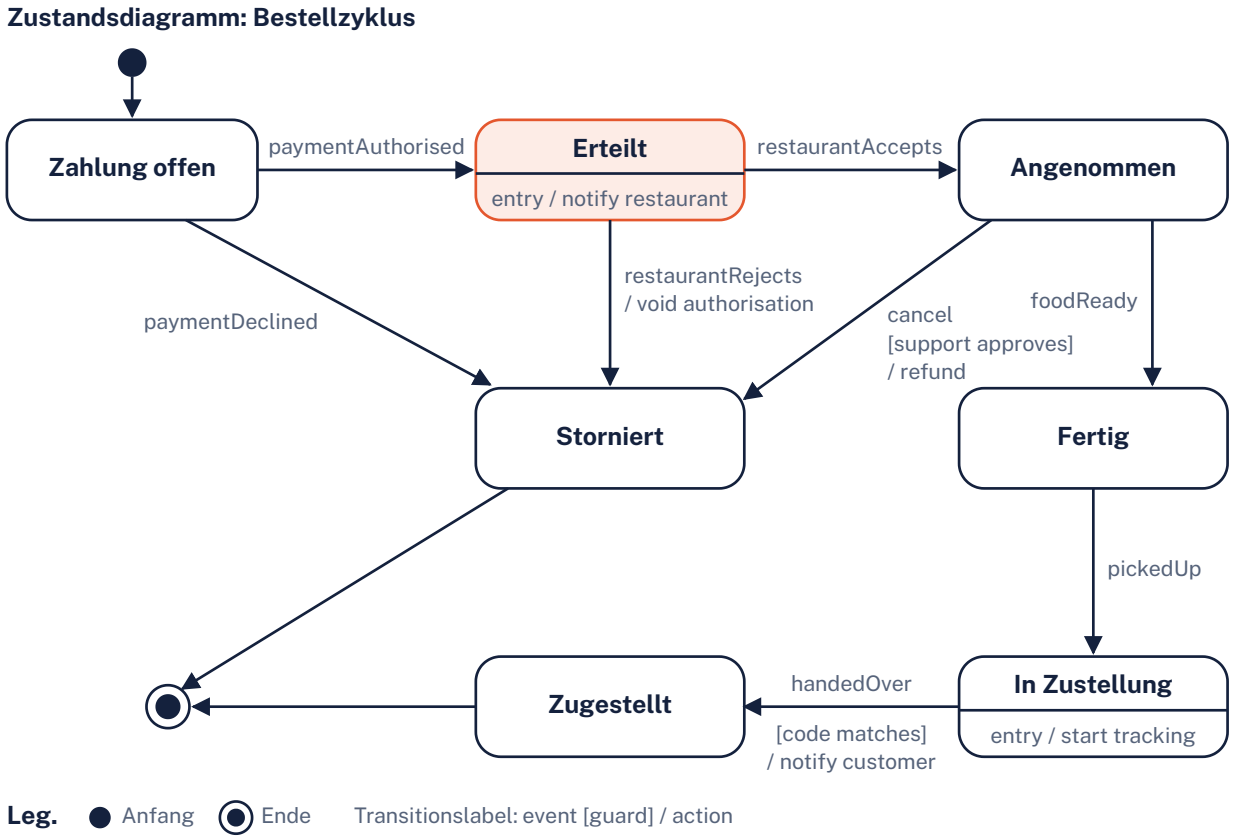
Eine Bestellung sicher aufgeben: zuerst die Bestellung speichern, die Karte mit einem Idempotenzschlüssel autorisieren, dann die Bestellung aktualisieren. Ein alt-Fragment zeigt die beiden Ergebnisse, und ein asynchrones Ereignis beendet den Ablauf.

Zustandsdiagramme: Lebenszyklen und Regeln

Viele Geschäftsobjekte haben einen Lebenszyklus: eine Bestellung, eine Zahlung, ein Abonnement, ein Versicherungsfall. Ein Zustandsdiagramm beschreibt ihn präzise. **Zustände** sind abgerundete Rechtecke; ein gefüllter Kreis markiert den **Anfangs**-Pseudozustand und ein Kreis mit Punkt (Bullseye) den **End**zustand. **Transitionen** sind Pfeile, beschriftet mit `event [guard] / action`: dem Ereignis, das die Transition auslöst, einer optionalen Guard-Bedingung, die wahr sein muss, und einer optionalen Aktion, die unterwegs ausgeführt wird. Zustände können interne Verhaltensweisen haben, etwa `entry / notify restaurant`.

Zustandsautomaten sind öfter architekturrelevant, als man erwartet. Der folgende Bestelllebenszyklus wird von mehreren Services geteilt: Die Bestellung besitzt ihn, Disposition und Benachrichtigungen reagieren auf seine Transitionen, und das Supportpersonal kann einige davon auslösen. Sich in einem Diagramm auf die Zustände und die zulässigen Transitionen zu einigen, verhindert eine ganze Klasse von Bugs, bei denen ein Service eine Bestellung für storniert hält, während ein anderer einen Kurier losschickt. Das Diagramm legt auch Fragen offen, die Text verbirgt: Kann eine Bestellung storniert werden, wenn sie bereits in Zustellung ist? (Hier nein: Es gibt keine Transition von diesem Zustand nach Storniert.) Ist eine Erstattung eine Aktion an einer Transition oder ein eigener Lebenszyklus? Auch das Geld folgt den Zuständen: Die Karte wird beim Aufgeben der Bestellung nur autorisiert und erst belastet, wenn das Restaurant annimmt; eine Ablehnung hebt daher die Autorisierung auf, und nur eine spätere Stornierung erfordert eine Erstattung.

Zustandsdiagramm



Der Bestelllebenszyklus: jede zulässige Transition, mit Ereignissen, Guards und Aktionen, auf einer Seite.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: ein Transitionslabel lesen

Die schwierigste Idee dieses Abschnitts ist die kleine Sprache, die an jedem Pfeil eines Zustandsdiagramms steht: event [guard] / action. Verfolgen Sie eine Bestellung durch den Bestelllebenszyklus dieses Abschnitts und lesen Sie jedes Label der Reihe nach.

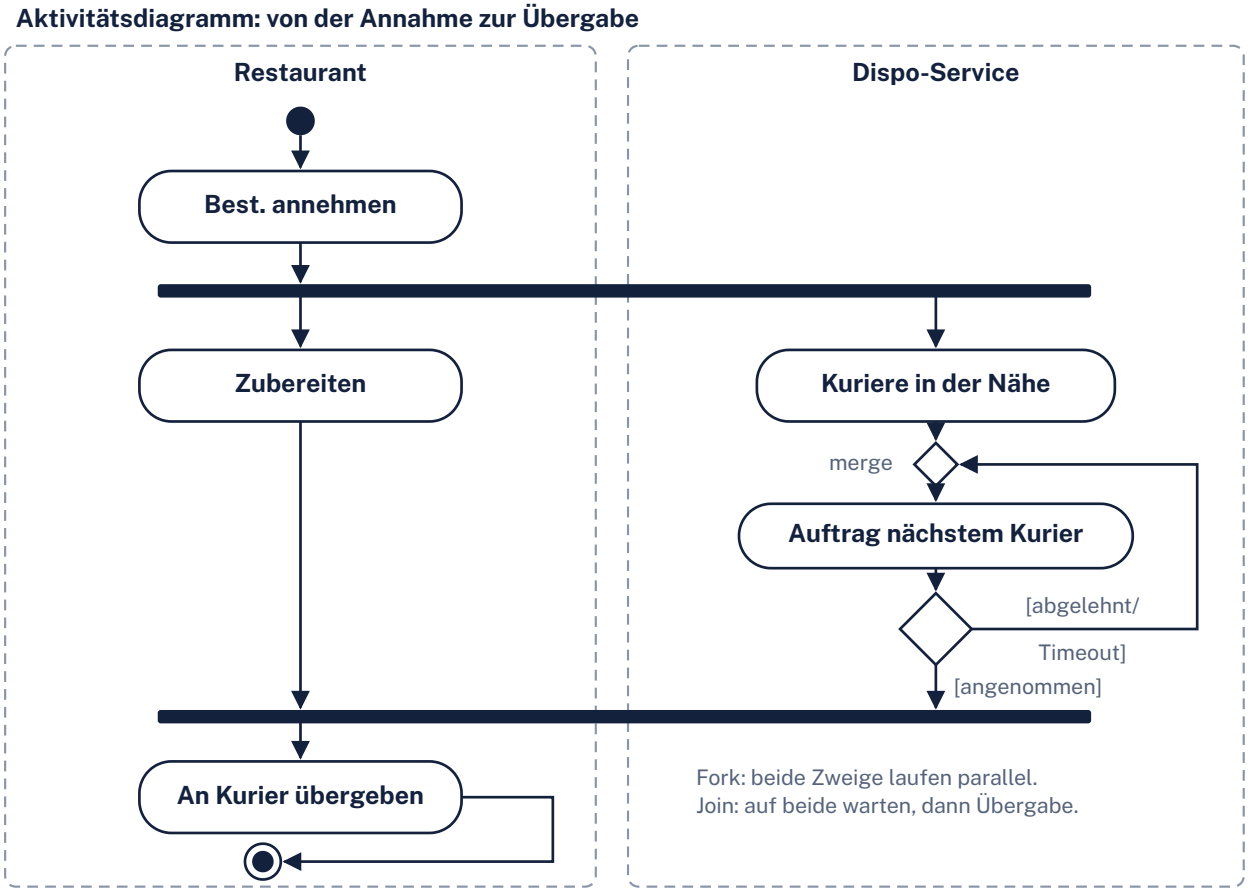
- Eine neue Bestellung beginnt in Zahlung offen, worauf der gefüllte Kreis zeigt. Sie bleibt dort, bis ein Ereignis eintrifft, das einer ihrer ausgehenden Pfeile nennt.
- Das Ereignis paymentAuthorised trifft ein, und sein Pfeil führt zu Erteilt. Dieses Label ist nur ein Ereignis: keine Bedingung und keine Aktion. Beim Eintreffen läuft das Eintrittsverhalten von Erteilt, notify restaurant, sodass das Restaurant von der Bestellung erfährt.
- Das Ereignis restaurantAccepts bringt die Bestellung nach Angenommen. Nun kommt eine Stornierungsanfrage herein. Der Pfeil von Angenommen lautet cancel [support approves] / refund: cancel ist das Ereignis, der Guard in eckigen Klammern ist die Bedingung, die wahr sein muss, und refund nach dem Schrägstrich ist die Aktion, die unterwegs ausgeführt wird.
- Hat der Support nicht zugestimmt, ist der Guard falsch: Die Transition feuert nicht, und die Bestellung bleibt einfach in Angenommen.
- Hat der Support zugestimmt, feuert die Transition: Die Erstattung erfolgt unterwegs, und die Bestellung landet in Storniert.
- Wäre die Bestellung bereits In Zustellung gewesen, verließe kein mit cancel beschrifteter Pfeil diesen Zustand, also hätte die Anfrage sie nicht stornieren können. Ein fehlender Pfeil ist auch eine Regel.
- **Die Lehre:** Lesen Sie jeden Pfeil als ‚wenn dieses Ereignis eintritt und diese Bedingung wahr ist, tue dies und wechsle dorthin‘. Sich in einem Diagramm auf diese Labels zu einigen, hilft mehreren Services, sich an dieselben Regeln für eine Bestellung zu halten.

Aktivitätsdiagramme: Workflows und Parallelität

Ein Aktivitätsdiagramm ist ein Flussdiagramm mit präziser Semantik. Abgerundete Rechtecke sind **Aktionen**; Rauten sind **Entscheidungen** (mit Guards an den ausgehenden Flüssen) und Zusammenführungen; dicke Balken sind **Gabelungen** (Forks), die einen Fluss in parallele Flüsse aufteilen, und **Vereinigungen** (Joins), die auf alle eingehenden Flüsse warten, bevor es weitergeht. **Partitionen**, oft Swimlanes genannt, zeigen, wer oder was jede Aktion ausführt. Eine Schleife muss über eine Zusammenführungsraute zurückfließen: Zwei Flüsse, die direkt in eine Aktion gezeichnet werden, würden bedeuten, dass die Aktion wie ein Join auf beide wartet.

Architekten nutzen Aktivitätsdiagramme für Geschäftsprozesse und Workflows, die mehrere Teile eines Systems umspannen, besonders wo Parallelität und Verantwortung wichtig sind. Das Beispiel zeigt, dass Essenszubereitung und Kurierzuweisung parallel laufen, dass die Zuweisung so lange wiederholt wird, bis ein Kurier annimmt, und dass die Übergabe auf beides wartet. Das wirft sofort eine architektonische Frage auf: Was, wenn innerhalb von zwanzig Minuten kein Kurier annimmt, während das Essen fertig ist? Das Diagramm macht die Lücke sichtbar; die Antwort könnte eine Timeout-Transition, ein Kompensationsschritt oder eine Änderung am Bestelllebenszyklus sein. Langlaufende Workflows dieser Art werden oft als Sagas oder orchestrierte Workflows implementiert, behandelt in Kapitel 6.

Aktivitätsdiagramm



Essenszubereitung und Kurierzuweisung laufen zwischen einem Fork und einem Join parallel; die Entscheidung wiederholt sich, bis ein Kurier annimmt.

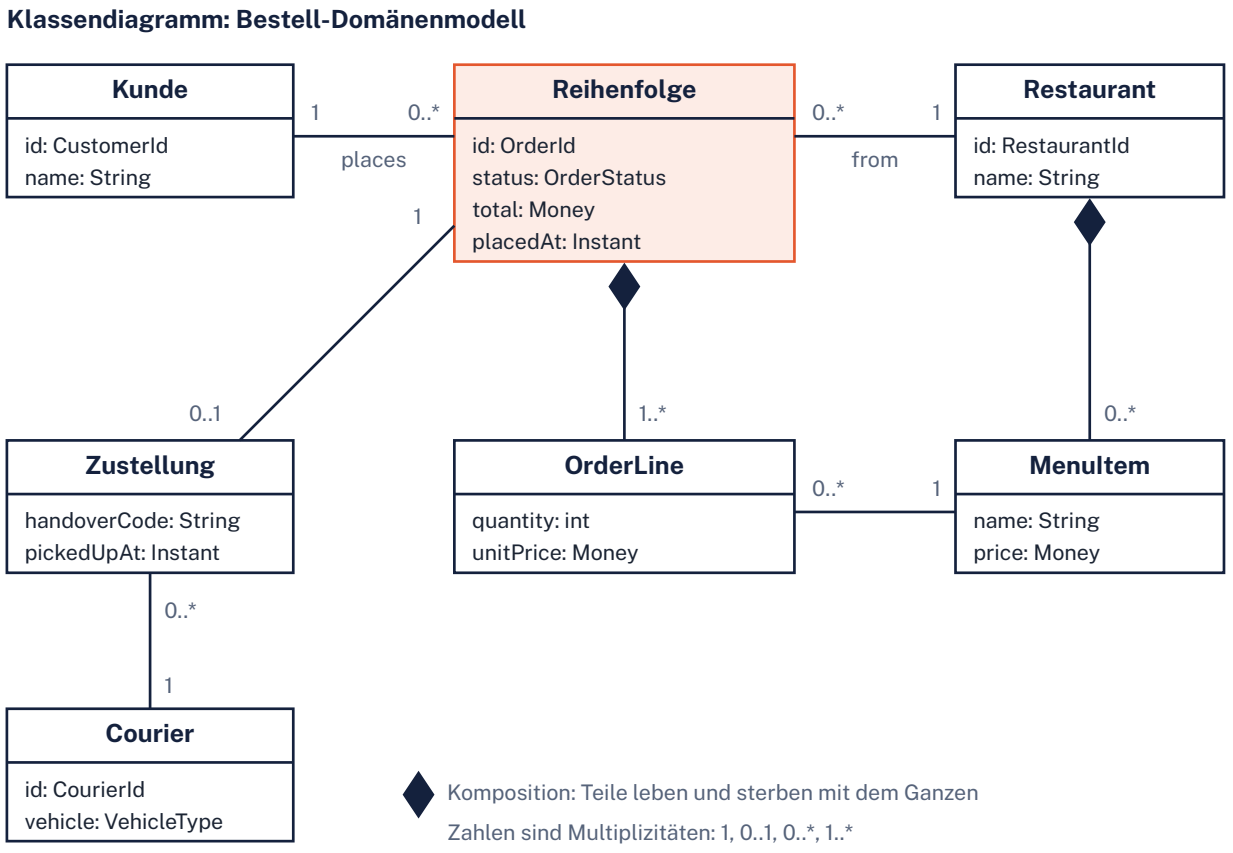
Klassendiagramme, sparsam eingesetzt: das Domänenmodell

Das Klassendiagramm ist das bekannteste UML-Diagramm und dasjenige, das Architekten am sparsamsten einsetzen sollten. Jede Klasse eines Service von Hand zu zeichnen ist der Bauplanmodus in seiner am wenigsten nützlichen Form: Es ist binnen Wochen veraltet, und der Code sagt dasselbe präziser. Wo Klassendiagramme ihren Platz verdienen, ist das

Domänenmodell: ein kleines Diagramm der zentralen Geschäftsbegriffe, ihrer Attribute und Beziehungen, über das Entwicklung, Produktmanagement und Fachexperten gemeinsam sprechen können. In der Begriffswelt des Domain-Driven Design (Kapitel 5) erfasst es die Ubiquitous Language eines Bounded Context.

Das Wesentliche ist überschaubar. Ein Klassenkasten hat einen Namen, Attribute und optional Operationen. Eine einfache Linie ist eine **Assoziation**, optional benannt, mit Multiplizitäten an jedem Ende (`1`, `0..1`, `0..*`, `1..*`). Eine gefüllte Raute markiert eine **Komposition**: Die Teile gehören zu genau einem Ganzen und überleben es nicht, so wie Bestellpositionen zu einer Bestellung gehören. Ein hohles Dreieck markiert eine Generalisierung (Vererbung) und eine gestrichelte Linie mit offener Pfeilspitze eine Abhängigkeit. Multiplizitäten tragen echte Designentscheidungen: `Order 1 to 0..1 Delivery` besagt, dass eine Bestellung höchstens eine Lieferung hat, was ausschließt, eine große Bestellung auf zwei Kuriere aufzuteilen. Falls das Unternehmen das wollen könnte, ist das Diagramm der Ort, an dem die Frage auftaucht.

Klassendiagramm des Domänenmodells



Ein Domänenmodell, klein genug, um es mit dem Fachbereich zu besprechen: zentrale Begriffe, Attribute, Assoziationen und Multiplizitäten.

Die sechs Diagramme auf einen Blick

Diagramm	Beantwortet	Wichtige Notation	Typischer Einsatz durch Architekten
Komponente	Was sind die Teile, und über welche Schnittstellen hängen sie voneinander ab?	Komponentenkästen, bereitgestellte (Ball) und benötigte (Socket) Schnittstellen, Abhängigkeiten	Abhängigkeitsrichtung und austauschbare Teile zeigen; Ports and Adapters
Deployment	Wo läuft jeder Teil, wie viele Kopien, wie verbunden?	Knoten (3D-Kästen), Artefakte, Kommunikationspfade	Produktions- und Staging-Infrastruktur, Netzwerkzonen
Sequenz	In welcher Reihenfolge interagieren die Beteiligten in diesem Szenario?	Lebenslinien, Aktivierungen, synchrone und asynchrone Nachrichten, Antworten, Fragmente	Wichtige Laufzeitszenarien, besonders Fehler, Wiederholungen und Integrationen
Zustandsautomat	In welchen Zuständen kann dieses Objekt sein, und was bewegt es zwischen ihnen?	Zustände, Anfangs- und Endzustände, Transitionen mit event [guard] / action	Lebenszyklen, die sich Services teilen: Bestellungen, Zahlungen, Schadensfälle
Aktivität	Wie sieht der Workflow aus, wer erledigt jeden Schritt, was läuft parallel?	Aktionen, Entscheidungen, Forks und Joins, Partitionen	Geschäftsprozesse, langlaufende Workflows, Sagas
Klasse	Was sind die zentralen Begriffe, und wie hängen sie zusammen?	Klassen, Assoziationen, Multiplizitäten, Komposition, Generalisierung	Kleine Domänenmodelle; aus Code generierte Diagramme

Wann UML zu viel ist und wann Präzision zählt

UML ist zu viel, wenn das Publikum sie nicht kennt und die Frage sie nicht braucht: Ein C4-Kontextdiagramm vermittelt Führungskräften mehr als ein UML-Anwendungsfalldiagramm, und eine Whiteboard-Skizze ist das richtige Werkzeug für eine Designdiskussion, die in einer Stunde vorbei ist. Sie ist zu viel, wenn Diagramme von Hand gepflegt werden, die man generieren könnte, und wenn strikte Notation in einer Skizze erzwungen wird, die dem Erkunden dient. Fowlers Beobachtung, dass die in UML 2 hinzugekommene Präzision vor allem den Nutzern von Bauplan und Programmiersprache dient und die Skizzierenden belastet, gilt nach wie vor.

Präzision zahlt sich aus, wenn Missverständnisse teuer sind. **Schnittstellen zwischen Unternehmen** sind der klarste Fall: Wenn das Essenslieferunternehmen einen Zahlungsanbieter oder das Bestellsystem einer großen Restaurantkette anbindet, baut jede

Seite gegen die Beschreibung der anderen und entdeckt Unstimmigkeiten erst im Test oder in der Produktion. Ein präzises Sequenzdiagramm des Zahlungsablaufs, einschließlich Timeouts, Wiederholungen, Idempotenzschlüsseln und Webhooks, an die Schnittstellenspezifikation angehängt, ist eine günstige Versicherung. **Regulierte und sicherheitskritische Branchen** (Medizinprodukte, Bahn, Luftfahrt, Automobil, manche Finanzsysteme) verlangen oft eine Designdokumentation, die Prüfer von den Anforderungen bis zur Implementierung nachverfolgen können; formale UML oder ihr Verwandter aus dem Systems Engineering, SysML, ist dort verbreitet. **Nebenläufigkeits- und Lebenszyklusregeln**, die sich mehrere Teams teilen, wie die obigen Bestellzustände, profitieren von einer Notation, die jede Transition ausdrücklich verlangt. Und **ausgelagerte oder verteilte Entwicklung**, bei der die Umsetzenden den Autor eines Designs nicht ohne Weiteres fragen können, braucht Beschreibungen, die weniger Raum für Interpretation lassen.

