

BE SOFTWARE ARCHITECTURE WISE · KAPITEL 1

Was Softwarearchitektur ist

Dieses Kapitel erklärt, was Softwarearchitektur ist und warum sie zählt. Es vergleicht die Standarddefinitionen, grenzt Architektur von Design ab, zeigt an echten Vorfällen, wie strukturelle Entscheidungen Kosten, Qualität und Risiko bestimmen, und beschreibt, was Architekten tun. Anschließend behandelt es Stakeholder und ihre Anliegen, Entscheidungen und Abwägungen, technische Schulden und Erosion sowie den Platz der Architektur in der modernen iterativen Softwareauslieferung. Es richtet sich an Entwicklerinnen und Entwickler, die in Architekturverantwortung hineinwachsen, und an alle, die mit Architekten zusammenarbeiten, und setzt nur die Bausteine aus Kapitel 0 voraus.

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

AUF EINEN BLICK

Das lernen Sie

- Die Definitionen von Softwarearchitektur nach ISO/IEC/IEEE 42010:2022 und nach dem SEI sowie ihre gemeinsamen Ideen erklären.
- Modul-, Komponenten-und-Konnektoren- und Zuordnungsstrukturen unterscheiden sowie eine Architektur von ihrer Beschreibung.
- Entscheidungen anhand von Umkehrbarkeit, Reichweite und Wirkung auf Qualitätsmerkmale als Architektur- oder Designentscheidungen einordnen.
- Analysieren, wie strukturelle Entscheidungen zu echten Vorfällen wie bei Knight Capital und HealthCare.gov beigetragen haben.
- Die Verantwortlichkeiten, Fähigkeiten und Arten von Architekten beschreiben und wie Architekten praktisch am Ball bleiben.
- Stakeholder und ihre Anliegen erkennen und widersprüchliche Anliegen explizit machen.
- Architekturentscheidungen als Abwägungen bewerten und dabei Umkehrbarkeit und Zeitpunkt berücksichtigen.
- Technische Schulden, Erosion und Drift erklären und wie man sie über den Lebenszyklus erkennt und steuert.

Bevor Sie beginnen

Kapitel 0 oder praktische Grundkenntnisse über Komponenten, Datenbanken, APIs, Netzwerke und die Cloud.

Dauer

Etwa 3 Stunden Lesezeit; 45 Minuten für den Test.

Schlüsselbegriffe

Softwarearchitektur · Architekturbeschreibung · Struktur · Architekturrelevant · Qualitätsmerkmal · Änderungskosten · Stakeholder · Anliegen · Abwägung (Trade-off) · Umkehrbarkeit · Technische Schulden · Architekturerosion · Fitness Function · Gesetz von Conway

Warum dieses Kapitel am Anfang steht

Jedes Softwaresystem hat eine Architektur, ob sie jemand entworfen hat oder nicht. Sie ist die Menge der grundlegenden Strukturen, aus denen das System besteht (seine Hauptteile, wie sie verbunden sind und welche Eigenschaften sie haben), und der bedeutsamen Entscheidungen, die sie hervorgebracht haben. Diese Strukturen entscheiden weit mehr, als den meisten bewusst ist: wie schnell und zuverlässig das System ist, wie viel sein Betrieb kostet, wie viele Teams gleichzeitig daran arbeiten können und vor allem, wie teuer es sein wird, es im nächsten Jahr zu ändern. Features sind das, was ein System tut; die Architektur entscheidet weitgehend darüber, wie gut es das tut und was es kostet, etwas anderes zu tun.

Kapitel 0 hat die Bausteine vorgestellt: Programme, Datenspeicher, Netzwerke, APIs, Server und die Cloud. Dieses Kapitel tritt einen Schritt zurück und fragt, was es bedeutet, diese Bausteine bewusst anzuordnen, wer das tut und warum sich der Aufwand lohnt. Es gibt Ihnen das Vokabular, auf dem der restliche Kurs aufbaut, und eine Denkweise, die Sie auf jedes System anwenden können, von einer kleinen Buchungs-App bis zu einer nationalen Zahlungsplattform.

Was Architektur ist und was nicht

In den ersten beiden Abschnitten geht es um Definitionen, denn unklare Definitionen führen zu unklarer Arbeit. Abschnitt 1.1 vergleicht die maßgeblichen Definitionen: den internationalen Standard ISO/IEC/IEEE 42010:2022, der von den grundlegenden Konzepten oder Eigenschaften eines Systems in seiner Umgebung und den Prinzipien spricht, die seine Weiterentwicklung bestimmen; die Definition des Software Engineering Institute, die von den Strukturen spricht, die man braucht, um über ein System nachzudenken, bestehend aus Elementen, Beziehungen und deren Eigenschaften; und Definitionen aus der Praxis, die gemeinsames Verständnis und schwer änderbare Entscheidungen betonen. Er zeigt, was sie gemeinsam haben, und stellt drei Arten von Strukturen vor, die jedes System hat: wie sein Code aufgeteilt ist, was zur Laufzeit läuft und mit wem kommuniziert und wie die Software auf Infrastruktur und Teams abgebildet wird.

Abschnitt 1.2 geht dann die häufigste Verwechslung an: wo Architektur endet und Design beginnt. Die Antwort lautet: Es gibt keine Mauer, nur ein Kontinuum, geordnet nach Bedeutsamkeit. Eine Entscheidung ist architekturrelevant, wenn sie teuer rückgängig zu machen ist, eine große Reichweite hat oder für ein zentrales Qualitätsmerkmal ausschlaggebend ist; deshalb kann eine unscheinbare Konvention wie die Speicherung von Geldbeträgen architekturrelevanter sein als ein großer Kasten in einem Diagramm. Der Abschnitt unterscheidet außerdem die Ebenen, auf denen das Wort verwendet wird: Unternehmens-, Lösungs- und Softwarearchitektur, quer dazu Daten-, Sicherheits- und Infrastrukturarchitektur.

Warum sie zählt und wer sie macht

Abschnitt 1.3 begründet, warum Architektur wichtig ist. Er zeigt, wie Qualitätsmerkmale wie Verfügbarkeit und Performance aus der Struktur entstehen statt aus einzelnen Codezeilen, wie die Änderungskosten steigen, je mehr auf einer Entscheidung aufgebaut wird, und wie Architektur Risiko, Teamorganisation (über das Gesetz von Conway), Kommunikation und Schätzungen prägt. Drei echte Fälle, anhand von Primärquellen erzählt, untermauern das Argument: der Handelsverlust von Knight Capital im August 2012, der holprige Start von HealthCare.gov im Oktober 2013 und die Entscheidung von Amazon Prime Video im Jahr 2023,

einen Monitoring-Dienst von verteilten Serverless-Schritten auf einen einzigen Prozess umzustellen, was dessen Kosten Berichten zufolge um über 90 % senkte und zeigt, dass es bei Architektur um Passung geht, nicht um Mode.

Abschnitt 1.4 wendet sich den Menschen zu. Er beschreibt, was Architekten tatsächlich tun, anhand der acht Erwartungen von Mark Richards und Neal Ford und der Aufgaben im iSAQB-Foundation-Lehrplan; die verschiedenen Arten von Architekten; warum Architektur eine Rolle ist, bevor sie eine Berufsbezeichnung ist; warum Architekten breites Wissen schätzen; und die lange Debatte darüber, ob Architekten noch selbst Code schreiben sollten.

Abschnitt 1.5 weitet den Blick auf die Stakeholder (alle, die ein Interesse am System haben, von Nutzern und Entwicklern bis zu Betriebsteams, Datenschutzbeauftragten und Aufsichtsbehörden) und ihre Anliegen und zeigt, wie widersprüchliche Anliegen explizit gemacht werden, statt von dem entschieden zu werden, der am lautesten ruft. Abschnitt 1.6 widmet sich Entscheidungen und Abwägungen: dem Aufbau einer guten Entscheidung, dem Unterschied zwischen leicht umkehrbaren und nicht umkehrbaren Entscheidungen, der Idee, zum Last Responsible Moment zu entscheiden, und den Denkfallen, die auch fähige Menschen in die Irre führen.

Architektur im Lauf der Zeit

In den letzten beiden Abschnitten geht es um Zeit. Abschnitt 1.7 erklärt technische Schulden so, wie Ward Cunningham sie 1992 ursprünglich gemeint hat, wie sie sich von bloßer Unordnung unterscheiden und wie Architekturen erodieren und von ihrer beabsichtigten Form abdriften; er zeigt, wie man den Schaden erkennt und in geschäftlichen Begriffen erklärt. Abschnitt 1.8 verortet Architektur im Lebenszyklus der Auslieferung: den alten Streit zwischen Big Design Up Front und rein emergentem Design und den praktischen Mittelweg einer risikogetriebenen, kontinuierlichen und evolutionären Architektur, bei der ein Team gerade so viel Architektur betreibt wie nötig, zum richtigen Zeitpunkt, und laufend prüft, ob sie noch passt.

Der eine Gedanke zum Mitnehmen

Wenn Sie sich aus diesem Kapitel nur eines merken, dann dies: Architektur ist die Menge der Entscheidungen, bei denen ein Fehler teuer wird, und die Aufgabe des Architekten besteht darin, diese Entscheidungen zu erkennen, sie bewusst mit den richtigen Leuten zu treffen, die Gründe festzuhalten und sie im Blick zu behalten, während möglichst viele andere Entscheidungen billig änderbar bleiben. Alles Weitere in diesem Kurs, von den Qualitätsmerkmalen in Kapitel 2 bis zum Entwurf KI-gestützter Systeme in Kapitel 12, ist eine Anwendung dieses Gedankens.

ABSCHNITT 1.1 VON 1.8 · EINSTEIGER · 21 MIN. LESEZEIT · ZULETZT GEPRÜFT IM NOVEMBER 2026

Was Softwarearchitektur ist: die Standarddefinitionen und ihre Gemeinsamkeiten

DIE IDEE IN KÜRZE

Denken Sie an ein Krankenhausgebäude. Seine Architektur ist nicht der Bauplan an der Wand, sondern die tatsächliche Anordnung des Gebäudes: wo die Stationen liegen, welche Flure sie verbinden, wo die Aufzüge fahren, welche Türen verschlossen bleiben müssen und welche Regeln alle befolgen, wenn sie es erweitern. Manche Entscheidungen, etwa wo die Aufzüge sind, wären nur mit einem Vermögen zu ändern; die Farbe der Vorhänge nicht. Softwarearchitektur ist dieselbe Idee für ein Softwaresystem: seine Hauptteile, wie sie verbunden sind, welche Eigenschaften sie haben und die schwer änderbaren Entscheidungen dahinter.

WAS SIE WISSEN MÜSSEN

- ISO/IEC/IEEE 42010:2022 definiert Architektur als die grundlegenden Konzepte oder Eigenschaften eines Betrachtungsgegenstands in seiner Umgebung sowie die maßgebenden Prinzipien für seine Realisierung und Weiterentwicklung.
- Bass, Clements und Kazman definieren Softwarearchitektur als die Menge der Strukturen, die man braucht, um über ein System nachzudenken: Softwareelemente, die Beziehungen zwischen ihnen und die Eigenschaften beider.
- Definitionen aus der Praxis ergänzen, dass Architektur das gemeinsame Verständnis der erfahrenen Entwickler ist (Ralph Johnson) und die bedeutsamen Entscheidungen umfasst, wobei Bedeutsamkeit an den Änderungskosten gemessen wird (Grady Booch).
- Die Definitionen teilen sieben Ideen: Strukturen, Elemente, Beziehungen, Eigenschaften, Entscheidungen, Bedeutsamkeit und Umgebung.
- Systeme haben drei Arten von Strukturen: Modulstrukturen (Code), Komponenten-und-Konnektoren-Strukturen (Laufzeit) und Zuordnungsstrukturen (Abbildung auf Infrastruktur, Teams und Dateien).
- Jedes System hat eine Architektur, ob sie bewusst entworfen oder aufgeschrieben wurde oder nicht; eine Architekturbeschreibung ist ein eigenes Arbeitsergebnis, das sie ausdrückt.

Warum das Wort überhaupt eine Definition braucht

Fragen Sie zehn erfahrene Fachleute, was Softwarearchitektur ist, und Sie hören zehn verschiedene Antworten: das Kästchen-und-Pfeile-Diagramm an der Wand, die Wahl des Cloud-Anbieters, das Datenbankschema, die Aufteilung in Services, die Regeln, welcher Code welchen aufrufen darf, oder einfach „die schwierigen Entscheidungen“. Jede Antwort erfasst etwas Reales, und jede für sich allein lässt etwas aus. Ein Kurs, der Architektur lehren will, muss daher zuerst genau sagen, wovon er spricht.

Das hat praktische Gründe. Wenn ein Team glaubt, Architektur sei „das Diagramm“, hört es auf hinzusehen, sobald das Diagramm gezeichnet ist, obwohl sich das System darunter weiter verändert. Wenn es glaubt, Architektur sei „der Technologie-Stack“, streitet es über Frameworks und übersieht, wie Verantwortlichkeiten aufgeteilt sind und wie Daten fließen. Eine gute Definition sagt Ihnen, worauf Sie achten, was Sie schützen und was Sie mit anderen besprechen sollten.

Die folgenden Definitionen stammen aus einem internationalen Standard, vom Software Engineering Institute (SEI) der Carnegie Mellon University und aus der Praxis. Sie wurden für unterschiedliche Zwecke geschrieben, stimmen aber mehr überein, als sie sich widersprechen, und die Punkte, in denen sie übereinstimmen, bilden den Kern der Disziplin.

Die Standarddefinition: ISO/IEC/IEEE 42010:2022

ISO/IEC/IEEE 42010 ist der internationale Standard für die Beschreibung von Architekturen. Seine aktuelle Ausgabe, veröffentlicht im November 2022, hat die Ausgabe von 2011 ersetzt. Er definiert Architektur, eng umschrieben, als die grundlegenden Konzepte oder Eigenschaften eines Betrachtungsgegenstands in seiner Umgebung, zusammen mit den maßgebenden Prinzipien für dessen Realisierung und Weiterentwicklung und für die zugehörigen Lebenszyklusprozesse.

Drei Merkmale dieser Definition verdienen Aufmerksamkeit. Erstens spricht sie von einem *Betrachtungsgegenstand*, nicht nur von einem Softwaresystem. Die Ausgabe von 2022 hat den Geltungsbereich bewusst erweitert, sodass dieselben Ideen für Software, Systemverbünde, Produkte, Produktlinien und ganze Unternehmen gelten. Zweitens stellt sie den Gegenstand *in seine Umgebung*: Eine Architektur lässt sich nicht verstehen ohne die Nutzer, anderen Systeme, Organisation, Gesetze und Budgets um sie herum. Drittens umfasst sie *maßgebende Prinzipien* dafür, wie der Gegenstand gebaut wird und wie er sich verändert. Architektur ist nicht nur eine Momentaufnahme der Struktur; sie umfasst die Regeln, die die Struktur stimmig halten, während sich das System weiterentwickelt, etwa „jeder Service besitzt seine eigenen Daten“ oder „der Domänencode hängt nie vom Web-Framework ab“.

Der Standard zieht außerdem eine scharfe Linie zwischen einer Architektur und einer Architekturbeschreibung. Die Architektur ist die Menge der grundlegenden Eigenschaften, die das System tatsächlich hat. Die Beschreibung ist ein Arbeitsergebnis (Dokumente, Modelle, Diagramme, Entscheidungsprotokolle), das sie ausdrückt. ISO/IEC/IEEE 42010 ist vor allem ein Standard über Beschreibungen: Er legt fest, was eine gute Beschreibung enthalten sollte, etwa die identifizierten Stakeholder, ihre Anliegen und Sichten, die diese Anliegen behandeln. Er sagt Ihnen nicht, wie Sie eine Architektur entwerfen. Abschnitt 1.5 kommt auf Stakeholder und Anliegen zurück, und Kapitel 4 behandelt Sichten.

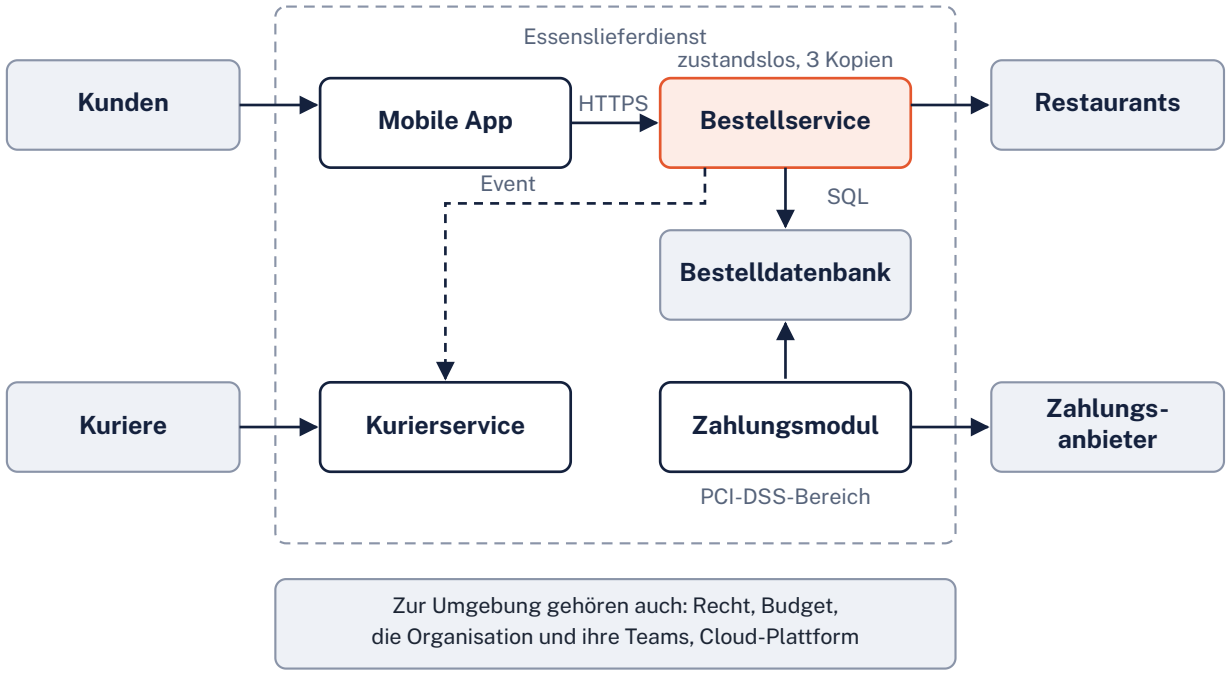
Die SEI-Definition: Strukturen, Elemente, Beziehungen und Eigenschaften

Die am häufigsten gelehrte Definition stammt aus *Software Architecture in Practice* von Len Bass, Paul Clements und Rick Kazman (4. Auflage, Addison-Wesley, 2021), deren Autoren am SEI tätig waren. Sinngemäß in ihren Worten: Die Softwarearchitektur eines Systems ist die Menge der Strukturen, die man braucht, um über das System nachzudenken, und diese Strukturen bestehen aus Software-Elementen, den Beziehungen zwischen ihnen und den Eigenschaften beider.

Jeder Teil dieses Satzes leistet etwas. **Strukturen**, im Plural: Ein System hat viele Strukturen zugleich (seine Codemodule, seine laufenden Prozesse, seine Verteilung auf Maschinen), und keine einzelne davon ist „die“ Architektur. **Die man braucht, um über das System nachzudenken**: Ob etwas architekturrelevant ist, zeigt sich daran, ob Sie es brauchen, um wichtige Fragen zu beantworten, etwa „Übersteht das die Spitzenlast?“ oder „Können zwei Teams unabhängig daran arbeiten?“. **Elemente** sind die Teile: Module, Services, Datenbanken, Queues. **Beziehungen** sind die Art, wie sie verbunden sind: *nutzt*, *ruft auf*, *veröffentlicht an*, *ist deployt auf*. **Eigenschaften** bestimmen, wie sich Elemente und Beziehungen verhalten: Ein Service ist zustandslos, ein Aufruf ist synchron mit einem Timeout von zwei Sekunden, eine Datenbank enthält personenbezogene Daten und muss in der EU bleiben.

Die Formulierung über das Nachdenken erklärt auch, was *keine* Architektur ist. Der Name einer lokalen Variable, der interne Algorithmus einer Sortierroutine oder das Layout eines Einstellungsbildschirms helfen meist nicht, über das System als Ganzes nachzudenken; sie gehören daher zum Feinentwurf und nicht zur Architektur. Abschnitt 1.2 untersucht, wo diese Grenze verläuft.

Elemente, Beziehungen, Eigenschaften und Umgebung



Elemente (Kästen), verbunden durch Beziehungen (Pfeile), mit danebengeschriebenen Eigenschaften, innerhalb einer Systemgrenze, die in einer weiteren Umgebung liegt.

Definitionen aus der Praxis: gemeinsames Verständnis, das Wichtige und Entscheidungen

Praktiker haben kürzere Definitionen vorgeschlagen, die Menschen und Entscheidungen statt Strukturen betonen. Martin Fowler hat zwei von Ralph Johnson bekannt gemacht (einem der vier Autoren des Buchs *Design Patterns* von 1994). Die erste lautet, Architektur sei das gemeinsame Verständnis, das die erfahrenen Entwickler davon haben, wie das System entworfen ist. Die zweite, aus derselben Diskussion, ist bewusst offen gehalten: „Bei Architektur geht es um das Wichtige. Was immer das ist.“ Johnson schlug außerdem vor, Architektur sei die Menge der Entscheidungen, die man gern früh in einem Projekt richtig treffen würde, nicht weil man sie früh eher richtig trifft, sondern weil ein Fehler bei ihnen so teuer ist.

Grady Booch, einer der Schöpfer der UML, fasste es ökonomisch: Architektur besteht aus den bedeutsamen Designentscheidungen, die ein System prägen, wobei Bedeutsamkeit an den Änderungskosten gemessen wird. Das macht aus einer abstrakten Idee einen brauchbaren Test. Die Farbe eines Buttons zu wählen, lässt sich billig rückgängig machen. Die Wahl zwischen einer relationalen Datenbank und einem Dokumentenspeicher oder zwischen einer

deploybaren Einheit und vierzig ist teuer rückgängig zu machen, sobald Daten, Code und Teams darum herum gewachsen sind. Booch stellte auch fest, dass jede Architektur Design ist, aber nicht jedes Design Architektur.

Diese Definitionen ergänzen etwas, das die strukturellen offenlassen. Architektur lebt zum Teil in den Köpfen der Menschen; ein System, dessen erfahrene Fachleute ein klares gemeinsames Bild davon haben, wie es funktioniert, hat eine stimmigere Architektur als eines, bei dem jeder ein anderes Bild im Kopf hat, selbst wenn keines von beiden ein Diagramm besitzt. Und Architektur wird durch Entscheidungen geprägt, von denen viele eine Abwägung erfordern: Wer sich für eine Option entscheidet, verzichtet auf die Vorteile einer anderen. Abschnitt 1.6 behandelt Entscheidungen und Abwägungen im Detail.