ACHTUNG

Häufige Fehler mit UML

- Falsche Pfeilspitzen in Sequenzdiagrammen, sodass ein asynchrones Ereignis wie ein blockierender Aufruf aussieht oder umgekehrt. Leser ziehen daraus architektonische Schlüsse.
- Klassendiagramme des ganzen Systems von Hand zeichnen. Generieren Sie sie bei Bedarf, und zeichnen Sie nur das Domänenmodell.
- Die Legende weglassen, weil „das Standard-UML ist“. Viele Leser kennen die Notation nicht; eine zweizeilige Legende kostet nichts.
- Ein riesiges Sequenzdiagramm mit jeder Verzweigung. Zeichnen Sie ein Diagramm je Szenario, und machen Sie die Fehlerszenarien zu eigenen Diagrammen.
- Auf vollständiger UML in einer Skizze bestehen, was die Diskussion bremst und Menschen ganz vom Diagrammzeichnen abschreckt.

Häufige Fragen

Sollte unser Team überhaupt UML lernen?
Lernen Sie die Notationsgrundlagen der sechs Diagramme dieses Abschnitts; das dauert ein paar Stunden und gibt dem Team ein gemeinsames Vokabular für Schnittstellen, Interaktionen und Lebenszyklen. Die übrigen Diagrammtypen oder die vollständige Spezifikation müssen Sie kaum lernen, es sei denn, Sie arbeiten in einer Domäne, die sie verlangt.

Wie verhält sich UML zu C4?
Beide ergänzen sich. C4 liefert eine kleine Menge von Abstraktionen und eine Hierarchie struktureller Diagramme; UML liefert präzise Notationen, besonders für Verhalten. Viele Teams nutzen C4 für Kontext-, Container- und Komponentensichten und UML für Sequenz-, Zustands- und gelegentlich Deployment-Diagramme.

Kann ich UML mit Text statt mit einem Zeichenwerkzeug erstellen?
Ja. PlantUML und Mermaid erzeugen beide Sequenz-, Zustands-, Klassen- und Aktivitätsdiagramme aus kurzen Textdateien, die in der Versionsverwaltung liegen können. Abschnitt 4.7 zeigt Beispiele.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Wie unterscheiden Sie in einem Sequenzdiagramm einen synchronen Aufruf von einer asynchronen Nachricht?

Ein synchroner Aufruf hat eine durchgezogene Linie mit gefüllter Pfeilspitze (der Sender wartet); eine asynchrone Nachricht hat eine durchgezogene Linie mit offener Pfeilspitze (der Sender macht weiter). Antworten sind gestrichelte Linien mit offenen Pfeilspitzen.

▼ Was bedeutet das Label `handedOver [code matches] / notify customer` an einer Transition?

Wenn das Ereignis handedOver eintritt und der Guard „code matches“ (der Code stimmt) wahr ist, feuert die Transition, und die Aktion „notify customer“ (Kunden benachrichtigen) wird ausgeführt.

▼ Warum sollten Architekten Klassendiagramme ganzer Services nur selten von Hand zeichnen?

Sie wiederholen, was der Code präziser aussagt, und veralten schnell; kleine Domänenmodelle lohnen das Zeichnen, und vollständige Diagramme lassen sich bei Bedarf generieren.

▼ Nennen Sie zwei Situationen, in denen sich präzise UML lohnt.

Zum Beispiel Schnittstellen zwischen Unternehmen, regulierte oder sicherheitskritische Entwicklung, Lebenszyklen, die sich mehrere Teams teilen, oder ausgelagerte Entwicklung, bei der die Umsetzenden die Designer nicht ohne Weiteres fragen können.

▼ **Probieren Sie es aus: Drei Teams eines Essenslieferdienstes sind sich immer wieder uneinig, wann eine Bestellung storniert werden kann. Welches UML-Diagramm würden Sie zeichnen? Geben Sie einen Beispielübergang in der korrekten Notation an, und sagen Sie, warum sich mehr Präzision hier bezahlt macht.**

Ein Zustandsdiagramm, denn der Streit dreht sich um den Lebenszyklus eines Objekts. Beispiel: Confirmed zu Cancelled bei cancelRequested [Küche hat noch nicht begonnen] / refundPayment, nach dem Muster Ereignis [Bedingung] / Aktion. Ein Sequenzdiagramm würde einen Pfad zeigen, aber nicht alle erlaubten Zustände. Präzision lohnt sich, weil mehrere Teams diesen Lebenszyklus teilen und Fehler echtes Geld kosten. Eine sorgfältige Skizze von einem Dutzend Zuständen schlichtet den Streit, ohne ein vollständiges UML-Modell des Systems zu brauchen.

QUELLEN

- UML specification versions (Object Management Group): https://www.omg.org/spec/UML/
- About the Unified Modeling Language 2.5.1 (OMG): https://www.omg.org/spec/UML/2.5.1/About-UML/
- UmlMode (Martin Fowler): https://martinfowler.com/bliki/UmlMode.html
- UmlAsSketch (Martin Fowler): https://martinfowler.com/bliki/UmlAsSketch.html

Architecture Decision Records

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.8 Wie Software entsteht
- 1.2 Architektur und Design
- 1.6 Entscheidungen und Abwägungen

Begriffe von früher, die er verwendet: Architekturrelevant, Qualitätsmerkmal, Abwägung (Trade-off), Pull Request

DIE IDEE IN KÜRZE

Stellen Sie sich vor, Sie ziehen in ein Haus ein und finden eine Wand, die keinen Zweck zu erfüllen scheint. Reißen Sie sie ein? Hätte der Bauunternehmer einen Zettel hinterlassen: „Diese Wand trägt das Bad im Obergeschoss“, wüssten Sie Bescheid. Ein Architecture Decision Record ist dieser Zettel für Software: eine kurze, datierte Seite, die eine wichtige Entscheidung erklärt, die erwogenen Alternativen nennt und sagt, was die Entscheidung kostet. Die Zettel werden nie umgeschrieben; ändern die Eigentümer ihre Meinung, fügen sie einen neuen Zettel hinzu, der sagt, was sich geändert hat und warum.

WAS SIE WISSEN MÜSSEN

- Ein Architecture Decision Record (ADR) hält eine bedeutsame Entscheidung mit ihrem Kontext, den betrachteten Optionen, der Entscheidung und ihren Konsequenzen fest; Michael Nygard machte die schlanke Form 2011 in einem Blogbeitrag bekannt.
- Nygards Format hat Titel, Kontext, Entscheidung, Status und Konsequenzen; der Kontext wird neutral formuliert, die Entscheidung im Aktiv („Wir werden …“), und die Konsequenzen umfassen Kosten ebenso wie Nutzen.
- MADR (Version 4.0.0, 2024) ergänzt Entscheidungstreiber, betrachtete Optionen mit Vor- und Nachteilen sowie die Bestätigung; ein Y-Statement verdichtet eine Entscheidung zu einem Satz: im Kontext von, angesichts, haben wir uns entschieden für, und gegen, um … zu erreichen, und nehmen in Kauf, dass.
- ADRs wechseln von „vorgeschlagen“ zu „akzeptiert“ oder „abgelehnt“ und später zu „veraltet“ oder „abgelöst“; der Inhalt eines akzeptierten ADR wird nie umgeschrieben, weil ein neues, ablösendes ADR die Historie bewahrt.

- ADRs liegen im Code-Repository als nummerierte Markdown-Dateien (zum Beispiel docs/decisions/0012-title.md), werden in Pull Requests geprüft und in einem Entscheidungsprotokoll indexiert.
- Schreiben Sie ein ADR für Entscheidungen, die teuer rückgängig zu machen sind, zentrale Qualitäten betreffen, andere einschränken oder umstritten waren; vermeiden Sie ADRs, die im Nachhinein geschrieben werden, keine echten Alternativen enthalten oder nur positive Konsequenzen nennen.

Was ein ADR ist und welches Problem es löst

Ein Architecture Decision Record (ADR) ist ein kurzes Textdokument, das eine architekturrelevante Entscheidung festhält: die Situation, die sie erzwungen hat, die abgewogenen Optionen, was gewählt wurde und was aus der Wahl folgt. Eine geordnete Sammlung solcher Records bildet das Entscheidungsprotokoll eines Systems. Die Idee, Entscheidungen festzuhalten, ist alt (die Forschung zu Design Rationale reicht Jahrzehnte zurück, und Jeff Tyree und Art Akerman veröffentlichten 2005 in *IEEE Software* eine ausführliche Entscheidungsvorlage), doch die schlanke Form, die die meisten Teams heute nutzen, stammt aus Michael Nygards Blogbeitrag *Documenting Architecture Decisions* vom 15. November 2011.

Nygards Ausgangspunkt war ein Problem, das jedes langlebige Team kennt. Eine neue Entwicklerin oder ein neuer Entwickler stößt im Code auf eine Entscheidung, die seltsam aussieht: eine zusätzliche Queue, eine doppelte Tabelle, eine Bibliothek, die heute niemand mehr wählen würde. Ohne zu wissen, warum sie getroffen wurde, bleiben der neuen Person zwei schlechte Möglichkeiten. Sie kann sie blind hinnehmen und damit etwas bewahren, dessen Gründe vielleicht nicht mehr gelten, oder sie blind ändern und damit etwas kaputtmachen, dessen Gründe noch gelten. Nygards Gegenmittel bestand darin, die Gründe zum Zeitpunkt der Entscheidung aufzuschreiben, in einer kleinen Datei neben dem Code, damit die Motivation mit dem System mitwandert. Große Architekturdokumente erreichen das selten, weil niemand sie aktualisiert; ein kurzer Record über eine einzige Entscheidung, einmal geschrieben und nie umgeschrieben, bleibt wahr, weil er nur beansprucht, zu beschreiben, was an einem bestimmten Datum bekannt war und entschieden wurde.

Das baut direkt auf der Anatomie einer Entscheidung aus Kapitel 1: Entscheidungen und Abwägungen auf: Kontext, Optionen, Kriterien, Entscheidung mit Begründung, Konsequenzen und Status. Ein ADR ist einfach diese Anatomie, in einem vorhersehbaren Format aufgeschrieben. Es beantwortet auch die Frage, die Architektur und Design in der Praxis offenlässt: Wenn Architektur die Menge der bedeutsamen Entscheidungen ist, dann ist das ADR-Protokoll das, was einer Liste der eigenen Architektur für ein Team am nächsten kommt.

Nygards Format

Nygard schlug fünf Teile vor. Die meisten Teams verwenden sie noch heute und setzen den Status oft an den Anfang, damit Lesende sofort sehen, ob der Record in Kraft ist. Seine Hinweise, wie man jeden Teil schreibt, sind ebenso wichtig wie die Überschriften.

Die fünf Teile eines ADR nach Nygard

Teil	Was er enthält	Wie man ihn gut schreibt
Titel	Eine kurze Nominalphrase, die die Entscheidung benennt, mit einer laufenden Nummer, etwa „ADR 12: Bestellereignisse über einen Message Broker“.	Benennen Sie die Entscheidung, nicht das Problem: Wer das Protokoll überfliegt, sollte schon am Titel erkennen, was entschieden wurde.
Kontext	Die wirkenden Kräfte: das Problem, die relevanten Qualitätsanforderungen und Randbedingungen, technische und organisatorische Fakten und die Spannung zwischen ihnen.	Nygard verlangt wertneutrale Sprache: Beschreiben Sie Fakten und Kräfte, nicht die bevorzugte Antwort. Nennen Sie Zahlen, wo es welche gibt (Spitzenlast, Zielwerte, Daten von Vorfällen).
Entscheidung	Die Antwort auf diese Kräfte.	Ganze Sätze im Aktiv, beginnend mit „Wir werden …“, damit kein Zweifel besteht, was entschieden wurde und von wem.
Status	Vorgeschlagen, akzeptiert und später eventuell veraltet oder abgelöst, mit Verweis auf den ersetzenden Record.	Halten Sie ihn aktuell; er ist das einzige Feld, das sich nach der Annahme noch ändert.
Konsequenzen	Der Kontext, der sich ergibt, nachdem die Entscheidung umgesetzt ist.	Führen Sie alle Konsequenzen auf, nicht nur die positiven: was leichter wird, was schwerer wird, neue Risiken, neue Arbeit und Optionen, die nun verschlossen sind.

Nygard schlug außerdem vor, dass ein Record auf ein oder zwei Seiten passen sollte, dass Records fortlaufend nummeriert und Nummern nie wiederverwendet werden und dass sie als einfacher Text im Projekt-Repository liegen. Jede dieser kleinen Regeln hat einen Grund: Kurze Records werden gelesen, stabile Nummern lassen sich in Code-Kommentaren und Commit-Nachrichten zitieren, und einfacher Text im Repository lässt sich wie Code prüfen und versionieren.

Weitere Vorlagen: MADR, Y-Statements und ausführlichere Formen

Nygards Format sagt nichts ausdrücklich über die verworfenen Optionen, und viele Teams stellten fest, dass gerade die Alternativen das sind, was spätere Lesende am meisten wissen wollen. Mehrere Vorlagen ergänzen sie.

MADR (Markdown Architectural Decision Records, gepflegt auf adr.github.io) ist die am weitesten verbreitete strukturierte Vorlage. Die zum Zeitpunkt der Abfassung aktuelle Version ist 4.0.0, veröffentlicht im September 2024. Die vollständige Vorlage hat einen Titel, dann *Context and Problem Statement*, optional *Decision Drivers*, *Considered Options*, *Decision Outcome* mit den optionalen Unterabschnitten *Consequences* und *Confirmation*, optional *Pros and Cons of the Options* und optional *More Information*. Metadaten wie Status, Datum, Entscheidende sowie wer konsultiert und wer informiert wurde, können in einen YAML-Header. Zwei Ideen aus MADR lohnen sich auch dann, wenn Sie ein anderes Format verwenden. Entscheidungstreiber machen die Kriterien explizit, bevor die Optionen beurteilt werden; das ist die Disziplin der gewichteten Matrix aus Kapitel 1 in kompakter Form. *Confirmation* (Bestätigung) sagt, wie das Team prüfen wird, ob die Entscheidung tatsächlich umgesetzt ist und weiterhin gilt, etwa durch ein Review, einen Test oder eine automatisierte Architekturprüfung.

Das Y-Statement, vorgeschlagen von Olaf Zimmermann und Kollegen und erstmals 2012 auf der SATURN-Konferenz vorgestellt, verdichtet eine Entscheidung zu einem einzigen strukturierten Satz: *Im Kontext von* (einem Anwendungsfall oder einer Komponente), *angesichts* (eines Anliegens oder einer Qualitätsanforderung) *haben wir uns entschieden für* (die Option) *und gegen* (die Alternativen), *um … zu erreichen* (den Nutzen), *und nehmen in Kauf, dass* (die Nachteile). Der Name ist ein Wortspiel mit „why“ („warum“). Ein Y-Statement eignet sich ideal für eine Zusammenfassung im Entscheidungsprotokoll, eine Folie, einen Code-Kommentar oder die Beschreibung eines Pull Requests; für eine umstrittene Einwegtür ist es meist zu knapp, und Teams bauen es zu einem vollständigen Record aus.

Für regulierte oder sehr große Umgebungen gibt es schwerere Vorlagen. Die Vorlage von Tyree und Akerman aus dem Jahr 2005 ergänzt Felder wie Annahmen, Randbedingungen, verwandte Entscheidungen, verwandte Anforderungen und Implikationen. Sie sind nützlich, wenn Entscheidungen für einen Prüfer auf Anforderungen zurückgeführt werden müssen; für die meisten Produktteams bedeuten sie Aufwand, den niemand liest.

Eine Vorlage wählen

Vorlage	Umfang	Hält verworfene Optionen fest?	Am besten für
Nygard (2011)	Eine halbe bis zwei Seiten	Nur wenn Sie sie in Kontext oder Konsequenzen aufnehmen	Teams am Anfang; die kleinste Gewohnheit, die funktioniert.
MADR 4.0.0 (2024)	Eine bis drei Seiten	Ja: betrachtete Optionen, optional mit Vor- und Nachteilen jeder Option	Die meisten Produktteams; umstrittene Entscheidungen, bei denen Alternativen sichtbar sein müssen.
Y-Statement (2012)	Ein Satz	Ja, im Teil *und gegen*	Zusammenfassungen im Entscheidungsprotokoll, kleine Entscheidungen, Beschreibungen von Pull Requests, Folien.
Tyree und Akerman (2005)	Mehrere Seiten	Ja, dazu Annahmen, Randbedingungen und Nachverfolgbarkeit	Regulierte Umgebungen, die Nachverfolgbarkeit bis zu den Anforderungen brauchen.

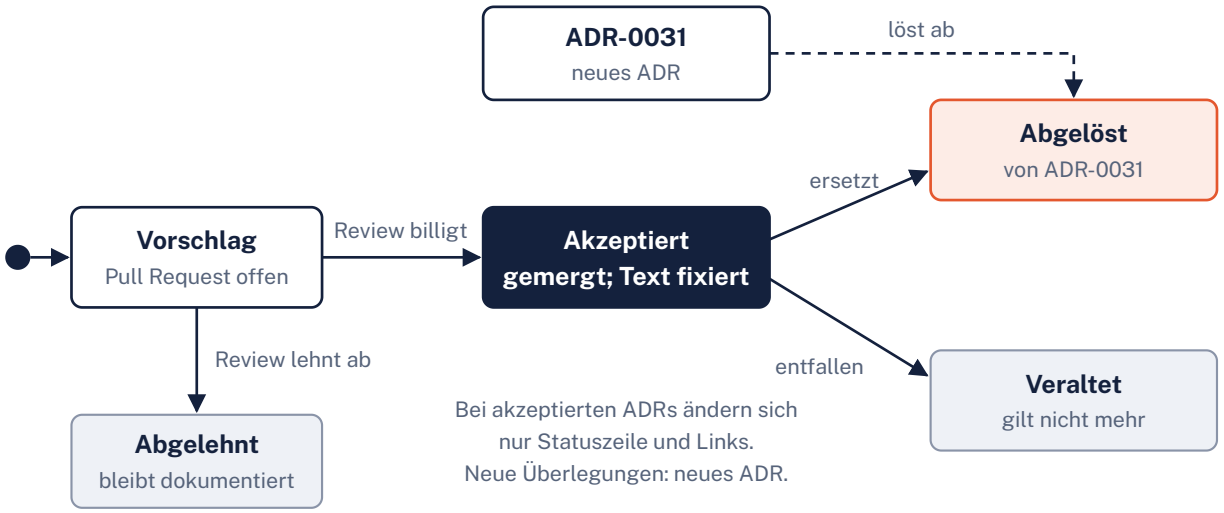
Status, Lebenszyklus und warum akzeptierte Records nicht bearbeitet werden

Ein ADR durchläuft eine kleine Zahl von Zuständen. Es beginnt als **vorgeschlagen**, meist als Pull Request, der die Datei hinzufügt, sodass die Diskussion in Review-Kommentaren stattfindet. Wenn die Verantwortlichen zustimmen, wird es als **akzeptiert** gemergt. Ein Vorschlag, der abgelehnt wird, kann als **abgelehnt** erhalten bleiben, und das ist wertvoll: Wer als Nächstes dieselbe Idee vorschlägt, findet die frühere Begründung. Später kann ein akzeptierter Record **veraltet** sein (die Entscheidung gilt nicht mehr, etwa weil die Komponente stillgelegt wurde) oder durch einen neueren Record, der ihn ersetzt, **abgelöst** werden, mit Links in beide Richtungen. MADR nennt dieselben Zustände: proposed, rejected, accepted, deprecated und superseded.

Die zentrale Regel lautet: Der Inhalt eines akzeptierten Records wird nicht umgeschrieben. Ändert das Team seine Meinung, schreibt es ein neues ADR, das das alte ablöst, und ändert am alten Record nur die Statuszeile, die nun nach vorn verweist. Die Gründe sind praktischer Natur. Erstens ist der Record ein Beleg dafür, was zu einem Zeitpunkt bekannt war und entschieden wurde; ihn im Nachhinein zu bearbeiten, macht aus Geschichte Fiktion und zerstört das Vertrauen, das das Protokoll nützlich macht. Zweitens erklärt der alte Record, warum das System eine Zeit lang so aussah, wie es aussah, und das zählt, wenn man alte Daten debuggt oder alten Code liest. Drittens zwingt ein neuer Record das Team, zu benennen, was sich am

Kontext geändert hat, und das ist oft die wichtigste Erkenntnis. Kleine Korrekturen (ein Tippfehler, ein defekter Link) sind harmlos; die Entscheidung, die Gründe oder die Konsequenzen zu ändern, ist es nicht. Martin Fowlers Beschreibung der Praxis sagt dasselbe: Ein akzeptierter Record wird nicht wieder geöffnet; stattdessen wird ein ablösender Record geschrieben.

Der Lebenszyklus eines ADR



Records werden vorgeschlagen und geprüft, dann akzeptiert oder abgelehnt; akzeptierte Records veralten später oder werden abgelöst, aber nie umgeschrieben.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: die Meinung ändern, ohne die Geschichte umzuschreiben

Die Idee, die hier die meiste Sorgfalt verlangt, ist, warum ein akzeptierter Record nie umgeschrieben wird, auch wenn das Team seine Meinung ändert. Kapitel 1 hat die Regel aufgestellt; verfolgen Sie eine echte Änderung im Entscheidungsprotokoll der Essenslieferplattform, um zu sehen, was sie in der Praxis bedeutet.