Die Definitionen im Vergleich

Quelle	Kernidee (sinngemäß)	Was sie betont	Am nützlichsten für
ISO/IEC/IEEE 42010:2022	Grundlegende Konzepte oder Eigenschaften eines Betrachtungsgegenstands in seiner Umgebung sowie maßgebende Prinzipien für seine Realisierung und Weiterentwicklung	Umgebung, Prinzipien, Weiterentwicklung; Architektur getrennt von ihrer Beschreibung	Verträge, Normungsarbeit, einheitliche Beschreibung von Architekturen in einer Organisation
Bass, Clements und Kazman (2021)	Die Menge der Strukturen, die man braucht, um über das System nachzudenken: Elemente, Beziehungen und die Eigenschaften beider	Mehrere Strukturen; Nachdenken über Qualitätsmerkmale	Analyse und Entwurf: die Frage, ob eine Struktur ihre Ziele erreicht
Ralph Johnson, über Martin Fowler	Das gemeinsame Verständnis erfahrener Entwickler; das Wichtige, was immer das ist	Menschen, Kommunikation, Urteil darüber, was wichtig ist	Teampraxis: ein gemeinsames mentales Modell lebendig halten
Grady Booch	Die bedeutsamen Designentscheidungen, die ein System prägen, wobei Bedeutsamkeit an den Änderungskosten gemessen wird	Entscheidungen und ihre Ökonomie	Entscheiden, wohin der Analyseaufwand fließt

Was die Definitionen gemeinsam haben

Nebeneinander gelesen, laufen die Definitionen auf sieben Ideen zusammen, die gemeinsam eine Arbeitsdefinition für den restlichen Kurs bilden.

Sieben gemeinsame Ideen

1. **Strukturen**
   Ein System hat mehrere Strukturen zugleich, die es jeweils aus einem anderen Blickwinkel zeigen. Kein einzelnes Diagramm ist die ganze Architektur.

2. **Elemente**
   Die Teile, aus denen diese Strukturen bestehen: Module, Komponenten, Services, Datenspeicher, Queues, Server.

3. **Beziehungen**
   Wie die Elemente verbunden sind und voneinander abhängen: Aufrufe, Nachrichten, gemeinsame Daten, Deployment, Zuständigkeit.

4. **Eigenschaften**
   Die von außen sichtbaren Merkmale von Elementen und Beziehungen, etwa zustandslos, verschlüsselt, asynchron oder repliziert, die zusammen die Qualitätsmerkmale des Systems bestimmen.

5. **Entscheidungen**
   Die Entscheidungen, aus denen die Strukturen hervorgegangen sind, mit den Gründen und verworfenen Alternativen dahinter. Ohne die Gründe machen Menschen später gute Entscheidungen versehentlich rückgängig.

6. **Bedeutsamkeit**
   Nur manches ist architekturrelevant: was teuer zu ändern ist, viele Teile betrifft oder ein zentrales Qualitätsmerkmal bestimmt. Architektur ist per Definition selektiv.

7. **Umgebung**
   Das System steht inmitten von Nutzern, anderen Systemen, einer Organisation, Gesetzen und Budgets. Diese prägen die Architektur und werden von ihr geprägt.

Drei Arten von Strukturen

Die Arbeit des SEI zur Dokumentation, veröffentlicht als *Documenting Software Architectures: Views and Beyond* (Clements und Kollegen, 2. Auflage, 2010), teilt Strukturen in drei Arten ein, und *Software Architecture in Practice* verwendet dieselbe Einteilung. Wer die drei Arten kennt, ist vor dem häufigsten Fehler in Architekturdiskussionen geschützt: über eine Art von Struktur zu sprechen, während das Gegenüber an eine andere denkt.

Modulstrukturen zeigen, wie der Code in Implementierungseinheiten aufgeteilt ist, etwa Pakete, Bibliotheken und Namensräume, und wie diese Module voneinander abhängen. Sie existieren zur Entwicklungszeit. Sie beantworten Fragen wie: Welches Team kann was ändern, ohne anderen etwas kaputtzumachen, und welcher Code lässt sich wiederverwenden?

Komponenten-und-Konnektoren-Strukturen zeigen, was existiert, wenn das System läuft: Prozesse, Services, Clients, Datenbanken und die Konnektoren dazwischen, etwa HTTP-Aufrufe, Message Queues und Event-Streams. Sie beantworten Laufzeitfragen: Wie schnell ist eine Anfrage, was passiert, wenn ein Teil ausfällt, wo liegen die Daten?

Zuordnungsstrukturen bilden Software auf Dinge außerhalb der Software ab: auf Hardware und Cloud-Infrastruktur (Deployment), auf Teams (Arbeitszuteilung) und auf Dateien und Repositorys (Implementierung). Sie beantworten Fragen zu Kosten, Betrieb und Organisation: Wo läuft es, wer baut es, wer wird nachts geweckt, wenn es ausfällt?

Ein Element kann in allen dreien vorkommen. Eine Checkout-Funktion ist in der Modulstruktur ein *Paket*, in der Komponenten-und-Konnektoren-Struktur ein laufender *Service* und in den Zuordnungsstrukturen eine Menge von *Containern in einer Cloud-Region, die dem Checkout-Team gehören*. Die Abbildung zwischen den Strukturen ist nicht immer eins zu eins: Ein einzelnes Modul kann als mehrere Services deployt werden, oder mehrere Module laufen in einem Prozess. Genau in dieser Lücke verbergen sich viele Überraschungen.

Ein System, drei Arten von Strukturen



Modul-, Komponenten-und-Konnektoren- und Zuordnungsstrukturen zeigen denselben Onlineshop aus drei Blickwinkeln und beantworten unterschiedliche Fragen.

Jedes System hat eine Architektur

Weil Architektur aus Strukturen und den Eigenschaften besteht, die sie tatsächlich haben, hat jedes Softwaresystem eine Architektur, ob sie jemand absichtlich entworfen hat oder nicht und ob sie aufgeschrieben ist oder nicht. Ein Tabellenkalkulationsmakro, das zu einem geschäftskritischen Werkzeug gewachsen ist, hat eine Architektur. Ebenso die Codebasis eines Start-ups, die drei Leute in Eile geschrieben haben. Die Frage ist nie, ob ein System eine Architektur hat, sondern ob diese Architektur ihren Zweck erfüllt, von den Menschen verstanden wird, die daran arbeiten, und bewusst gestaltet ist.

Daraus ergibt sich eine nützliche Unterscheidung. Die *beabsichtigte* (oder präskriptive) Architektur ist das, was das Team für die richtige Struktur hält oder beschlossen hat: das Diagramm im Wiki, die vereinbarten Regeln. Die *tatsächliche* (oder deskriptive) Architektur ist das, was Code und Infrastruktur wirklich tun. Beide driften mit der Zeit auseinander, wenn niemand sie in Einklang hält, und diese Lücke ist eine der Hauptquellen technischer Schulden, die in Abschnitt 1.7 behandelt werden. Eine undokumentierte Architektur fehlt nicht; sie ist nur schwerer zu sehen, zu durchdenken und sicher zu ändern.

WOHER DAS KOMMT

Woher die Definitionen kommen

Softwarearchitektur entstand Anfang der 1990er-Jahre als eigenständige Disziplin. Dewayne Perry und Alexander Wolf schlugen 1992 in ihrem Aufsatz *Foundations for the Study of Software Architecture* vor, dass eine Architektur aus Elementen, ihrer Form und der Begründung dahinter besteht, und die Arbeit von David Garlan und Mary Shaw, die 1996 in ihrem Buch *Software Architecture: Perspectives on an Emerging Discipline* gipfelte, katalogisierte wiederkehrende Architekturstile. Die erste Auflage von *Software Architecture in Practice* erschien 1998, und IEEE 1471, der Vorläufer von ISO/IEC/IEEE 42010, wurde im Jahr 2000 veröffentlicht.

Die Ideen sind noch älter. Edsger Dijkstras geschichtetes Betriebssystem THE (1968) und David Parnas' Aufsatz von 1972 über die Zerlegung von Systemen in Module durch das Verbergen von Designentscheidungen sind beide Architekturdenken, bevor es das Wort dafür gab.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: die Definitionen auf eine Essensliefer-App anwenden

Ein Essenslieferunternehmen bittet eine neue Architektin: „Sagen Sie uns, was unsere Architektur ist.“ Es gibt ein veraltetes Diagramm. Mit den gemeinsamen Ideen als Checkliste erstellt die Architektin innerhalb einer Woche eine erste Beschreibung.

Elemente	Mobile App, Web-App für Restaurants, Bestellservice, Kurierzuordnungsservice, Zahlungsmodul, eine gemeinsame PostgreSQL-Datenbank, ein Message Broker, ein externer Kartenzahlungsanbieter.
Beziehungen	Die Apps rufen den Bestellservice über HTTPS auf; der Bestellservice schreibt in die gemeinsame Datenbank und veröffentlicht „Bestellung aufgegeben“-Events; die Kurierzuordnung konsumiert diese Events; das Zahlungsmodul ruft die API des Anbieters auf.
Eigenschaften	Der Bestellservice ist zustandslos und läuft in drei Kopien; die Kurierzuordnung hält ihren Zustand im Arbeitsspeicher und läuft als einzelne Kopie; die Datenbank enthält personenbezogene Daten und Karten-Tokens.
Umgebung	Zwei Entwicklungsteams, eine Lebensmittelaufsichtsbehörde, Datenschutzrecht, ein festes Cloud-Budget, Restaurants mit unzuverlässigen Tablets.
Gefundene bedeutsame Entscheidungen	Eine gemeinsame Datenbank für alle Services; eine einzelne Kopie der Kurierzuordnung; Events für die Kurierdisposition, aber direkte Aufrufe für Zahlungen.
Beabsichtigt vs. tatsächlich	Das alte Diagramm zeigt getrennte Datenbanken pro Service. In Wirklichkeit liest der Kurierservice die Bestelltabellen direkt, sodass die beiden Services nicht unabhängig voneinander geändert werden können.

Die Übung bringt zwei Risiken zutage, die niemand benannt hatte: Die einzelne Kopie der Kurierzuordnung ist ein Single Point of Failure, und die gemeinsame Datenbank koppelt zwei Teams aneinander.

Keines der beiden Risiken war im Code eines einzelnen Service sichtbar. Beide sind Eigenschaften von Strukturen und Beziehungen, und deshalb zählen sie zur Architektur.

Die Beschreibung dauerte eine Woche, weil niemand die Entscheidungen oder ihre Gründe festgehalten hatte. Das Festhalten ist Thema von Kapitel 4.

ACHTUNG

Verbreitete Irrtümer

- **„Architektur ist das Diagramm.“** Ein Diagramm ist eine Teilbeschreibung einer Struktur. Die Architektur ist die Menge der Eigenschaften, die das System tatsächlich hat.
- **„Architektur ist der Technologie-Stack.“** Technologieentscheidungen können architekturrelevant sein, aber wie Verantwortlichkeiten, Daten und Abhängigkeiten aufgeteilt sind, ist meist wichtiger als die Wahl des Frameworks.
- **„Architektur gibt es nur bei großen Systemen.“** Auch kleine Systeme haben Architekturen; sie haben nur weniger bedeutsame Entscheidungen, sodass weniger Aufwand nötig ist, sie richtig zu treffen.
- **„Architektur wird einmal, am Anfang, entschieden.“** Alle Definitionen schließen die Weiterentwicklung ein. Architektur wird während des gesamten Lebens eines Systems geformt, wie Abschnitt 1.8 erklärt.

Wie Fachleute die Definitionen nutzen

In der Praxis sagen erfahrene Architekten selten eine Definition auf, aber sie nutzen jede als Linse. Die ISO-Sicht erinnert sie daran, nach außen auf die Umgebung zu schauen und das, was das System *ist*, von dem zu trennen, was die Dokumente *sagen*. Die SEI-Sicht gibt ihnen ein Vokabular aus Strukturen, Elementen, Beziehungen und Eigenschaften für die Analyse und einen Test („Brauchen wir das, um über das System nachzudenken?“). Johnsons Sicht erinnert sie daran, dass ein gemeinsames mentales Modell im Team selbst ein Wert ist, der gepflegt werden muss. Boochs Sicht sagt ihnen, wofür sie ihre begrenzte Zeit einsetzen sollten: für die Entscheidungen, die teuer rückgängig zu machen wären.

Daraus folgt eine praktische Faustregel. Wenn Sie unsicher sind, ob etwas architekturrelevant ist, stellen Sie drei Fragen. Wäre es später teuer oder riskant, es zu ändern? Betrifft es mehrere Teile des Systems oder mehrere Teams? Entscheidet es darüber, ob das System ein wichtiges Qualitätsziel erreicht, etwa Geschwindigkeit, Verfügbarkeit oder Sicherheit? Lautet die Antwort auf eine dieser Fragen Ja, behandeln Sie es als Architektur: Entscheiden Sie es bewusst, halten Sie die Gründe fest und sorgen Sie dafür, dass die Betroffenen es verstehen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Was ist der Unterschied zwischen einer Architektur und einer Architekturbeschreibung?**

Die Architektur ist die Menge der grundlegenden Eigenschaften und Strukturen, die das System tatsächlich hat. Die Beschreibung ist ein Arbeitsergebnis, etwa Diagramme und Dokumente, das sie ausdrückt, und sie kann unvollständig oder veraltet sein.

▼ **Welche Art von Struktur würden Sie untersuchen, um herauszufinden, welches Team geweckt wird, wenn ein Service ausfällt?**

Eine Zuordnungsstruktur, genauer die Abbildung der Arbeitszuteilung oder Zuständigkeit von Software auf Teams.

▼ **Ist nach Boochs Test die Wahl der Logging-Bibliothek eine Architekturentscheidung?**

Meist nicht, wenn sie hinter einer kleinen Schnittstelle verborgen und billig austauschbar ist. Sie wird architekturrelevant, wenn jedes Modul sie direkt aufruft und sie dadurch teuer zu ändern ist.

▼ **Warum erwähnt die ISO-Definition die Umgebung?**

Weil sich eine Architektur nur im Verhältnis zu den Nutzern, Systemen, der Organisation, den Gesetzen und Budgets um sie herum beurteilen lässt; dieselbe Struktur kann in einer Umgebung gut und in einer anderen schlecht sein.

▼ **Probieren Sie es aus: Eine Führungskraft bei einem Essenslieferdienst sagt: „Wir haben keine Architektur, nur Code und ein altes Diagramm.“ Antworten Sie darauf, und ordnen Sie dann diese Punkte den Strukturtypen zu: Das Paket orders importiert payments; Bestellungen werden an einen Broker veröffentlicht; die Disposition läuft auf drei Servern, die von Team B betrieben werden.**

Jedes System hat eine Architektur, ob entworfen oder nicht; was fehlt, ist eine aktuelle Beschreibung davon. Der Paketimport ist eine Modulstruktur, weil er den Code betrifft. Das Veröffentlichen an einen Broker ist eine Komponenten-und-Konnektoren-Struktur, weil es das laufende System betrifft. Die Disposition auf von Team B betriebenen Servern ist eine Zuordnungsstruktur, die Software auf Infrastruktur und Menschen abbildet. Kein einzelnes Diagramm zeigt alle drei, eine gute Beschreibung braucht also mehrere.

QUELLEN

- ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description (ISO): https://www.iso.org/standard/74393.html

- Software Architecture in Practice, Fourth Edition (SEI, Carnegie Mellon University): https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- Software Architecture Guide (Martin Fowler): https://martinfowler.com/architecture/
- Documenting Software Architectures: Views and Beyond, Second Edition (SEI): https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/

Architektur, Design und Code: Bedeutsamkeit, Reichweite und Ebenen

DIE IDEE IN KÜRZE

Wenn ein Haus geplant wird, legen manche Entscheidungen alles andere fest: wie viele Stockwerke, wo die Treppe verläuft, wo die tragenden Wände stehen. Andere lassen sich an einem Wochenende ändern: Farbe, Regale, Lampen. Beides ist Design, aber nur die erste Art ist Architektur, weil eine spätere Änderung bedeutet, das Haus umzubauen. Bei Software ist es ähnlich. Zu entscheiden, wie ein System in Teile zerlegt wird oder wie Geldbeträge überall gespeichert werden, ist Architektur, weil es teuer rückgängig zu machen ist. Zu entscheiden, wie eine Funktion innerhalb eines Teils geschrieben wird, ist Design.

WAS SIE WISSEN MÜSSEN

- Architektur und Design bilden ein Kontinuum: Jede Architektur ist Design, aber nur die bedeutsamen Designentscheidungen sind Architektur.
- Eine Entscheidung ist architekturrelevant, wenn sie schwer oder teuer rückgängig zu machen ist, eine große Reichweite hat, ein zentrales Qualitätsmerkmal bestimmt, andere einschränkt oder wichtige Abwägungen beinhaltet.
- Die Abstraktionsebene allein ist ein schlechter Test: Konventionen auf niedriger Ebene wie Formate für Kennungen oder Geldbeträge können architekturrelevant sein, und manche Kästen auf hoher Ebene sind belanglos.
- Strategische Entscheidungen geben vielen Menschen eine langfristige Richtung vor; taktische Entscheidungen lösen abgegrenzte, unmittelbare Probleme, doch wiederholte taktische Entscheidungen können architekturrelevant werden.
- Unternehmensarchitektur umfasst eine ganze Organisation (TOGAF ist das bekannteste Framework), Lösungsarchitektur ein Geschäftsproblem über mehrere Systeme hinweg und Softwarearchitektur ein System; Daten-, Sicherheits- und Infrastrukturarchitektur ziehen sich durch alle Ebenen.
- Entscheidungen sollten auf der niedrigsten Ebene getroffen werden, die genug Kontext und Verantwortung hat, mit breiterer Abstimmung bei Entscheidungen, die viele Teams betreffen.

Ein Kontinuum, keine Mauer

Wenn Architektur die Menge der bedeutsamen Entscheidungen und Strukturen ist (Abschnitt 1.1), dann ist Design alles Übrige: die vielen kleineren Entscheidungen darüber, wie jeder Teil im Inneren funktioniert. Die Schwierigkeit ist, dass es keine saubere Grenze zwischen beidem gibt. Eine Essenslieferplattform in ein Bestellsystem und ein Kuriersystem aufzuteilen, ist eindeutig eine Architekturentscheidung. Eine Variable zu benennen, ist es eindeutig nicht. Zwischen diesen Extremen liegt ein breites Mittelfeld von Entscheidungen (das Fehlerformat einer API, eine Caching-Regel, die Aufteilung eines Moduls in Pakete), deren Status vom Kontext abhängt.

Grady Booch hat es treffend formuliert: Jede Architektur ist Design, aber nicht jedes Design ist Architektur. Architektur und Feinentwurf sind Punkte auf einem Kontinuum von Designentscheidungen, geordnet danach, wie bedeutsam sie sind. Deshalb lautet die nützlichste Frage nicht abstrakt „Ist das Architektur oder Design?“, sondern „Wie bedeutsam ist diese Entscheidung hier, und wie viel Sorgfalt verdient sie?“.

Eine zweite, beliebte Art, die Grenze zu ziehen, ist die Abstraktionsebene: Architektur ist die Sicht von oben („Kästchen und Pfeile“), Design die Sicht im Detail („Klassen und Methoden“). Das ist eine vernünftige erste Näherung, scheitert aber in beide Richtungen. Ein Detail auf niedriger Ebene kann architekturrelevant sein: Das Format einer Kennung, die zwanzig Services gemeinsam nutzen, oder die Wahl einer Zeitzonenkonvention für gespeicherte Datumsangaben kann Monate kosten, wenn man es ändern will. Und ein Kasten auf hoher Ebene kann belanglos sein: Ob ein internes Admin-Tool ein eigener Service ist, spielt womöglich gar keine Rolle.

Was eine Entscheidung architekturrelevant macht

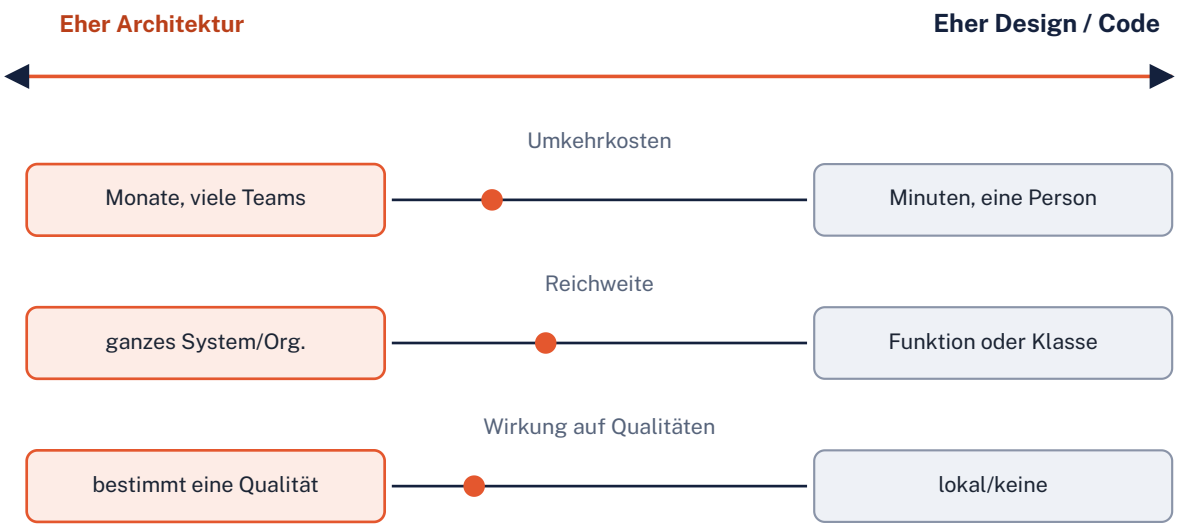
Eine Entscheidung ist architekturrelevant, wenn sie eines oder mehrere der folgenden Merkmale aufweist. Das sind keine formalen Regeln aus einem einzelnen Standard; sie fassen zusammen, was das SEI, Martin Fowler, Mark Richards und Neal Ford und andere mit unterschiedlichen Worten sagen.

Anzeichen dafür, dass eine Entscheidung architekturrelevant ist

Merkmal	Zu stellende Frage	Beispiel
Schwer oder teuer rückgängig zu machen	Wenn sich das in einem Jahr als falsch erweist, was kostet dann die Änderung?	Ein ereignisbasiertes Datenmodell (Event-Sourcing) wählen, statt den aktuellen Zustand zu speichern
Große Reichweite	Wie viele Komponenten, Teams oder Systeme berührt es?	Das Format der Kundenkennungen festlegen, die in allen Services verwendet werden
Bestimmt ein zentrales Qualitätsmerkmal	Entscheidet es darüber, ob wir ein Ziel bei Performance, Verfügbarkeit, Sicherheit oder Kosten erreichen?	Einen Cache vor den Produktkatalog setzen, um ein Antwortzeitziel von 200 ms zu erreichen
Setzt anderen eine Randbedingung	Schränkt es ein, was andere Teams wählen dürfen?	Vorschreiben, dass alle Services Events an einen gemeinsamen Broker veröffentlichen
Beinhaltet bedeutsame Abwägungen	Geben wir etwas Wichtiges auf, um etwas anderes zu bekommen?	Eventual Consistency zwischen Bestellungen und Lagerbestand in Kauf nehmen, damit der Checkout verfügbar bleibt
Hohe Neuartigkeit oder hohes Risiko	Hat das hier schon jemand gemacht? Was, wenn uns die Technologie im Stich lässt?	Eine Datenbank einführen, die die Organisation noch nie in Produktion betrieben hat

Mark Richards und Neal Ford beschreiben in *Fundamentals of Software Architecture* das Spektrum zwischen Architektur und Design ähnlich: Eine Entscheidung liegt umso näher an der Architektur, je strategischer sie ist (langfristig, viele Beteiligte), je mehr Aufwand eine Änderung erfordern würde und je bedeutsamer ihre Abwägungen sind. Martin Fowler bietet eine kompakte Fassung: Architektur ist das, was schwer zu ändern ist, und daraus folgt auch, dass es eines der Ziele eines Architekten ist, mehr Dinge leicht änderbar zu machen. Eine Entscheidung billig umkehrbar zu machen, etwa indem man sie hinter eine Schnittstelle legt, verschiebt sie faktisch von der Architektur hin zum Design. Diese Idee, die Umkehrbarkeit, wird in Abschnitt 1.6 weiter vertieft.