- 2025 akzeptiert das Team ADR-0009: Der Benachrichtigungsservice versendet Push-Benachrichtigungen über ein SDK eines Drittanbieters. Kontext, Entscheidung und Konsequenzen beschreiben, was das Team zu diesem Zeitpunkt weiß und wählt.
- 2026, nachdem mit ADR-0012 ein Message Broker eingeführt wurde, beschließt das Team, das SDK durch eine eigene Push-Zustellung zu ersetzen, die auf dem Broker läuft.
- Die verlockende Abkürzung wäre, ADR-0009 zu öffnen und so umzuschreiben, dass es den neuen Entwurf beschreibt. Dann aber behauptet das Protokoll, das Team habe sich schon 2025 für die eigene Push-Zustellung entschieden, und wer alten Code oder alte Benachrichtigungsdaten liest, findet keinen Hinweis darauf, dass das SDK je verwendet wurde, oder warum.
- Stattdessen schreibt das Team einen neuen Record, ADR-0015, und schlägt ihn wie jeden anderen in einem Pull Request vor. Sein Kontext sagt, was sich seit ADR-0009 geändert hat, und das ist oft der nützlichste Teil des Records.
- Wenn ADR-0015 akzeptiert ist, ändert sich an ADR-0009 nur die Statuszeile, die nun ‚Abgelöst durch ADR-0015' lautet. ADR-0015 verlinkt zurück, und das Entscheidungsprotokoll führt beide auf.
- Wer neu dazukommt und das Protokoll liest, sieht nun die ganze Geschichte: was 2025 entschieden wurde, was es 2026 ersetzt hat und warum.
- **Die Lehre:** Ein akzeptierter Record beschreibt, was an seinem Datum bekannt war und entschieden wurde, und deshalb bleibt er wahr. Neue Überlegungen kommen in einen neuen Record; abgesehen von kleinen Korrekturen wie einem Tippfehler ändert sich am alten nur die Statuszeile.

Wo man ADRs aufbewahrt

Bewahren Sie ADRs im Repository des Codes auf, für den sie gelten, als Textdateien (meist Markdown), eine Entscheidung pro Datei, mit laufender Nummer und kurzem Slug im Dateinamen. Zwei Verzeichniskonventionen sind verbreitet: `doc/adr`, der Standard von Nat Pryces Kommandozeilenwerkzeug `adr-tools`, und `docs/decisions`, von MADR empfohlen, mit Namen wie `0012-publish-order-events-through-broker.md`. Thoughtworks setzte

schlanke ADRs in den *Adopt*-Ring seines Technology Radar (zuletzt 2018) und empfahl ausdrücklich die Versionsverwaltung statt eines Wikis, damit die Records mit dem Code, den sie beschreiben, Schritt halten.

Records neben dem Code aufzubewahren, hat konkrete Vorteile. Sie werden in denselben Pull Requests geprüft wie der Code, der sie umsetzt, sie werden mit ihm versioniert und verzweigt, man findet sie mit denselben Suchwerkzeugen, und Code kann auf sie verweisen: Ein Kommentar wie „siehe ADR-0012“ über einem Outbox-Relay sagt dem nächsten Leser, wo die Gründe stehen. Entscheidungen, die viele Repositorys betreffen, etwa die organisationsweite Wahl eines Identity Providers, brauchen ein eigenes Zuhause; ein eigenes Architektur-Repository mit denselben Konventionen funktioniert gut, mit Links aus jeder betroffenen Codebasis.

```
docs/
  decisions/
    README.md                                  ← the decision log (index)
    0001-record-architecture-decisions.md
    0004-use-hosted-card-fields.md
    0007-use-managed-postgresql-for-orders.md
    0012-publish-order-events-through-broker.md
    template.md
src/
  ordering/
    outbox_relay.py      # comment: see docs/decisions/0012-...
```

Eine typische Struktur: nummerierte ADR-Dateien neben dem Code, ein Index, eine Vorlage und ein Code-Kommentar, der auf den Record zurückverweist

Welche Entscheidungen ein ADR verdienen

Ein ADR für jede Wahl zu schreiben, begräbt die wichtigen; gar keines zu schreiben, verliert die Gründe. Der Maßstab ist die Bedeutsamkeit, wie in Kapitel 1 definiert: Eine Entscheidung verdient einen Record, wenn sie teuer rückgängig zu machen ist, ein zentrales Qualitätsmerkmal betrifft, eine große Reichweite hat, andere Teams einschränkt oder umstritten war. In der Praxis sind das Entscheidungen wie der primäre Datenspeicher, der Kommunikationsstil zwischen Services, ein Identity Provider, eine Richtlinie zur API-Versionierung, eine Hosting-Plattform, eine Make-or-Buy-Entscheidung und Konventionen, deren Verletzung teuer ist (wie Geldbeträge dargestellt, wie Zeitzonen gespeichert werden). Dazu gehören auch Entscheidungen, etwas *nicht* zu tun, das immer wieder vorgeschlagen wird, etwa vorerst keinen Cache einzuführen.

Entscheidungen, die lokal sind, billig rückgängig zu machen und außerhalb eines Moduls unsichtbar, brauchen meist keinen Record: die Wahl einer Assertion-Bibliothek für Tests, der Name einer Klasse, der interne Aufbau einer Komponente. Ein nützliches Signal ist die Frage,

die jemand Neues stellen würde: Wenn innerhalb eines Jahres vermutlich jemand fragt „Warum um Himmels willen haben die das so gemacht?“, schreiben Sie den Record jetzt, solange die Gründe frisch sind.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: Braucht diese Entscheidung ein ADR?

Stellen Sie diese Fragen der Reihe nach, wenn in einer Designdiskussion oder einem Code-Review eine technische Entscheidung ansteht.

Q1 **Würde es in einem Jahr mehr als ein paar Tage Arbeit, eine Datenmigration oder Änderungen anderer Teams kosten, diese Entscheidung rückgängig zu machen?**

- **Ja** Schreiben Sie ein ADR. Es ist eine Einwegtür oder kommt ihr nahe.
- **Nein** Weiter zur nächsten Frage.

Q2 **Beeinflusst sie spürbar ein Qualitätsmerkmal, das den Stakeholdern wichtig ist (zum Beispiel Verfügbarkeit, Latenz, Sicherheit oder Kosten), oder eine Randbedingung wie Regulierung?**

- **Ja** Schreiben Sie ein ADR und verknüpfen Sie es mit dem Qualitätsszenario, dem es dient.
- **Nein** Weiter zur nächsten Frage.

Q3 **War sie umstritten, oder hat das Team eine Option verworfen, die vermutlich erneut vorgeschlagen wird?**

- **Ja** Schreiben Sie ein kurzes ADR (ein Y-Statement kann genügen), damit die Begründung nicht von Grund auf neu diskutiert wird.
- **Nein** Weiter zur nächsten Frage.

Q4 **Wird jemand Neues wahrscheinlich auf das Ergebnis schauen und fragen, warum es so gemacht wurde?**

- **Ja** Schreiben Sie ein kurzes ADR oder zumindest einen Code-Kommentar und eine Commit-Nachricht, die den Grund erklären.
- **Nein** Kein ADR nötig. Eine klare Commit-Nachricht und der Code selbst genügen.

Ein vollständiges ausgearbeitetes ADR

Die Essenslieferplattform, die sich durch dieses Kapitel zieht, hat Apps für Kunden, Restaurants und Kuriere, einen Bestellservice, Zahlungen über einen externen Zahlungsanbieter, Kurierdisposition und Benachrichtigungen. Der folgende Record folgt der MADR-Struktur, leicht gekürzt. Achten Sie darauf, wie der Kontext Zahlen und Daten statt

Adjektiven nennt, wie die Treiber vor den Optionen stehen, wie die verworfenen Optionen fair angehört werden und wie die Konsequenzen echte Kosten enthalten. Die Entscheidung selbst (asynchrones Messaging für Bestellereignisse) wird in Kapitel 6: ereignisgesteuerte Architektur gründlich behandelt; hier geht es darum, wie sie festgehalten wird.

```
# ADR-0012: Publish order events through a managed message broker

- Status: Accepted
- Date: 2026-09-14
- Deciders: ordering team lead, platform lead, principal architect
- Consulted: restaurant, courier and notifications teams; SRE
- Related: ADR-0007 (managed PostgreSQL for the order service)

## Context and problem statement

When a customer's payment succeeds, the ordering service calls the restaurant,
courier-dispatch and notification services synchronously over HTTP before it
confirms the order. On Friday evenings (peak about 40 orders/s) p99 order
confirmation time reaches 4.2 s against our target of 1.5 s (scenario QS-3).
On Friday 21 August a notification-service outage blocked all new orders for
23 minutes. Two new consumers (loyalty points, analytics) want order data.

## Decision drivers

- Order confirmation must not depend on notification or dispatch availability.
- p99 confirmation under 1.5 s at three times today's peak.
- No order event may be lost, even if a service crashes mid-request.
- New consumers should not require changes to the ordering service.
- The platform team has three people; operational load must stay small.

## Considered options

1. Keep synchronous calls; add timeouts, retries and circuit breakers.
2. Publish events to a managed message broker using a transactional outbox.
3. Let consumers poll a shared "order_events" table in the order database.

## Decision outcome

Chosen option: 2. We will publish OrderPlaced, OrderAccepted, OrderPickedUp,
OrderDelivered and OrderCancelled events to our cloud provider's managed
broker. The ordering service writes each event to an outbox table in the
same database transaction as the order; a relay publishes it. Payment
authorisation stays synchronous, because the customer must see the result.

### Consequences

- Good: confirmation no longer waits for downstream services.
- Good: loyalty and analytics can subscribe without ordering-team work.
- Bad: services become eventually consistent; the restaurant tablet may
  show a new order a second or two after the customer's confirmation.
- Bad: a broker to pay for and monitor; consumers must be idempotent
  because delivery is at-least-once.
- Neutral: event schemas become public contracts and need versioning.

### Confirmation
```

```
Load test at 120 orders/s in staging before go-live; a build check fails
if the ordering service gains an HTTP client for notification or dispatch.

## Pros and cons of the other options

- Option 1: least new infrastructure, but a slow dependency still delays
  every order, and each new consumer means another call to add.
- Option 3: no new infrastructure, but couples every consumer to our schema
  and adds polling load to the order database (see ADR-0007).
```

Markdown-Quelltext von ADR-0012 in docs/decisions/0012-publish-order-events-through-broker.md

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: dieselbe Entscheidung als Y-Statement

Für das Entscheidungsprotokoll, die Beschreibung eines Pull Requests oder eine Folie schreibt das Team die Entscheidung zusätzlich als ein Y-Statement.

Y-Statement	Im Kontext der **Bestellbestätigung auf der Essenslieferplattform**, angesichts **langsamer oder nicht verfügbarer nachgelagerter Services, die Bestellungen zu Spitzenzeiten blockieren**, haben wir uns entschieden für **die Veröffentlichung von Bestellereignissen über einen verwalteten Message Broker mit Transactional Outbox** und gegen **synchrone Aufrufe mit Circuit Breakern sowie Consumer, die eine gemeinsame Tabelle abfragen**, um **eine Bestellbestätigung, die nicht von Benachrichtigungs-, Dispositions- oder Restaurantservices abhängt, und neue Consumer ohne Änderungen am Bestellservice** zu erreichen, und nehmen in Kauf, dass **die Services nur noch Eventual Consistency bieten, der Broker betrieben und bezahlt werden muss und jeder Consumer doppelte Ereignisse verarbeiten muss**.
Was es bewahrt	Den Kontext, das treibende Anliegen, die Wahl, die verworfenen Optionen, den Nutzen und die in Kauf genommenen Kosten: jedes Element des vollständigen Records, auf einen Blick.
Was es verliert	Die Zahlen (p99 von 4,2 s, 23-minütiger Ausfall), die Entscheidenden, den Bestätigungsplan und die Gründe gegen jede verworfene Option. Für eine so bedeutsame Entscheidung bleibt der vollständige Record nötig.

Ein guter Test für jedes ADR ist, ob es sich zu einem sinnvollen Y-Statement verdichten lässt. Wenn Sie die Teile *und gegen* oder *und nehmen in Kauf, dass* nicht füllen können, hat der Record vermutlich keine echten Alternativen betrachtet oder seine Kosten verschwiegen.

ADRs prüfen

Weil ein ADR eine Datei im Repository ist, ist der Pull Request der natürliche Review-Mechanismus. Wer es vorschlägt, öffnet ihn mit dem Status *vorgeschlagen*; die Betroffenen kommentieren; der Record wird überarbeitet; wenn die Verantwortlichen zustimmen, wechselt der Status zu *akzeptiert*, und er wird gemergt. So bleibt die Diskussion mit dem Record verbunden und für alle sichtbar, die später die Historie lesen.

Wer prüfen muss, hängt davon ab, wie Entscheidungen gesteuert werden. In einem zentralisierten Modell genehmigt ein Architekturgremium bedeutsame Records. Im Advice Process, den Andrew Harmel-Law beschrieben hat (martinfowler.com, 2021), darf jede Person eine Architekturentscheidung treffen, sofern sie vorher den Rat aller wesentlich Betroffenen und von Personen mit einschlägiger Expertise einholt und diesen Rat festhält; wer entscheidet, muss dem Rat nicht folgen, ihn aber anhören. ADRs sind das Rückgrat dieses Prozesses: Der Rat steht im Record, und ein regelmäßiges, offenes Beratungsforum prüft die vorgeschlagenen Records. Governance-Modelle behandelt Kapitel 11: Governance in agilen Organisationen.

Olaf Zimmermanns Checkliste *ecADR* (2020) gibt Prüfenden eine kompakte Definition of Done für eine Entscheidung. **E**vidence (Evidenz): Es gibt Grund zur Annahme, dass die Option funktioniert, aus einem Spike, einem Proof of Concept oder bewährter Erfahrung. **C**riteria (Kriterien): Mindestens zwei Optionen wurden anhand der Anliegen der Stakeholder verglichen. **A**greement (Einigung): Die Entscheidung wurde von Fachkollegen hinterfragt, und das Team trägt sie mit. **D**ocumentation (Dokumentation): Sie ist festgehalten und mit den Betroffenen geteilt. **R**ealisation and review (Umsetzung und Überprüfung): Die Umsetzung ist geplant, und ein Datum oder Auslöser für die erneute Prüfung ist festgelegt. Wer diese fünf Punkte prüft und dazu die Ehrlichkeit der Konsequenzen, erkennt die meisten schwachen Records.

Anti-Patterns

ADRs scheitern auf erkennbare Weise. Zimmermann hat 2023 mehrere solche Muster mit einprägsamen Namen katalogisiert, und die schädlichsten sollte man kennen.

ACHTUNG

Wie ADRs schiefgehen

- **Im Nachhinein zur Rechtfertigung geschrieben.** Die Entscheidung fiel auf dem Flur, und der Record wird nachträglich so konstruiert, dass sie rational wirkt. Seine Optionen sind erfunden, sein Kontext ist selektiv. Schreiben Sie den Record während der Entscheidung, als Vorschlag, nicht hinterher als Pressemitteilung.
- **Keine Optionen betrachtet** (Zimmermanns *Sprint*) oder eine *Dummy Alternative*: eine Strohmann-Option, die nie funktionieren könnte und nur hinzugefügt wurde, damit der Favorit geprüft wirkt. Wenn es keine glaubwürdige Alternative gibt, sagen Sie das und warum.
- **Keine echten Konsequenzen** (der *Free Lunch Coupon*) oder nur Vorteile (das *Fairy Tale*). Jede bedeutsame Entscheidung hat Kosten; ein Record, der sie verschweigt, fliegt auf, und die Lesenden verlieren das Vertrauen in das Protokoll.
- **Das Mega-ADR.** Diagramme, Komponentenspezifikationen und Code, in einen mehrseitigen Record gestopft. Verlinken Sie stattdessen die Architekturdokumentation; das ADR hält die Entscheidung und ihre Gründe fest.
- **Akzeptierte Records bearbeiten**, damit sie zum aktuellen Entwurf passen; das zerstört genau die Historie, die das Protokoll bewahren soll.
- **ADRs für alles,** auch für triviale Zweiwegtüren, bis niemand sie mehr liest; oder ADRs, die niemand findet, verstreut über Wikis, Chat-Threads und Foliensätze.

Das Entscheidungsprotokoll als Index

Einzelne Records sind nützlich; das Protokoll als Ganzes ist nützlicher. Eine kurze Indexdatei (oft die `README.md` des Entscheidungsordners) führt jeden Record mit Nummer, Titel, Status und Datum auf. Wer neu ist, liest sie in fünf Minuten und erhält eine komprimierte Geschichte der Architektur: was entschieden, was verworfen, was ersetzt wurde und wann. Werkzeuge wie `adr-tools` können den Index erzeugen und ablösende Records anlegen, die den Status des alten Records automatisch aktualisieren, und Dokumentationswerkzeuge wie die Structurizr DSL können einen Ordner mit ADRs in die veröffentlichte Dokumentation importieren.

Behandeln Sie das Protokoll als Teil der Architekturbeschreibung. Die arc42-Vorlage hat einen Abschnitt für Architekturentscheidungen, der einfach darauf verlinken kann, und eine Docs-as-Code-Pipeline kann es zusammen mit der übrigen Dokumentation veröffentlichen.

```
# Decision log

| ADR  | Title                                                | Status                | Date       |
|------|------------------------------------------------------|-----------------------|------------|
| 0001 | Record architecture decisions                        | Accepted              | 2025-03-02 |
| 0004 | Use the payment provider's hosted card fields        | Accepted              | 2025-04-18 |
| 0007 | Use managed PostgreSQL for the order service         | Accepted              | 2025-06-11 |
| 0009 | Send push notifications through a third-party SDK    | Superseded by 0015    | 2025-09-30 |
| 0012 | Publish order events through a managed broker        | Accepted              | 2026-09-14 |
| 0013 | Store courier locations in a time-series database    | Rejected              | 2026-09-20 |
| 0015 | Replace the SDK with our own push delivery on broker | Accepted              | 2026-10-08 |
```

docs/decisions/README.md: das Entscheidungsprotokoll der Essenslieferplattform

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum sollte der Abschnitt „Kontext“ eines ADR nicht für die gewählte Option argumentieren?

Weil seine Aufgabe darin besteht, die Kräfte und Fakten so festzuhalten, wie sie waren, damit spätere Lesende beurteilen können, ob sie noch gelten. Ist der Kontext als Plädoyer geschrieben, kann man die Situation nicht von der Schlussfolgerung trennen.

▼ Ihr Team ändert seine Meinung zu einer akzeptierten Entscheidung. Was machen Sie mit dem alten ADR?

Seinen Inhalt unverändert lassen, seinen Status mit einem Link auf den neuen Record auf „abgelöst“ setzen und ein neues ADR schreiben, das erklärt, was sich am Kontext geändert hat und was nun entschieden ist.

▼ Was ergänzt der MADR-Abschnitt „Confirmation“, das Nygards Format fehlt?

Eine Aussage darüber, wie das Team prüfen wird, ob die Entscheidung umgesetzt ist und weiterhin gilt, etwa durch einen Lasttest, ein Review oder eine automatisierte Prüfung der Abhängigkeiten.

▼ **Woran lässt sich schnell prüfen, ob ein ADR wirklich Alternativen und Kosten betrachtet hat?**

Versuchen Sie, es als Y-Statement zu schreiben. Lassen sich die Teile „und gegen“ oder „und nehmen in Kauf, dass“ nicht ehrlich füllen, fehlen dem Record echte Optionen oder echte Konsequenzen.

▼ **Probieren Sie es aus: Kritisieren Sie dieses ADR: „Wir haben Kafka gewählt, weil es der beste Broker ist. Konsequenzen: schnell und skalierbar.“ Sechs Monate später ändert jemand den Text in RabbitMQ. Was ist falsch, und was hätte jeweils getan werden müssen?**

Es nennt keinen Kontext, keine Alternativen und keine Kriterien, und „am besten“ ist keine Begründung. Die Konsequenzen führen nur Vorteile auf, keine Kosten wie den Betrieb des Clusters. Es sollte das Problem mit Zahlen darstellen, mindestens zwei Optionen fair vergleichen, dann „Wir werden …“ sagen und benennen, worauf verzichtet wird. Spätere Änderungen zerstören die Historie: Schreiben Sie ein neues ADR, das es ablöst, und markieren Sie das alte als abgelöst.

QUELLEN

- Documenting Architecture Decisions, Michael Nygard (2011): https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions
- MADR: Markdown Architectural Decision Records: https://adr.github.io/madr/
- Architecture Decision Record, Martin Fowler (bliki): https://martinfowler.com/bliki/ArchitectureDecisionRecord.html
- Scaling the Practice of Architecture, Conversationally, Andrew Harmel-Law: https://martinfowler.com/articles/scaling-architecture-conversationally.html
- A Definition of Done for Architectural Decision Making, Olaf Zimmermann: https://ozimmer.ch/practices/2020/05/22/ADDefinitionOfDone.html
- Jeff Tyree and Art Akerman: Architecture Decisions: Demystifying Architecture (IEEE Software, 2005): https://doi.org/10.1109/MS.2005.27

arc42 und andere Dokumentationsvorlagen

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 4.2 Sichten und Blickwinkel
- 4.3 Das C4-Modell
- 4.5 Entscheidungsdokumentation

Begriffe von früher, die er verwendet: Architekturbeschreibung, Blickwinkel (Viewpoint), Korrespondenz, Qualitätsszenario, Architecture Decision Record (ADR)

DIE IDEE IN KÜRZE

Eine Vorlage ist wie der Standardaufbau eines Autohandbuchs: Jedes Handbuch hat einen Abschnitt zu Reifen, einen zu Warnleuchten, einen zur Wartung, sodass Sie in jedem Auto wissen, wo Sie nachschlagen müssen. arc42 leistet dasselbe für Softwarearchitektur. Es gibt zwölf Standardabschnitte vor, etwa Ziele, Kontext, Struktur und Entscheidungen, damit Schreibende wissen, was sie abdecken müssen, und Lesende wissen, wo sie es finden. Sie müssen nicht jeden Abschnitt füllen; ein kleines System braucht vielleicht nur vier oder fünf kurze.

WAS SIE WISSEN MÜSSEN

- arc42, entwickelt von Gernot Starke und Peter Hruschka und frei unter einer Creative-Commons-Lizenz verfügbar, ist eine Vorlage aus zwölf Abschnitten; sie legt fest, wohin Informationen gehören, nicht, wie man entwirft oder welche Notation man verwendet.
- Die Abschnitte sind Einführung und Ziele, Randbedingungen, Kontextabgrenzung, Lösungsstrategie, Bausteinsicht, Laufzeitsicht, Verteilungssicht, Querschnittliche Konzepte, Architekturentscheidungen, Qualitätsanforderungen, Risiken und technische Schulden sowie Glossar.
- Jeder Abschnitt ist optional; eine minimale Beschreibung umfasst drei bis fünf Qualitätsszenarien, den Kontext, eine kurze Lösungsstrategie, die Bausteinsicht der Ebene 1 und die wichtigsten querschnittlichen Konzepte.
- Nutzen Sie arc42 sparsam: Verlinken Sie Anforderungen, ADRs und Infrastrukturcode, statt sie zu kopieren, und halten Sie bewusst fest, wenn ein Abschnitt nicht benötigt wird.

- Die SEI-Vorlage Views and Beyond ist gründlicher, mit einer Roadmap, Paketen pro Sicht und einer Zuordnung zwischen den Sichten; IEEE 1016-2009 legt Software Design Descriptions fest und hat seit 2020 den Status „inactive-reserved“.
- ISO/IEC/IEEE 42010 schreibt den Inhalt einer Architekturbeschreibung vor, nicht eine Vorlage; eine arc42-Beschreibung ist konform, sobald Stakeholder, Anliegen, Blickwinkel, Sichten, Korrespondenzen und Begründungen explizit gemacht sind.

Was arc42 ist

arc42 ist eine kostenlose Vorlage zur Beschreibung und Kommunikation von Softwarearchitektur, entwickelt von Gernot Starke und Peter Hruschka und seit 2005 in der Praxis im Einsatz. Sie ist unter einer Creative-Commons-Lizenz (Namensnennung – Weitergabe unter gleichen Bedingungen) veröffentlicht, in mehr als einem Dutzend Sprachen und in Formaten wie AsciiDoc, Markdown und Word, mit einer „leeren“ Fassung und einer Fassung mit eingebauten Hilfetexten. Besonders verbreitet ist sie im deutschsprachigen Raum und in der iSAQB-Ausbildung, sie wird aber weltweit genutzt.

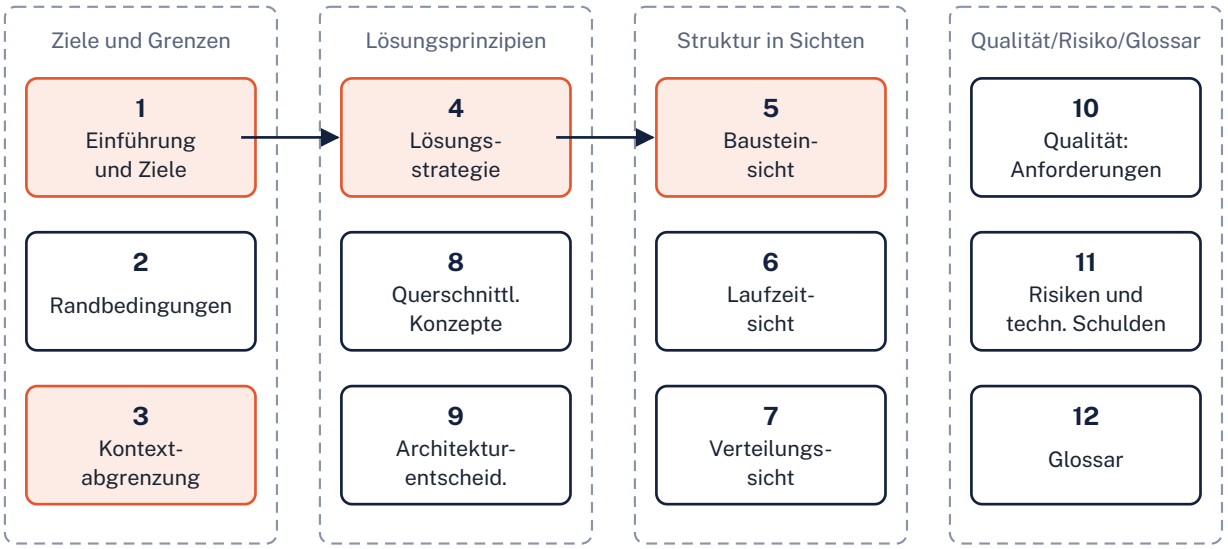
arc42 ist bewusst weder eine Methode noch eine Notation. Es sagt Ihnen nicht, wie Sie entwerfen sollen, und schreibt weder UML noch C4 noch einen anderen Diagrammstil vor. Es ist ein Satz von zwölf beschrifteten Schubladen: Für jede Art von Architekturinformation sagt es, wohin sie gehört, damit alle Lesenden wissen, wo sie suchen müssen, und alle Schreibenden wissen, was fehlt. Die Autoren sagen ausdrücklich, dass jeder Abschnitt optional ist; ein kleines System füllt vielleicht vier Abschnitte auf wenigen Seiten, ein großes alle zwölf auf vielen.

Die Vorlage baut auf Ideen aus diesem Kapitel auf. Ihre drei Sichten (Baustein-, Laufzeit- und Verteilungssicht) entsprechen weitgehend den SEI-Kategorien Modul, Komponenten und Konnektoren sowie Zuordnung und Kruchtens Entwicklungs-, Prozess- und physischer Sicht aus Sichten und Blickwinkel. In ihren Entscheidungsabschnitt gehören die Architecture Decision Records, in ihren Qualitätsabschnitt die Qualitätsszenarien aus Kapitel 2.

Die zwölf Abschnitte

Das Diagramm gruppiert die Abschnitte nach der Frage, die sie beantworten. Die Abschnitte 1 bis 3 legen Ziele und Grenzen fest: wofür das System da ist, was die Lösung einschränkt und wo das System endet. Abschnitt 4 nennt die grundlegenden Ideen der Lösung, und die Abschnitte 8 und 9 enthalten die Prinzipien und Entscheidungen, die für die ganze Lösung gelten. Die Abschnitte 5 bis 7 zeigen die Struktur aus drei Blickrichtungen. Die Abschnitte 10 bis 12 machen Qualität, Risiken und Vokabular explizit.

Der Aufbau von arc42



Schattiert: der Kern auch einer minimalen Beschreibung (Ziele, Kontext, Strategie, Bausteine der Ebene 1); die arc42-FAQ verlangt außerdem 3 bis 5 Qualitätsszenarien und Kernkonzepte (8).

Die zwölf arc42-Abschnitte, nach Zweck gruppiert; die hervorgehobenen Abschnitte bilden den Kern auch einer minimalen Beschreibung.

Was in jeden Abschnitt gehört, und wie viel

Abschnitt	Was hineingehört	Kleines System (ein Team)	Großes System (viele Teams)
1. Einführung und Ziele	Ein kurzer Überblick über die Anforderungen, die drei bis fünf wichtigsten Qualitätsziele nach Priorität und eine Stakeholder-Tabelle mit den Erwartungen jeder Gruppe.	Eine halbe Seite: Zweck, drei Qualitätsziele, fünf oder sechs Stakeholder.	Zwei oder drei Seiten, die auf die Anforderungsdokumente verlinken, statt sie zu kopieren.
2. Randbedingungen	Alles, was die Entwurfsfreiheit einschränkt: Vorschriften, vorgegebene Technologien, organisatorische Regeln, Termine, Budget, Konventionen.	Eine Aufzählung.	Eine Tabelle, gegliedert in technische, organisatorische und rechtliche Randbedingungen, jeweils mit Quelle.
3. Kontextabgrenzung	Das System als Blackbox: Nutzer und Nachbarsysteme, was zwischen ihnen fließt (fachlicher Kontext) und die verwendeten Kanäle und Protokolle (technischer Kontext).	Ein Kontextdiagramm (oft ein C4-Systemkontextdiagramm) und eine kurze Tabelle der externen Schnittstellen.	Fachliche und technische Kontextdiagramme sowie eine Schnittstellentabelle mit Verantwortlichen und Verträgen.
4. Lösungsstrategie	Die wenigen grundlegenden Entscheidungen und Ansätze, die alles andere prägen: Zerlegungsstil, Schlüsseltechnologien, wie die wichtigsten Qualitätsziele erreicht werden.	Eine Handvoll Stichpunkte, jeder an ein Qualitätsziel gebunden.	Ein oder zwei Seiten; eine Tabelle, die jedem Qualitätsziel den Ansatz zuordnet, der ihm dient.
5. Bausteinsicht	Die statische Zerlegung in Bausteine (Subsysteme, Services, Komponenten, Module), Ebene für Ebene verfeinert, jeder mit Verantwortlichkeiten und Schnittstellen. Meist der umfangreichste Abschnitt.	Nur Ebene 1: ein Diagramm (ein C4-Containerdiagramm eignet sich) und ein Satz pro Baustein.	Ebene 1 für das ganze System, Ebenen 2 und 3 nur für komplexe oder riskante Teile.
6. Laufzeitsicht	Wie Bausteine in wichtigen Szenarien zusammenarbeiten: zentrale Anwendungsfälle, Start, Fehlerbehandlung, Failover.	Ein oder zwei Sequenzdiagramme für die wichtigsten Abläufe.	Die kritischen und die überraschenden Szenarien; niemals jeder Anwendungsfall.

Abschnitt	Was hineingehört	Kleines System (ein Team)	Großes System (viele Teams)
7. Verteilungssicht	Die technische Infrastruktur: Umgebungen, Knoten, Netzwerke und welche Bausteine wo laufen.	Ein Diagramm oder eine kurze Liste der gehosteten Services.	Diagramme pro Umgebung und Region, verlinkt mit dem Infrastructure as Code, das sie definiert.
8. Querschnittliche Konzepte	Konzepte, die für viele Bausteine gelten: Domänenmodell, Sicherheit, Fehlerbehandlung, Logging und Observability, Persistenz, Internationalisierung, Tests.	Je ein kurzer Absatz für die zwei oder drei Konzepte, auf die es ankommt.	Eine Seite oder mehr pro Konzept, oft verantwortet von einem Plattform- oder Enabling Team.
9. Architekturentscheidungen	Wichtige Entscheidungen mit ihren Gründen, sofern nicht anderswo festgehalten.	Ein Link auf den ADR-Ordner.	Ein Link auf das Entscheidungsprotokoll, vielleicht mit den fünf oder sechs Entscheidungen, die jede neue Person kennen muss.
10. Qualitätsanforderungen	Ein Qualitätsbaum (der Utility Tree aus Kapitel 2) und konkrete Qualitätsszenarien, die die Qualitätsziele aus Abschnitt 1 verfeinern.	Drei bis fünf Szenarien.	Ein Utility Tree mit priorisierten Szenarien, verknüpft mit Fitness Functions.
11. Risiken und technische Schulden	Bekannte technische Risiken und Schulden, möglichst priorisiert, mit Plänen zur Risikominderung.	Eine kurze Liste, jedes Quartal überprüft.	Ein Link auf das Schuldenregister, dazu die wichtigsten Risiken mit Verantwortlichen.
12. Glossar	Die wichtigen fachlichen und technischen Begriffe, mit Definitionen, auf die sich alle einigen.	Etwa zehn Begriffe.	Ein gemeinsames Glossar, manchmal pro Bounded Context.

arc42 sparsam nutzen

Der häufigste Weg, mit arc42 zu scheitern, ist, es als Formular zu behandeln, das ausgefüllt werden muss: Zwölf Überschriften werden angelegt, jede wird mit irgendetwas gefüllt, und das Ergebnis ist lang, generisch und wird nie gelesen. Die Hinweise der Vorlage selbst weisen in die andere Richtung. Die arc42-FAQ sagt, es gebe keine allgemeine Regel für das Minimum, weil es von Kritikalität, Größe, Komplexität und Risiko abhängt, schlägt aber vor, dass selbst eine

minimale Beschreibung drei bis fünf zentrale Qualitätsanforderungen als Szenarien, den Kontext und die externen Schnittstellen, eine kurze Lösungsstrategie, die Bausteinsicht der Ebene 1 und die wichtigsten querschnittlichen Konzepte enthalten sollte.

Vier Gewohnheiten halten arc42 schlank. **Löschen Sie leere Abschnitte oder kennzeichnen Sie sie bewusst** („Verteilung: eine einzige verwaltete Plattform; siehe Infrastruktur-Repository“), damit Lesende wissen, dass die Lücke beabsichtigt ist. **Verlinken statt kopieren**: Die Anforderungen liegen im Backlog, die Entscheidungen im ADR-Ordner, die Verteilung im Infrastructure as Code; arc42 verweist darauf. **Bevorzugen Sie Diagramme mit kurzen Elementtabellen** gegenüber seitenlanger Prosa. **Schreiben Sie für die Fragen der nächsten Lesenden**, nicht für Vollständigkeit: Wenn niemand nach dem Laufzeitverhalten der Admin-Oberflächen fragen wird, beschreiben Sie es nicht.

arc42 lässt sich gut mit dem C4-Modell kombinieren. Eine gängige Paarung ist ein C4-Systemkontextdiagramm in Abschnitt 3, ein Containerdiagramm als Ebene 1 der Bausteinsicht in Abschnitt 5 mit Komponentendiagrammen als Ebene 2 und ein C4-Deployment-Diagramm in Abschnitt 7. Als Diagrams as Code im selben Repository wie der arc42-Text gepflegt, lässt sich die gesamte Beschreibung wie Code versionieren und prüfen.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: ein Verweis, keine Kopie

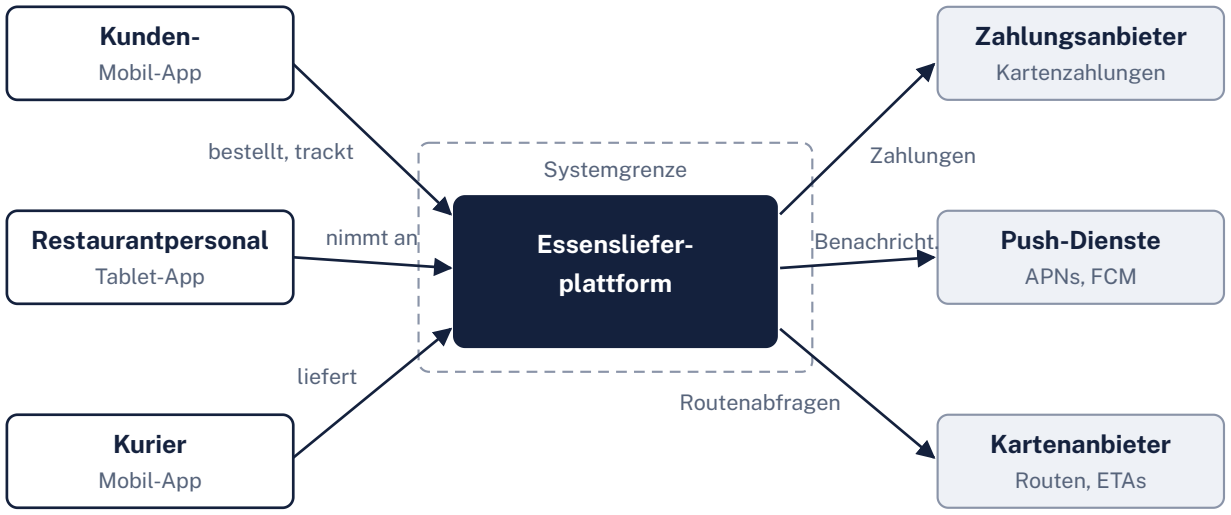
Der Punkt, an dem hier die meisten stolpern, ist, dass arc42 ein Satz beschrifteter Schubladen ist, kein Formular zum Ausfüllen. Viele Abschnitte sollten einen kurzen Verweis darauf enthalten, wo die eigentliche Information bereits liegt, keine Kopie davon.

- Das Team der Essenslieferplattform pflegt bereits zwei C4-Diagramme (Systemkontext und Container), ein Entscheidungsprotokoll in docs/decisions und seine Qualitätsszenarien in docs/quality.md. Nun ordnet es sie mit arc42.
- Die Diagramme kommen in ihre Schubladen: das Kontextdiagramm in Abschnitt 3 und das Containerdiagramm als Ebene 1 der Bausteinsicht in Abschnitt 5.
- Für Abschnitt 9, Architekturentscheidungen, schlägt jemand vor, den vollständigen Text jedes Entscheidungsdokuments hineinzukopieren, damit alles in einem Dokument steht.
- Dann löst ADR-0015 ADR-0009 ab, wie im vorigen Abschnitt. Der Pull Request, der ADR-0015 hinzufügt, ändert auch das Entscheidungsprotokoll, aber niemand denkt an die eingefügte Kopie, die ADR-0009 weiterhin als akzeptiert darstellt.
- Wer nun liest, findet zwei Antworten: Die arc42-Beschreibung sagt, Push-Benachrichtigungen laufen über ein SDK eines Drittanbieters, während das Entscheidungsprotokoll sagt, das Team betreibe inzwischen eine eigene Push-Zustellung. Man kann nicht erkennen, was stimmt, und beginnt, der ganzen Beschreibung zu misstrauen.
- Deshalb enthält Abschnitt 9 einen Link auf docs/decisions und die Nummern einiger Pflicht-Records. Jeder andere Abschnitt funktioniert genauso: Die Qualitätsszenarien bleiben in docs/quality.md, die Verteilung im Infrastruktur-Repository, und ein Abschnitt ohne Ergänzung wird gelöscht oder sagt in einer Zeile, warum.
- **Die Lehre:** arc42 sagt Lesenden, wo sie suchen müssen; es muss nicht selbst alles enthalten. Geben Sie jedem Fakt ein Zuhause und lassen Sie die Vorlage darauf verweisen, dann bleibt die Beschreibung kurz und wahr.

Ausgearbeitetes Beispiel: arc42-lite für die Essenslieferplattform

Die folgende Seite ist eine vollständige arc42-lite-Beschreibung der Essenslieferplattform, die sich durch dieses Kapitel zieht. Sie passt auf eine gedruckte Seite plus Kontextdiagramm, beantwortet aber die Fragen, die eine neue Ingenieurin, eine Betriebsleitung oder ein Prüfer zuerst stellen würden, und verweist darauf, wo die Details liegen.

Abschnitt 3: fachlicher Kontext



Der Kontext der Plattform: drei Nutzergruppen links, drei externe Dienste rechts, jeder Pfeil mit dem beschriftet, was ausgetauscht wird; Kanäle und Protokolle gehören in den technischen Kontext.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine einseitige arc42-lite-Beschreibung

Essenslieferplattform, Architekturbeschreibung, Fassung vom leitenden Architekten am 2026-10-01 geprüft. Abschnitte, die nichts beitragen würden, sind als solche ausgewiesen, statt leer zu bleiben.

1. Ziele	Kunden sollen bei lokalen Restaurants bestellen und die Lieferung in Echtzeit verfolgen können. Qualitätsziele nach Priorität: (1) keine Bestellung geht verloren oder wird doppelt belastet; (2) Bestellbestätigung p99 unter 1,5 s bei 120 Bestellungen/s; (3) ein neuer Consumer von Bestelldaten lässt sich hinzufügen, ohne den Bestellservice zu ändern. Stakeholder: Kunden, Restaurants, Kuriere, Betrieb, Finanzen, der Datenschutzbeauftragte (DSB), der Zahlungsanbieter.
2. Randbedingungen	Kartendaten berühren nie unsere Systeme (gehostete Kartenfelder des Zahlungsanbieters, um den Geltungsbereich von PCI DSS zu begrenzen). Personenbezogene Daten aus Großbritannien und der EU bleiben in EU-Regionen. Fünf Squads; serverseitiger Code nur in Java, Go oder TypeScript.
3. Kontext	Siehe Kontextdiagramm. Externe Schnittstellen: REST-API und Webhooks des Zahlungsanbieters; APNs und FCM für Push; Kartenanbieter für Routen und ETAs. Verantwortliche und Verträge der Schnittstellen sind in `docs/interfaces.md` aufgeführt.
4. Lösungsstrategie	Getrennte Services für Bestellungen, Disposition und Benachrichtigungen (Ziel 3); asynchrone Bestellereignisse über einen verwalteten Broker mit Transactional Outbox (Ziele 1 und 2, ADR-0012); verwaltetes PostgreSQL für Bestellungen (Ziel 1, ADR-0007); Idempotenzschlüssel bei jedem Zahlungsaufruf (Ziel 1).
5. Bausteine	Ebene 1 ist das C4-Containerdiagramm in `docs/diagrams/containers.dsl`: drei Client-Apps, API-Gateway, Bestell-, Dispositions- und Benachrichtigungsservice, Bestelldatenbank und das Topic für Bestellereignisse auf dem verwalteten Broker. Ebene 2 gibt es nur für den Bestellservice, den riskantesten Service.
6. Laufzeit	Zwei Szenarien: *Bestellung aufgeben und bezahlen* (Sequenzdiagramm) und *Timeout beim Zahlungsanbieter* (zeigt die Wiederholung mit Idempotenzschlüssel und den Status „ausstehend“).
7. Verteilung	Ein Cloud-Anbieter, zwei EU-Regionen, verwaltetes Kubernetes und verwaltete Datenbanken; definiert im Infrastruktur-Repository, das die Single Source of Truth ist.
8. Konzepte	Domänenmodell (Zustände im Lebenszyklus einer Bestellung); Idempotenz und Wiederholungen; Observability (Trace-IDs in jedem Ereignis); Umgang mit personenbezogenen Daten und deren Aufbewahrung.

9. Entscheidungen	Siehe das Entscheidungsprotokoll in `docs/decisions/`. Pflichtlektüre: ADR-0004, ADR-0007, ADR-0012.
10. Qualität	Szenarien QS-1 bis QS-5 in `docs/quality.md`, jeweils mit Antwortmaß; QS-3 (Latenz der Bestätigung) und QS-1 (keine verlorenen Bestellungen) haben automatisierte Fitness Functions.
11. Risiken und Schulden	Nur ein Broker-Anbieter (Lock-in, in ADR-0012 akzeptiert); die Disposition liest noch direkt aus der Bestelldatenbank (Schulden, Abbau für Q1 geplant); Rate Limits des Kartenanbieters zu Spitzenzeiten (Risiko, verantwortlich: Dispositions-Squad).
12. Glossar	Bestellung, Warenkorb, Annahme, Abholung, Übergabe, Disposition, ETA, Outbox, Idempotenzschlüssel: definiert in `docs/glossary.md`.