Eine Entscheidung auf dem Kontinuum einordnen



Punkte: eine Entscheidung (Wahl des Nachrichtenformats) auf jeder Dimension

Drei Dimensionen ordnen eine Entscheidung zwischen Architektur und Design ein; eine Entscheidung kann auf jeder anders abschneiden.

Strategische und taktische Entscheidungen

Eine verwandte Unterscheidung ist die zwischen strategischen und taktischen Entscheidungen. **Strategische** Entscheidungen geben für lange Zeit und für viele Menschen die Richtung vor: einen Cloud-Anbieter einführen, zwischen einem Monolithen und verteilten Services wählen, festlegen, dass Kundendaten in einem einzigen System geführt werden. Sie brauchen meist den Input mehrerer Stakeholder, und es lohnt sich, sie mit ihren Gründen aufzuschreiben. **Taktische** Entscheidungen lösen ein unmittelbares Problem in einem abgegrenzten Bereich: welche Bibliothek ein Service zum Parsen von Datumsangaben nutzt, wie eine bestimmte Datenbankabfrage aufgebaut ist, wie die Endpoints einer internen API heißen.

Strategisch heißt nicht langsam. Viele strategische Entscheidungen können und sollten schnell getroffen werden, sobald die Optionen klar sind, und manche sind umkehrbar, wenn sie gut strukturiert sind. Taktisch heißt nicht unwichtig: Eine schlechte taktische Entscheidung, die an fünfzig Stellen wiederholt wird (etwa wenn jeder Service seine eigene Wiederholungslogik ohne Back-off schreibt), kann sich zu einem Architekturproblem summieren. Architekten achten auf taktische Entscheidungen, die durch Wiederholung gerade strategisch werden, und machen daraus ein gemeinsames Prinzip oder eine Plattformfähigkeit.

Architektur oder Design? Beispielentscheidungen mit Begründung

Entscheidung (für einen Onlineshop)	Urteil	Begründung
Das System in separat deploybare Services für Katalog, Checkout und Lieferung aufteilen	Architektur	Betrifft jedes Team, das Deployment, die Datenhoheit und das Ausfallverhalten; sehr teuer rückgängig zu machen.
PostgreSQL statt einer Dokumentendatenbank als Hauptspeicher für Bestellungen verwenden	Architektur	Prägt das Datenmodell, die Transaktionen und die nötigen Betriebskenntnisse; eine spätere Migration ist teuer und riskant.
Eine bestimmte JSON-Bibliothek im Checkout-Service verwenden	Design	Lokal auf einen Service begrenzt und an einem Tag austauschbar, wenn sie hinter dem eigenen Code des Service verborgen ist.
Verlangen, dass jeder Service strukturierte Logs mit einer gemeinsamen Korrelations-ID ausgibt	Architektur	Ein Querschnittsanliegen: Es funktioniert nur, wenn sich alle daran halten, und es bestimmt, wie schnell sich Vorfälle diagnostizieren lassen.
Die Methode benennen, die Liefergebühren berechnet	Design (Code)	Rein lokal; Umbenennen erledigen Werkzeuge automatisch.
Das Format der Bestellnummern festlegen, die Kunden angezeigt und mit dem Lagerpartner geteilt werden	Architektur	Niedrige Ebene, aber es überschreitet System- und Unternehmensgrenzen und wird auf Pakete gedruckt; eine Änderung erfordert Abstimmung mit einem externen Partner.
Das Strategy-Muster verwenden, um mehrere Regeln für Liefergebühren zu unterstützen	Design	Prägt das Innenleben einer Komponente; andere Komponenten sind nicht betroffen.
Produktpreise 60 Sekunden lang cachen	Kommt darauf an	Design, wenn sich Preise selten ändern und veraltete Werte harmlos sind; Architektur, wenn ein veralteter Preis ein rechtliches oder finanzielles Problem verursacht, weil die Entscheidung dann ein zentrales Qualitätsmerkmal (Korrektheit) bestimmt.

Ebenen der Architektur

Das Wort Architektur wird außerdem auf mehreren Ebenen der Reichweite verwendet, und Berufsbezeichnungen folgen diesen Ebenen. Wer sie verwechselt, erzeugt viele Missverständnisse: Ein Unternehmensarchitekt und ein Softwarearchitekt können beide „der Architekt“ sein, und doch überschneidet sich ihre Arbeit kaum.

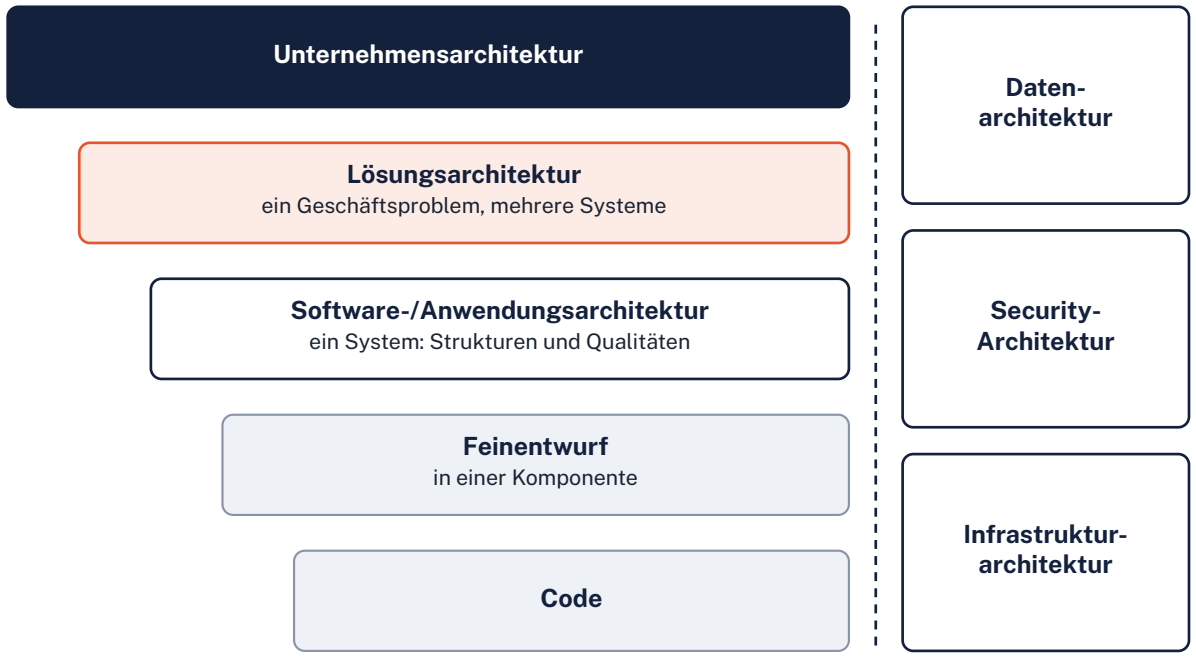
Unternehmensarchitektur betrachtet eine ganze Organisation: ihre Geschäftsfähigkeiten (was sie können muss, etwa „Schadensfälle prüfen“ oder „Kunden aufnehmen“), die Informationen, die sie hält, ihr Anwendungsportfolio und ihre Technologieplattformen und wie sich all das ändern sollte, um die Strategie zu unterstützen. Typische Ergebnisse sind Capability Maps, Roadmaps zum Zielzustand, Technologiestandards und Entscheidungen darüber, welche Systeme abgelöst werden. Das hier am weitesten verbreitete Framework ist TOGAF, herausgegeben von The Open Group, dessen aktuelle Version der TOGAF Standard, 10th Edition (2022) ist. Es bietet eine Methode (die Architecture Development Method) und einen Fundus an Inhalten für die Planung und Steuerung von Veränderungen im Unternehmen. Organisationen nutzen es in unterschiedlichem Maß; viele übernehmen Teile davon, manche finden es schwergewichtig, und dieser Kurs beschreibt es nur, damit Sie es wiedererkennen.

Lösungsarchitektur befasst sich mit einem Geschäftsproblem oder Programm, das typischerweise mehrere Anwendungen, Teams und Anbieter umfasst: etwa ein Krankenhaus, das eine Online-Terminbuchung einführt, die ein neues Web-Frontend, das bestehende Patientenverwaltungssystem, einen SMS-Anbieter und den Identitätsdienst verbinden muss. Der Lösungsarchitekt sorgt dafür, dass die Teile zusammenpassen, oft unter kaufmännischen und vertraglichen Randbedingungen.

Software- oder Anwendungsarchitektur, das Hauptthema dieses Kurses, betrifft die Strukturen und Qualitäten eines Systems: seine Zerlegung, seine Interaktionen zur Laufzeit, sein Deployment, seine Daten und seine Weiterentwicklung.

Drei Fachgebiete ziehen sich durch alle diese Ebenen. **Datenarchitektur** umfasst, wie Daten modelliert, verantwortet, gespeichert, bewegt, gesteuert und für Analysen genutzt werden (Kapitel 7). **Sicherheitsarchitektur** umfasst Identität, Zugriff, Datenschutz, Bedrohungsmodelle und die Platzierung von Sicherheitskontrollen (Kapitel 8). **Infrastrukturarchitektur** (heute oft Cloud- oder Plattformarchitektur) umfasst Rechenleistung, Netzwerke, Speicher, Umgebungen und Deployment-Plattformen (Kapitel 10).

Ebenen und übergreifende Fachgebiete



Jede Ebene beschränkt die darunter; Fachgebiete gehen durch alle Ebenen

Die Reichweite verengt sich vom Unternehmen bis zum Code; Daten-, Sicherheits- und Infrastrukturarchitektur gelten auf jeder Ebene.

Wer welche Entscheidungen trifft

In einer gesunden Organisation werden Entscheidungen auf der niedrigsten Ebene getroffen, die genug Kontext hat und für die Folgen verantwortlich ist. Architektur ist nicht das alleinige Eigentum von Leuten mit „Architekt“ im Titel. Eine vernünftige Standardaufteilung sieht so aus, auch wenn jede Organisation sie anpasst.

Art der Entscheidung	Meist getroffen von	Meist einbezogen
Technologiestandards und Plattformen für die Organisation (z. B. zugelassener Cloud-Anbieter, Identitätsplattform)	Unternehmensarchitekten, CTO, Architekturgremium	Lösungs- und Softwarearchitekten, Sicherheit, Finanzen
Wie mehrere Systeme und Anbieter zusammenwirken, um ein Geschäftsproblem zu lösen	Lösungsarchitekt	Product Owner, Unternehmensarchitekten, Architekten der Anbieter
Die Hauptstruktur eines Systems: Zerlegung, Kommunikationsstil, Datenhoheit	Softwarearchitekt oder technische Leitung des Teams, gemeinsam mit dem Team	Betrieb, Sicherheit, andere betroffene Teams
Design innerhalb einer Komponente: Klassen, Muster, Algorithmen	Das Entwicklungsteam	Architekt nur, wenn ein Prinzip oder Qualitätsmerkmal auf dem Spiel steht
Entscheidungen auf Codeebene: Benennung, lokales Refactoring	Einzelne Entwickler, über Code-Reviews	Normalerweise niemand sonst

Zwei Warnsignale zeigen, dass die Aufteilung falsch ist. Wenn Architekten interne Klassenstrukturen festlegen, werden sie zum Engpass, und das Team hört auf mitzudenken. Wenn Teams unabhängig voneinander Entscheidungen mit systemweiter Wirkung treffen (etwa jedes seinen eigenen Message Broker wählt), zahlt die Organisation später für die Uneinheitlichkeit. Wie man festhält, welche Entscheidungen eine breitere Abstimmung brauchen, und wie man das Festhalten leicht macht, wird unter Entscheidungen und Abwägungen und ausführlich in Kapitel 4 bei den Architecture Decision Records behandelt.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Entscheidungen für eine Fahrdienst-App einordnen

Ein Fahrdienst-Start-up listet fünf anstehende Entscheidungen auf. Die technische Leitung nutzt die drei Anzeichen (Umkehrkosten, Reichweite, Wirkung auf zentrale Qualitätsmerkmale), um zu entscheiden, welche ein schriftliches Protokoll und eine breitere Prüfung brauchen.

1. Fahrtpositionen in einer Zeitreihendatenbank speichern	Umkehrbarkeit: teuer (Datenmigration). Reichweite: Fahrten, Abrechnung, Analytics. Qualität: Performance des Live-Trackings. Urteil: Architektur; aufschreiben und mit Daten- und Betriebsteams prüfen.
2. Eine bestimmte HTTP-Client-Bibliothek im Preisservice verwenden	Umkehrbarkeit: billig. Reichweite: ein Service. Qualität: nichts Wesentliches. Urteil: Design; das Preisteam entscheidet.
3. Fahrpreise auf dem Server berechnen, nie auf dem Smartphone	Umkehrbarkeit: mittel. Reichweite: Apps und Backend. Qualität: Sicherheit und Integrität der Preise. Urteil: Architektur; es ist ein maßgebendes Prinzip.
4. Geld überall als ganzzahlige kleinste Einheiten (Pence, Cent) darstellen	Umkehrbarkeit: sehr teuer, sobald Daten und Partner-APIs es verwenden. Reichweite: jeder Service. Qualität: Korrektheit. Urteil: Architektur, obwohl es eine niedrige Ebene betrifft.
5. Farben für den Dunkelmodus in der Fahrer-App verwenden	Umkehrbarkeit: trivial. Reichweite: eine Gruppe von Bildschirmen. Urteil: Design.

Zwei der drei Architekturentscheidungen (3 und 4) haben überhaupt nichts mit Kästchen und Pfeilen zu tun; es sind Prinzipien und Konventionen. Nach der Abstraktionsebene allein wären sie falsch eingeordnet worden.

Entscheidung 4 ist die Art, die am häufigsten übersehen wird: Sie wirkt wie ein Codierdetail, wird aber sehr teuer zu ändern, sobald andere Systeme davon abhängen.

Häufige Fragen

Ist die Wahl einer Programmiersprache eine Architekturentscheidung?
Oft ja. Sie beeinflusst die Personalgewinnung, die verfügbaren Bibliotheken, die Performance-Eigenschaften und wie leicht sich Code zwischen Komponenten teilen lässt. In einem System aus vielen unabhängigen Services kann die Sprache eines kleinen Service eine lokale Entscheidung sein; die Standardsprache für die ganze Organisation ist Architektur.

Ist ein Datenbankschema Architektur oder Design?
Das hängt davon ab, wer es nutzt. Eine Tabelle, die nur innerhalb eines Service hinter dessen eigener API verwendet wird, ist Design. Ein Schema, das mehrere Services, Berichte und Partnerexporte direkt lesen, ist Architektur, weil jede Änderung mit allen abgestimmt werden muss. Viele Teams machen Schemas genau deshalb zur Privatsache eines Service, um sie zurück ins Design zu holen.

Bedeutet agile Entwicklung, dass es keine Architektur gibt?
Nein. Jedes System hat eine Architektur; agile Ansätze ändern, wann und wie über sie entschieden wird: Sie bevorzugen kleine, umkehrbare Schritte und schieben nicht umkehrbare Entscheidungen auf, bis genug bekannt ist. Abschnitt 1.8 behandelt das.

Wer entscheidet, ob eine Entscheidung bedeutsam ist?
Meist das Team und sein Architekt oder seine technische Leitung gemeinsam, anhand vereinbarter Kriterien wie der Anzeichen in diesem Abschnitt. Wer diese Kriterien aufschreibt, vermeidet zwei Fehlentwicklungen: dass über jede Entscheidung debattiert wird und dass man erst hinterher entdeckt, dass ein Team allein eine systemweite Entscheidung getroffen hat.

Wie Fachleute die Unterscheidung anwenden

Erfahrene Teams machen aus dem Kontinuum eine schlanke Routine. Sie vereinbaren eine kurze Liste der Arten von Entscheidungen, die ein schriftliches Protokoll und eine breitere Abstimmung brauchen, zum Beispiel alles, was eine öffentliche API ändert, einen neuen Datenspeicher oder eine neue Technologie einführt, Teamgrenzen überschreitet oder ein festgelegtes Qualitätsziel berührt. Alles andere entscheiden die Menschen, die die Arbeit machen, über das normale Code-Review. Sie überarbeiten die Liste, wenn etwas durchrutscht oder wenn sich Reviews nach Bürokratie anfühlen.

Sie suchen auch aktiv nach Wegen, die Menge der Architekturentscheidungen zu verkleinern, denn jede Entscheidung, die billig umkehrbar wird, ist eine weniger, die beim ersten Mal richtig sein muss. Einen Anbieter hinter einer Schnittstelle zu verbergen, eine Datenbank zur Privatsache eines Service zu machen, Feature-Flags für riskantes Verhalten zu nutzen und in kleinen Schritten zu deployen, verschiebt Entscheidungen vom architektonischen Ende des Kontinuums zum Design-Ende.

VERTIEFUNG

Die Idee von Intension und Lokalität

Amnon Eden und Rick Kazman schlugen 2003 eine formalere Unterscheidung vor, die auf zwei Kriterien beruht. *Intension* fragt, ob eine Aussage allgemein ist (sie beschreibt eine ganze Klasse möglicher Implementierungen) oder spezifisch (sie beschreibt eine Implementierung); *Lokalität* fragt, ob sie sich in einem Teil des Systems prüfen lässt oder das Ganze einschränkt. Architekturaussagen sind allgemein und *nicht lokal*: Sie schränken das ganze System ein („alle Komponenten kommunizieren über den Message Bus“) und können daher überall verletzt werden. Designaussagen sind allgemein, aber lokal („in diesem Modul das Observer-Muster verwenden“). Implementierungsaussagen sind spezifisch und lokal („diese Klasse hat diese Methoden“). Der Test ist nützlich beim Formulieren von Regeln: Eine Aussage, die überall gelten muss, ist ein Kandidat für ein Architekturprinzip und für eine automatisierte Prüfung, eine Fitness Function.

ACHTUNG

Fehler, die Sie vermeiden sollten

- Alles als Architektur behandeln, was zu Prüfgremien führt, die jede Änderung bremsen, und zu Teams, die keine Verantwortung mehr übernehmen.
- Nichts als Architektur behandeln, wodurch systemweite Eigenschaften zufällig aus Hunderten lokaler Entscheidungen entstehen.
- Annehmen, eine Entscheidung sei Design, weil sie eine niedrige Ebene betrifft. Datenformate, Kennungen, Zeit- und Geldformate und API-Verträge sind häufig architekturrelevant.
- Annehmen, eine Entscheidung sei Architektur, weil sie eine große Komponente betrifft. Wenn sie leicht umkehrbar und in ihrer Wirkung lokal ist, braucht sie kein Zeremoniell.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum kann eine Entscheidung über das Format einer Kennung architekturrelevant sein?

Weil ihre Änderung, sobald sich viele Services, Datenbanken und externe Partner darauf verlassen, abgestimmte Arbeit bei allen erfordert; ihre Änderungskosten und ihre Reichweite sind also hoch.

▼ Was ist der Unterschied zwischen Unternehmens- und Lösungsarchitektur?

Unternehmensarchitektur umfasst die Fähigkeiten, Informationen, Anwendungen und Technologien der ganzen Organisation; Lösungsarchitektur befasst sich mit einem Geschäftsproblem, das meist mehrere Systeme und Teams umfasst.

▼ Wie kann ein Architekt eine Entscheidung von der Architektur in Richtung Design verschieben?

Indem er sie billiger umkehrbar macht, etwa indem er eine Technologie hinter einer Schnittstelle verbirgt, damit sie sich lokal ersetzen lässt.

▼ Probieren Sie es aus: Ein Ticketing-Team listet drei Entscheidungen auf: eine Hilfsfunktion benennen; alle Geldbeträge in jedem Service als ganze Zahl in Pence speichern; jedem Service eine eigene Datenbank geben oder eine gemeinsam nutzen. Welche sind architektonisch? Ziehen Sie Umkehrkosten, Reichweite und Wirkung auf Qualitäten heran.

Der Name der Hilfsfunktion ist lokales Design: billig umkehrbar, winziger Geltungsbereich, keine Wirkung auf Qualitäten. Das Geldformat wirkt untergeordnet, ist aber architektonisch, weil es sich durch jeden Service und jeden gespeicherten Datensatz zieht, später teuer zu ändern ist und die Korrektheit bestimmt. Gemeinsame oder getrennte Datenbanken sind eindeutig architektonisch: sehr schwer umkehrbar, breiter Geltungsbereich, und sie prägen Deploybarkeit und Konsistenz. Diese beiden verdienen eine schriftliche Dokumentation und ein breiteres Review, die erste nicht.

QUELLEN

- Who Needs an Architect? Martin Fowler, IEEE Software, 2003 (PDF): https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- Software Architecture Guide (Martin Fowler): https://martinfowler.com/architecture/
- The TOGAF Standard (The Open Group): https://www.opengroup.org/togaf
- iSAQB CPSA-Foundation curriculum: https://public.isaqb.org/curriculum-foundation/curriculum-foundation-en.html

Warum Architektur zählt: Änderungskosten, Qualität, Risiko und echte Ausfälle

DIE IDEE IN KÜRZE

Stellen Sie sich zwei Küchen vor, die dieselbe Speisekarte kochen. In der einen sind alle Stationen so angeordnet, dass sich die Köche selten in die Quere kommen; in der anderen steht der einzige Kühlschrank hinter der Spülmaschine. An einem ruhigen Abend servieren beide dieselben Gerichte. An einem vollen Samstag oder wenn sich die Speisekarte ändert, gerät die zweite Küche ins Stocken. Bei Software ist es ähnlich: Die Funktionen mögen identisch aussehen, aber die Anordnung der Teile entscheidet, wie schnell, zuverlässig und günstig das System ist und wie mühsam es sich ändern lässt. Einige der teuersten Softwareausfälle der Geschichte waren Probleme der Anordnung, nicht einzelner Codezeilen.

WAS SIE WISSEN MÜSSEN

- Die Architektur bestimmt weitgehend Qualitätsmerkmale wie Performance, Verfügbarkeit, Sicherheit und Kosten, weil diese daraus entstehen, wie die Teile angeordnet sind und zusammenwirken.
- Die Kosten, eine Entscheidung zu ändern, wachsen, je mehr Code, Daten, Integrationen und Teams darauf aufbauen; frühe strukturelle Entscheidungen prägen daher Kosten und Tempo aller späteren Arbeit.
- Die Architektur bestimmt das Risiko, wie sich Arbeit auf Teams aufteilen lässt (Gesetz von Conway), die Kommunikation und die Fähigkeit, Änderungen abzuschätzen.
- Knight Capital verlor am 1. August 2012 mehr als 460 Millionen USD, nachdem ein manuelles Deployment einen von acht Servern mit altem Code zurückgelassen hatte, den ein wiederverwendetes Flag reaktivierte.
- Das GAO stellte fest, dass HealthCare.gov im Oktober 2013 startete, ohne dass geprüft war, ob es die Performance-Anforderungen erfüllte – nach ungeklärten Anforderungen, schwacher Aufsicht und unvollständigen Tests.
- Prime Video verlagerte 2023 einen Monitoring-Service von verteilten Serverless-Schritten in einen einzigen Prozess und senkte damit Berichten zufolge die Infrastrukturkosten um über 90 % – ein Beleg dafür, dass es bei Architektur um Passung geht, nicht um Mode.

Die Entscheidungen, deren Fehler teuer sind

Abschnitt 1.1 hat Architektur als die Menge der Strukturen und bedeutenden Entscheidungen definiert, die ein System prägen, wobei sich die Bedeutung an den Änderungskosten bemisst. Dieser Abschnitt stellt die praktische Anschlussfrage: Na und? Warum sollte sich ein Team, ein Product Owner oder eine Finanzdirektorin für Architektur interessieren und nicht einfach nur für Funktionen?

Die kurze Antwort lautet: Die Architektur entscheidet mehr als jeder andere technische Faktor darüber, wie das System *beschaffen* sein wird: wie schnell, wie zuverlässig, wie sicher, wie günstig im Betrieb und vor allem wie leicht zu ändern. Funktionen sind das, was ein System tut. Die Architektur bestimmt, wie gut es das tut und was es kosten wird, im nächsten Jahr etwas anderes zu tun. Viele große Softwareausfälle und viele teure Erfolge lassen sich auf eine Handvoll struktureller Entscheidungen zurückführen.

Die Architektur bestimmt die Qualitätsmerkmale

Ein Qualitätsmerkmal ist eine messbare Eigenschaft dafür, wie gut ein System seine Aufgabe erfüllt: Performance, Verfügbarkeit, Sicherheit, Änderbarkeit, Skalierbarkeit, Benutzbarkeit, Kosteneffizienz und weitere. Kapitel 2 macht sie präzise. Entscheidend ist hier, dass die meisten Qualitätsmerkmale *emergent* sind: Sie entstehen daraus, wie die Teile angeordnet sind und zusammenwirken, nicht aus einem einzelnen Stück Code.

Nehmen Sie die Verfügbarkeit. Muss der Checkout eines Onlineshops nacheinander fünf synchrone Aufrufe an Services machen, die jeweils zu 99,9 % der Zeit verfügbar sind, kann die Kette als Ganzes nicht verfügbarer sein als ungefähr 99,5 %, so gut jeder Service auch geschrieben ist, weil sich die Wahrscheinlichkeiten multiplizieren. Eine Queue, damit einige dieser Schritte später stattfinden, oder ein Cache für Ergebnisse, damit manche Aufrufe entfallen, ändert die Rechnung. Keine der beiden Änderungen ist innerhalb einer einzelnen Funktion sichtbar; beide sind architektonisch. Performance, Sicherheit und Kosten verhalten sich genauso. Eine sichere Anmeldeseite macht ein System nicht sicher, wenn ein interner Service ohne Authentifizierung aus dem Internet erreichbar ist.

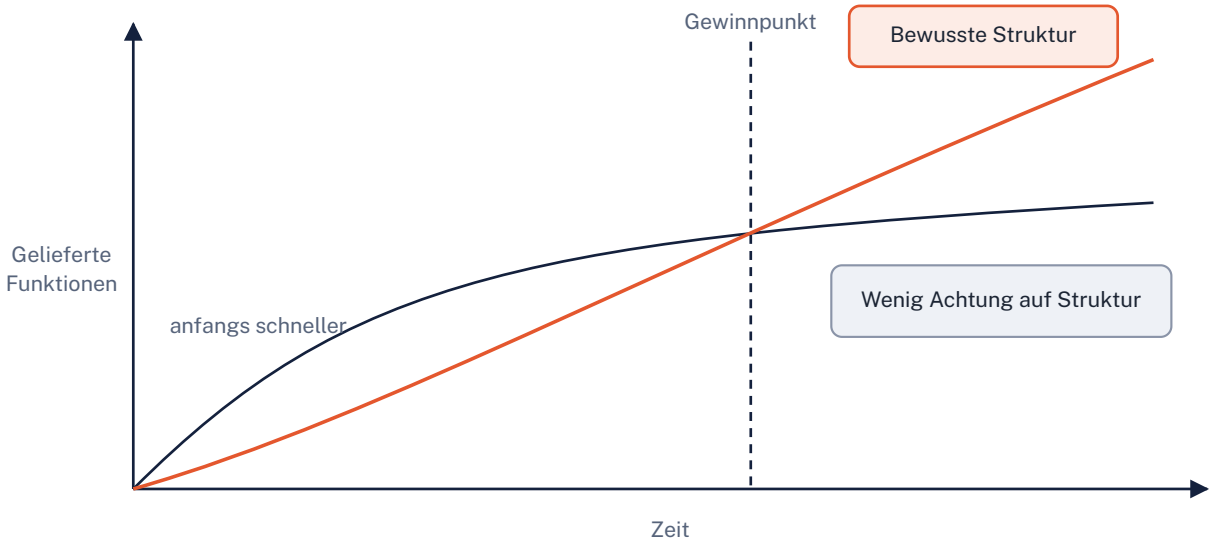
Die Funktionalität ist dagegen meist weitgehend unabhängig von der Struktur. Sie können dieselben Funktionen einer Banking-App als eine einzige deploybare Einheit oder als fünfzig Services umsetzen; die Nutzer sehen dieselben Schaltflächen. Was sich ändert, ist alles andere: wie schnell eine Korrektur ausgeliefert werden kann, was passiert, wenn ein Teil ausfällt, wie viele Entwicklerinnen und Entwickler das System aufnehmen kann und wie die Cloud-Rechnung aussieht.

Kosten und Tempo von Änderungen

Der Großteil der Kosten eines erfolgreichen Systems entsteht nach dem ersten Release, wenn es an neue Anforderungen angepasst wird. Die Struktur ist deshalb vor allem wichtig, weil sie bestimmt, wie teuer jede künftige Änderung wird. In einem gut strukturierten System betrifft eine Änderung der Steuerregeln ein Modul. In einem verworrenen System betrifft sie zwanzig, die alle getestet und abgestimmt werden müssen.

Martin Fowlers *Design Stamina Hypothesis* (2007) fasst den Effekt. Ein Team, das wenig auf die innere Struktur achtet, liefert anfangs oft schneller. Mit der Zeit bremst das angesammelte Durcheinander jede Änderung, und ein Team, das in Struktur investiert hat, überholt es. Fowler nennt dies eine Hypothese, weil es schwer streng zu messen ist, und er vermutet, dass der Kreuzungspunkt meist nach Wochen statt Monaten kommt. Die Verlangsamung sind die Zinsen auf die technischen Schulden, die Abschnitt 1.7 behandelt.

Die Design Stamina Hypothesis



Eine konzeptionelle Skizze nach Martin Fowler: Weniger Achtung auf Struktur ist anfangs schneller, aber bewusste Struktur hält das Liefertempo aufrecht.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: die Kosten einer Entscheidung im Zeitverlauf

Ein Onlineshop speichert Geldbeträge als Gleitkommazahlen. Gleitkommaarithmetik kann viele Dezimalwerte nicht exakt darstellen (0,1 + 0,2 ist nicht genau 0,3), sodass Summen gelegentlich um einen Penny von dem abweichen, was die Buchhaltung erwartet. Die Zahlen unten sind veranschaulichende Annahmen, keine Branchendaten, und sollen zeigen, wie die Änderungskosten wachsen.

Monat 1: anders entscheiden	Zwei Entwickler vereinbaren, stattdessen ganzzahlige kleinste Währungseinheiten (Pence) zu speichern. Aufwand: etwa ein halber Tag. Kosten bei angenommenen £600 pro Personentag: etwa £300.
Jahr 3: was auf der Entscheidung aufgebaut wurde	35 Services verarbeiten Geldbeträge; 120 Datenbankspalten speichern sie; 200 Millionen historische Bestellzeilen; 4 Partnerintegrationen (Zahlungsanbieter, Lager, Buchhaltung, Marktplatz) tauschen sie aus.
Jahr 3: Codeänderungen	35 Services zu je etwa 3 Personentagen für Änderung, Test und Release = 105 Personentage.
Jahr 3: Datenmigration	Spalten umwandeln, Historie migrieren, Summen abgleichen: etwa 20 Personentage, dazu ein geplantes Wartungsfenster.
Jahr 3: Abstimmung mit Partnern	4 Partner zu je etwa 10 Personentagen, einschließlich ihrer Tests und Vertragsänderungen = 40 Personentage.
Jahr 3: End-to-End-Tests und Abgleich mit der Finanzabteilung	Etwa 30 Personentage.
Jahr 3: Summe	Etwa 195 Personentage oder rund £117.000, dazu das Fehlerrisiko während der Migration und monatelange Durchlaufzeit. Das ist ein Mehrhundertfaches der Kosten in Monat 1.

Die Entscheidung selbst wurde nie größer. Gewachsen ist alles, was darauf aufgebaut wurde: Code, Daten, Partner und Gewohnheiten.

Deshalb beschrieb Ralph Johnson Architektur als die Entscheidungen, die man gern früh richtig treffen würde, und deshalb achten Architekten auf Konventionen auf niedriger Ebene mit großer Reichweite.

Eine alternative Strategie ist, solche Entscheidungen billiger umkehrbar zu machen, etwa indem man die gesamte Geldbehandlung von Anfang an hinter einem gemeinsamen Typ oder einer Bibliothek kapselt, sodass eine spätere Änderung lokal bleibt.

Risiko, Teams, Kommunikation und Schätzung

Risiko. In der Architektur werden die meisten technischen Risiken entweder eingedämmt oder verstärkt. Ein Entwurf mit einem Single Point of Failure, einer ungetesteten Integration zwischen unabhängig gebauten Teilen oder einer unerprobten Technologie im Kern trägt ein Risiko, das keine noch so sorgfältige Programmierung beseitigt. Ein Großteil der Architekturarbeit ist Risikomanagement: erkennen, was gründlich schiefgehen könnte, und die Struktur so gestalten, dass es entweder nicht passieren kann oder nur begrenzten Schaden anrichtet, wenn es passiert.

Arbeit auf Teams aufteilen. Melvin Conway beobachtete 1968, dass Organisationen, die Systeme entwerfen, dazu neigen, Entwürfe hervorzubringen, die ihre eigenen Kommunikationsstrukturen abbilden. Das heißt heute Gesetz von Conway, und es wirkt in beide Richtungen. Ein System, dessen Teile klare Grenzen und schmale Schnittstellen haben, erlaubt mehreren Teams, mit wenig Abstimmung parallel zu arbeiten. Ein System, in dem alles von allem abhängt, zwingt jedes Team in jedes Meeting. Architektur und Organisationsgestaltung sind daher eng verflochten, wie der Abschnitt zum Gesetz von Conway in Kapitel 3 erklärt.

Kommunikation. Eine Architektur gibt Menschen ein gemeinsames Vokabular und eine gemeinsame Landkarte. Neue Entwicklerinnen und Entwickler lernen das System schneller, Produktmanager verstehen, warum manche Wünsche billig und andere teuer sind, und der Betrieb weiß während eines Incidents, was wovon abhängt.

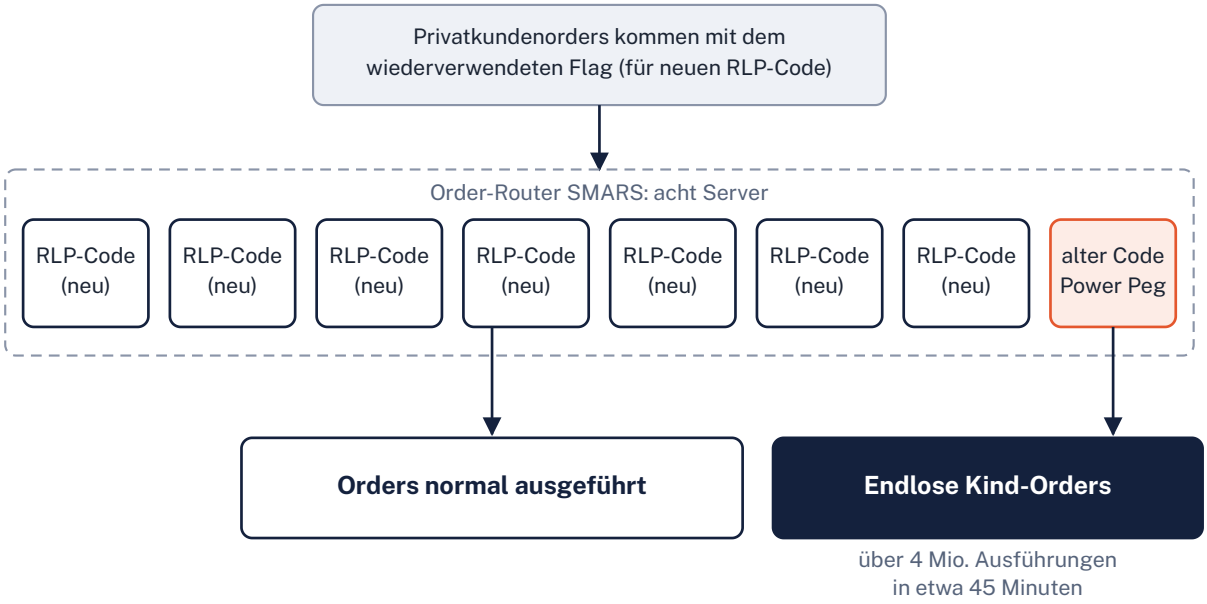
Schätzung und Planung. Die Zerlegung in bekannte Teile macht Schätzungen erst möglich: Eine Änderung lässt sich auf die Komponenten zurückführen, die sie berührt, und jede kann von dem Team geschätzt werden, das sie kennt. Ohne klare Struktur sind Schätzungen geraten, weil niemand mit Sicherheit sagen kann, wie weit eine Änderung Kreise zieht.

Fallstudie: Knight Capital, 1. August 2012

Knight Capital war einer der größten Market Maker für US-Aktien. Laut der Verfügung der US-Börsenaufsicht SEC (Securities and Exchange Commission) vom 16. Oktober 2013 sandte Knights automatischer Order-Router (SMARS genannt) in den ersten 45 Minuten des Handels am 1. August 2012 Millionen unbeabsichtigter Orders an den Markt, während er 212 kleine Privatkundenorders verarbeitete. Er erhielt mehr als 4 Millionen Ausführungen in 154 Aktien, über 397 Millionen Stück, und blieb auf großen ungewollten Positionen sitzen. Knight verlor mehr als 460 Millionen USD, brauchte binnen Tagen eine Notfallinvestition von außen, und die SEC verhängte später eine Geldbuße von 12 Millionen USD wegen Verstößen gegen die Marktzugangsregeln.

Die SEC-Verfügung beschreibt den Mechanismus. Jahre zuvor hatte Knight eine alte Funktion namens Power Peg nicht mehr verwendet, doch ihr Code blieb im Router und konnte weiterhin ausgelöst werden: Es war toter Code, der nicht tot war. 2012 schrieb Knight neuen Code für ein neues NYSE-Programm, das Retail Liquidity Program, und dieser neue Code verwendete ein Flag wieder, das früher Power Peg eingeschaltet hatte. Der neue Code wurde manuell auf die acht Server des Routers deployt. Ein Techniker kopierte ihn nicht auf einen davon, kein zweiter Techniker prüfte das Deployment, und Knight hatte kein schriftliches Verfahren, das eine solche Prüfung vorschrieb. Als das Flag eingeschaltet wurde, führten sieben Server den neuen Code aus; der achte deutete das Flag als „Power Peg ausführen“ und sandte Orders, ohne zu erkennen, dass sie bereits ausgeführt waren. Automatische E-Mails, die vor einem Problem warnten, wurden vor Markteröffnung verschickt, aber niemand handelte danach.

Wie ein veralteter Server den Verlust von Knight Capital verursachte



Kein zweiter Techniker prüfte das Deployment; keine automatische Prüfung bestätigte gleiche Server

Ein wiederverwendetes Flag, ein veralteter Server und keine Deployment-Prüfung: Der Fehler lag in Struktur und Prozess, nicht in einer einzelnen Codezeile.

Es wäre unfair, dies einfach einen Programmierfehler zu nennen, und ebenso unfair, es als reinen Betriebsfehler abzutun. Die architektonischen Lehren betreffen Struktur und Deploybarkeit: veraltete Funktionalität, die in einem Produktivsystem erreichbar blieb; ein Flag mit zwei Bedeutungen in verschiedenen Codeversionen; ein Deployment-Prozess ohne automatische Prüfung, ob alle Server dieselbe Version ausführten; und, wie die SEC betonte, unzureichende Kontrollen, um den Schaden zu begrenzen, sobald etwas schiefging. Jede davon ist eine Entscheidung darüber, wie das System gebaut und ausgeliefert wird, nicht über eine einzelne Codezeile.

Fallstudie: der Start von HealthCare.gov, Oktober 2013

HealthCare.gov, der bundesweite Marktplatz für Krankenversicherungen in den USA, öffnete am 1. Oktober 2013 für Anmeldungen. Viele Menschen konnten keine Konten anlegen oder Anträge abschließen, und die Website litt wochenlang unter Ausfällen. Der US-Rechnungshof GAO (Government Accountability Office) und das Office of Inspector General (OIG) des Gesundheitsministeriums (Department of Health and Human Services) untersuchten den Fall später; ihre Ergebnisse sind die beste Quelle dafür, was schiefging.

Der GAO-Bericht vom Juli 2014 (GAO-14-694) stellte fest, dass die Centers for Medicare and Medicaid Services (CMS) Aufträge vergaben, bevor zentrale Anforderungen geklärt waren, dass sich die Anforderungen ständig änderten, dass die Aufsicht schwach war und dass CMS das System startete, ohne zu prüfen, ob es seine Performance-Anforderungen erfüllte. Die Verpflichtungen aus dem Hauptvertrag für den Marktplatz stiegen zwischen 2011 und Anfang 2014 von 56 Millionen USD auf mehr als 209 Millionen USD. Ein zweiter GAO-Bericht vom März 2015 (GAO-15-238) stellte fest, dass die Systeme vor dem Start nicht vollständig getestet worden waren und dass zu den Problemen beim Start eine unzureichende Kapazitätsplanung, Programmierfehler und geplante, aber nicht umgesetzte Funktionen gehörten. Die Fallstudie des OIG von 2016 nannte das Fehlen einer klaren Leitung und die schlechte Abstimmung zwischen fachpolitischer und technischer Arbeit als zentrale Ursachen. Nach intensiven Korrekturarbeiten konnte die Website innerhalb von etwa zwei Monaten eine hohe Nachfrage bewältigen.

Die Geschichte von HealthCare.gov handelt vor allem vom Programmmanagement, doch ihre architektonische Lehre ist direkt. Der Marktplatz wurde von mehreren Auftragnehmern als getrennte Teile gebaut, die zusammenarbeiten mussten, darunter ein Daten-Hub mit Anbindung an andere Bundesbehörden, und niemand prüfte durchgängig unter realistischer Last, ob das zusammengesetzte Ganze seine Kapazitäts- und Performance-Ziele erreichte. Qualitätsmerkmale wie Kapazität sind Eigenschaften des Gesamtsystems; sie lassen sich daher nur von jemandem sicherstellen, der die Struktur des Gesamtsystems verantwortet und es als Ganzes testet.

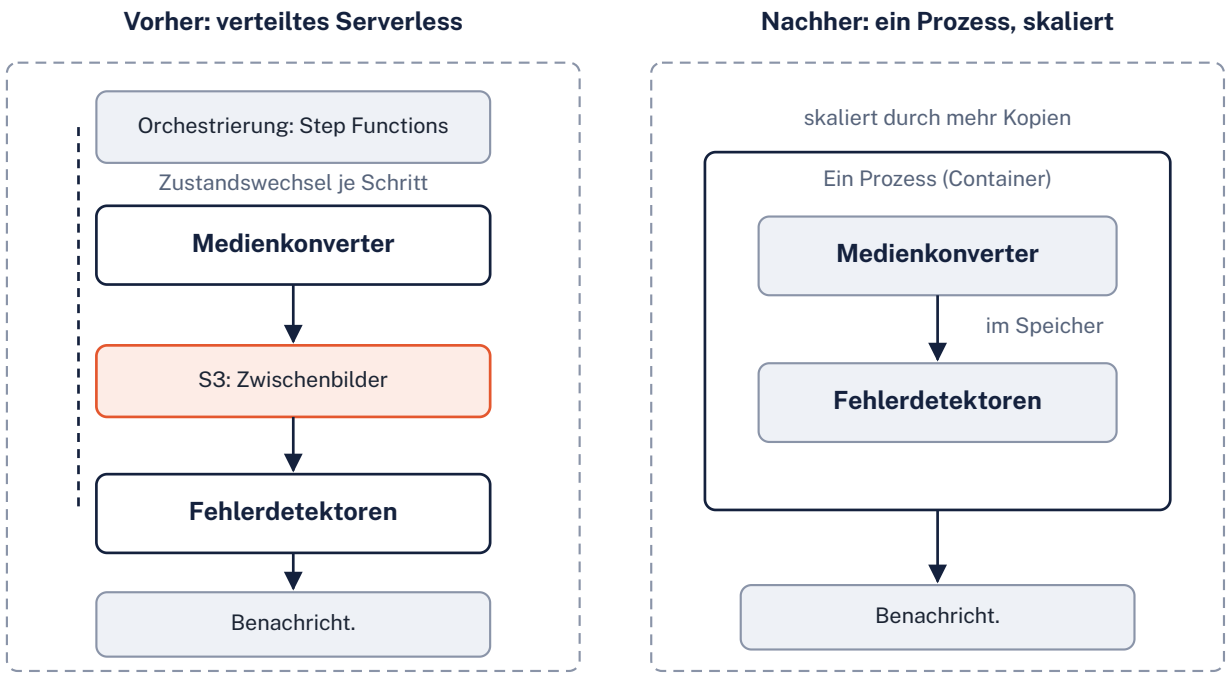
Fallstudie: Prime Video und Architektur als Passung, nicht als Mode

Im März 2023 veröffentlichte ein Engineer aus dem Video-Quality-Analysis-Team von Amazon Prime Video im Prime Video Tech Blog einen Beitrag über einen Service, der Videostreams auf Qualitätsmängel überwacht. Die erste Version war aus verteilten Serverless-Komponenten gebaut: AWS Step Functions orchestrierte die Schritte, und AWS-Lambda-Funktionen erledigten die Arbeit, wobei Zwischenbilder des Videos über den Speicher Amazon S3 zwischen den Schritten weitergereicht wurden. Das Team berichtete, dass dieser Entwurf bei

etwa 5 % der erwarteten Last an eine harte Skalierungsgrenze stieß und bei voller Last zu teuer gewesen wäre, vor allem wegen der Orchestrierungsgebühren pro Schritt und der vielen Speicheranfragen.

Das Team verlagerte die Medienkonvertierung und die Fehlererkennung in einen einzigen Prozess, der in Containern auf Amazon ECS lief, reichte die Bilder im Arbeitsspeicher statt über den Speicher weiter und skalierte, indem es mehr Kopien dieses Prozesses startete. Laut dem Beitrag sanken die Infrastrukturkosten um über 90 %, und der Service konnte weiter skalieren. Der Beitrag wurde vielfach als „Amazon gibt Microservices auf“ geteilt, was er nicht sagte: Die Änderung betraf einen Service, und der Autor merkte an, dass Microservices und Serverless-Komponenten bei hoher Last durchaus funktionieren, die Wahl zwischen ihnen und einem Monolithen aber von Fall zu Fall getroffen werden muss.

Der Monitoring-Service von Prime Video vorher und nachher



Dieselben Schritte, neu gruppiert: Daten im Arbeitsspeicher innerhalb eines Prozesses weiterzureichen beseitigte die Orchestrierungs- und Speicherkosten pro Schritt.

Die Lehre ist nicht, dass Monolithen Serverless schlagen. Sie lautet, dass es bei Architektur um Passung geht. Der Serverless-Entwurf war schnell gebaut und für eine frühe, kleinere Arbeitslast angemessen. Als das prägende Merkmal der Arbeitslast (sehr große Datenmengen, die für jede Sekunde jedes Streams zwischen den Schritten weitergereicht werden) dominierte, war eine Struktur, die diese Daten viele Male über das Netzwerk bewegte, die falsche Wahl, und eine andere Struktur passte besser. Ein guter Architekt fragt, was die Arbeitslast, das Team und das Budget brauchen, nicht, was gerade in Mode ist.