Jeder Abschnitt ist entweder knapp gefüllt oder verweist auf eine einzige maßgebliche Quelle; nichts ist kopiert, also kann nichts auseinanderlaufen.

Die Qualitätsziele in Abschnitt 1 bestimmen die Strategie in Abschnitt 4, die die Entscheidungen in Abschnitt 9 zitiert: Lesende können jede wichtige Wahl auf ein Ziel zurückführen.

Weitere Vorlagen

Die SEI-Vorlage *Views and Beyond*. Das Software Engineering Institute veröffentlicht eine Dokumentationsvorlage, die *Documenting Software Architectures: Views and Beyond* von Paul Clements und Kollegen folgt. Ihre oberste Ebene hat sechs Teile: eine Roadmap und Übersicht der Dokumentation, den Architekturhintergrund, die Sichten, die Zuordnung zwischen den Sichten, referenzierte Materialien und ein Glossar mit Abkürzungen. Jede Sicht wird auf dieselbe Weise dokumentiert, mit einer primären Darstellung, einem Elementkatalog, einem Kontextdiagramm, einem Variabilitätsleitfaden und der Begründung (beschrieben im Kasten „Vertiefung“ in Abschnitt 4.2). Sie ist gründlicher als arc42 und eignet sich für große, langlebige oder vertraglich spezifizierte Systeme, bei denen jede Sicht vollständig und mit Querverweisen versehen sein muss; ihre Zuordnung zwischen den Sichten ist eine Stärke, die arc42 implizit lässt.

IEEE 1016. IEEE Std 1016-2009, *Software Design Descriptions*, legt Inhalt und Aufbau einer Software Design Description (SDD) fest, des detaillierten Entwurfsdokuments, das traditionell in Verteidigung, Luft- und Raumfahrt und anderen vertragsgetriebenen Projekten verlangt wird. Es übernahm die Begriffe Stakeholder, Anliegen, Blickwinkel und Sicht aus IEEE 1471 (dem Vorgänger von 42010) und listet Entwurfsblickwinkel wie Kontext, Komposition, logische Struktur, Abhängigkeiten, Information, Schnittstellen, Interaktion, Zustandsdynamik,

Algorithmen und Ressourcen auf. IEEE setzte es 2020 auf den Status *inactive-reserved*, doch es wird weiterhin in Verträgen und in der Lehre zitiert, und ältere Programme liefern SDDs noch in dieser Form. Es steht dem Feinentwurf näher als der Architektur.

Designdokumente und RFCs, die viele Engineering-Organisationen verwenden, sind eine andere Art von Vorlage: ein Vorschlag für eine einzelne Änderung statt einer Beschreibung des ganzen Systems. Sie werden im übernächsten Abschnitt behandelt, Architektur kommunizieren.

Konformität mit ISO/IEC/IEEE 42010

ISO/IEC/IEEE 42010:2022 stellt Anforderungen an Architekturbeschreibungen und an Architekturbeschreibungsrahmen (ADFs): etablierte Konventionen wie arc42 oder den Ansatz Views and Beyond, mit denen in einer bestimmten Community Beschreibungen erstellt werden. Die Norm schreibt keine Vorlage, keine Notation und keinen Prozess vor. Eine Beschreibung ist *konform*, wenn sie die Informationen enthält, die die Norm verlangt, gleich welche Vorlage sie gliedert.

Grob gesagt bedeutet das: das beschriebene System und seine Umgebung benennen, die Stakeholder und ihre Anliegen benennen, angeben, welche Blickwinkel verwendet werden und welche Anliegen jeder abdeckt, Sichten liefern, die diesen Blickwinkeln folgen, Korrespondenzen festhalten (wie Elemente in verschiedenen Sichten zusammenhängen) und die Begründung für zentrale Entscheidungen festhalten. arc42 beansprucht keine Konformität von Haus aus, und einfachen arc42-Beschreibungen fehlt oft eine explizite Verbindung von jedem Anliegen zu der Sicht, die es behandelt. Die Lücken sind klein und leicht zu schließen, wie die Tabelle zeigt. Die meisten Teams brauchen keine formale Konformität; Teams in regulierten Branchen oder mit vertraglichen Pflichten manchmal schon, und es ist beruhigend, dass eine schlanke Vorlage die Norm mit wenigen Ergänzungen erfüllen kann.

Eine arc42-Beschreibung konform zu 42010 machen

42010 verlangt	Wo arc42 es hat	Was zu ergänzen ist
Stakeholder und ihre Anliegen	Abschnitt 1.3, Stakeholder-Tabelle; Qualitätsziele in 1.2 und 10	Anliegen pro Stakeholder explizit auflisten, nicht nur Erwartungen.
Blickwinkel, die jeweils benannte Anliegen abdecken	Implizit in den Abschnitten 3, 5, 6 und 7	Eine kurze Tabelle, die jeden Blickwinkel, die Anliegen, die er abdeckt, und seine Notation nennt (zum Beispiel C4-Containerdiagramme).
Sichten, die den Blickwinkeln folgen	Abschnitte 3, 5, 6 und 7	Nichts, sofern jede Sicht angibt, welchem Blickwinkel sie folgt.
Korrespondenzen zwischen Sichten	Teilweise in Abschnitt 7 (was wo läuft)	Zuordnungstabellen, zum Beispiel Baustein zu Verteilungsknoten, Baustein zu verantwortlichem Team.
Begründung zentraler Entscheidungen	Abschnitte 4 und 9 (ADRs)	Jede Entscheidung mit den Anliegen verknüpfen, die sie behandelt.

ACHTUNG

Häufige Fehler mit Vorlagen

- Jeden Abschnitt füllen, weil es ihn gibt. Ein leerer Abschnitt mit einer Zeile, die erklärt, warum, ist besser als eine Seite Füllstoff.
- Anforderungen, Entscheidungen und Infrastrukturdetails in die Vorlage kopieren, wo sie mit den Originalen auseinanderlaufen. Verlinken Sie stattdessen.
- Eine Bausteinsicht mit Dutzenden von Kästen auf Ebene 1. Ebene 1 sollte auf einen Bildschirm passen; Details gehören in tiefere Ebenen, und nur dort, wo es sich lohnt.
- Die Beschreibung einmal am Anfang schreiben und nie wieder. Stellen Sie sie unter Versionsverwaltung, prüfen Sie sie bei bedeutsamen Änderungen und zeigen Sie ein Datum „zuletzt geprüft“ an.
- Vorlagen ohne Übersicht mischen. Wenn Sie arc42 für die Struktur und C4 für die Diagramme verwenden, sagen Sie das in Abschnitt 1, damit Lesende die Konventionen kennen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Welcher arc42-Abschnitt wird meist der umfangreichste, und wie verhindern Sie, dass er ausufert?

Abschnitt 5, die Bausteinsicht. Beschränken Sie Ebene 1 auf ein Diagramm, das auf einen Bildschirm passt, und verfeinern Sie nur komplexe oder riskante Bausteine auf die Ebenen 2 und 3.

▼ Wo erscheinen Qualitätsziele in arc42, und wie unterscheiden sich die Abschnitte 1 und 10?

Abschnitt 1 nennt zur Orientierung die drei bis fünf wichtigsten Qualitätsziele nach Priorität; Abschnitt 10 verfeinert sie und alle weiteren Qualitäten zu einem Qualitätsbaum und konkreten, testbaren Szenarien.

▼ Ihr Team pflegt bereits ADRs im Repository. Was sollte Abschnitt 9 von arc42 enthalten?

Einen Link auf das Entscheidungsprotokoll, vielleicht mit einer kurzen Liste der Entscheidungen, die jede neue Person lesen muss, statt einer Kopie der Records.

▼ Macht die Verwendung von arc42 eine Architekturbeschreibung konform zu ISO/IEC/IEEE 42010?

Nicht automatisch. Konformität hängt vom Inhalt ab: Stakeholder, Anliegen, Blickwinkel, Sichten, Korrespondenzen und Begründungen müssen explizit sein, was meist einige kleine Ergänzungen zu arc42 erfordert.

▼ Probieren Sie es aus: Ein dreiköpfiges Team betreibt einen kleinen Service für Restaurant-Speisekarten. Die Führungskraft weist es an, alle zwölf arc42-Abschnitte auszufüllen. Was würden Sie stattdessen raten, und wie sollte das Team mit Abschnitten umgehen, die es nicht braucht?

Verwenden Sie arc42-lite. Schreiben Sie drei bis fünf Qualitätsszenarien, den Kontext, eine kurze Lösungsstrategie, die Bausteine der Ebene 1 und die wichtigsten Querschnittskonzepte, idealerweise auf etwa einer Seite. Verlinken Sie ADRs und Infrastruktur-Code, statt sie zu kopieren. Jeder Abschnitt ist optional, geben Sie also bewusst an, dass ein Abschnitt nicht nötig ist, statt ihn leer zu lassen oder aufzublähen. arc42 sagt, wohin Informationen gehören; es sollte nie Umfang verlangen.

QUELLEN

- arc42 overview: https://arc42.org/overview

- arc42 documentation: section 1, Introduction and Goals: https://docs.arc42.org/section-1/
- arc42 FAQ: What is the minimal amount of an arc42 documentation?: https://faq.arc42.org/questions/B-4/
- SEI Software Architecture Documentation Template (Views and Beyond): https://wiki.sei.cmu.edu/confluence/display/SAD/Software+Architecture+Documentation+Template
- IEEE 1016-2009: Software Design Descriptions: https://standards.ieee.org/ieee/1016/4502/

Diagrams as Code und aktuelle Dokumentation

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.8 Wie Software entsteht
- 4.1 Warum dokumentieren
- 4.3 Das C4-Modell

Begriffe von früher, die er verwendet: Versionskontrolle, Pull Request, Continuous Integration (CI), Sequenzdiagramm, Lebende Dokumentation

DIE IDEE IN KÜRZE

Stellen Sie sich ein Familienkochbuch vor, in das jede Änderung von dem, der gerade kocht, mit Bleistift eingetragen wird, ohne festzuhalten, wer was geändert hat. Nach ein paar Jahren traut ihm niemand mehr. Stellen Sie sich nun vor, die Rezepte sind abgetippt, jede Änderung wird vorgeschlagen, von einer zweiten Köchin oder einem zweiten Koch geprüft und datiert, und das gedruckte Buch wird nach jeder genehmigten Änderung automatisch neu gedruckt. Diagrams as Code überträgt diese Disziplin auf Architekturdiagramme: Sie werden als kurze Textdateien geschrieben, zusammen mit der Software aufbewahrt, bei Änderungen an der Software geprüft und automatisch neu gezeichnet.

WAS SIE WISSEN MÜSSEN

- Diagrams as Code bedeutet, Diagramme in einer Textsprache zu schreiben und von einem Werkzeug zeichnen zu lassen, sodass sie versioniert, verglichen, in Pull Requests geprüft und von der Build-Pipeline gerendert werden können.
- PlantUML (2009) ist am stärksten bei UML, besonders bei Sequenzdiagrammen; Mermaid (2014) wird auf GitHub und GitLab direkt in Markdown gerendert; D2 (2022) bietet saubere Layouts; Structurizr DSL definiert ein einziges C4-Modell, aus dem viele Sichten erzeugt werden.
- Zeichenwerkzeuge kennen nur Formen, sodass dasselbe Element zwischen Diagrammen auseinanderlaufen kann; Modellierungswerkzeuge halten Elemente und Beziehungen in einem Modell, sodass jede Sicht konsistent bleibt und das Modell abgefragt und geprüft werden kann.

- Eine Docs-as-Code-Pipeline rendert Diagramme, prüft Links, baut die Dokumentationssite bei jedem Pull Request und veröffentlicht von main aus, sodass Lesende eine Dokumentation sehen, die zum Code passt.
- Dokumentation bleibt durch Gewohnheiten aktuell, nicht durch Werkzeuge: namentlich benannte Verantwortliche, Review zusammen mit Codeänderungen, Datum der letzten Prüfung, Links zwischen Dokumentation, Code und Entscheidungen sowie das Löschen veralteter Seiten (ADRs werden ersetzt, nicht gelöscht).

Was Diagrams as Code bedeutet

Diagrams as Code bedeutet, ein Diagramm als kurze Textdatei in einer kleinen Sprache zu schreiben und es von einem Werkzeug anordnen und zeichnen zu lassen. Statt Kästen auf einer Zeichenfläche zu verschieben, schreiben Sie `ordering → db: SQL`, und das Werkzeug erzeugt das Bild. Die Textdatei liegt im Code-Repository und wird wie Code behandelt: versioniert, geprüft, von automatisierten Builds getestet und von der Pipeline veröffentlicht.

Es ist ein Teil von Docs as Code, der von der Community Write the Docs vorangetriebenen Praxis, Dokumentation mit denselben Werkzeugen und demselben Arbeitsablauf wie Software zu erstellen: Issue-Tracker, Versionskontrolle (meist Git), Klartext-Auszeichnungssprachen wie Markdown oder AsciiDoc, Code-Review und automatisierte Tests. Die vorigen Abschnitte haben das Material geliefert: Sichten und C4-Diagramme, UML, wo Präzision zählt, ADRs und eine arc42-Struktur. In diesem Abschnitt geht es darum, dieses Material über die gesamte Lebensdauer des Systems korrekt zu halten, und genau daran scheitert die meiste Architekturdokumentation. Die Gründe für dieses Scheitern wurden in Warum und wie viel dokumentieren dargelegt: veraltete Wikis, große Dokumente, für die niemand verantwortlich ist, und Diagramme, die nicht mehr zum System passen.

Warum Diagramme als Text

Versionskontrolle. Ein Diagramm in Textform hat eine Historie. Sie sehen, wer es wann und warum geändert hat, und können zwei beliebige Versionen Zeile für Zeile vergleichen; ein Binärbild oder eine Datei im Format eines Zeichenwerkzeugs zeigt nur, dass sich etwas geändert hat. Sie können es auch verzweigen: Das Diagramm für eine vorgeschlagene Änderung liegt auf demselben Branch wie der Code, der sie umsetzt.

Review in Pull Requests. Wenn eine Entwicklerin oder ein Entwickler eine neue Abhängigkeit zwischen Services einführt, erscheint die Änderung am Diagramm im selben Pull Request wie der Code, als lesbarer Text-Diff. Die Reviewer können fragen, ob der neue Pfeil gewollt ist, und das ist ein Architektur-Review genau in dem Moment, in dem es am wenigsten kostet.

Konsistenz. Das Werkzeug wendet auf jedes Diagramm dieselben Formen, Schriften und Layoutregeln an, sodass Diagramme von zwanzig Personen trotzdem wie ein zusammengehöriger Satz aussehen. Mit einer Modellierungssprache (siehe unten) erscheint dasselbe Element in jeder Sicht mit demselben Namen und derselben Beschreibung.

Generierung. Text kann von Programmen erzeugt werden. Diagramme lassen sich aus dem Code selbst (Abhängigkeitsgraphen), aus Infrastructure as Code (Deployment-Diagramme), aus API-Spezifikationen oder aus einem zentralen Modell generieren und dann automatisch gegen das System prüfen.

Die Abwägungen sind real. Automatisches Layout ist bequem, aber manchmal hässlich, und der Kampf mit einer Layout-Engine um die Position eines einzigen Kastens kann länger dauern, als ihn zu zeichnen; die meisten Werkzeuge bieten Hinweise wie Richtung oder Gruppierung, keine exakten Positionen. Textsprachen haben eine Lernkurve für Menschen, die nicht programmieren, etwa im Produktmanagement. Und eine schöne Skizze am Whiteboard während einer Designdiskussion ist immer noch der schnellste Weg zu denken; Diagrams as Code sind für die Diagramme gedacht, die Sie behalten wollen.

Vier Sprachen, ein Beispiel

Vier Sprachen dominieren. Jedes Beispiel unten beschreibt einen Teil derselben Essenslieferplattform.

PlantUML, von Arnaud Roques entwickelt und 2009 erstmals veröffentlicht, ist die älteste der vier. Es deckt alle gängigen UML-Diagramme ab (Sequenz-, Komponenten-, Deployment-, Zustands-, Aktivitäts- und Klassendiagramm) und viele weitere und nutzt Graphviz für das Layout der meisten Diagrammtypen. Die Bibliothek C4-PlantUML ergänzt C4-Formen. Seine Sequenzdiagramme sind besonders gut, was es zur naheliegenden Wahl für die Laufzeitszenarien aus UML-Grundlagen macht.

```
@startuml
title Place an order (happy path)
actor Customer
participant "Customer app" as App
participant "Ordering service" as Ord
participant "Payment provider" as Pay
queue "Order events topic" as MB

Customer -> App : Tap "Pay"
App -> Ord : POST /orders (HTTPS)
Ord -> Pay : Authorise payment
Pay --> Ord : Authorised
Ord ->> MB : OrderPlaced (async)
Ord --> App : 201 Created
@enduml
```

PlantUML: ein Sequenzdiagramm zum Aufgeben einer Bestellung; `->>` zeichnet eine asynchrone Nachricht und `-->` eine Antwort

Mermaid, 2014 von Knut Sveidqvist entwickelt, ist in JavaScript geschrieben und wird im Browser gerendert. Sein größter Vorteil ist die Verbreitung: GitHub rendert Mermaid-Blöcke in Markdown-Dateien seit Februar 2022, und GitLab sowie viele Wiki- und Notizwerkzeuge tun dasselbe, sodass ein Diagramm in einer README überhaupt keinen Build-Schritt braucht. Es unterstützt unter anderem Flussdiagramme, Sequenz-, Klassen-, Zustands- und Entity-Relationship-Diagramme und hat experimentelle C4-Unterstützung. Bei großen Diagrammen ist die Kontrolle über das Layout begrenzt.

```
flowchart LR
    app["Customer app<br/>Kotlin, Swift"]
    gw["API gateway"]
    ord["Ordering service<br/>Java"]
    db[("Order database<br/>PostgreSQL")]
    topic[["Order events<br/>Kafka topic"]]
    disp["Dispatch service<br/>Go"]
    notif["Notification service<br/>Node.js"]

    app -->|HTTPS/JSON| gw
    gw -->|HTTPS/JSON| ord
    ord -->|SQL| db
    ord -->|publishes OrderPlaced| topic
    disp -->|consumes| topic
    notif -->|consumes| topic
```

Mermaid: ein Flussdiagramm auf Container-Ebene; `[( )]` zeichnet einen Datenbankzylinder, und die Beschriftungen sitzen an den Pfeilen; das Topic wird als Container gezeichnet, wie Abschnitt 4.3 empfiehlt

Structurizr DSL, von Simon Brown parallel zum C4-Modell entwickelt, ist grundsätzlich anders. Es beschreibt kein Diagramm, sondern ein *Modell* (Personen, Softwaresysteme, Container, Komponenten und die Beziehungen zwischen ihnen) und definiert dann *Sichten* auf dieses Modell. Jedes Element wird einmal definiert, mit Name, Beschreibung und Technologie, und erscheint in jeder Sicht, die es enthält. Beziehungen zwischen Containern werden auf Systemebene automatisch abgeleitet, sodass die Kontextsicht keine eigenen Pfeile braucht. Die DSL kann dem Workspace außerdem Markdown-Dokumentation und einen Ordner mit ADRs zuordnen, und die Werkzeuge können Sichten nach PlantUML, Mermaid, D2 und in andere Formate exportieren.

```
workspace "Food delivery" {
    model {
        customer = person "Customer" "Orders food from local restaurants."
        payments = softwareSystem "Payment provider" "Takes card payments." "External"
        platform = softwareSystem "Food-delivery platform" {
            app = container "Customer app" "Ordering and tracking." "Kotlin, Swift"
            gateway = container "API gateway" "Authenticates and routes requests."
"Managed API gateway"
            ordering = container "Ordering service" "Accepts and tracks orders." "Java"
            db = container "Order database" "Orders and their states." "PostgreSQL"
"Database"
            orderEvents = container "Order events" "Order lifecycle events." "Kafka
topic" "Topic"
            dispatch = container "Dispatch service" "Assigns couriers." "Go"
        }
        customer → app "Places and tracks orders using"
        app → gateway "Makes API calls to" "JSON/HTTPS"
        gateway → ordering "Routes requests to" "JSON/HTTPS"
        ordering → db "Reads from and writes to" "SQL/TCP"
        ordering → payments "Takes card payments using" "JSON/HTTPS"
        ordering → orderEvents "Publishes order events to" "Kafka"
        dispatch → orderEvents "Consumes order events from" "Kafka"
    }
    views {
        systemContext platform "Context" {
            include *
            autoLayout lr
        }
        container platform "Containers" {
            include *
            autoLayout lr
        }
        styles {
            element "Database" {
                shape cylinder
            }
            element "Topic" {
                shape pipe
            }
        }
    }
}
```

Structurizr DSL: ein Modell und zwei Sichten der Essenslieferplattform; die Kontextsicht wird aus denselben Definitionen erzeugt

D2 (Declarative Diagramming), 2022 von Terrastruct als Open Source veröffentlicht, ist die jüngste. Sie zielt auf saubere, moderne Ausgabe und lesbare Syntax, unterstützt verschachtelte Container (mit geschweiften Klammern geschrieben und mit Punktnamen

referenziert) und bietet eine Auswahl an Layout-Engines: dagre und ELK sind enthalten und kostenlos, und TALA, für Softwarearchitekturdiagramme entworfen, ist eine separate kommerzielle Engine.

```
direction: right

customer: Customer {shape: person}
payments: Payment provider

platform: Food-delivery platform {
  app: Customer app
  ordering: Ordering service
  db: Order database {shape: cylinder}
  events: Order events topic {shape: queue}
  dispatch: Dispatch service

  app → ordering: Places order (HTTPS)
  ordering → db: SQL
  ordering → events: Publishes OrderPlaced
  dispatch → events: Consumes OrderPlaced
}

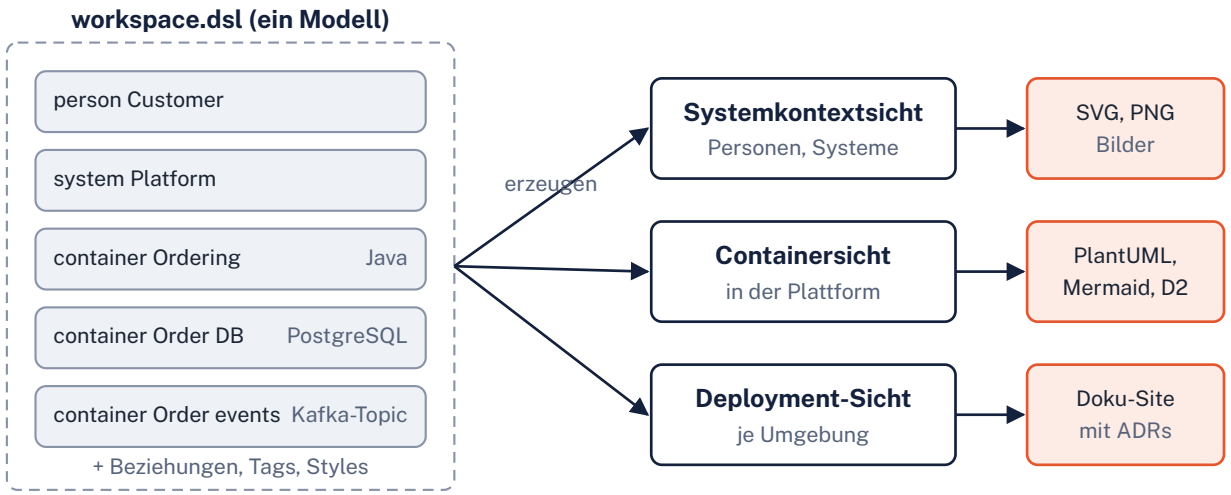
customer → platform.app: Places orders using
platform.ordering → payments: Authorise (HTTPS)
```

D2: verschachtelte Container, Formen und beschriftete Verbindungen

Die vier im Vergleich

	PlantUML	Mermaid	Structurizr DSL	D2
Beschreibt	Ein Diagramm pro Datei	Ein Diagramm pro Block	Ein Modell plus Sichten darauf	Ein Diagramm pro Datei
Am stärksten bei	UML, besonders Sequenzdiagrammen	Diagrammen in Markdown ohne Build-Schritt	Konsistenten C4-Diagrammen auf mehreren Ebenen	Ansprechenden allgemeinen Architekturdiagrammen
Rendering	Java-Werkzeug oder Server; Graphviz für die meisten Layouts	Im Browser; nativ auf GitHub und GitLab	Structurizr-Werkzeuge; Export in andere Formate	Kommandozeilenwerkzeug; Layout-Engine wählbar
Worauf zu achten ist	Altbackenes Standardaussehen; Layoutsteuerung nur über Hinweise	Begrenzte Layoutsteuerung bei großen Diagrammen	Nur C4-Konzepte; braucht eigene Werkzeuge	Die beste Layout-Engine ist kommerziell

Ein Modell, viele Sichten



Ordering einmal im Modell umbenennen, und jede Sicht und jeder Export ändert sich mit.