ACHTUNG

Fallstudien sorgfältig lesen

- Öffentliche Darstellungen vereinfachen. Nutzen Sie Primärquellen (Verfügungen von Aufsichtsbehörden, Prüfberichte, die eigenen Beiträge der Engineers) statt Nacherzählungen, die oft übertreiben oder Details hinzufügen.
- Die meisten echten Ausfälle haben mehrere Ursachen: technische Struktur, Prozess, Organisation und Anreize. Einem einzelnen Engineer oder einer einzelnen Codezeile die Schuld zu geben, ist fast immer falsch.
- Eine Entscheidung, die in einem Kontext gescheitert ist, kann in einem anderen richtig sein. Fragen Sie nach dem Kontext, bevor Sie eine allgemeine Regel ableiten.

VERTIEFUNG

Warum Architektur schwer zu begründen ist und wie es trotzdem gelingt

Der Nutzen guter Architektur besteht überwiegend aus Dingen, die *nicht* geschehen: der vermiedene Ausfall, die unnötige Neuentwicklung, das nicht blockierte Team. Deshalb wird er leicht unterschätzt. Wirksame Architekten übersetzen strukturelle Entscheidungen in betriebswirtschaftliche Größen: Durchlaufzeit von Änderungen, Kosten pro Transaktion, Ausfallstunden pro Jahr, die Zahl der Teams, die parallel arbeiten können, die Größe des getragenen Risikos. Kapitel 11 kommt darauf zurück, wie man Business Cases für Architekturarbeit aufstellt.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum kann ein gut geschriebener Service trotzdem Teil eines unzuverlässigen Systems sein?

Weil Verfügbarkeit daraus entsteht, wie Services voneinander abhängen. Eine Kette synchroner Aufrufe multipliziert die Ausfallwahrscheinlichkeiten, sodass das Ganze weniger verfügbar sein kann als jedes Teil.

▼ Was machte den Ausfall bei Knight Capital zu einem architektonischen Problem statt zu einem einfachen Bug?

Erreichbarer veralteter Code, ein Flag mit zwei Bedeutungen über Versionen hinweg und ein manuelles Deployment ohne automatische Prüfung, ob alle Server übereinstimmten: alles Entscheidungen über Struktur und Release, nicht eine fehlerhafte Zeile.

▼ **Was ist die wichtigste Lehre aus dem Fall Prime Video?**

Dass eine Architektur zu ihrer Arbeitslast und ihrem Kontext passen muss. Ein Entwurf, der bei einer bestimmten Größenordnung und einem bestimmten Datenmuster geeignet ist, kann unpassend werden, und eine geänderte Struktur kann die Kosten grundlegend verändern.

▼ **Warum wachsen die Kosten, eine Entscheidung zu ändern, mit der Zeit?**

Weil immer mehr Code, gespeicherte Daten, Integrationen, Partner und Gewohnheiten davon abhängen, sodass eine Änderung bedeutet, all das zu ändern.

▼ **Probieren Sie es aus: Ein neuer Ticketing-Service speichert Preise als Gleitkommazahlen. Die Umstellung auf ganze Pence kostet heute zwei Entwickler einen Tag. Die Teamleitung sagt: „Das machen wir nächstes Jahr, es ist nur ein Rundungsproblem." Wie lautet Ihr Argument, und an wen richten Sie es?**

Beheben Sie es jetzt. Heute hängt nichts vom Format ab; in einem Jahr steckt es in gespeicherten Buchungen, Berichten, Integrationen und im Code anderer Teams, sodass dieselbe Änderung eine Datenmigration, koordinierte Releases und einen Abgleich erfordert. Frühe strukturelle Entscheidungen bestimmen die Kosten aller späteren Arbeit. Tragen Sie es dem Lead und der Finanzverantwortung als geschäftliches Argument vor: ein Tag jetzt gegen Wochen später, während um einen Penny abweichende Summen die Buchhaltung so lange weiter verärgern.

QUELLEN

- SEC press release 2013-222: SEC charges Knight Capital with violations of market access rule: https://www.sec.gov/news/press-release/2013-222
- GAO-14-694: Healthcare.gov: Ineffective Planning and Oversight Practices Underscore the Need for Improved Contract Management: https://www.gao.gov/products/gao-14-694
- GAO-15-238: Healthcare.gov: CMS Has Taken Steps to Address Problems, but Needs to Further Implement Systems Development Best Practices: https://www.gao.gov/products/GAO-15-238
- Marcin Kolny: Scaling up the Prime Video audio/video monitoring service and reducing costs by 90% (Prime Video Tech, 2023; the original post, which may no longer be online): https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90
- InfoQ: Prime Video moves from serverless to ECS, cutting costs (report on the original post, 2023): https://infoq.com/news/2023/05/prime-ec2-ecs-saves-costs
- Design Stamina Hypothesis (Martin Fowler): https://martinfowler.com/bliki/DesignStaminaHypothesis.html

Was ein Architekt tut: Verantwortlichkeiten, Fähigkeiten und Arten von Architekten

DIE IDEE IN KÜRZE

Bei einem Bauprojekt setzt der Architekt nicht jeden Ziegel selbst, sondern entscheidet die Dinge, bei denen ein Fehler verheerend wäre, erklärt sie den Bauleuten und den Bauherren, prüft auf der Baustelle, ob die Pläne eingehalten werden, und passt an, wenn sich der Boden als weicher erweist als erwartet. Ein Softwarearchitekt tut dasselbe für ein Softwaresystem: trifft oder lenkt die großen Entscheidungen, erklärt sie, prüft fortlaufend, ob sie eingehalten werden und noch sinnvoll sind, und muss ein wenig über viele Technologien und eine Menge über Menschen und das Geschäft wissen.

WAS SIE WISSEN MÜSSEN

- Architekten übernehmen die Verantwortung dafür, dass bedeutende Entscheidungen gut getroffen, verstanden und bei Änderungen des Systems eingehalten werden; Architektur ist eine Rolle, bevor sie eine Berufsbezeichnung ist.
- Richards und Ford beschreiben acht Erwartungen: Entscheidungen treffen, fortlaufend analysieren, auf dem neuesten Stand bleiben, die Einhaltung sicherstellen, vielfältige Erfahrung haben, die Fachdomäne kennen, zwischenmenschliche Fähigkeiten besitzen und mit Politik umgehen.
- Der iSAQB-Lehrplan nennt als Aufgaben von Architekten: Anforderungen klären, Strukturen und querschnittliche Konzepte entwerfen, kommunizieren, die Umsetzung begleiten und bewerten – und zwar iterativ.
- Zu den Arten von Architekten gehören Enterprise-, Solution-, Software- oder Anwendungs-, Daten-, Sicherheits- sowie Cloud- oder Infrastrukturarchitekten; Staff und Principal Engineers tragen oft dieselben Verantwortlichkeiten.
- Architekten schätzen technische Breite (viele Optionen und ihre Abwägungen kennen) höher als tiefes Fachwissen in wenigen, wie die Wissenspyramide zeigt.
- Architekten sollten in der Regel praktisch mitarbeiten, ohne Code auf dem kritischen Pfad zu verantworten: Proofs of Concept, Fitness Functions, Bugfixes und Code-Reviews.

Wozu es Architekten gibt

Wenn Architektur die Menge der bedeutenden Entscheidungen und Strukturen eines Systems ist (Abschnitt 1.1), dann ist Architekt, wer die Verantwortung dafür übernimmt, dass diese Entscheidungen gut getroffen, von anderen verstanden und bei Änderungen des Systems eingehalten werden. Das klingt einfach, doch die Rolle gehört zu den am uneinheitlichsten definierten in der Softwareentwicklung. In einem Unternehmen ist ein Architekt ein erfahrener Engineer, der noch täglich Code schreibt; in einem anderen ein Mitglied eines zentralen Teams, das Entwürfe prüft und Standards pflegt; in einem dritten jemand, der vor allem mit der Geschäftsleitung und mit Anbietern spricht.

Hinter der Vielfalt steht ein gemeinsamer Kern. Architekten treffen oder lenken Entscheidungen, die sich nur teuer umkehren lassen, kümmern sich um die Qualitäten des Gesamtsystems statt nur eines Teils und verbinden technische Entscheidungen mit Geschäftszielen. Der Rest dieses Abschnitts beschreibt, was dazugehört, welche Fähigkeiten es erfordert und wie die Rolle organisiert ist.

Acht Erwartungen an einen Architekten

Mark Richards und Neal Ford beschreiben in *Fundamentals of Software Architecture* (O'Reilly, 1. Auflage 2020, 2. Auflage 2025) acht zentrale Erwartungen an einen Architekten, unabhängig von Berufsbezeichnung oder Unternehmen. Sie sind unten sinngemäß wiedergegeben, jeweils mit einem Beispiel aus einem Essenslieferunternehmen.

Die acht Erwartungen (nach Richards und Ford)

Erwartung	Was sie bedeutet	Beispiel
Architekturentscheidungen treffen	Entscheidungen und Prinzipien festlegen, die Technologieentscheidungen lenken, meist indem man das Team anleitet, statt bestimmte Produkte vorzuschreiben.	Festlegen, dass Restaurantintegrationen über eine einzige Adapterschicht laufen, statt für jedes Restaurant die Bibliothek auszuwählen.
Die Architektur fortlaufend analysieren	Neu bewerten, ob die Architektur noch passt, wenn sich Technologie und Geschäft ändern, und nach Verfall suchen.	Bemerken, dass der Kurierzuordnungsservice inzwischen 40 % aller fehlgeschlagenen Deployments verursacht, und nach dem Grund fragen.
Bei Trends auf dem Laufenden bleiben	Entwicklungen in Technologie und Praxis verfolgen, weil Architekturentscheidungen langlebig sind.	Neue verwaltete Datenbankangebote vor der nächsten Kapazitätsprüfung verstehen.
Die Einhaltung von Entscheidungen sicherstellen	Prüfen, ob vereinbarte Entscheidungen und Prinzipien tatsächlich befolgt werden, idealerweise mit automatischen Prüfungen.	Eine Build-Prüfung hinzufügen, die fehlschlägt, wenn die App den Zahlungsanbieter direkt aufruft.
Vielfältige Erfahrung und Einblicke haben	Eine breite Palette von Technologien, Frameworks und Plattformen gut genug kennen, um sie zu vergleichen.	Message Broker hinsichtlich Reihenfolge, Dauerhaftigkeit und Betriebskosten vergleichen.
Fachdomänenwissen haben	Das Geschäft gut genug verstehen, um mit Stakeholdern in deren Begriffen zu sprechen.	Wissen, wie Auszahlungen an Restaurants und Erstattungen funktionieren, bevor man die Zahlungen neu entwirft.
Zwischenmenschliche Fähigkeiten besitzen	Führen, zusammenarbeiten, mentoren und Meinungsverschiedenheiten lösen; der Großteil der Architekturarbeit geschieht durch andere Menschen.	Eine Sitzung moderieren, in der sich zwei Teams auf einen Event-Vertrag einigen.
Politik verstehen und damit umgehen	In einer Organisation, in der Budgets, Prioritäten und Zuständigkeiten umstritten sind, Unterstützung für Entscheidungen gewinnen.	Die Zustimmung von Finanzabteilung und Betrieb einholen, bevor ein altes Berichtssystem abgeschaltet wird.

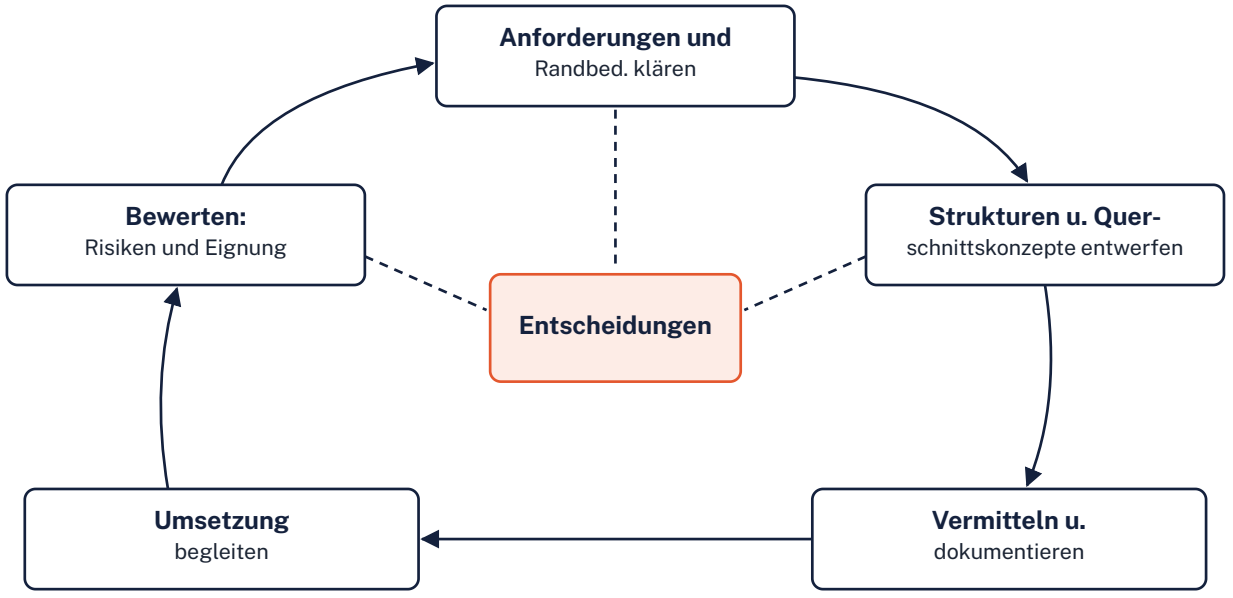
Beachten Sie, wie wenige der acht rein technisch sind. Richards und Ford betonen, dass fast jede Entscheidung eines Architekten von jemandem infrage gestellt wird (Entwicklern, anderen Architekten, Product Ownern, Projektmanagern), sodass Verhandlungsgeschick und Überzeugungskraft Kernfähigkeiten sind, keine Extras. Eine Architektur, die auf dem Papier richtig ist, aber nicht verstanden oder akzeptiert wird, wird nicht wie beabsichtigt gebaut.

Die iSAQB-Sicht auf die Aufgaben von Architekten

Das International Software Architecture Qualification Board (iSAQB), dessen Foundation-Level-Lehrplan (CPSA-F) in Europa weit verbreitet ist, beschreibt die Aufgaben und Verantwortlichkeiten von Architekten ähnlich. Zusammengefasst: Architekten klären und hinterfragen Anforderungen und Randbedingungen; entscheiden, wie das System in Bausteine mit ihren Abhängigkeiten und Schnittstellen zerlegt wird; entscheiden über Querschnittsanliegen wie Persistenz, Kommunikation und Benutzeroberfläche; kommunizieren und dokumentieren die Architektur; begleiten ihre Realisierung und Umsetzung; analysieren und bewerten sie, besonders auf Risiken für die Erfüllung der Anforderungen; und erklären anderen Stakeholdern die Folgen von Architekturentscheidungen.

Der Lehrplan betont außerdem, dass diese Aufgaben iterativ sind: Architekten arbeiten in Zyklen und greifen Anforderungen und Entscheidungen erneut auf, wenn Rückmeldungen eintreffen, statt einmal zu entwerfen und dann zu übergeben. Abschnitt 1.8 zeigt, wie sich das in iterative und kontinuierliche Auslieferung einfügt.

Die Arbeit des Architekten als Zyklus



Jede Aufgabe speist und überprüft Entscheidungen; der Zyklus läuft das ganze Systemleben lang

Nach dem iSAQB-Lehrplan: Klären, Entwerfen, Vermitteln, Begleiten und Bewerten wiederholen sich, und alles dreht sich um Entscheidungen.

Arten von Architekten

Die Bezeichnungen folgen den Ebenen der Architektur, die in Abschnitt 1.2 beschrieben sind. Die genauen Verantwortlichkeiten unterscheiden sich stark zwischen Organisationen; betrachten Sie die Tabelle unten daher als typisch, nicht als verbindlich.

Art	Schwerpunkt	Typische Ergebnisse
Enterprise-Architekt	Die gesamte Organisation: Fähigkeiten, Anwendungsportfolio, Technologiestandards, Ausrichtung an der Strategie	Capability Maps, Roadmaps zum Zielzustand, Standards, Investitionsempfehlungen
Lösungsarchitekt	Ein Geschäftsproblem oder Programm, das mehrere Systeme, Anbieter und Teams umspannt	Lösungsentwürfe, Integrationsentwürfe, Anbieterbewertungen
Software- oder Anwendungsarchitekt	Struktur und Qualitäten eines Systems oder einer Gruppe eng verwandter Systeme	Komponentenstrukturen, Entscheidungsdokumente, Qualitätsszenarien, technische Anleitung für Teams
Datenarchitekt	Wie Daten modelliert, verantwortet, gespeichert, bewegt, gesteuert und analysiert werden	Datenmodelle, Übersichten der Datenverantwortung, Entwürfe für Pipelines und Plattformen
Sicherheitsarchitekt	Identität, Zugriff, Schutz von Daten und die Platzierung von Sicherheitskontrollen	Bedrohungsmodelle, Sicherheitsmuster, Kontrollanforderungen
Cloud- oder Infrastrukturarchitekt	Rechenleistung, Netzwerk, Speicher, Umgebungen und Plattformen	Landing Zones, Netzwerkentwürfe, Plattformstandards, Kostenmodelle

Eine Rolle oder eine Berufsbezeichnung?

Architektur ist eine Reihe von Verantwortlichkeiten, bevor sie eine Berufsbezeichnung ist. In vielen Organisationen, besonders kleineren und solchen, die um autonome Produktteams herum organisiert sind, heißt niemand „Architekt“. Die Architekturarbeit wird trotzdem erledigt, von technischen Leads, erfahrenen Engineers oder dem Team gemeinsam. Entscheidend ist, dass jemand die bedeutenden Entscheidungen und die systemweiten Qualitäten klar verantwortet. Tut das niemand, entsteht trotzdem eine Architektur, aber zufällig.

Viele Technologieunternehmen haben eine technische Karriereleiter für erfahrene Fachkräfte ohne Führungsverantwortung, mit Stufen wie Staff Engineer und Principal Engineer. Will Larsons *Staff Engineer* (2021) beschreibt vier verbreitete Archetypen solcher Rollen, von denen einer ausdrücklich „Architect“ heißt: eine Person, die für Richtung und Qualität eines kritischen

technischen Bereichs verantwortlich ist. Auch die anderen (Tech Lead, Solver und Right Hand) tragen erhebliche architektonische Verantwortung. In diesen Organisationen leisten Menschen die Arbeit des Architekten, die zugleich Engineers sind.

Martin Fowlers Artikel *Who Needs an Architect?* von 2003 stellt zwei Zerrbilder gegenüber: einen Architekten, der alle wichtigen Entscheidungen allein trifft, und einen, der eng mit dem Team arbeitet, Entwickler anleitet und ihnen hilft, selbst gute Entscheidungen zu treffen. Fowler sprach sich entschieden für den zweiten aus. Die meisten Fachleute sind sich heute einig, dass Architekten, die aus der Ferne Vorgaben machen, meist Entwürfe hervorbringen, die die Teams stillschweigend umgehen.

Konzeptionelle Integrität: warum jemand das Ganze im Blick behalten muss

Fred Brooks lieferte in *The Mythical Man-Month* (1975) die klassische Begründung für diese Rolle: „Ich behaupte, dass konzeptionelle Integrität die wichtigste Überlegung beim Systementwurf ist." Ein System sollte wirken, als hätte es ein einziger Kopf entworfen, mit einem Satz von Ideen, der einheitlich angewendet wird, selbst wenn dafür gute, aber nicht dazu passende Funktionen wegfallen. Er verglich das mit der Einheit der Kathedrale von Reims, deren Baumeister einige ihrer eigenen Ideen zugunsten des Ganzen aufgaben. Als Abhilfe empfahl er, Architektur und Implementierung zu trennen, und er nahm den Vorwurf der Aristokratie in Kauf. Moderne Teams teilen diese Arbeit breiter, doch der Bedarf bleibt: Wenn die Essenslieferplattform drei Arten hat, eine Rückerstattung zu behandeln, sollte das jemandem auffallen und jemanden kümmern.

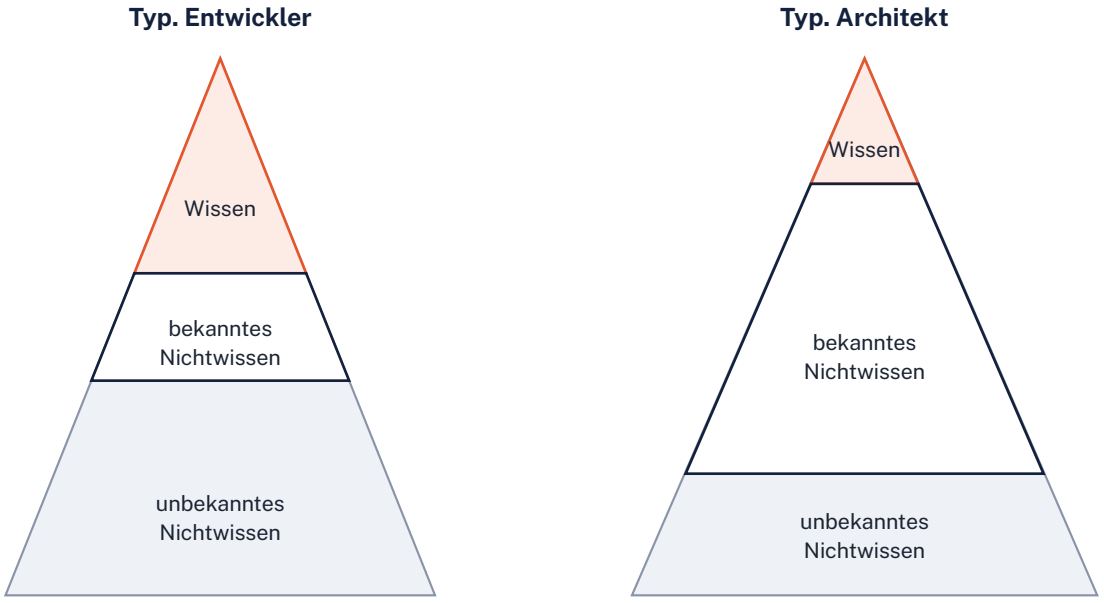
Technische Breite oder Tiefe

Entwickler werden meist für Tiefe belohnt: eine Sprache, ein Framework oder eine Plattform äußerst gut zu kennen. Architekten brauchen ein anderes Profil. Richards und Ford beschreiben eine Wissenspyramide mit drei Schichten: Dinge, die Sie wissen (wo Sie echtes Fachwissen haben), Dinge, von denen Sie wissen, dass Sie sie nicht wissen (Sie wissen, dass es sie gibt und ungefähr wofür sie gut sind, könnten aber heute nichts damit bauen), und Dinge, von denen Sie nicht wissen, dass Sie sie nicht wissen. Fachwissen in der obersten Schicht muss gepflegt werden, und das kostet Zeit. Die mittlere Schicht lässt sich günstig erweitern und ermöglicht es, Optionen zu erkennen.

Der Wert eines Architekten liegt großenteils in technischer Breite: zu wissen, dass es fünf Wege gibt, ein Problem zu lösen, wofür jeder gut und schlecht ist und welcher einen genaueren Blick verdient. Ein Architekt, der nur eine Datenbank kennt, wird sie jedes Mal empfehlen. Breite hat ihren Preis, denn Zeit, die man in die Verbreiterung der mittleren Schicht steckt, fehlt für die Vertiefung der obersten; Architekten lassen deshalb bewusst manches alte Fachwissen

verblassen und investieren darin, über mehr Dinge Bescheid zu wissen. Richards und Ford warnen vor dem umgekehrten Fehler: dem Architekten, dessen Ansichten von einer schlechten Erfahrung vor Jahren geprägt wurden und der jede Option ablehnt, die ihr ähnelt.

Die Wissenspyramide



Tiefe (oben) bleibt wichtig, aber Breite (Mitte) lässt einen Architekten Optionen vergleichen

Nach Richards und Ford: Ein Architekt verbreitert bewusst die mittlere Schicht, also Dinge, von denen man weiß, dass es sie gibt, ohne sie zu beherrschen.

Sollten Architekten Code schreiben?

Hier sind sich Fachleute wirklich uneinig, und der Streit ist nützlich, weil beide Seiten in einem Punkt recht haben.

Was für das Programmieren spricht. Architekten, die programmieren, verstehen die tatsächlichen Kosten ihrer Entscheidungen, merken, wenn ein Prinzip in der Praxis schmerzt, halten ihr technisches Urteilsvermögen aktuell und behalten den Respekt der Entwickler, die ihre Entwürfe umsetzen müssen. Architekten, die jahrelang nicht programmieren, laufen Gefahr, Systeme zu entwerfen, die am Whiteboard elegant und im Editor sperrig sind.

Was dagegen spricht. Die Zeit eines Architekten verteilt sich auf viele Teams, Meetings und Entscheidungen. Ein Architekt, der Features auf dem kritischen Pfad verantwortet, wird zum Engpass: Das Team wartet auf Code, der sich wegen der anderen Aufgaben des Architekten verzögert. Seine Aufmerksamkeit wird außerdem für teamübergreifende Anliegen gebraucht, um die sich kein einzelnes Team kümmert.

Die übliche Lösung. Die meisten erfahrenen Stimmen, darunter Richards und Ford, empfehlen, dass Architekten auf eine Weise praktisch mitarbeiten, die sie nicht auf den kritischen Pfad bringt. Beispiele sind ein Proof of Concept, um eine riskante Option zu testen, bevor sich das Team darauf festlegt; das Schreiben von Fitness Functions, automatischen Prüfungen, die Architekturregeln verifizieren, etwa „das Domänenmodul darf nicht vom Web-Framework abhängen“; Bugs beheben und kleinere Arbeiten an technischen Schulden erledigen; Werkzeuge und Automatisierung für das Team bauen; und regelmäßig Code-Reviews machen, die zeigen, wie Entscheidungen tatsächlich umgesetzt werden.

```
import pathlib, re

FORBIDDEN = re.compile(r"^\s*(from|import)\s+flask\b", re.M)

def test_domain_does_not_depend_on_web_framework():
    offenders = []
    for path in pathlib.Path("shop/domain").rglob("*.py"):
        if FORBIDDEN.search(path.read_text(encoding="utf-8")):
            offenders.append(str(path))
    assert not offenders, f"Domain code imports Flask: {offenders}"
```

Eine einfache Fitness Function, wie ein Architekt sie schreiben könnte: Der Build schlägt fehl, wenn ein Domänenmodul das Web-Framework importiert.

Eine Woche im Leben eines Softwarearchitekten (zur Veranschaulichung)

Tag	Hauptaktivitäten	Wahrgenommene Erwartung
Montag	Planung mit zwei Produktteams; bemerken, dass ein neues Treueprogramm-Feature Daten braucht, die der Bestellservice nicht bereitstellt; Optionen entwerfen	Entscheidungen treffen; die Fachdomäne kennen
Dienstag	Workshop mit dem Datenschutzbeauftragten und der IT-Sicherheit zur Speicherung von Kundenadressen; ein Entscheidungsdokument verfassen	Zwischenmenschliche Fähigkeiten; Einhaltung sicherstellen
Mittwoch	Zweitägiger Proof of Concept eines verwalteten Event-Streaming-Service, um Reihenfolgegarantien unter Last zu testen	Vielfältige Erfahrung; auf dem Laufenden bleiben
Donnerstag	Code-Reviews im Zahlungsteam; nach Entdecken einer verbotenen Abhängigkeit eine Fitness Function hinzufügen; Einzelmentoring	Einhaltung sicherstellen; zwischenmenschliche Fähigkeiten
Freitag	Incident-Berichte und Durchlaufzeit-Kennzahlen auf Anzeichen architektonischen Verfalls prüfen; Termin mit der Finanzabteilung zu Cloud-Kosten; Lektüre	Fortlaufend analysieren; mit Politik umgehen; auf dem Laufenden bleiben

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine Entscheidung, fünf Aufgaben

Das Terminbuchungssystem eines Krankenhauses muss Erinnerungen per SMS verschicken. Der Softwarearchitekt arbeitet für diesen einen Bedarf die iSAQB-Aufgaben durch.

Anforderungen klären	Fragt, wie viele Erinnerungen pro Tag (etwa 20.000), wie schnell sie hinausgehen müssen, was passiert, wenn der SMS-Anbieter ausfällt, und ob Nachrichten klinische Details enthalten dürfen (aus Datenschutzgründen nicht).
Design	Schlägt vor, dass der Buchungsservice Events „Termin gebucht“ veröffentlicht und eine separate Benachrichtigungskomponente Nachrichten aus einer Queue versendet, sodass ein SMS-Ausfall nie Buchungen blockiert.
Kommunizieren	Schreibt ein einseitiges Entscheidungsdokument mit den erwogenen Optionen (direkter Aufruf oder Queue) und den Gründen; stellt es dem Team und der Leitung für Information Governance vor.
Umsetzung begleiten	Arbeitet beim ersten Release im Pair Programming mit einer Entwicklerin, prüft die Retry-Logik und fügt eine Fitness Function hinzu, die klinische Felder in der Nachrichtenvorlage verhindert.
Bewerten	Prüft drei Monate später Zustellzeiten und Fehlerquoten; stellt fest, dass Erinnerungen bei einem Ausfall des Anbieters 30 Minuten in der Queue warten, und bestätigt mit dem klinischen Personal, dass dies akzeptabel ist.

Der Großteil des Aufwands des Architekten floss in Klären, Kommunizieren und Prüfen, nicht in das Zeichnen des Entwurfs.

Die Entscheidung wurde mit ihren Gründen festgehalten, sodass ein künftiges Team sie neu prüfen kann, wenn sich der Kontext ändert.

Häufige Fragen

Wie werde ich Architekt?

Meist, indem man Architekturarbeit leistet, bevor man den Titel hat: querschnittliche Entscheidungen im eigenen Team verantworten, Entscheidungsdokumente schreiben, die eigene technische Breite bewusst erweitern, die Fachdomäne kennenlernen und üben, Abwägungen für nicht technische Menschen verständlich zu machen. Zertifizierungen wie der iSAQB CPSA-F können dieses Lernen strukturieren, ersetzen aber keine Erfahrung.

Braucht jedes Team einen Architekten?
Jedes Team braucht klar zugeordnete Architekturverantwortung, aber nicht unbedingt eine eigene Person dafür. Kleine Teams übertragen sie oft einem technischen Lead; größere Organisationen haben möglicherweise Architekten, die jeweils mehrere Teams unterstützen.

Was ist der Unterschied zwischen einem technischen Lead und einem Architekten?
Ein technischer Lead ist meist für die technische Lieferung eines Teams verantwortlich und verbringt einen Großteil seiner Zeit mit Programmieren; ein Architekt ist meist für Entscheidungen und Qualitäten verantwortlich, die Teams oder Systeme übergreifen. Die Rollen überschneiden sich stark, und in vielen Organisationen ist der technische Lead der Architekt für das System seines Teams.

Wie viel Zeit sollte ein Architekt in Meetings verbringen?
Es gibt keine richtige Zahl, aber ein nützlicher Test ist, ob die Meetings Entscheidungen, gemeinsames Verständnis oder beseitigte Hindernisse hervorbringen. Gregor Hohpes Bild vom Architekten, der mit dem Aufzug zwischen Vorstandsetage und Maschinenraum pendelt, erinnert daran, dass beide Enden regelmäßig besucht werden müssen.

ACHTUNG

Anti-Patterns in der Architektenrolle

- **Der Elfenbeinturm:** Entwürfe werden ohne Beteiligung an der Umsetzung vorgegeben und von den Teams ignoriert oder umgangen.
- **Der Engpass:** Jede noch so kleine Entscheidung muss über den Architekten laufen, sodass sich die Lieferung verlangsamt und die Teams aufhören mitzudenken.
- **Der erstarrte Experte:** ein Urteilsvermögen, das in Technologien oder schlechten Erfahrungen von vor Jahren verankert ist.
- **Lebenslaufgetriebene Architektur:** Technologien werden gewählt, weil sie den Architekten interessieren, nicht weil sie zum Problem passen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Warum zählen Richards und Ford Politik zu den Erwartungen an einen Architekten?**

Weil fast jede Architekturentscheidung von jemandem infrage gestellt wird und eine Entscheidung, die über Budgets, Prioritäten und Teams hinweg keine Unterstützung gewinnt, nicht wie beabsichtigt umgesetzt wird.

▼ Welche Schicht der Wissenspyramide sollte ein Architekt bewusst erweitern, und warum?

Die mittlere Schicht, also die Dinge, von denen Sie wissen, dass Sie sie nicht wissen, denn zu wissen, dass es Optionen gibt und wofür sie gut sind, ermöglicht es einem Architekten, Lösungen zu vergleichen.

▼ Nennen Sie zwei Wege, wie ein Architekt praktisch mitarbeiten kann, ohne zum Engpass zu werden.

Zwei beliebige aus: Proofs of Concept bauen, Fitness Functions schreiben, Bugs oder kleine Schuldenposten beheben, Werkzeuge und Automatisierung bauen und Code reviewen.

▼ Probieren Sie es aus: Eine Architektin bei einem Essenslieferdienst, die Stärkste beim Programmieren im Raum, bietet an, das zentrale Zahlungsmodul selbst zu schreiben, weil es für alle anderen zu riskant sei. Ist das ein guter Einsatz der Rolle? Was würden Sie ihr stattdessen raten?

Nein. Wenn sie Code auf dem kritischen Pfad verantwortet, wird sie zum Engpass, und wenn Architekturarbeit sie abzieht, steckt das Team beim wichtigsten Modul fest. Sie sollte auf andere Weise praktisch beteiligt bleiben: einen Proof of Concept des riskanten Teils bauen, den Code reviewen, Fitness Functions ergänzen, die die Regeln schützen, und einige Fehler beheben. Eine starke Entwicklerin oder ein starker Entwickler verantwortet dann das Modul, was Wissen verteilt und ihr erlaubt, die Verantwortung für die weiterreichenden Entscheidungen zu tragen.

QUELLEN

- iSAQB CPSA-Foundation curriculum (LG 01-04: tasks and responsibilities of software architects): https://public.isaqb.org/curriculum-foundation/curriculum-foundation-en.html
- Who Needs an Architect? Martin Fowler, IEEE Software, 2003 (PDF): https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- Staff archetypes (Will Larson, StaffEng): https://staffeng.com/guides/staff-archetypes/
- The Architect Elevator (Gregor Hohpe): https://architectelevator.com/
- Brooks: The Mythical Man-Month, chapter 4 (sample chapter): https://ptgmedia.pearsoncmg.com/images/0201835959/samplechapter/chap4.html

Stakeholder, ihre Anliegen und wie Architektur ihnen dient

DIE IDEE IN KÜRZE

Denken Sie an den Bau eines Hauses. Die Familie, die darin wohnen wird, wünscht sich Platz und Licht. Die Bank, die das Geld leiht, will, dass das Haus den Kredit wert ist. Das Bauamt will, dass es in die Straße passt. Der Bauunternehmer will Pläne, die sich tatsächlich bauen lassen, und die Nachbarn wollen nachts keinen Lärm. Sie alle sind Stakeholder, und das, worüber sich jeder Einzelne Gedanken macht, ist ein Anliegen. Auch ein Softwaresystem hat seine Stakeholder: Nutzer, Zahlende, Entwicklungsteams, die Menschen, die es am Laufen halten, Aufsichtsbehörden. Architektur ist die Kunst, alle anzuhören und offen zu entscheiden, wo ihre Wünsche kollidieren.

WAS SIE WISSEN MÜSSEN

- Ein Stakeholder ist jede Rolle, Person, Organisation oder Gruppe davon mit einem Interesse, Recht, Anteil oder Anspruch an einem System, einschließlich derer, die es nie direkt nutzen, etwa Aufsichtsbehörden und Betreiber.
- Ein Anliegen ist nach ISO/IEC/IEEE 42010 jede Angelegenheit, die für einen Stakeholder relevant oder wichtig ist: Qualitäten, Kosten, Risiken, Randbedingungen, Machbarkeit und organisatorische Fragen zählen alle dazu.
- Anliegen sind noch keine Anforderungen: Halten Sie sie in den Worten des Stakeholders fest und schärfen Sie sie dann zu messbaren Qualitätsszenarien, Randbedingungen oder funktionalen Anforderungen.
- Die Macht-Interesse-Matrix hilft zu planen, wie Sie jeden Stakeholder einbinden, sagt aber nichts darüber, wessen Anliegen am wichtigsten sind; Gruppen mit wenig Macht können am meisten zu verlieren haben.
- Widersprüchliche Anliegen sind normal; die Architektin oder der Architekt macht sie explizit, beziffert die Optionen, benennt, wer entscheidet, und hält die Begründung fest.
- Anliegen strukturieren die Architekturdokumentation: Jeder Blickwinkel erfasst bestimmte Anliegen, und jede Sicht zeigt, wie das System sie adressiert.

Wer die Stakeholder sind

Ein Stakeholder ist jeder, der ein berechtigtes Interesse an einem System hat: Menschen, die es nutzen, bezahlen, bauen, betreiben, regulieren oder von ihm betroffen sind. ISO/IEC/IEEE 42010:2022, die internationale Norm für Architekturbeschreibungen, definiert den Begriff weit:

als Rolle, Position, Einzelperson, Organisation oder Gruppe davon, die ein Interesse, Recht, Anteil oder einen Anspruch an dem beschriebenen Gegenstand hat. Zwei Merkmale dieser Definition sind in der Praxis wichtig. Erstens ist ein Stakeholder oft eine *Gruppe* und keine Person: „Patienten“ oder „die Bereitschaftsingenieure“ statt einer namentlich bekannten Person. Zweitens genügt ein Anspruch oder ein Recht: Eine Aufsichtsbehörde, die sich nie in Ihr System einloggt, ist trotzdem ein Stakeholder, weil sie Ihnen den Betrieb untersagen kann.

Die Liste ist länger, als die meisten Teams erwarten. Bei einem typischen kommerziellen System umfasst sie:

Nutzer (die Menschen, die die Software im Alltag bedienen, oft mehrere unterschiedliche Gruppen); **Kunden** (wer bezahlt, ist womöglich gar kein Nutzer); den **Product Owner** oder Produktmanager, der entscheidet, was gebaut wird; den **Sponsor**, der die Arbeit finanziert und für ihren Erfolg geradesteht; **Entwickler**, die den Code verstehen und ändern müssen; **Tester**, die Daten bereitstellen und Verhalten beobachten können müssen; **Betriebspersonal und Site Reliability Engineers (SREs)**, die das System deployen, überwachen und um drei Uhr morgens wiederherstellen; das **Sicherheitsteam**; den **Datenschutzbeauftragten**, wo personenbezogene Daten verarbeitet werden (nach UK GDPR und DSGVO müssen manche Organisationen einen benennen); **Recht und Einkauf**, denen es um Lizenzen, Verträge und Haftung geht; die **Finanzabteilung**, der die Betriebskosten ebenso wichtig sind wie die Entwicklungskosten; **Aufsichtsbehörden und Prüfer**; sowie **Integratoren von Drittanbietern**, also die anderen Teams und Unternehmen, deren Software Ihre aufruft oder von ihr aufgerufen wird.

Manche Stakeholder fehlen in jedem Meeting: die Entwickler, die das System in fünf Jahren warten werden, der Support, der die Beschwerden entgegennimmt, Menschen mit Behinderungen, die die Oberfläche nutzen werden, und Bürgerinnen und Bürger, die von seinen Entscheidungen betroffen sind. Zu den Aufgaben einer Architektin oder eines Architekten gehört es, für diejenigen zu sprechen, die nicht im Raum sind.

Anliegen: worauf es Stakeholdern ankommt

In 42010 ist ein Anliegen jede Angelegenheit, die für einen Stakeholder relevant oder wichtig ist. Der Begriff ist bewusst weit gefasst. Ein Anliegen kann eine Qualität sein („Bleibt die Buchungsseite am Montagmorgen schnell?“), Kosten („Was kostet der Betrieb pro Monat?“), ein Risiko („Was passiert, wenn der SMS-Anbieter ausfällt?“), eine Randbedingung („Müssen Patientendaten im Vereinigten Königreich bleiben?“), eine Frage der Machbarkeit („Kann unser Team das mit seinen vorhandenen Fähigkeiten bauen?“) oder eine organisatorische Frage („Welches Team ist für die Integration mit dem Hausarztsystem zuständig?“).

Anliegen sind noch keine Anforderungen. Meist werden sie vage formuliert, im eigenen Vokabular des Stakeholders, und oft sind es eher Sorgen als Spezifikationen. „Ich will nicht noch einmal einen Ausfall wie im letzten Winter“ ist ein Anliegen; „der Buchungsservice ist in jedem Kalendermonat zu 99,9 % verfügbar, gemessen am Load Balancer“ ist eine daraus abgeleitete Anforderung. Wenn Sie das Anliegen zuerst in den Worten des Stakeholders festhalten und es erst dann schärfen, bleibt nachvollziehbar, woher eine Anforderung stammt und warum sie wichtig ist.

Ein Anliegen wird oft von mehreren Stakeholdern geteilt, und ein Stakeholder hat meist viele Anliegen. Sicherheit ist ein Anliegen des Sicherheitsteams, des Datenschutzbeauftragten, des Sponsors (der Schlagzeilen fürchtet) und der Patienten (die fürchten, dass ihre Akten nach außen dringen). Genau wegen dieser Viele-zu-viele-Beziehung behandelt 42010 Anliegen als Scharnier zwischen Menschen und Architektur: Entscheidungen beantworten Anliegen, und Sichten werden nach den Anliegen gegliedert, die sie adressieren.

Typische Stakeholder und ihre Anliegen

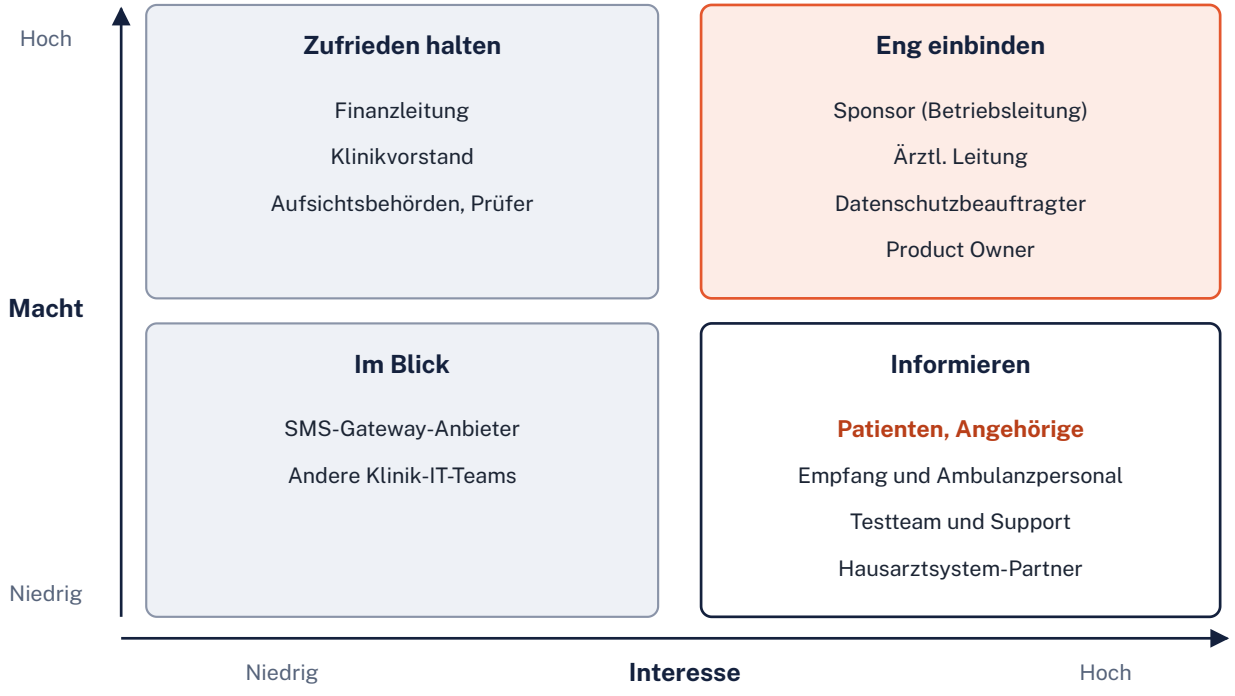
Stakeholder	Typische Anliegen	Was die Architektur ihnen bieten muss
Nutzer	Geschwindigkeit, Benutzerfreundlichkeit, Barrierefreiheit, ein System, das da ist, wenn man es braucht	Ziele für Antwortzeit und Verfügbarkeit; eine Oberflächenschicht, die Barrierefreiheitsstandards erfüllen kann
Kunden und Sponsoren	Geschäftlicher Nutzen, Time-to-Market, Gesamtkosten, Reputationsrisiko	Ein glaubwürdiger Lieferplan, ein Kostenmodell und eine Liste der Hauptrisiken
Product Owner	Wie schnell neue Features hinzukommen können, wie Optionen offen bleiben	Modulare Struktur entlang der Fachdomäne; wenige schwer umkehrbare Entscheidungen
Entwickler	Verständlichkeit, Build- und Testgeschwindigkeit, klare Zuständigkeiten	Saubere Modulgrenzen, ein schneller lokaler Build, dokumentierte Konventionen
Tester	Testbarkeit, steuerbare Testdaten, beobachtbares Verhalten	Nahtstellen, an denen sich Abhängigkeiten ersetzen lassen; Testumgebungen, die der Produktion ähneln
Betrieb und SRE	Deploybarkeit, Monitoring, Wiederherstellung, Kapazität, Bereitschaftslast	Health Checks, Logs, Metriken und Traces; sicherer Rollback; Runbooks
Sicherheitsteam	Angriffsfläche, Identität, Secrets, Reaktion auf Schwachstellen	Vertrauensgrenzen, Design der Authentifizierung, Strategie für Patches und Abhängigkeiten
Datenschutzbeauftragter	Rechtsgrundlage, Datenminimierung, Speicherdauer, Betroffenenrechte, Übermittlungen	Ein Dateninventar und eine Datenflusssicht; Mechanismen für Löschung und Export
Recht und Einkauf	Lizenzen, Verträge, Haftung, Lock-in beim Anbieter	Eine Liste der Komponenten und Dienste von Drittanbietern mit ihren Bedingungen
Finanzen	Entwicklungskosten, Betriebskosten, Planbarkeit der Kosten	Ein Kostenmodell pro Umgebung und pro Nutzungseinheit
Aufsichtsbehörden und Prüfer	Einhaltung branchenspezifischer Vorschriften, Nachweis der Kontrolle	Audit-Trails, Rückverfolgbarkeit von der Anforderung zum Test, dokumentierte Entscheidungen
Integratoren von Drittanbietern	Stabile Schnittstellen, Versionierung, Dokumentation, Test-Sandboxes	Veröffentlichte API-Verträge und eine Versionierungsrichtlinie

Stakeholder erfassen: die Macht-Interesse-Matrix

Bei zwanzig oder mehr Stakeholder-Gruppen können Sie nicht jeder die gleiche Aufmerksamkeit schenken. Ein weit verbreitetes Werkzeug ist die Macht-Interesse-Matrix, die meist Aubrey Mendelow zugeschrieben wird, dessen Aufsatz über Stakeholder und Umfeldanalyse aus dem Jahr 1981 stammt (oft wird er mit 1991 zitiert). Jeder Stakeholder wird danach eingeordnet, wie viel Macht er über das System hat (es zu finanzieren, zu genehmigen, zu blockieren oder umzugestalten) und wie viel Interesse er daran zeigt. Die Quadranten legen eine Strategie der Einbindung nahe: **eng einbinden** (viel Macht, viel Interesse), **zufrieden halten** (viel Macht, wenig Interesse), **informiert halten** (wenig Macht, viel Interesse) und **im Blick behalten** (wenig Macht, wenig Interesse).