Eine Modellierungssprache definiert jedes Element einmal; Sichten und Exporte werden generiert und können sich daher nicht widersprechen.

Zeichenwerkzeuge und Modellierungswerkzeuge

Der Unterschied zwischen Structurizr und den anderen drei ist ein Fall der in Abschnitt 4.3 eingeführten Unterscheidung zwischen Zeichen- und Modellierungswerkzeugen. Ein Zeichenwerkzeug (diagrams.net, Excalidraw, Visio, ein Online-Whiteboard und auch PlantUML, Mermaid und D2) kennt Formen, Linien und Text. Ein Kasten mit der Beschriftung „Bestellservice“ in einem Diagramm hat keine Verbindung zum gleich beschrifteten Kasten in einem anderen. Benennen Sie den Service um, müssen Sie jedes Diagramm von Hand suchen und korrigieren; vertippen Sie sich einmal, haben Sie, soweit das Werkzeug weiß, zwei Services.

Ein Modellierungswerkzeug (Structurizr, Archi für ArchiMate-Modelle, Sparx Enterprise Architect für UML und SysML, Capella) führt ein Modell: ein Repository von Elementen und Beziehungen mit Eigenschaften. Diagramme sind Sichten auf das Modell. Benennen Sie ein Element um, ändert sich jede Sicht. Das Modell lässt sich abfragen („Welche Container hängen vom Broker ab?“), prüfen („Keine Komponente der Kunden-App darf die Datenbank aufrufen“) und mit dem laufenden System vergleichen. Der Preis ist mehr Struktur im Voraus und, bei den schwergewichtigen kommerziellen Werkzeugen, Lizenzen, Schulungen und die Neigung, weit mehr zu modellieren, als jemand liest.

Eine praktische Regel: Verwenden Sie Zeichenwerkzeuge, textbasiert oder grafisch, für Skizzen, einmalige Erklärungen und Diagramme, die für sich stehen, etwa ein einzelnes Sequenzdiagramm in einem ADR. Verwenden Sie ein Modell, wenn dieselben Elemente in vielen Diagrammen vorkommen, wenn mehrere Teams beitragen oder wenn Sie die

Dokumentation gegen den Code prüfen wollen. Viele Teams kombinieren beides: ein Structurizr-Modell für die C4-Struktur und PlantUML oder Mermaid für einige Sequenz- und Zustandsdiagramme.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: eine Umbenennung, drei Dateien

Die schwierigste Idee dieses Abschnitts ist, dass Diagramme als Text sie nicht schon von selbst konsistent halten. Abschnitt 4.3 hat Werkzeuge in Zeichenwerkzeuge und Modellierungswerkzeuge unterteilt, und dieser Abschnitt ergänzt, dass PlantUML, Mermaid und D2 Zeichenwerkzeuge sind, obwohl man sie tippt, und dass nur ein Modellierungswerkzeug wie Structurizr weiß, dass zwei Kästen dasselbe Ding sind.

- Das Team der Essenslieferplattform führt drei Diagramme als Text: das PlantUML-Sequenzdiagramm zum Aufgeben einer Bestellung, das Mermaid-Container-Flussdiagramm und das D2-Diagramm. Jede Datei schreibt die Beschriftung „Bestellservice“ selbst aus.
- Angenommen, das Team benennt den Service in „Auftragsservice“ um. Jemand muss jede Datei finden, die ihn erwähnt, und jede Beschriftung von Hand ändern, denn nichts verbindet die drei Kästen: Sie sind einfach Text, der zufällig übereinstimmt.
- Wird eine Datei übersehen, widersprechen sich die Diagramme. Vertippt man sich bei einer Beschriftung, etwa „Auftragsservce“, ist es für dieses Werkzeug ein anderer Service. Die Pipeline rendert trotzdem jedes Diagramm, weil jede Datei für sich gültig ist.
- In der Structurizr-DSL-Datei wird der Bestellservice einmal definiert, im Modell, mit Name, Beschreibung und Technologie. Die Containersicht und jede andere Sicht, die ihn enthält, wird aus dieser einen Definition erzeugt.
- Ändern Sie dort den Namen, übernimmt ihn jede Sicht beim nächsten Build. Weil das Modell Elemente und Beziehungen enthält, lässt sich auch abfragen, welche Container vom Topic für Bestellereignisse abhängen, oder es lässt sich gegen eine Regel prüfen.
- **Die Lehre:** Text gibt Diagrammen eine Historie, Review in Pull Requests und automatisches Rendering; nur ein Modell sorgt dafür, dass dasselbe Element überall dasselbe bedeutet. Setzen Sie eines ein, wenn dieselben Teile in vielen Diagrammen vorkommen.

Dokumentation in der Pipeline erzeugen

Sobald Dokumentation als Text im Repository liegt, kann die Continuous-Integration-Pipeline aus Kapitel 0 sie wie Code bauen und prüfen. Eine typische Pipeline rendert jedes Diagramm (und lässt den Build bei einem Syntaxfehler scheitern), prüft Links, baut mit einem Generator wie MkDocs, Antora oder Docusaurus eine statische Dokumentationssite und veröffentlicht sie,

sobald Änderungen den Main-Branch erreichen. Lesende sehen dann immer die Dokumentation, die zum Code auf main passt, und niemand muss daran denken, ein Bild zu exportieren und in ein Wiki hochzuladen.

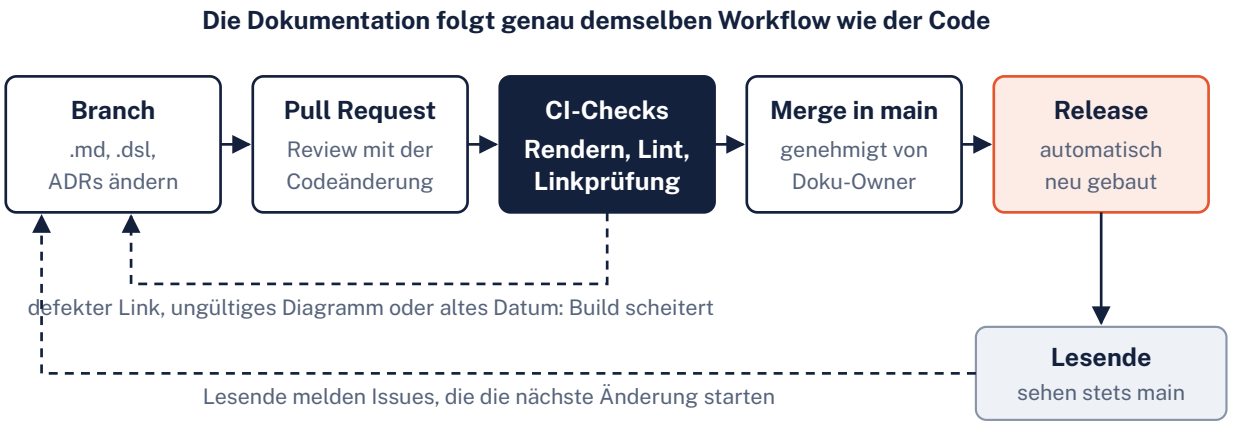
Weitere Prüfungen kommen hinzu, wenn das Team reifer wird: eine Rechtschreibprüfung und ein Prosa-Linter (Vale ist beliebt), eine Prüfung, dass jedes ADR einen gültigen Status hat, eine Warnung, wenn das Datum der „letzten Prüfung“ einer Seite älter als ihr Prüfintervall ist, und Architekturprüfungen, die die dokumentierten Abhängigkeiten mit den tatsächlichen im Code vergleichen und eine Form von Fitness Function sind. Manche Teams generieren Teile der Dokumentation aus dem System selbst: Abhängigkeitsgraphen aus dem Build, API-Referenzen aus OpenAPI-Spezifikationen, Deployment-Diagramme aus Infrastructure as Code.

```
name: architecture-docs
on:
  pull_request:
    paths: ["docs/**"]
  push:
    branches: [main]

jobs:
  docs:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      # assumes plantuml, d2, lychee and mkdocs are available on the runner
      - name: Render diagrams (fails on syntax errors)
        run: |
          plantuml -tsvg docs/diagrams/*.puml
          for f in docs/diagrams/*.d2; do d2 "$f" "${f%.d2}.svg"; done
      - name: Check links
        run: lychee --offline docs/
      - name: Build site (warnings such as broken links fail the build)
        run: mkdocs build --strict
      - name: Publish
        if: github.ref == 'refs/heads/main'
        run: ./scripts/publish-docs.sh
```

Ein GitHub-Actions-Workflow, der Diagramme rendert, Links prüft, die Dokumentationssite bei jedem Pull Request baut und von main aus veröffentlicht

Der Docs-as-Code-Workflow



Änderungen an der Dokumentation nehmen denselben Weg wie Codeänderungen: Branch, Review, automatisierte Prüfungen, Merge und automatische Veröffentlichung.

Dokumentation aktuell halten

Werkzeuge halten Dokumentation nicht wahr, Gewohnheiten schon. Fünf Gewohnheiten machen den Unterschied.

Verantwortung. Jedes Dokument hat einen namentlich benannten Owner, normalerweise das Team, dem der beschriebene Code gehört. In GitHub und GitLab kann eine `CODEOWNERS`-Datei dieses Team zum Pflicht-Reviewer für Änderungen in seinem Ordner `docs/` machen. Dokumente, für die „alle“ verantwortlich sind, pflegt niemand.

Review mit Codeänderungen. Eine Pull-Request-Vorlage, die fragt: „Ändert dies eine Schnittstelle, eine Abhängigkeit, ein Deployment oder eine Entscheidung? Wenn ja, welches Dokument oder ADR haben Sie aktualisiert?“, fängt den meisten Drift genau dann ab, wenn er entsteht. Manche Teams gehen weiter und blockieren das Mergen von Features ohne Dokumentation, eine Praxis, die der Leitfaden von Write the Docs erwähnt.

Datum der letzten Prüfung. Ein kleiner Kopfbereich auf jeder Seite hält ihren Owner fest, das Datum, an dem sie zuletzt mit der Wirklichkeit abgeglichen wurde, und wie oft sie geprüft werden soll. Lesende können einschätzen, wie weit sie ihr trauen können, und die Pipeline kann überfällige Seiten markieren. Eine Seite mit „zuletzt geprüft 2023“ ist ehrlich; eine Seite, die stillschweigend das System von 2023 beschreibt, ist eine Falle.

Dokumentation und Code verknüpfen. Seiten verweisen auf den Code, den sie beschreiben (Pfade, Module), Code-Kommentare verweisen auf ADRs, und Commit-Nachrichten nennen die Records, die sie umsetzen. Links in beide Richtungen machen offensichtlich, welche Dokumente eine Änderung betreffen könnte.

Veraltetes Material löschen. Falsche Dokumentation ist schlimmer als keine, weil Lesende danach handeln. Wenn eine Seite das System nicht mehr beschreibt und niemand sie korrigieren wird, löschen Sie sie oder verschieben Sie sie in ein klar gekennzeichnetes Archiv. ADRs sind die Ausnahme: Sie sind Aufzeichnungen der Geschichte und werden daher ersetzt oder als veraltet markiert, nicht gelöscht. Das ist die Idee der lebenden Dokumentation, die Cyrille Martraire 2019 in seinem gleichnamigen Buch entwickelt hat: Dokumentation, die sich im selben Tempo wie das System weiterentwickelt, idealerweise weil vieles davon aus dem System abgeleitet wird.

```
---
title: "Ordering service: building block view (level 2)"
owner: "@ordering-team"
last_reviewed: 2026-09-14
review_every: 6 months
code: src/ordering/
decisions: [ADR-0007, ADR-0012]
---
```

Front Matter am Anfang einer Markdown-Seite: Owner, Prüfdatum und -intervall sowie Links zum beschriebenen Code und zu den Entscheidungen

Checkliste: Dokumentation, die aktuell bleibt

Verwenden Sie sie, wenn Sie die Architekturdokumentation eines Repositorys aufsetzen, und gehen Sie sie alle paar Monate erneut durch.

- [ ] **Die Dokumentation liegt in dem Repository, das sie beschreibt**
  Markdown- oder AsciiDoc-Seiten, Diagrammquellen und ADRs unter `docs/`, nicht in einem separaten Wiki, das niemand in derselben Änderung aktualisiert.
- [ ] **Diagrammquellen sind Text und werden von der Pipeline gerendert**
  Keine von Hand exportierten Bilder, die stillschweigend hinter ihren Quellen zurückbleiben; der Build scheitert an einem ungültigen Diagramm.
- [ ] **Jedes Element wird einmal definiert, wo sich Diagramme überschneiden**
  Verwenden Sie ein Modell (etwa Structurizr DSL), wenn dieselben Container in mehreren Diagrammen vorkommen.
- [ ] **Jede Seite hat einen Owner und ein Datum der letzten Prüfung**
  Durchgesetzt durch `CODEOWNERS` und eine Pipeline-Prüfung, die warnt, wenn das Prüfintervall überschritten ist.
- [ ] **Pull Requests fragen nach der Dokumentation**
  Die Vorlage fragt, ob sich eine Schnittstelle, Abhängigkeit, ein Deployment oder eine Entscheidung geändert hat und welches Dokument aktualisiert wurde.
- [ ] **Links werden automatisch geprüft**
  Defekte interne Links und Verweise auf gelöschte Dateien lassen den Build scheitern.
- [ ] **Die Doku verweist auf den Code und der Code auf Entscheidungen**
  Pfade in der Dokumentation, ADR-Nummern in Code-Kommentaren und Commit-Nachrichten.
- [ ] **Die veröffentlichte Site wird aus main neu gebaut**
  Lesende sehen immer die Version, die zum deployten Code passt, ohne manuellen Veröffentlichungsschritt.
- [ ] **Veraltete Seiten werden gelöscht oder archiviert**
  Seiten, die niemand korrigieren wird, werden entfernt; ADRs werden ersetzt oder als veraltet markiert, nie gelöscht.
- [ ] **Mindestens eine Prüfung vergleicht die Dokumentation mit der Wirklichkeit**
  Zum Beispiel ein Test, der scheitert, wenn im Code eine Abhängigkeit auftaucht, die das Containerdiagramm nicht zeigt.

ACHTUNG

Häufige Fehler

- Ein Werkzeug wählen, bevor entschieden ist, was dokumentiert werden soll. Das Werkzeug zählt weit weniger als die Gewohnheiten von Verantwortung und Review.
- Alles in eine riesige Diagrammquelle packen, die niemand lesen kann, weder als Text noch als Bild.
- Gerenderte Bilder ohne ihre Quellen committen, oder Quellen ohne einen Build, der sie rendert, sodass beide auseinanderlaufen.
- Diagrams as Code als Ausrede nehmen, um Legenden, Titel und Beschriftungen wegzulassen: Die C4-Notationsregeln gelten auch für generierte Diagramme.
- Änderungen an der Dokumentation im Review optional machen, sodass sie unter Termindruck immer als Erstes wegfallen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Was kann ein Reviewer in einem Pull Request sehen, wenn ein Diagramm Code ist, was er bei einem exportierten Bild nicht sehen kann?

Einen lesbaren Zeile-für-Zeile-Diff dessen, was sich geändert hat, etwa eine neue Beziehung zwischen zwei Services, im selben Pull Request wie der Code, der sie eingeführt hat.

▼ Warum verursacht das Umbenennen eines Service mit einem Zeichenwerkzeug mehr Arbeit als mit einem Modellierungswerkzeug?

In einem Zeichenwerkzeug hält jedes Diagramm seine eigene Kopie der Beschriftung, sodass jedes Diagramm gefunden und geändert werden muss; in einem Modellierungswerkzeug ist das Element einmal definiert, und jede Sicht übernimmt den neuen Namen.

▼ Warum sollte veraltete Dokumentation gelöscht statt stehen gelassen werden, und was ist die Ausnahme?

Lesende handeln nach dem, was sie lesen, daher führt falsche Dokumentation zu falschen Entscheidungen. ADRs sind die Ausnahme: Sie halten Geschichte fest und werden stattdessen als ersetzt oder veraltet markiert.

▼ **Nennen Sie zwei Prüfungen, die eine Dokumentations-Pipeline neben dem Rendern von Diagrammen ausführen kann.**

Linkprüfung, Prosa-oder Rechtschreib-Linting, ein strikter Site-Build, eine Warnung bei überfälligen Prüfdaten und der Vergleich dokumentierter Abhängigkeiten mit denen im Code.

▼ **Probieren Sie es aus: Ihr Team hat acht Diagramme in einem Zeichenwerkzeug. Nach der Umbenennung eines Services zeigen drei noch den alten Namen, und ein Dispositionspfeil unterscheidet sich zwischen zweien. Was würden Sie bei Werkzeugen und Gewohnheiten ändern, und warum?**

Zeichenwerkzeuge kennen nur Formen, also driftet jedes Bild für sich. Wechseln Sie zu einem Modell, etwa Structurizr DSL, das im Repository liegt: Definieren Sie jedes Element einmal, und generieren Sie die Sichten, sodass eine Umbenennung jedes Diagramm erreicht. Rendern Sie sie in der Pipeline bei jedem Pull Request. Werkzeuge allein genügen nicht, benennen Sie also eine verantwortliche Person, prüfen Sie die Doku zusammen mit Code-Änderungen, zeigen Sie ein Datum der letzten Prüfung, und löschen Sie veraltete Seiten. PlantUML oder Mermaid passen für einzelne Sequenzdiagramme.

QUELLEN

- Docs as Code, Write the Docs: https://www.writethedocs.org/guide/docs-as-code/
- Structurizr DSL language reference: https://docs.structurizr.com/dsl/language
- Mermaid documentation: https://mermaid.js.org/
- PlantUML: https://plantuml.com/
- Include diagrams in your Markdown files with Mermaid, GitHub Blog (2022): https://github.blog/2022-02-14-include-diagrams-markdown-files-mermaid/

Architektur für verschiedene Zielgruppen kommunizieren

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 1.5 Stakeholder und Anliegen
- 1.6 Entscheidungen und Abwägungen
- 4.3 Das C4-Modell

Begriffe von früher, die er verwendet: Abwägung (Trade-off), Anliegen, C4-Modell, Gewichtete Entscheidungsmatrix, Last Responsible Moment

DIE IDEE IN KÜRZE

Eine Fremdenführerin erklärt dieselbe Kathedrale Schulkindern, Architekten und Pilgern unterschiedlich: andere Fakten, andere Worte, andere Wege durch das Gebäude. Softwarearchitekten müssen dasselbe tun. Eine Finanzdirektorin will wissen, was ein Design kostet und was schiefgehen könnte; ein Entwickler will wissen, wie die Teile zusammenhängen; eine Prüferin will Nachweise. Gute Kommunikation beginnt mit der Frage, wer zuhört und was diese Person entscheiden muss, und zeigt dann genau das, klar und ehrlich, einschließlich der Nachteile.