Bei der Matrix geht es um Einbindung, nicht darum, wessen Anliegen zählen. Die Patienten eines Krankenhauses haben wenig formale Macht über das Design eines Buchungssystems, aber mehr auf dem Spiel als alle anderen, und manche gesetzlichen Pflichten (Barrierefreiheit, Datenschutz) gibt es gerade, um Menschen mit wenig Macht zu schützen. Verstehen Sie die Matrix als Erinnerung daran, mit wem Sie wie oft sprechen sollten, und urteilen Sie getrennt davon, welche Anliegen kritisch sind. Auch Positionen verschieben sich: Eine Aufsichtsbehörde mit geringem Interesse wird in der Woche nach einer Datenpanne zum interessiertesten Stakeholder von allen.

Eine Macht-Interesse-Matrix für ein Krankenhaus-Buchungssystem



Patienten liegen im Quadranten mit wenig Macht, haben aber am meisten auf dem Spiel; deshalb steuert die Matrix die Einbindung und nicht die Priorität.

Eine schlanke Stakeholder-Analyse

1. **Identifizieren**
   Sammeln Sie mit dem Sponsor und dem Team Stakeholder-Gruppen und gleichen Sie die Liste mit einem Standardsatz ab (Nutzer, Zahlende, Entwickelnde, Betreibende, Schützende, Regulierende, Integrierende). Fragen Sie jede gefundene Gruppe: Mit wem sollten wir noch sprechen?

2. **Anliegen erheben**
   Führen Sie mit jeder wichtigen Gruppe ein Interview oder einen Workshop durch. Halten Sie die Anliegen in ihren eigenen Worten fest und notieren Sie dann die Qualität, Randbedingung oder das Risiko hinter jedem einzelnen.

3. **Einordnen und priorisieren**
   Ordnen Sie die Stakeholder in der Macht-Interesse-Matrix ein, um die Einbindung zu planen. Bewerten Sie getrennt davon die Anliegen nach geschäftlicher Auswirkung und danach, wie stark sie die Architektur einschränken.

4. **Konflikte explizit machen**
   Wo zwei Anliegen in entgegengesetzte Richtungen ziehen, schreiben Sie den Konflikt auf, benennen Sie die Personen, die zustimmen müssen, und vereinbaren Sie, wer entscheidet.

5. **Nachverfolgen und überprüfen**
   Verknüpfen Sie jede bedeutsame Entscheidung mit den Anliegen, die sie adressiert, und wiederholen Sie die Analyse an wichtigen Meilensteinen, denn Stakeholder und ihre Anliegen ändern sich.

Widersprüchliche Anliegen und wie man sie explizit macht

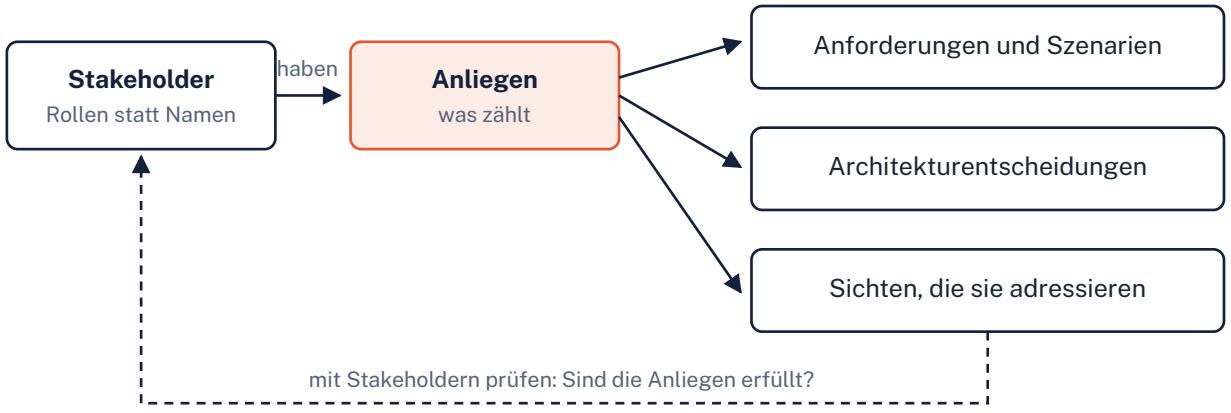
Anliegen widersprechen sich ständig, und eine Architektur, die alle zufriedenstellt, bedeutet meist, dass ein Konflikt verdeckt statt gelöst wurde. Das Sicherheitsteam will jede Anmeldung mit Multi-Faktor-Authentifizierung schützen; die Patientenvertretung weist darauf hin, dass viele ältere Patienten kein Smartphone haben. Die Finanzabteilung will einen einzigen kleinen Datenbankserver; der Betrieb will ein Standby-Replikat in einem zweiten Rechenzentrum. Der Product Owner will jede Woche ein neues Feature; die Tester wollen Zeit für Regressionstests der Integration mit dem Patientenverwaltungssystem des Krankenhauses.

Der Beitrag der Architektur besteht nicht darin, stillschweigend einen Sieger zu küren, sondern den Konflikt sichtbar, beziffert und verantwortet zu machen. In der Praxis heißt das vier Dinge. **Formulieren Sie beide Anliegen präzise**, idealerweise als testbare Szenarien, damit über Zahlen statt über Adjektive gestritten wird. **Stellen Sie die Optionen und ihre Konsequenzen** nebeneinander: zum Beispiel MFA für das Personal und für Patienten, die sich dafür

entscheiden, mit einem Einmalcode per SMS oder Festnetzanruf als Ausweichlösung. **Benennen Sie, wer entscheidet**: Manche Abwägungen gehören dem Sponsor oder einem Risikoausschuss, nicht der Architektur. **Halten Sie das Ergebnis und seine Begründung fest**, damit es eine Antwort gibt, wenn in zwei Jahren jemand fragt, warum sich Patienten ohne MFA anmelden können. Der nächste Abschnitt, Entscheidungen und Abwägungen, behandelt, wie man solche Optionen gegeneinander abwägt.

Manche Konflikte lösen sich bei näherem Hinsehen auf. Das Sicherheitsteam braucht MFA vielleicht nicht für jede Aktion der Patienten, sondern nur für das Abrufen von Befunden; damit kann die Patientengruppe womöglich leben. Wenn Sie jede Seite fragen, *warum* sie das Anliegen hat und welches Ergebnis sie eigentlich fürchtet, zeigt sich oft ein Design, das beide zufriedenstellt.

Wohin Anliegen führen



Anliegen sind das Bindeglied zwischen Menschen und Architektur: Sie werden zu Anforderungen, treiben Entscheidungen und bestimmen, welche Sichten Sie zeichnen.

Wie Anliegen in Anforderungen und Dokumentation einfließen

Anliegen sind das Rohmaterial für zwei spätere Tätigkeiten. Die erste ist das Requirements Engineering. Ein Anliegen zur Geschwindigkeit wird zu einer Anforderung an ein Qualitätsmerkmal, die Sie in Kapitel 2 als messbares Szenario formulieren lernen (Quelle, Stimulus, Artefakt, Umgebung, Antwort und Antwortmaß): siehe Testbare Qualitätsszenarien schreiben. Ein Anliegen wie „Patientendaten müssen im Vereinigten Königreich bleiben“ wird zu einer *Randbedingung*, einer Entscheidung, die bereits für Sie getroffen ist. Anliegen, die sich als bedeutungslos für die Struktur des Systems erweisen, bleiben gewöhnliche funktionale oder Projektanforderungen.

Die zweite ist die Dokumentation. 42010 gliedert eine Architekturbeschreibung nach Anliegen: Ein Blickwinkel benennt die Anliegen, die er erfasst, und die Stakeholder, denen er dient, und eine Sicht wendet diesen Blickwinkel auf Ihr System an. Die Anliegen des Betriebsteams

verlangen eine Verteilungssicht mit Servern, Regionen und Failover; die des Datenschutzbeauftragten eine Datenflusssicht, die zeigt, wohin personenbezogene Daten fließen und wo sie gespeichert sind; Entwickler brauchen eine Modulsicht, die zeigt, was wovon abhängt. Ein Diagramm, das kein Anliegen eines Stakeholders beantwortet, ist Dekoration. Kapitel 4 behandelt das ausführlich in Sichten und Blickwinkel.

Rückverfolgbarkeit verbindet beides. Wenn Sie eine Linie von einem Stakeholder über ein Anliegen zu einer Anforderung, einer Entscheidung und einer Sicht ziehen können, können Sie die zwei Fragen beantworten, die jede Prüferin und jeder Prüfer irgendwann stellt: Warum ist es so gebaut, und haben wir jemanden vergessen?

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: die Stakeholder eines Systems zur Terminbuchung im Krankenhaus

Ein regionaler Krankenhausverbund möchte ein Onlinesystem, mit dem Patienten ambulante Termine buchen, ändern und absagen können, als Ersatz für die telefonische Buchung. Es muss mit dem bestehenden Patientenverwaltungssystem (PAS) integriert werden und Erinnerungen per SMS versenden. Hier ist die erste Stakeholder-Analyse, die eine Architektin oder ein Architekt in den ersten zwei Wochen erstellen könnte.

Patienten, Angehörige	Anliegen: einfache Buchung auf dem Smartphone, Barrierefreiheit (Screenreader, große Schrift), Erinnerungen, Privatsphäre. Auswirkung auf die Architektur: responsives Web-Frontend, das WCAG 2.2 AA erfüllt; keine App-Installation nötig; nur minimale Daten auf dem Bildschirm.
Empfang und Ambulanzpersonal	Anliegen: keine Doppelbuchungen, Sprechstundenlisten, die der Realität entsprechen, weniger Anrufe. Auswirkung: Das PAS bleibt die Single Source of Truth für Termine; das Buchungssystem darf keine eigene Kopie halten, die auseinanderlaufen kann.
Ärztl. Leitung	Anliegen: Dringende Überweisungen dürfen nicht von Routinebuchungen verdrängt werden; klinische Sicherheit. Auswirkung: Terminregeln, die von Ärzten verantwortet werden und ohne Release konfigurierbar sind; ein klinischer Sicherheitsnachweis (in England nach dem Standard DCB0129), der das System abdeckt.
Datenschutzbeauftragter	Anliegen: Gesundheitsdaten sind nach UK GDPR besondere Kategorien personenbezogener Daten; Minimierung; Speicherdauer; Audit. Auswirkung: eine Datenflusssicht, Verschlüsselung bei der Übertragung und im Ruhezustand, Audit-Logs darüber, wer was eingesehen hat, eine Löschrichtlinie.
Betriebsteam	Anliegen: Spitzen am Montagmorgen, die begrenzte Kapazität des PAS, Support außerhalb der Dienstzeiten. Auswirkung: eine Queue oder ein Rate Limiter vor dem PAS; Caching der Terminverfügbarkeit mit kurzer Ablaufzeit; Monitoring der Antwortzeiten des PAS.
Finanzleitung	Anliegen: festes Budget, planbare Betriebskosten, Einsparungen beim Callcenter-Personal. Auswirkung: Managed Cloud Services, auf die Spitzenlast ausgelegt; ein Kostenmodell pro tausend Buchungen.
PAS-Anbieter (Integrator)	Anliegen: keine nicht abgestimmte Last auf seinem System; nur Nutzung seiner veröffentlichten Schnittstelle. Auswirkung: Integration über die Schnittstelle des Anbieters mit vereinbarten Rate Limits; eine Contract-Test-Suite.

Wichtigster gefundener Konflikt	Der Betrieb will die Terminverfügbarkeit cachen, um das PAS zu schützen; das Empfangspersonal fürchtet Doppelbuchungen durch veraltete Daten. Vorgeschlagene Lösung: Cache nur für die Anzeige (30 Sekunden), aber jede Buchung in Echtzeit gegen das PAS bestätigen, mit einer klaren Meldung, falls der Termin gerade vergeben wurde.

Sieben Stakeholder-Gruppen führten zu mindestens fünf Entscheidungen mit strukturellen Folgen (Single Source of Truth, Rate Limiting, Audit-Logging, konfigurierbare Regeln, barrierefreies Frontend), bevor irgendein Feature im Detail entworfen war.

Der folgenreichste Konflikt bestand zwischen zwei unauffälligen Gruppen, dem Betrieb und dem Empfangspersonal, von denen keine eine Lenkungsausschusssitzung dominiert hätte.

Jede Zeile lässt sich vorwärts zu einem Qualitätsszenario, einem Entscheidungsdokument und einer Sicht verfolgen und rückwärts zu einer benannten Gruppe von Menschen.

ACHTUNG

Häufige Fehler

So geht die Arbeit mit Stakeholdern am häufigsten schief.

- „Das Business“ als einen einzigen Stakeholder behandeln. Vertrieb, Finanzen, Compliance und Kundensupport wollen unterschiedliche, oft widersprüchliche Dinge.
- Den Betrieb vergessen. Systeme, die nur um Nutzer und Entwickler herum entworfen werden, sind regelmäßig schwer zu deployen, zu überwachen und wiederherzustellen.
- Lautstärke mit Wichtigkeit verwechseln. Wer an jedem Meeting teilnimmt, ist nicht unbedingt derjenige, dessen Anliegen das Projekt zum Scheitern bringen wird.
- Sich auf Stellvertreter verlassen. Ein Product Owner kann Nutzer vertreten, aber nicht immer sagen, was jemand, der auf Barrierefreiheit angewiesen ist, oder eine Pflegekraft in der Nachtschicht tatsächlich erlebt. Sprechen Sie mit einigen echten Nutzern.
- Die Analyse nur einmal machen. Stakeholder kommen und gehen, Vorschriften ändern sich, und ein neuer Integrationspartner bringt neue Anliegen mit. Überprüfen Sie die Liste an jedem wichtigen Meilenstein.

Häufige Fragen

Ist das Entwicklungsteam wirklich ein Stakeholder? Wir bauen das System doch.
Ja. Entwickler leben länger mit der Architektur als fast alle anderen, und ihre Anliegen (Build-Zeit, Testbarkeit, klare Zuständigkeiten) schlagen sich direkt in der Liefergeschwindigkeit nieder. Die Definition in 42010 unterscheidet nicht zwischen denen, die bauen, und denen, die nutzen.

Was ist der Unterschied zwischen einem Anliegen eines Stakeholders und einer Anforderung?
Ein Anliegen ist das zugrunde liegende Interesse, oft vage („Ich will, dass es zuverlässig ist“). Eine Anforderung ist eine präzise, überprüfbare Aussage, die aus einem oder mehreren Anliegen abgeleitet ist („99,9 % monatliche Verfügbarkeit“). Mehrere Anforderungen können einem Anliegen dienen, und manche Anliegen werden stattdessen zu Randbedingungen.

Wer entscheidet, wenn zwei mächtige Stakeholder uneins sind?
Meist nicht die Architektin oder der Architekt allein. Die Architektur stellt die Optionen und Konsequenzen dar; die Entscheidung liegt bei dem, der für das Ergebnis verantwortlich ist, typischerweise beim Sponsor oder einem Governance-Gremium. Wer vorab vereinbart, wer welche Art von Abwägung entscheidet, spart wochenlange Diskussionen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Nennen Sie drei Stakeholder eines typischen kommerziellen Systems, die selten an Designbesprechungen teilnehmen.**

Zum Beispiel: die Entwickler, die es in künftigen Jahren warten werden, der Support, Nutzer mit Behinderungen, Betriebspersonal in nächtlicher Bereitschaft und Bürgerinnen und Bürger, die von seinen Entscheidungen betroffen sind.

▼ **Ein Sponsor sagt: „Es muss sicher sein.“ Ist das ein Anliegen oder eine Anforderung?**

Ein Anliegen. Es muss zu konkreten, testbaren Anforderungen verfeinert werden, etwa zu Authentifizierung, Verschlüsselung oder Audit-Logging, bevor es Entscheidungen leiten kann.

▼ **Warum kann ein Anliegen wie Sicherheit mehreren Stakeholdern zugleich gehören?**

Weil verschiedene Menschen unterschiedliche Folgen derselben Qualität fürchten: Das Sicherheitsteam fürchtet Angriffe, der Datenschutzbeauftragte Verletzungen des Schutzes personenbezogener Daten, der Sponsor Reputationsschäden und die Nutzer, dass ihre Daten nach außen dringen.

▼ **Was verbindet ein Anliegen eines Stakeholders mit einer Architektursicht?**

Ein Blickwinkel: Er legt fest, welchen Anliegen und Stakeholdern er dient und wie sie zu modellieren sind, und die Sicht wendet ihn auf das System an.

▼ **Probieren Sie es aus: Das Marketing, einflussreich und engagiert, wünscht sich für den Vorverkaufstag einen animierten Saalplan. Eine kleine Gruppe von Menschen mit Behinderung sagt, dass Screenreader ihn nicht nutzen können. Eine Führungskraft ordnet beide in der Macht-Interesse-Matrix ein und sagt, das Marketing gewinne. Kritisieren Sie diese Begründung und sagen Sie, was zu tun ist.**

Das Raster hilft nur bei der Planung, wie man Menschen einbindet; es sagt nichts darüber aus, wessen Anliegen am wichtigsten ist, und bei Gruppen mit wenig Macht kann am meisten auf dem Spiel stehen. Hier konnten manche Kunden gar nicht kaufen. Halten Sie beide Anliegen in deren eigenen Worten fest, und machen Sie dann den Konflikt ausdrücklich: etwa eine per Tastatur und Screenreader nutzbare Liste neben der Karte, mit ihren Kosten. Beziffern Sie die Optionen, benennen Sie, wer entscheidet, und dokumentieren Sie die Begründung, einschließlich etwaiger Pflichten zur Barrierefreiheit.

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
- Stakeholder analysis for UX projects (Nielsen Norman Group): https://www.nngroup.com/articles/stakeholder-analysis/
- arc42 section 3: Context and scope: https://docs.arc42.org/section-3/

Architekturentscheidungen und Abwägungen: Umkehrbarkeit und Zeitpunkt

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 1.2 Architektur und Design
- 1.5 Stakeholder und Anliegen

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

DIE IDEE IN KÜRZE

Die Wahl eines Autos ist eine Abwägung: Ein größerer Motor verbraucht mehr Kraftstoff, ein kleineres Auto lässt sich leichter einparken, transportiert aber weniger. Niemand bekommt alles. Manche Entscheidungen lassen sich leicht rückgängig machen, etwa eine Lackfarbe, die man später ändern kann; andere, wie der Kauf eines Hauses direkt an der Autobahn, nur sehr schwer. Softwarearchitektur ist voll von solchen Entscheidungen. Gute Architektinnen und Architekten treffen die leicht umkehrbaren schnell, gehen mit den schwer umkehrbaren sorgfältig um, entscheiden zum richtigen Zeitpunkt statt zu früh oder zu spät und schreiben auf, warum sie sich so entschieden haben, damit ihre Nachfolger es verstehen.

WAS SIE WISSEN MÜSSEN

- Jede bedeutsame Architekturentscheidung ist eine Abwägung: Sie verbessert manche Qualitäten auf Kosten anderer, und die Nachteile müssen gefunden und benannt werden.
- Eine gut aufgebaute Entscheidung hält ihren Kontext fest, die betrachteten Optionen, die Kriterien, die Entscheidung mit ihrer Begründung, die Konsequenzen und ihren Status.
- Umkehrbarkeit ist die entscheidende Eigenschaft einer Entscheidung: Über Zweiwegtüren sollte schnell und vor Ort entschieden werden, über Einwegtüren mit Bedacht und gemeinsam mit denen, die die Konsequenzen tragen.
- Der Last Responsible Moment ist der Zeitpunkt, ab dem ein Aufschub eine wichtige Option zunichtemachen würde; wer dann entscheidet, nutzt das meiste Wissen, ohne die Entscheidung dem Zufall zu überlassen.
- Abstraktionen an Grenzen, Feature-Flags, schrittweise Migration und Spikes machen Entscheidungen umkehrbarer, zu Kosten, die sich nur lohnen, wo sowohl die Unsicherheit als auch die Kosten einer Umkehr hoch sind.

- Gewichtete Entscheidungsmatrizen und Architecture Decision Records machen Überlegungen sichtbar und angreifbar; vereinbaren Sie Kriterien und Gewichte vor der Bewertung und prüfen Sie, wie empfindlich das Ergebnis auf sie reagiert.

Alles ist eine Abwägung

Jede bedeutsame Architekturentscheidung gewinnt etwas, indem sie etwas anderes aufgibt. Wer ein System in unabhängig deploybare Services aufteilt, gewinnt Autonomie der Teams und unabhängige Skalierung und bezahlt mit Netzwerklatenz, betrieblicher Komplexität und schwierigerer Datenkonsistenz. Caching gewinnt Geschwindigkeit und bezahlt mit veralteten Daten und Invalidierungslogik. Eine verwaltete Cloud-Datenbank befreit von Patches und Backups und kostet dafür Geld und Abhängigkeit von einem Anbieter. Mark Richards und Neal Ford stellen dies an den Anfang von *Fundamentals of Software Architecture* (2020; zweite Auflage 2025), als ihr erstes Gesetz: „Alles in der Softwarearchitektur ist eine Abwägung." Als Folgerung leiten sie ab: Wenn Sie glauben, etwas gefunden zu haben, das keine Abwägung ist, haben Sie die Abwägung wahrscheinlich nur noch nicht erkannt.

Ihr zweites Gesetz lautet, dass das *Warum* wichtiger ist als das *Wie*. Jeder kann sich ein laufendes System ansehen und erkennen, wie es gebaut ist: drei Services, ein Message Broker, eine Dokumentendatenbank. Was sich aus dem Code nicht rekonstruieren lässt, ist das Warum: welche Alternativen erwogen wurden, welche Anliegen sich durchsetzten, welche Risiken in Kauf genommen wurden. Ohne das Warum bewahren spätere Entwickler entweder Entscheidungen, die keinen Sinn mehr ergeben, oder sie machen Entscheidungen rückgängig, deren Gründe weiterhin gelten. Die zweite Auflage fügt ein drittes Gesetz hinzu: Die meisten Architekturentscheidungen sind nicht binär, sondern liegen auf einem Spektrum zwischen Extremen. Selten wählen Sie zwischen „Monolith" und „Microservices"; Sie wählen, wie viele deploybare Einheiten es gibt, entlang welcher Linien geschnitten wird und was geteilt wird.

In Abwägungen zu denken ist die Gewohnheit, die architektonisches Denken am deutlichsten von Begeisterung unterscheidet. Wenn jemand eine Technologie vorschlägt, lauten die nützlichen Fragen nicht „Ist sie gut?", sondern „Was kostet sie, was erschwert sie, und sind diese Kosten angesichts unserer Anliegen akzeptabel?" Der vorige Abschnitt hat gezeigt, woher diese Anliegen kommen.