WAS SIE WISSEN MÜSSEN

- Die Architektur zu kommunizieren ist eine zentrale Architekturaufgabe; beginnen Sie jede Erklärung damit, die Zielgruppe zu benennen und was sie danach entscheiden oder tun muss.
- Führungskräfte brauchen die Empfehlung, Kosten, Risiko und die nötige Entscheidung; Entwicklungsteams brauchen Struktur, Schnittstellen und Begründungen; Betrieb, Sicherheit und Prüfer brauchen jeweils Sichten auf Ausfälle, Vertrauensgrenzen und Nachweise.
- Erklären Sie Abwägungen als Optionen, Kriterien, Empfehlung und Konsequenzen, beginnen Sie mit der Antwort und sagen Sie klar, was aufgegeben wird.
- Eine einseitige Architekturübersicht und eine nummerierte Geschichte nach dem Prinzip „einer Anfrage folgen“ machen jedes System verständlich; beschriften Sie beim Skizzieren jeden Kasten und jede Linie und halten Sie die Ergebnisse nach dem Whiteboard fest.
- Designdokumente und RFCs schlagen eine Änderung schriftlich zur Kommentierung vor, bevor sie gebaut wird; ADRs halten die Entscheidungen dann dauerhaft fest.

- Klare Diagramme haben einen Titel, eine Legende, eine konsistente Notation, wenige bedeutungstragende Farben, eine einzige Flussrichtung und beschriftete Pfeile; Meinungsverschiedenheiten werden gelöst, indem man Kriterien vereinbart, Belege sammelt, Rough Consensus anstrebt und abweichende Meinungen festhält.

Kommunikation ist die Aufgabe

Eine Architektur, die nur ihr Architekt versteht, ist gescheitert, wie gut sie auch sein mag. Entscheidungen prägen, was Dutzende Menschen bauen, betreiben, finanzieren und prüfen, und sie funktionieren nur, wenn diese Menschen sie gut genug verstehen, um danach zu handeln und zu bemerken, wenn die Wirklichkeit davon abweicht. Deshalb zählt der iSAQB-Lehrplan die Kommunikation der Architektur zu den Kernaufgaben einer Architektin oder eines Architekten, und deshalb hängt die Architektenrolle aus Kapitel 1 so stark von zwischenmenschlichen Fähigkeiten ab.

Die früheren Abschnitte dieses Kapitels haben das Material geliefert: Sichten, C4- und UML-Diagramme, ADRs, eine arc42-Struktur und eine Pipeline, die alles aktuell hält. In diesem Abschnitt geht es darum, dieses Material mit Menschen zu nutzen. Die Kernfähigkeit ist überall dieselbe: Entscheiden Sie, wer die Zielgruppe ist, was sie tun muss, nachdem sie Sie gehört hat, und geben Sie ihr genau das, was dafür nötig ist, in einer Form, die sie aufnehmen kann. Gregor Hohpes Bild vom *Architect Elevator* (*The Software Architect Elevator*, 2020) fasst die Bandbreite: Ein Architekt fährt zwischen der Vorstandsetage, wo Strategie in geschäftlichen Begriffen festgelegt wird, und dem Maschinenraum, wo Systeme gebaut werden, und muss auf jeder Etage verstanden werden.

Auf die Zielgruppe zuschneiden

Jede Zielgruppe hat ihre eigenen Anliegen, genau im Sinne von Stakeholder und Anliegen, und ihr eigenes Vokabular und ihre eigene Aufmerksamkeitsspanne. Die Tabelle ist ein Ausgangspunkt; die eigentliche Kunst besteht darin, vor jeder Vorbereitung zu fragen: „Was wird diese Person nach diesem Gespräch entscheiden oder anders machen?“

Was verschiedene Zielgruppen brauchen

Zielgruppe	Was ihnen wichtig ist	Was zeigen	Was vermeiden
Führungskräfte und Sponsoren	Kosten, Risiko, Time-to-Market, strategische Optionen, was sie entscheiden müssen	Eine Seite: die Empfehlung, die geschäftlichen Auswirkungen, die Kosten, die größten Risiken und die Entscheidung, die Sie von ihnen brauchen; höchstens ein Systemkontextdiagramm	Technologienamen ohne geschäftliche Bedeutung; Komponentendiagramme; ein Rundgang durch die Optionen vor der Antwort
Produktmanagement	Was möglich oder schwieriger wird, Liefertermine, für Nutzer sichtbares Verhalten	Fähigkeiten und Einschränkungen in Produktbegriffen; wie die Architektur die Roadmap beeinflusst; eine Laufzeitgeschichte einer User Journey	Interne Struktur um ihrer selbst willen
Entwicklungsteams	Wie man innerhalb der Architektur baut: Grenzen, Schnittstellen, Konventionen, warum die Dinge so sind, wie sie sind	C4-Container- und Komponentendiagramme, Sequenzdiagramme, ADRs, Codebeispiele, das Entscheidungsprotokoll	Vage Prinzipien ohne Beispiele; Diagramme, die nicht zum Code passen
Betrieb und SRE	Deployment, Ausfallarten, Monitoring, Kapazität, Wiederherstellung	Deployment-Diagramme, Ausfallszenarien zur Laufzeit, SLOs, Runbooks, Abhängigkeitskarten	Abläufe nur für den Happy Path
Sicherheit	Vertrauensgrenzen, Datenflüsse, Identitäten, Angriffsfläche	Datenflussdiagramme mit markierten Vertrauensgrenzen, Authentifizierungs- und Autorisierungsabläufe, Bedrohungsmodell	Diagramme, die verbergen, wohin personenbezogene oder Zahlungsdaten gehen
Prüfer und Aufsichtsbehörden	Nachweise, dass Kontrollen existieren und eingehalten werden; Nachverfolgbarkeit	Versionierte Dokumente mit Ownern und Prüfdaten, ADRs, Zuordnung von Anforderungen zu Kontrollen	Informelle Skizzen ohne Datum, Owner oder Version

Zwei Gewohnheiten helfen bei jeder Zielgruppe. **Beginnen Sie mit der Antwort.** Beschäftigte Lesende sollten die Schlussfolgerung im ersten Satz erfahren, dann die stützenden Gründe, dann die Details; das ist die Struktur, die Barbara Minto Beratern als Pyramid Principle beigebracht hat, und es ist dieselbe Idee wie Fowlers umgekehrte Pyramide für ADRs.

Verwenden Sie ihre Worte. Für eine Finanzdirektorin ist „Eventual Consistency“ Rauschen; „Das Restaurant sieht eine neue Bestellung bis zu zwei Sekunden nach dem Kunden, und nichts geht verloren“ ist eine Tatsache, die sie abwägen kann.

Abwägungen erklären

Die meiste Architekturkommunikation ist im Kern die Erklärung einer Abwägung. Laien misstrauen Empfehlungen, die keinen Nachteil zu haben scheinen, und das zu Recht. Eine klare, konsequent verwendete Struktur macht Abwägungen leicht nachvollziehbar: die **Optionen** (einschließlich Nichtstun), die **Kriterien**, die für diese Zielgruppe zählen, wie jede Option abschneidet, die **Empfehlung** und ihre **Konsequenzen**, einschließlich dessen, was Sie aufgeben. Das ist die Anatomie einer Entscheidung aus Kapitel 1 und der Inhalt eines ADR, umgeformt für Zuhörende.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine Abwägung gegenüber dem Führungsteam erklären

Die Essenslieferplattform hatte zwei Ausfälle an Freitagabenden, verursacht durch einen langsamen Benachrichtigungsservice. Die Architektin hat in der Führungsrunde fünf Minuten, um die Zustimmung zu ADR-0012 (asynchrone Bestellereignisse) zu bekommen.

Zuerst die Antwort	„Ich empfehle, dass wir ändern, wie Bestellungen zwischen unseren Systemen fließen, damit ein Ausfall bei den Benachrichtigungen oder der Kurierdisposition Kunden nicht mehr am Bestellen hindern kann. Das kostet etwa sechs Teamwochen und ein paar hundert Pfund im Monat.“
Warum jetzt	„Am Freitag, dem 21. August, hat ein Ausfall der Benachrichtigungen 23 Minuten lang während der Freitagsspitze jede Bestellung blockiert. Bei etwa 40 Bestellungen pro Sekunde sind das rund 55.000 Bestellungen, die nicht aufgegeben werden konnten, gefolgt von Erstattungen und Beschwerden.“
Optionen in ihren Begriffen	„Wir könnten (A) das jetzige Design behalten und Sicherheitsgrenzen ergänzen, was billig ist, aber weiterhin zulässt, dass ein langsamer Service jede Bestellung verlangsamt; (B) die empfohlene Änderung umsetzen; oder (C) andere Systeme unsere Bestelldaten direkt lesen lassen, was schnell geht, aber jede künftige Änderung riskanter macht.“
Was wir aufgeben	„Restaurants sehen neue Bestellungen ein oder zwei Sekunden nach der Bestätigung für den Kunden, und wir müssen einen weiteren Managed Service betreiben. Wir halten beides für vertretbar; der Betrieb stimmt zu.“
Die nötige Entscheidung	„Ich brauche die Zustimmung, in diesem Quartal sechs Teamwochen aufzuwenden, was das Treueprogramm-Feature um zwei Wochen verzögert.“

Keine Technologie wird genannt: „Message Broker“ und „Transactional Outbox“ bleiben im ADR. Die Bestellzahl (40 Bestellungen pro Sekunde über 23 Minuten) wird vor dem Meeting mit echten Daten abgeglichen.

Die Kosten der Empfehlung und die Alternativen werden klar benannt, sodass die Führungskräfte in der Sache widersprechen können statt aus Misstrauen.

Die einseitige Architekturübersicht

Jedes System profitiert von einer einseitigen Architekturübersicht: einer einzigen Seite, die neue Teammitglieder, Führungskräfte oder Reviewer aus einem anderen Team in fünf Minuten lesen können. Sie ersetzt nicht die vollständige Dokumentation, sondern ist ihre Eingangstür. Eine nützliche Struktur ist: was das System tut und für wen (zwei Sätze); die drei wichtigsten Qualitätsziele; ein Systemkontextdiagramm; ein Containerdiagramm oder eine Liste der

wichtigsten Bausteine; die etwa fünf Entscheidungen, die alles prägen (je eine Zeile, mit Link zu den ADRs); die größten Risiken; und wo man mehr findet. Die arc42-lite-Seite in arc42 und Vorlagen ist eine solche Übersicht.

Die Disziplin, die Architektur auf eine Seite zu bringen, ist schon für sich wertvoll. Wenn sich die Hauptideen nicht auf einer Seite darstellen lassen, sind sie vermutlich auch dem Team nicht klar.

Storytelling: einer Anfrage folgen

Menschen merken sich Geschichten besser als Strukturen. Eine wirkungsvolle Technik, um ein beliebiges System zu erklären, ist es, einer Anfrage zu folgen: Wählen Sie ein einzelnes, konkretes, wichtiges Ereignis, etwa einen Kunden, der eine Bestellung aufgibt, und gehen Sie Schritt für Schritt durch, was damit quer durch die Architektur geschieht. Ein statisches Kastendiagramm zeigt, was existiert; die Geschichte zeigt, warum es existiert und wie die Teile zusammenarbeiten. Sie bringt auch die interessanten Momente ganz natürlich zum Vorschein: wo der Kunde wartet, wo Daten eine Vertrauensgrenze überschreiten, was passiert, wenn der Zahlungsanbieter langsam ist.

Wählen Sie die Geschichte passend zur Zielgruppe. Für Entwicklungsteams erst den Happy Path, dann einen Fehlerfall („jetzt läuft beim Zahlungsanbieter ein Timeout ab“). Für den Betrieb den Ausfall und die Wiederherstellung. Für Führungskräfte das Kundenerlebnis mit nur den geschäftlich relevanten Momenten. Nummerieren Sie die Schritte im Diagramm, damit Sie darauf zeigen können, und lassen Sie dasselbe Diagramm auf dem Bildschirm, während Sie sprechen, statt zwischen Folien hin- und herzuspringen.

Einer Bestellung durch die Plattform folgen



Durchgezogene Pfeile: synchrone Aufrufe. Gestrichelte Pfeile: asynchrone Events. Zahlen: Ablauf der Geschichte.

Ein nummeriertes Story-Diagramm: Die Schritte 1 bis 4 geschehen, während der Kunde wartet; die Schritte 5 bis 8 laufen asynchron und können die Bestellung nicht blockieren.

Gut am Whiteboard arbeiten und skizzieren

Viel Architekturkommunikation geschieht live, an einem Whiteboard oder auf einer gemeinsamen Online-Zeichenfläche, und ein paar Gewohnheiten machen Skizzen weit nützlicher. **Sagen Sie, was Sie zeichnen, bevor Sie es zeichnen** („Das ist die Container-Ebene: Dinge, die wir getrennt deployen“), damit das Publikum die Abstraktionsebene kennt. **Beginnen Sie mit dem Kontext**: die Nutzer und externen Systeme am Rand, dann das Innere. **Beschriften Sie jeden Kasten und jede Linie** beim Zeichnen, einschließlich der Richtung und dessen, was fließt; ein unbeschrifteter Pfeil wird von drei Personen auf drei verschiedene Arten gelesen. **Zeichnen Sie eine kleine Legende** in eine Ecke, wenn Sie mehr als eine Art von Linie oder Kasten verwenden. **Lassen Sie Platz**, denn die besten Ideen kommen mitten in der Diskussion und brauchen einen Ort.

Skizzen sind Denkwerkzeuge und dürfen weggeworfen werden, die Schlussfolgerungen aber nicht. Fotografieren Sie das Whiteboard und machen Sie innerhalb eines Tages aus allem, was wichtig sein wird, ein richtiges Diagramm, ein ADR oder eine Änderung an der Dokumentation. Simon Brown, der das C4-Modell geschaffen hat, beobachtete, dass die Whiteboard-Diagramme vieler Teams gerade deshalb mehrdeutige Kästen und Linien sind, weil ihnen ein vereinbarter Satz von Abstraktionen fehlt; die C4-Ebenen am Whiteboard zu verwenden ist einer der einfachsten Wege, das zu beheben.

Designdokumente und RFCs

Viele Entwicklering-Organisationen verwenden ein schriftliches Designdokument, oft RFC (Request for Comments) genannt, um eine bedeutende Änderung vorzuschlagen, bevor sie gebaut wird. Der Name greift die RFC-Reihe des Internets auf, die im April 1969 mit RFC 1 von Steve Crocker begann. Die Autorin oder der Autor beschreibt das Problem, den Kontext, die Ziele und Nicht-Ziele, das vorgeschlagene Design, die erwogenen Alternativen und die übergreifenden Anliegen wie Sicherheit, Datenschutz und Observability; das Dokument wird an die Betroffenen verteilt, die schriftlich kommentieren; und es wird genehmigt oder überarbeitet. Malte Ubls viel gelesene Beschreibung von Design Docs bei Google (2020) nennt genau diese Abschnitte und schlägt zehn bis zwanzig Seiten für große Projekte und ein bis drei Seiten für kleine „Mini Design Docs“ vor. Gergely Orosz beschrieb, wie Uber von seinen frühen Jahren an einen RFC-Prozess nutzte, um Wissen zu verbreiten und Probleme früh zu erkennen (2018).

Amazon treibt das Schreiben weiter: Sein Aktionärsbrief von 2017 erklärt, dass das Unternehmen in Meetings keine Folienpräsentationen verwendet, sondern „narrativ strukturierte sechsseitige Memos“, die die Teilnehmenden zu Beginn still lesen. Der gemeinsame Nutzen all dieser Praktiken ist, dass Schreiben klares Denken erzwingt, schriftliche Kommentare auch ruhigeren und entfernt arbeitenden Kollegen eine Stimme geben und das Dokument zu einer Aufzeichnung wird. Auch die Risiken sind gemeinsam: lange Dokumente, die niemand zu Ende liest, Review-Prozesse, die zu bürokratischen Hürden werden, und Design Docs, die nach der Umsetzung nie aktualisiert werden.

Design Docs und ADRs ergänzen sich. Ein Design Doc untersucht ein Problem und schlägt ein ganzes Design vor, das oft mehrere Entscheidungen umfasst; sobald es genehmigt ist, werden die bedeutenden Entscheidungen darin als ADRs festgehalten, die kürzer, dauerhaft und indexiert sind. Eine vernünftige Regel lautet: Design Doc zum Denken und Überzeugen, ADR zum Erinnern.

Bei einem Architektur-Review präsentieren

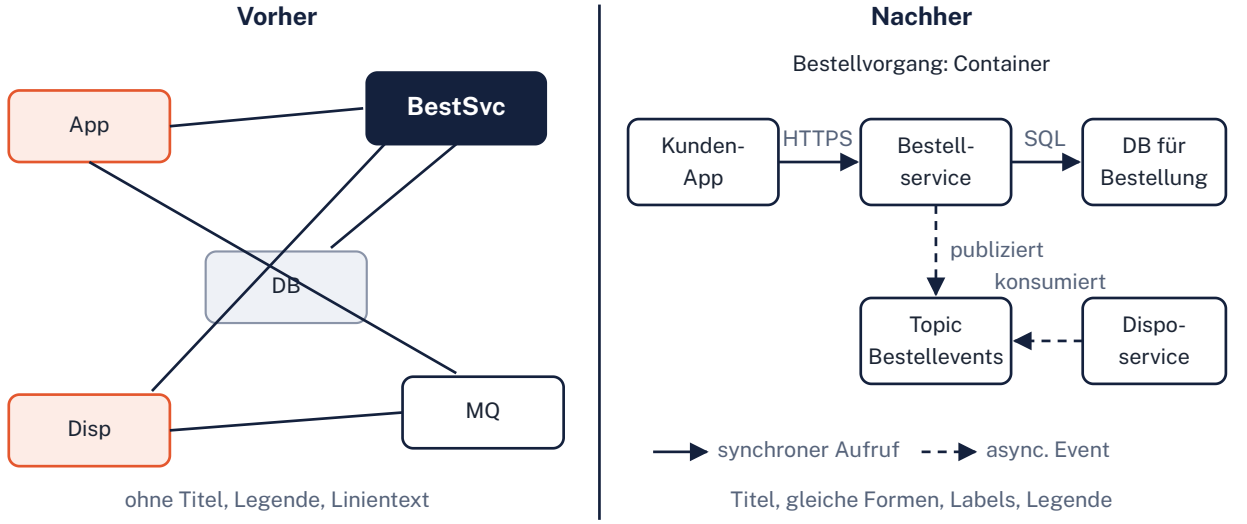
Ein Architektur-Review ist ein Meeting, in dem andere erfahrene Personen ein Design untersuchen, um Risiken zu finden, bevor sie teuer werden. Strukturierte Methoden wie ATAM und leichtgewichtigere Reviews werden in Kapitel 11: Leichtgewichtige Reviews behandelt; hier geht es darum, wie man gut präsentiert. Senden Sie das Material mindestens zwei Tage vorher: die einseitige Übersicht, die wichtigsten Diagramme, die relevanten ADRs und die Qualitätsszenarien. Sagen Sie zu Beginn, was Sie vom Review wollen: eine Genehmigung, eine Liste von Risiken oder eine Wahl zwischen zwei Optionen. Präsentieren Sie die Treiber vor dem Design, denn ein Reviewer kann ein Design nicht beurteilen, ohne zu wissen, welche Qualitäten es erreichen muss. Gehen Sie ein oder zwei Szenarien mit der Technik „einer Anfrage folgen“

durch, einschließlich eines Fehlerfalls. Zeigen Sie die Abwägungen, die Sie getroffen haben, und die Risiken, die Sie kennen; ein Reviewer, der ein Risiko findet, das Sie bereits aufgelistet haben, vertraut Ihnen mehr, nicht weniger. Halten Sie jeden angesprochenen Punkt mit einer verantwortlichen Person fest und senden Sie die Liste danach herum.

Regeln für die visuelle Gestaltung von Diagrammen

Diagramme kommunizieren, bevor jemand ein Wort liest, deshalb zählt ihre visuelle Gestaltung. Die folgenden Regeln stammen aus den C4-Notationsempfehlungen in Das C4-Modell und aus allgemeinem Informationsdesign. Das Vorher-nachher-Diagramm wendet sie auf dieselben fünf Elemente an.

Dasselbe Diagramm, vorher und nachher



Links tragen Farben und Abkürzungen keine Bedeutung, und Linien kreuzen sich unbeschriftet; rechts ein Titel, einheitliche Formen, ein Fluss von links nach rechts, beschriftete Pfeile und eine Legende.

Regeln für klare Architekturdiagramme

1. **Geben Sie jedem Diagramm einen Titel**
   Nennen Sie Typ und Umfang, etwa „Container: Essenslieferplattform“, damit Lesende die Abstraktionsebene kennen.

2. **Verwenden Sie eine Notation konsequent**
   Dieselbe Form für dieselbe Art von Ding in allen Diagrammen; mischen Sie C4- und UML-Symbole nicht in einem Diagramm, ohne es zu sagen.

3. **Fügen Sie eine Legende hinzu**
   Erklären Sie jede Form, Farbe, Linienart und Pfeilspitze, die Bedeutung trägt. Wenn Sie sie nicht erklären können, entfernen Sie sie.

4 **Begrenzen Sie Farben und geben Sie ihnen Bedeutung**
Zwei oder drei Farben, jede mit einer Bedeutung (extern, neu, gefährdet). Verlassen Sie sich nie allein auf Farbe: Diagramme werden schwarz-weiß gedruckt, und viele Lesende haben eine Farbsehschwäche.

5 **Fluss von links nach rechts oder von oben nach unten**
Folgen Sie der Leserichtung Ihres Publikums, mit Anfragen in eine Richtung, damit das Auge der Geschichte folgt.

6 **Beschriften Sie jedes Element und jeden Pfeil**
Namen ohne unerklärte Abkürzungen; Pfeile in eine Richtung, mit dem, was fließt, und zwischen Containern mit dem Protokoll.

7 **Vermeiden Sie sich kreuzende Linien, wo möglich**
Ordnen Sie zuerst die Kästen neu an; bleiben Kreuzungen, teilen Sie das Diagramm. Ein Diagramm mit vielen Kreuzungen will meist zu viel zeigen.

8 **Bleiben Sie bei einem Bildschirm**
Höchstens etwa ein oder zwei Dutzend Elemente. Größere Bilder gehören auf eine tiefere Ebene oder in mehrere Diagramme.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: entscheiden, wenn nicht alle zustimmen

Die Idee, die hier die meiste Sorgfalt verlangt, ist Rough Consensus: Eine Entscheidung kann getroffen werden, sobald jeder Einwand fair angehört und begründet beantwortet wurde, auch wenn manche weiterhin eine andere Option bevorzugen. Stellen Sie sich vor, wie ein Vorschlag ablaufen könnte.

- Das Bestellteam eröffnet ADR-0012, Bestellereignisse über einen Message Broker zu veröffentlichen, als vorgeschlagenen Record in einem Pull Request. Die Teams für Restaurants, Kuriere und Benachrichtigungen sowie das SRE-Team werden um Kommentare gebeten.
- Zwei Einwände kommen. Ein Engineer aus dem Benachrichtigungsteam möchte die synchronen Aufrufe mit Timeouts und Wiederholungen beibehalten, die Option mit der wenigsten neuen Infrastruktur. Eine Entwicklerin aus dem Restaurantteam befürchtet, dass Tablets neue Bestellungen verspätet anzeigen.
- Jeder Einwand wird schriftlich beantwortet. Der Record erklärt, dass bei synchronen Aufrufen ein langsamer Service weiterhin jede Bestellung verzögert, und räumt ein, dass ein Restaurant-Tablet ein oder zwei Sekunden hinter der Bestätigung für den Kunden zurückliegen kann: Kosten, die er festhält.
- Der Engineer aus dem Benachrichtigungsteam ist weiterhin nicht überzeugt. Sein Einwand wurde fair angehört und beantwortet, aber nicht berücksichtigt: Das Design ändert sich nicht, um ihm entgegenzukommen. Für Rough Consensus genügt das.
- Vor der Entscheidung fragen die Entscheidenden: „Kann jemand damit nicht leben, und warum?“ Kein neuer blockierender Punkt kommt auf. Wäre ein Einwand unbeantwortet geblieben, gäbe es noch keinen Rough Consensus, egal wie viele dafür wären.
- Der Record wird angenommen, mit der abweichenden Meinung des Engineers und ihren Gründen darin. Der Engineer verpflichtet sich, es zum Laufen zu bringen; behält er später recht, zeigt der Record, was zu überdenken ist.
- **Die Lehre:** Ziel ist, jeden Einwand zu beantworten, nicht, alle zur Zustimmung zu bringen. Halten Sie die abweichende Meinung mit der Entscheidung fest und legen Sie vorab fest, wer entscheidet, wenn kein Konsens erreicht wird.

Mit Meinungsverschiedenheiten umgehen und Konsens aufbauen

Meinungsverschiedenheiten über Architektur sind normal und gesund: Meist bedeuten sie, dass Menschen unterschiedliche Risiken sehen. Ziel ist nicht zu gewinnen, sondern eine Entscheidung zu erreichen, die diejenigen, die damit leben müssen, verstehen und mittragen. Mehrere Praktiken helfen.

Trennen Sie Interessen von Positionen. „Wir müssen Kafka verwenden“ ist eine Position; „Ich mache mir Sorgen, dass wir bei Deployments Ereignisse verlieren“ ist ein Interesse, und Interessen lassen sich oft auf mehrere Arten erfüllen. **Vereinbaren Sie die Kriterien, bevor Sie Optionen vergleichen**, wie in einer gewichteten Entscheidungsmatrix; die meisten hitzigen Debatten sind eigentlich Uneinigkeit darüber, was zählt, und sie kühlen ab, sobald das ausgesprochen ist. **Klären Sie Unsicherheit mit Belegen**: Ein zweitägiger Spike oder ein Lasttest entscheidet viele Debatten, die Meetings nicht entscheiden können. **Schreiben Sie es auf**: Ein vorgeschlagenes ADR oder Design Doc lässt alle in ihrem eigenen Tempo kommentieren und gibt ruhigeren Kollegen eine Stimme.

Streben Sie Zustimmung statt Einstimmigkeit an. RFC 7282 der IETF (Pete Resnick, 2014) beschreibt Rough Consensus: Eine Entscheidung ist erreicht, wenn jeder Einwand behandelt wurde, also erwogen und beantwortet, wenn auch nicht unbedingt berücksichtigt. Die Frage „Kann jemand damit nicht leben, und warum?“ bringt blockierende Punkte schneller ans Licht als die Frage, ob alle zustimmen. Ist Konsens nicht möglich, muss jemand entscheiden, und es sollte vorab klar sein, wer. Jeff Bezos’ Aktionärsbrief von 2016 machte *Disagree and Commit* populär: Menschen äußern ihre abweichende Meinung offen und stehen dann voll hinter der Entscheidung, sobald sie gefallen ist. Halten Sie die abweichende Meinung und die Gründe im ADR fest; behalten die Abweichenden recht, zeigt der Record, was zu überdenken ist und warum.

Zwei Fehlmuster verdienen eine Warnung. Die Architektin, die jede Diskussion kraft ihres Dienstalters gewinnt, hört nichts mehr von Problemen. Das Team, das ohne Einstimmigkeit nicht entscheiden kann, lässt wichtige Entscheidungen durch Untätigkeit fallen, das Gegenteil des Last Responsible Moment.

Checkliste: eine Architekturpräsentation

Verwenden Sie sie vor jeder Präsentation von Architektur, vom fünfminütigen Update für die Führung bis zum formalen Review.

- [ ] **Zielgruppe und Ergebnis benannt**
  Sie können in einem Satz sagen, wer im Raum sein wird und was diese Personen danach entscheiden oder tun sollen.
- [ ] **Zuerst die Antwort**
  Die Empfehlung oder Kernbotschaft steht auf der ersten Folie oder im ersten Absatz.
- [ ] **Treiber vor Design**
  Die Geschäftsziele, Qualitätsziele und Randbedingungen werden vor der Lösung gezeigt.
- [ ] **Abstraktionsebene passt zur Zielgruppe**
  Kontextebene für Führungskräfte, Container- und Komponentenebene für Engineers; nicht mehr als zwei Ebenen in einer Sitzung.
- [ ] **Eine Geschichte von Anfang bis Ende erzählt**
  Ein Durchgang nach dem Prinzip „einer Anfrage folgen“, für technisches Publikum mit einem Fehlerfall.
- [ ] **Abwägungen und Risiken ehrlich gezeigt**
  Optionen, was jede kostet und was Sie aufgeben; bekannte Risiken aufgelistet, bevor jemand anderes sie findet.
- [ ] **Jedes Diagramm hat Titel, Legende und Beschriftungen**
  Konsistente Notation, wenige Farben mit Bedeutung, lesbar aus der letzten Reihe.
- [ ] **Material vorab verschickt**
  Für Reviews und Entscheidungen gehen die Übersicht und die ADRs mindestens zwei Tage vorher hinaus, sodass das Meeting der Diskussion dient.
- [ ] **Das Anliegen ist ausdrücklich**
  Genehmigung, Risiken, eine Wahl zwischen Optionen oder einfach Kenntnisnahme; sagen Sie, was davon, und bis wann.
- [ ] **Nachbereitung festgehalten**
  Fragen, Einwände und Aufgaben mit Verantwortlichen festgehalten; Entscheidungen in ADRs überführt; Dokumentation aktualisiert.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Welche Frage sollten Sie beantworten, bevor Sie eine Architekturerklärung vorbereiten?**

Wer die Zielgruppe ist und was sie danach entscheiden oder anders machen soll; Inhalt, Abstraktionsebene und Vokabular ergeben sich daraus.

▼ **Warum eine Geschichte nach dem Prinzip „einer Anfrage folgen“ erzählen, statt nur ein Kastendiagramm zu zeigen?**

Ein Kastendiagramm zeigt, was existiert; die Geschichte zeigt, wie und warum die Teile zusammenarbeiten, und bringt Wartezeiten, Vertrauensgrenzen und Fehlerstellen ans Licht, die ein statisches Bild verbirgt.

▼ **Wie unterscheiden sich ein Designdokument und ein ADR in ihrem Zweck?**

Ein Designdokument untersucht ein Problem und schlägt vor dem Bau ein ganzes Design zur Kommentierung vor; ein ADR ist ein kurzer, dauerhafter Record einer Entscheidung. Genehmigte Design Docs bringen typischerweise mehrere ADRs hervor.

▼ **Was bedeutet Rough Consensus in RFC 7282?**

Jeder Einwand wurde erwogen und beantwortet, wenn auch nicht unbedingt berücksichtigt; es ist weder Einstimmigkeit noch eine Mehrheitsabstimmung.

▼ **Probieren Sie es aus: Eine Architektin sagt zur Unternehmensleitung: „Wir brauchen eine Transactional Outbox und einen Broker, um unseren synchronen Bestellpfad zu entkoppeln.“ Schreiben Sie das für die Leitung in etwa drei Sätzen um, einschließlich der Empfehlung, der Kosten und dessen, was sie entscheiden muss.**

„Ich empfehle, die Art zu ändern, wie Bestellungen zwischen unseren Systemen weitergegeben werden, damit eine Störung bei den Benachrichtigungen Kunden nicht mehr am Bestellen hindern kann, wie es kürzlich zweimal passiert ist. Das würde etwa sechs Teamwochen und geringe monatliche Betriebskosten erfordern; der Haken ist, dass manche Aktualisierungen ein paar Sekunden später ankämen. Bitte genehmigen Sie das heute.“ Die Antwort steht am Anfang, in geschäftlichen Begriffen, mit klar genannten Kosten und Nachteilen und ohne Fachjargon.

QUELLEN

- C4 model: notation: https://c4model.com/diagrams/notation
- Design Docs at Google, Malte Ubl (2020): https://www.industrialempathy.com/posts/design-docs-at-google/

- Scaling Engineering Teams via RFCs: Writing Things Down, Gergely Orosz: https://blog.pragmaticengineer.com/scaling-engineering-teams-via-writing-things-down-rfcs/
- RFC 7282: On Consensus and Humming in the IETF: https://www.rfc-editor.org/rfc/rfc7282.html
- Amazon 2017 letter to shareholders: https://www.aboutamazon.com/news/company-news/2017-letter-to-shareholders

Weiterführende Literatur

- *Documenting Software Architectures: Views and Beyond, 2nd edition*, Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord and Judith Stafford (2010), Addison-Wesley. Die gründlichste Darstellung der Architekturdokumentation: Sichten, Stile, die sieben Regeln für gute Dokumentation und eine Dokumentationsvorlage.
  https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/
- *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, 2nd edition*, Nick Rozanski and Eóin Woods (2012), Addison-Wesley. Ein praktischer Katalog von sieben Blickwinkeln und von Perspektiven wie Sicherheit und Performance, ausgerichtet auf große Informationssysteme.
  https://www.viewpoints-and-perspectives.info/
- *Architectural Blueprints: The "4+1" View Model of Software Architecture*, Philippe Kruchten (1995), IEEE Software 12(6). Der kurze Artikel, der die Architekturbeschreibung mit mehreren Sichten und die szenariobasierte Validierung populär machte.
  https://arxiv.org/abs/2006.04975
- *ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description*, ISO/IEC JTC 1/SC 7 and IEEE (2022), ISO. Die internationale Norm, die Stakeholder, Anliegen, Blickwinkel, Sichten und Modellarten definiert.
  https://www.iso.org/standard/74393.html
- *The C4 Model: Visualizing Software Architecture*, Simon Brown (2026), O'Reilly Media. Die Darstellung des Urhebers selbst zu den Abstraktionen, Diagrammen und der Notation von C4; die Website behandelt das Wesentliche kostenlos.
  https://c4model.com/
- *UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edition*, Martin Fowler (2003), Addison-Wesley. Ein kurzer, praxisnaher Leitfaden zu den Teilen der UML, die man kennen sollte, und die Quelle der Unterscheidung zwischen Skizze, Bauplan und Programmiersprache.
- *Living Documentation: Continuous Knowledge Sharing by Design*, Cyrille Martraire (2019), Addison-Wesley. Wie man Dokumentation aus dem System selbst ableitet und handgeschriebenes Material auf das beschränkt, was nur Menschen liefern können.

Glossar

4+1-Sichtenmodell
Der Ansatz von Philippe Kruchten aus dem Jahr 1995, der ein System durch eine logische Sicht, eine Prozesssicht, eine Entwicklungssicht und eine physische Sicht beschreibt, verbunden durch eine kleine Zahl von Szenarien.

Advice Process
Eine dezentrale Art der Entscheidungsfindung, bei der jede Person entscheiden darf, sofern sie vorher den Rat der Betroffenen und der zuständigen Fachleute einholt.

Aktivitätsdiagramm
Ein UML-Diagramm, das einen Ablauf als Aktionen, Entscheidungen, parallele Verzweigungen und Zusammenführungen zeigt, oft in Schwimmbahnen (Swimlanes) unterteilt, die zeigen, wer was tut.

arc42
Eine kostenlose Vorlage mit zwölf Abschnitten zur Dokumentation und Kommunikation von Softwarearchitektur, entwickelt von Gernot Starke und Peter Hruschka.

Architekturbeschreibungsrahmen
In ISO/IEC/IEEE 42010:2022 eine etablierte Menge von Konventionen zur Erstellung von Architekturbeschreibungen, etwa Blickwinkel (Viewpoints), Vorlagen und Regeln.

Baustein
In arc42 jeder Teil der statischen Struktur eines Systems, etwa ein Subsystem, Service, eine Komponente oder ein Modul, dargestellt in der Bausteinsicht.

Benötigte Schnittstelle
In einem UML-Komponentendiagramm eine Schnittstelle, die eine Komponente von einer anderen Komponente bereitgestellt braucht, dargestellt als Halbkreis (Socket).

Bereitgestellte Schnittstelle
In einem UML-Komponentendiagramm eine Schnittstelle, die eine Komponente anderen anbietet, dargestellt als kleiner Kreis (Ball) an einem Stiel.

Betrachtungsgegenstand (Entity of Interest)
In ISO/IEC/IEEE 42010:2022 das, dessen Architektur beschrieben wird: ein Softwaresystem, ein System von Systemen, eine Produktlinie oder ein Unternehmen.

C4-Modell
Der Ansatz von Simon Brown für Architekturdiagramme: eine Hierarchie aus Systemkontext-, Container-, Komponenten- und Codediagrammen sowie einigen ergänzenden Diagrammen.

Container (C4)
Im C4-Modell eine separat ausführbare oder deploybare Einheit, die Code ausführt oder Daten speichert, etwa eine Web-App, eine mobile App, ein Service oder eine Datenbank. Es ist kein Docker-Container.

Containerdiagramm
Ein C4-Diagramm, das die Apps, Services und Datenspeicher innerhalb eines Systems mit ihren Technologien und ihrer Kommunikation zeigt.

Deployment-Diagramm
Ein Diagramm (in C4 oder UML), das zeigt, wie Software in einer Umgebung auf Infrastruktur abgebildet wird, einschließlich Knoten, Replikaten und Netzwerkpfaden.

Designdokument (RFC)
Ein schriftlicher Vorschlag für eine wesentliche Änderung, der Problem, Ziele, Design und Alternativen behandelt und vor der Umsetzung zur Kommentierung in Umlauf gebracht wird.

Diagrams as Code
Diagramme als Text in einer kleinen Sprache schreiben, etwa PlantUML, Mermaid, D2 oder Structurizr DSL, die ein Werkzeug anordnet und zeichnet.

Docs as Code
Dokumentation mit denselben Werkzeugen und Abläufen erstellen wie Software: Textdateien in der Versionskontrolle, Code-Review, automatisierte Prüfungen und Veröffentlichung.

Domänenmodell
Ein kleines Diagramm oder eine Beschreibung der zentralen Geschäftskonzepte eines Systems, ihrer Attribute und ihrer Beziehungen, gemeinsam genutzt von Entwicklungsteam und Fachleuten der Domäne.

Dynamisches Diagramm
Ein ergänzendes C4-Diagramm, das zeigt, wie Elemente in einem Szenario zur Laufzeit zusammenarbeiten, meist als nummerierte Interaktionen.

Einer Anfrage folgen
Ein System erklären, indem man ein einzelnes konkretes Ereignis, etwa eine Bestellung, Schritt für Schritt durch jeden Teil verfolgt, den es berührt.

Einseitige Architekturübersicht
Eine einzelne Seite mit Zweck, Qualitätszielen, Kontext, wichtigsten Bausteinen, zentralen Entscheidungen und Risiken eines Systems, als Einstieg in die ausführlichere Dokumentation.

Entscheidungsprotokoll (Decision Log)
Die geordnete Sammlung der ADRs eines Systems, meist mit einem Index, der Nummer, Titel, Status und Datum jedes Eintrags auflistet.

Entscheidungstreiber
Eine Kraft oder ein Kriterium, etwa ein Qualitätsziel oder eine Randbedingung, die eine Architekturentscheidung erfüllen muss und anhand derer Optionen verglichen werden.

Guard
Eine Bedingung in eckigen Klammern an einem Übergang oder Fluss, die wahr sein muss, damit er genommen wird, etwa [code matches].

Just-enough-Dokumentation
Dokumentation, deren Umfang sich danach richtet, was ihre Leser tatsächlich brauchen, sodass mehr zu schreiben und zu pflegen mehr kosten würde, als es einbringt.

Komponente (C4)
Im C4-Modell eine Gruppe zusammengehöriger Funktionalität hinter einer klaren Schnittstelle, die innerhalb eines Containers läuft und mit ihm deployt wird.

Komponenten-und-Konnektoren-Sicht
Eine Sicht auf das, was zur Laufzeit existiert, etwa Services, Clients, Datenbanken und Queues, und auf die Konnektoren (Aufrufe, Nachrichten, gemeinsame Daten) zwischen ihnen.

Korrespondenz
Eine festgehaltene Beziehung zwischen Elementen einer Architekturbeschreibung, zum Beispiel zwischen einem Container in einer Sicht und dem Server, auf dem er in einer anderen läuft.

Lebende Dokumentation
Dokumentation, die sich mit dem beschriebenen System weiterentwickelt, oft weil sie aus dem System selbst erzeugt oder gegen es geprüft wird.

Lebenslinie
In einem Sequenzdiagramm die gestrichelte senkrechte Linie unter einem Teilnehmer, entlang der seine Nachrichten und Aktivierungen in zeitlicher Reihenfolge dargestellt werden.

MADR
Markdown Architectural Decision Records: eine weit verbreitete ADR-Vorlage mit Entscheidungstreibern, betrachteten Optionen, dem Ergebnis, den Konsequenzen und der Bestätigung.

Modellart (Model Kind)
In ISO/IEC/IEEE 42010 die Konventionen für eine Modellart, etwa ein UML-Sequenzdiagramm oder ein C4-Containerdiagramm.

Modellierungswerkzeug
Ein Werkzeug, das Elemente und Beziehungen in einem Modell speichert und Diagramme als Sichten darauf erzeugt, wobei alle Sichten konsistent bleiben.

Modulsicht
Eine Sicht auf die Implementierungseinheiten (Pakete, Bibliotheken, Schichten) und ihre Abhängigkeiten, wie sie zur Entwicklungszeit bestehen.

Multiplizität
In einem UML-Klassendiagramm, wie viele Objekte an jedem Ende einer Assoziation beteiligt sein dürfen, etwa 1, 0..1, 0..* oder 1..*.

Perspektive
Im Ansatz von Rozanski und Woods ein Qualitätsanliegen wie Sicherheit oder Performance, das quer über alle Sichten angewendet wird, statt als eigene Sicht gezeichnet zu werden.

Querschnittliches Konzept
Eine Regel oder Lösung, die für viele Teile eines Systems gilt, etwa Sicherheit, Fehlerbehandlung, Logging oder Persistenz.

Rough Consensus
Eine Entscheidung, die erreicht ist, wenn jeder Einwand geprüft und beantwortet, aber nicht unbedingt berücksichtigt wurde; beschrieben in IETF RFC 7282.

Sequenzdiagramm
Ein UML-Diagramm, das die zwischen Teilnehmern ausgetauschten Nachrichten für ein Szenario im zeitlichen Verlauf zeigt, wobei die Zeit von oben nach unten verläuft.

Sichtkomponente
In ISO/IEC/IEEE 42010:2022 ein Bestandteil einer Sicht, etwa ein Diagramm, eine Tabelle oder ein Modell, der von einer Modellart bestimmt wird. Die Ausgabe von 2011 nannte ihn Architekturmodell.

Software Design Description (SDD)
Ein Dokument, das einen Softwareentwurf im Detail beschreibt und dessen Inhalt IEEE Std 1016-2009 festlegt; verbreitet in vertragsgetriebenen Projekten.

Softwaresystem (C4)
Im C4-Modell die höchste Abstraktionsebene: etwas, das seinen Nutzern einen Wert liefert, meist im Besitz eines Teams oder einer Organisation.

Structurizr
Simon Browns Werkzeuge für das C4-Modell, bei denen ein textbasiertes Modell viele konsistente Diagramme erzeugt.

Systemkontextdiagramm
Ein C4-Diagramm, das ein System als einzelnen Kasten zeigt, zusammen mit den Personen, die es nutzen, und den anderen Systemen, mit denen es interagiert.

Unified Modeling Language (UML)

Eine universelle grafische Modellierungssprache, standardisiert von der Object Management Group. Die aktuelle Version ist 2.5.1 (2017).

Y-Statement

Ein Entscheidungsprotokoll in einem Satz von Olaf Zimmermann: Im Kontext von …, angesichts …, haben wir uns entschieden für … und gegen …, um … zu erreichen, und nehmen in Kauf, dass ….

Zeichenwerkzeug

Ein Diagrammwerkzeug, das nur Formen, Linien und Text kennt, sodass dasselbe Element, in zwei Diagrammen gezeichnet, keine Verbindung zwischen ihnen hat.

Zuordnungssicht

Eine Sicht, die Software auf Dinge außerhalb von ihr abbildet: Infrastruktur (Deployment), Dateien und Builds (Implementierung) oder Teams (Arbeitszuteilung).

Zustandsdiagramm

Ein UML-Diagramm, das die Zustände zeigt, in denen sich ein Objekt befinden kann, und die Übergänge zwischen ihnen, die jeweils durch ein Ereignis ausgelöst werden.