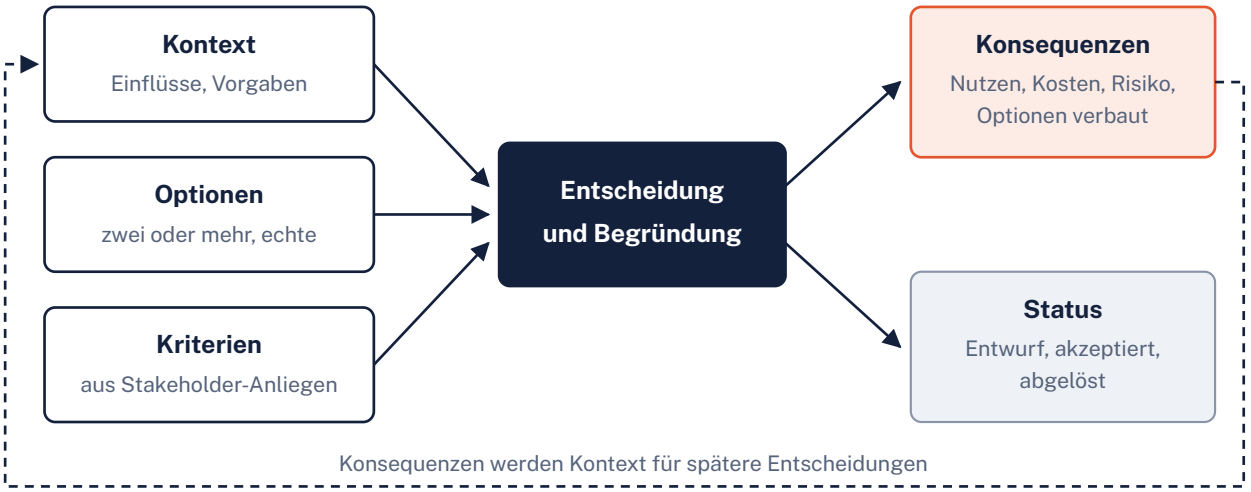
Die Anatomie einer Entscheidung

Eine Architekturentscheidung ist mehr als ihr Ergebnis. Eine Entscheidung, die sich erklären, prüfen und überdenken lässt, hat sechs Teile, und sie zu benennen ist der erste Schritt zu guten Entscheidungen.

Kontext: die wirkenden Kräfte. Das zu lösende Problem, die Anforderungen an Qualitätsmerkmale, die es betreffen, die Randbedingungen (Budget, Fristen, Vorschriften, bestehende Systeme, Fähigkeiten des Teams) und die getroffenen Annahmen. **Optionen**: mindestens zwei echte Alternativen, oft einschließlich „nichts tun“ oder „aufschieben“. Eine Entscheidung, bei der nur eine Option betrachtet wurde, ist eine Ankündigung, keine Entscheidung. **Kriterien**: wie die Optionen beurteilt werden, abgeleitet aus den Anliegen der Stakeholder und nach Wichtigkeit gewichtet. **Entscheidung und Begründung**: die gewählte Option und, ganz wesentlich, warum sie die anderen bei den Kriterien geschlagen hat. **Konsequenzen**: was leichter wird, was schwerer wird, welche neuen Risiken entstehen und welche künftigen Optionen nun verschlossen sind. **Status**: vorgeschlagen, akzeptiert, veraltet oder abgelöst, dazu, wer die Entscheidung wann getroffen hat.

Die Konsequenzen verdienen besondere Aufmerksamkeit, denn an ihnen wird die Ehrlichkeit geprüft. Jede Option hat Nachteile; wer sie aufschreibt, schützt das Team vor Überraschungen und gibt der nächsten Person, die entscheidet, eine faire Darstellung. Die Konsequenzen einer Entscheidung werden zudem Teil des Kontexts für die nächste: Sobald Sie sich für asynchrones Messaging entschieden haben, müssen Sie entscheiden, wie Sie mit doppelten und in falscher Reihenfolge eintreffenden Nachrichten umgehen.

Anatomie einer Architekturentscheidung



Kontext, Optionen und Kriterien fließen in eine Entscheidung ein; ihre Konsequenzen und ihr Status werden festgehalten und werden zum Kontext für spätere Entscheidungen.

Einwegtüren und Zweiwegtüren

Die nützlichste einzelne Eigenschaft einer Entscheidung ist ihre Umkehrbarkeit: wie billig und sicher sie sich rückgängig machen lässt. In seinem Brief an die Amazon-Aktionäre von 2015 unterschied Jeff Bezos Entscheidungen vom **Typ 1**, die folgenreich und unumkehrbar oder nahezu unumkehrbar sind („Einwegtüren“), von Entscheidungen vom **Typ 2**, die änderbar und

umkehrbar sind („Zweiwegtüren“). Typ-1-Entscheidungen verdienen langsame, sorgfältige Abwägung und Beratung. Typ-2-Entscheidungen sollten schnell von Einzelnen oder kleinen Gruppen mit gutem Urteilsvermögen getroffen werden, denn wenn sie sich als falsch erweisen, können Sie durch die Tür zurückgehen. Er warnte, dass große Organisationen dazu neigen, den schwerfälligen Typ-1-Prozess auf alles anzuwenden, was sie langsam und zaghaft macht.

Die Idee lässt sich direkt auf Architektur übertragen. Eine Einwegtür in Software ist eine Entscheidung, deren Rücknahme erfordern würde, große Teile des Systems neu zu schreiben, große Datenmengen zu migrieren, Verträge neu zu verhandeln oder viele Menschen umzuschulen: die Hauptprogrammiersprache einer großen Codebasis, das Datenmodell eines System of Record, die öffentliche API, an die Hunderte von Partnern angebunden sind, eine mehrjährige Cloud-Verpflichtung. Zu den Zweiwegtüren gehören die Wahl einer Logging-Bibliothek hinter Ihrer eigenen Schnittstelle, die interne Struktur eines Moduls, das Tuning eines Caches oder die Frage, welches von zwei gleichwertigen Nachrichtenformaten innerhalb eines Service verwendet wird.

Zwei Vorbehalte. Erstens ist Umkehrbarkeit ein Spektrum, nicht binär: Die meisten Entscheidungen liegen irgendwo zwischen „trivial rückgängig zu machen“ und „unmöglich rückgängig zu machen“, und die Kosten einer Umkehr wachsen meist, je mehr Code, Daten und Menschen von der Entscheidung abhängig werden. Eine Entscheidung, die im ersten Monat eine Zweiwegtür ist, kann im zwölften Monat zur Einwegtür geworden sein. Zweitens können Sie oft durch Design beeinflussen, vor welcher Art von Tür Sie stehen, wie der Abschnitt zur Umkehrbarkeit weiter unten zeigt.

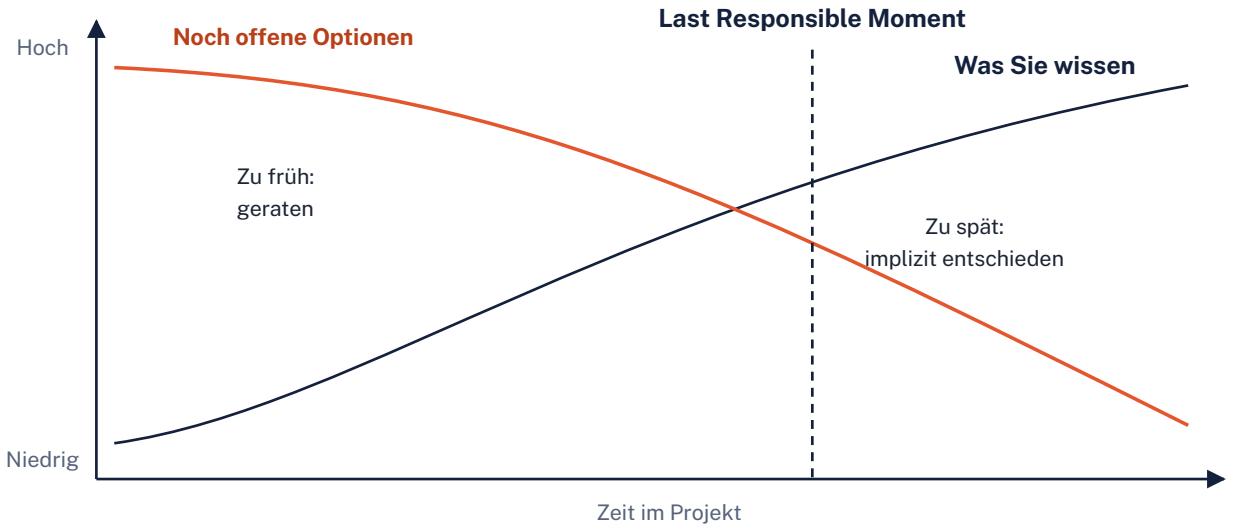
Der Zeitpunkt: der Last Responsible Moment

Wann sollte eine Entscheidung getroffen werden? Mary und Tom Poppendieck machten in *Lean Software Development: An Agile Toolkit* (2003) die Idee populär, im Last Responsible Moment zu entscheiden: in dem Moment, in dem eine ausbleibende Entscheidung eine wichtige Alternative zunichtemachen würde. Vor diesem Moment entscheiden Sie mit weniger Informationen, als Sie haben könnten; danach wird die Entscheidung stillschweigend für Sie getroffen, meist schlecht.

Das Prinzip wirkt in beide Richtungen. Es spricht dagegen, zu früh zu entscheiden: Die Datenbanktechnologie in der ersten Woche festzulegen, bevor Sie die Struktur der Daten und die Zugriffsmuster verstehen, ist Raten. Ebenso spricht es dagegen, zu spät zu entscheiden. Hat das Team bereits begonnen, Persistenzcode auf Grundlage einer unausgesprochenen Annahme zu schreiben, ist die Entscheidung schon gefallen, nur nicht bewusst. Braucht der Einkauf sechs Wochen, um einen Cloud-Anbieter zu genehmigen, liegt der Last Responsible Moment für diese Wahl mindestens sechs Wochen vor dem Zeitpunkt, zu dem Sie die Umgebung brauchen.

Fachleute sind sich uneinig, wie weit man das treiben sollte. Kritiker weisen darauf hin, dass der „Last Responsible Moment“ leicht als Freibrief zum Aufschieben missbraucht wird, dass manche Entscheidungen (Teamstruktur, Programmiersprache) alles Weitere prägen und deshalb früh fallen müssen und dass Unentschlossenheit ihre eigenen Kosten hat: Ohne vereinbarte Schnittstellen können Teams nicht parallel arbeiten. Befürworter entgegnen, das Prinzip habe nie „so spät wie möglich“ bedeutet, sondern „so spät, wie es verantwortbar ist“, und den Moment zu bestimmen sei selbst ein Akt der Planung. Beide Seiten sind sich im praktischen Kern einig: Wissen Sie, welche Entscheidungen anstehen, wissen Sie, wann jede fallen muss, und nutzen Sie die Zeit bis dahin, um zu lernen.

Zu früh, zu spät und zum richtigen Zeitpunkt entscheiden



Das Wissen wächst im Verlauf des Projekts, während sich offene Optionen schließen; der Last Responsible Moment ist der späteste Punkt, bevor eine wichtige Option verschwindet.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: wann die Datenbank des Bestellservice gewählt wird

Der schwierigste Gedanke dieses Abschnitts ist der Last Responsible Moment: Eine Entscheidung sollte weder so früh noch so spät wie möglich getroffen werden, sondern zum spätesten Zeitpunkt, bevor Warten eine wichtige Option zunichtemachen würde. Verfolgen Sie eine Entscheidung durch die Zeit: die Datenbank für den neuen Bestellservice des Essenslieferunternehmens, die später im ausgearbeiteten Beispiel dieses Abschnitts bewertet wird.

- Woche eins. Das Team könnte sofort eine Datenbank wählen, weiß aber noch nicht, wie Bestellungen gespeichert und gelesen werden. Jetzt zu wählen wäre geraten: eine zu frühe Entscheidung.
- Also wartet das Team und nutzt die Zeit, um zu lernen. Es stellt zum Beispiel fest, dass jede Bestellung und ihre Zahlung gemeinsam gespeichert werden müssen, ganz oder gar nicht, als eine Transaktion: genau die Art von Tatsache, auf die sich eine Datenbankwahl stützen sollte.
- Das Warten hat allerdings eine Frist. Angenommen, jeder neue Datenspeicher braucht eine dreiwöchige Sicherheitsprüfung und die Datenbank muss in Woche neun für Tests laufen. Dann ist spätestens Woche sechs der Last Responsible Moment für die Wahl.
- Es gibt auch eine leisere Frist. Wenn Entwickler beginnen, den Code zu schreiben, der Bestellungen speichert, und dieser Code stillschweigend von einer bestimmten Datenbank ausgeht, ist die Wahl implizit gefallen: Niemand hat sie getroffen, aber sie jetzt zu ändern heißt, diesen Code neu zu schreiben.
- Also benennt das Team die Entscheidung, gibt ihr eine verantwortliche Person und einen Termin vor beiden Fristen, entscheidet dann mit dem, was es gelernt hat, und hält die Annahmen dahinter fest.
- **Die Lehre:** Der Last Responsible Moment bedeutet nicht „so spät wie möglich“. Wissen Sie, welche Entscheidungen anstehen und wann jede fallen muss, nutzen Sie die Zeit bis dahin, um zu lernen, und entscheiden Sie, bevor die Wahl für Sie getroffen wird.

Design für Umkehrbarkeit

Der billigste Weg, eine schwere Entscheidung zu treffen, ist oft, sie in eine leichtere zu verwandeln. Mehrere Techniken senken die Kosten, später umzudenken, um den Preis von etwas Aufwand jetzt.

Abstraktion an der Grenze. Setzen Sie eine schmale eigene Schnittstelle zwischen Ihre Kernlogik und eine Technologie, die Sie vielleicht ersetzen, etwa einen Zahlungsanbieter, einen Message Broker oder die API eines großen Sprachmodells. Ein späterer Austausch der

Technologie betrifft dann einen Adapter statt der ganzen Codebasis. Die Idee geht auf David Parnas' Information Hiding (1972) zurück und taucht in der hexagonalen Architektur (Kapitel 5) wieder auf. **Feature-Flags.** Liefern Sie neues Verhalten ausgeschaltet aus, schalten Sie es für einen kleinen Teil der Nutzer ein und in Sekunden wieder aus, wenn es sich falsch verhält; das macht Releases zu Zweiwegtüren. **Schrittweise Migration.** Lassen Sie Alt und Neu nebeneinander laufen und verlagern Sie den Traffic nach und nach, statt in einem Schritt umzuschalten. **Spikes und Proofs of Concept.** Investieren Sie ein paar Tage, um den riskantesten Teil mit einer Kandidatentechnologie zu bauen, bevor Sie sich festlegen. **Vertragliche Ausstiege.** Bevorzugen Sie kürzere Bindungen, Standarddatenformate und die Möglichkeit, Ihre Daten zu exportieren.

Das ist *Optionsdenken*, entlehnt aus der Finanzwelt: Sie zahlen jetzt eine kleine Prämie (einen Adapter, ein Flag, einen Spike), um das Recht, aber nicht die Pflicht zu erwerben, später den Kurs zu ändern. Wie jede Option lohnt sie sich nur, wenn die Prämie im Verhältnis zum Wert der Flexibilität klein ist. Jede Bibliothek „für den Fall, dass wir sie wechseln" in eine Abstraktion zu hüllen, ist ein klassischer Weg, Kosten und Indirektion für eine Flexibilität hinzuzufügen, die niemand nutzt. Kaufen Sie Optionen für Entscheidungen, die sowohl unsicher als auch teuer umzukehren sind; bei allen anderen entscheiden Sie und machen weiter.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: jetzt entscheiden oder aufschieben?

Nutzen Sie dies für jede Entscheidung, die architekturrelevant sein könnte. Gehen Sie die Fragen der Reihe nach durch.

Q1 **Ist andere Arbeit blockiert, oder verschwindet bald eine wichtige Option, wenn niemand entscheidet (zum Beispiel wegen einer Vorlaufzeit im Einkauf oder weil ein Team beginnt, auf einer Annahme aufzubauen)?**

Ja Sie sind im Last Responsible Moment. Entscheiden Sie jetzt mit den vorhandenen Informationen, halten Sie die Entscheidung und ihre Annahmen fest und legen Sie einen Termin für die Überprüfung fest.

Nein Weiter zur nächsten Frage.

Q2 **Lässt sich die Entscheidung billig und sicher umkehren (eine Zweiwegtür)?**

Ja Entscheiden Sie schnell, auf der niedrigsten sinnvollen Ebene (oft durch das Team, dem der Code gehört), halten Sie es kurz fest und machen Sie weiter. Diskutieren Sie nicht eine Woche darüber.

Nein Es ist eine Einwegtür. Gehen Sie zur nächsten Frage.

Q3 **Können Sie sie billig umkehrbarer machen, zum Beispiel mit einer Schnittstelle um die Technologie, einem Feature-Flag oder einem Pfad für eine schrittweise Migration?**

Ja Kaufen Sie zuerst diese Option und behandeln Sie die verbleibende Entscheidung dann als Zweiwegtür.

Nein Weiter zur nächsten Frage.

Q4 **Würden Informationen, die Sie vor der Frist realistisch beschaffen können (ein Spike, ein Lasttest, ein Gespräch mit einem Stakeholder), die Wahl ändern?**

Ja Schieben Sie bewusst auf: Benennen Sie die Entscheidung, die verantwortliche Person, den Termin, bis zu dem sie getroffen werden muss, und das Experiment, das sie stützen wird.

Nein Weiteres Warten hilft nicht. Führen Sie jetzt einen strukturierten Vergleich durch (Kriterien, Gewichte, Optionen), beziehen Sie die Stakeholder ein, denen die Konsequenzen gehören, und schreiben Sie ein Architecture Decision Record.

Kognitive Fallen

Architektinnen und Architekten sind Menschen, und Architekturentscheidungen fallen unter Unsicherheit, Zeitdruck und sozialem Druck: ideale Bedingungen für kognitive Verzerrungen. Wer die häufigsten beim Namen kennt, erkennt sie leichter bei sich selbst und kann sie bei anderen freundlich hinterfragen.

Fallen, die Architekturentscheidungen verzerren

Falle	Woran man sie erkennt	Gegenmaßnahme
Ankereffekt	Die zuerst genannte Option oder die erste Schätzung wird zum Bezugspunkt, an dem jede andere Option gemessen wird.	Erzeugen Sie mehrere Optionen unabhängig voneinander, bevor Sie eine davon diskutieren; bitten Sie die Beteiligten, Schätzungen aufzuschreiben, bevor sie sie teilen.
Versunkene Kosten (Sunk Cost)	„Wir haben acht Monate in dieses Framework gesteckt, wir können es jetzt nicht fallen lassen“, obwohl die bisherigen Ausgaben verloren sind, egal wie Sie entscheiden.	Beurteilen Sie Optionen nur nach künftigen Kosten und Nutzen; fragen Sie, wie Sie sich entscheiden würden, wenn Sie heute anfingen.
Gruppendenken	Ein eng verbundenes Team kommt schnell zu einer Einigung, Widerspruch fühlt sich illoyal an, und niemand spricht den offensichtlichen Einwand aus.	Bestimmen Sie jemanden, der die Gegenposition vertritt; sammeln Sie Meinungen vor dem Meeting schriftlich ein; laden Sie jemanden von außen zur Prüfung ein.
Lebenslaufgetriebene Entwicklung	Eine Technologie wird gewählt, weil sie spannend zu lernen ist oder sich im Lebenslauf gut macht, nicht weil sie zum Problem passt.	Verlangen Sie, dass jede Wahl anhand der vereinbarten Kriterien begründet wird; planen Sie Lernzeit gesondert ein.
Hype und Mitläufereffekt	„Alle steigen auf X um“ ersetzt die Analyse; Konferenzvorträge sehr großer Unternehmen werden auf kleine Teams übertragen.	Fragen Sie, welches Problem X für die Anwender gelöst hat und ob Sie dieses Problem in dieser Größenordnung haben.
Bestätigungsfehler	Belege für die bevorzugte Option werden gesucht und behalten; Belege dagegen werden wegerklärt.	Schreiben Sie vorab auf, welches Ergebnis Ihre Meinung ändern würde, und führen Sie dann den Spike oder Test durch.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine gewichtete Entscheidungsmatrix

Ein Essenslieferunternehmen baut einen neuen Bestellservice. Bestellungen beinhalten Zahlungen und dürfen nie verloren gehen oder doppelt abgerechnet werden. Das sechsköpfige Team kennt PostgreSQL gut und hat keinen eigenen Datenbankadministrator. Drei Optionen liegen auf dem Tisch: (A) ein verwalteter PostgreSQL-Dienst ihres Cloud-Anbieters, (B) die proprietäre verwaltete Dokumentendatenbank des Anbieters und (C) selbst betriebenes PostgreSQL auf virtuellen Maschinen. Das Team erstellt eine gewichtete Entscheidungsmatrix: Es vereinbart Kriterien und Gewichte aus den Anliegen der Stakeholder (5 = am wichtigsten) und bewertet dann jede Option von 1 (schlecht) bis 5 (hervorragend).

Transaktionale Integrität (Gewicht 5)	A: 5, B: 3, C: 5. Transaktionen über mehrere Zeilen sind für Bestellungen und Zahlungen zentral; der Dokumentenspeicher unterstützt sie nur mit mehr Einschränkungen.
Betrieblicher Aufwand (Gewicht 4)	A: 4, B: 5, C: 2. Selbstbetrieb bedeutet Patches, Backups und Failover durch ein Team ohne Datenbankspezialisten.
Vertrautheit des Teams (Gewicht 3)	A: 5, B: 2, C: 4.
Kosten bei erwarteter Last (Gewicht 3)	A: 3, B: 4, C: 4.
Kapazitätsreserve für Skalierung (Gewicht 2)	A: 3, B: 5, C: 3. Das erwartete Wachstum liegt bei etwa dem Dreifachen in zwei Jahren, gut innerhalb einer einzigen großen PostgreSQL-Instanz.
Umkehrbarkeit und Lock-in (Gewicht 2)	A: 4, B: 2, C: 5. Standard-PostgreSQL lässt sich zwischen Anbietern umziehen; die proprietäre API nicht.
Gewichtete Summen (maximal 95)	A: 25 + 16 + 15 + 9 + 6 + 8 = **79**. B: 15 + 20 + 6 + 12 + 10 + 4 = **67**. C: 25 + 8 + 12 + 12 + 6 + 10 = **73**.
Sensitivitätsprüfung	Wird die Kapazitätsreserve auf Gewicht 5 angehoben (als wäre ein fünfzigfaches Wachstum zu erwarten): A 88, B 82, C 82. Wird der betriebliche Aufwand auf Gewicht 1 gesenkt (als würde ein Plattformteam die Datenbanken für sie betreiben): A 67, B 52, C 67, ein Gleichstand zwischen A und C.
Entscheidung	Option A, verwaltetes PostgreSQL. Sie gewinnt unter den vereinbarten Gewichten und bleibt bei plausiblen Änderungen auf Platz eins oder gleichauf an der Spitze. Festgehaltene Konsequenzen: eine höhere Monatsrechnung als beim Selbstbetrieb; Abhängigkeit von den Wartungsfenstern des Anbieters; ein Plan, Schema und Abfragen standardkonform zu halten, damit Option C als Ausweichlösung bleibt.

Die Zahlen treffen nicht die Entscheidung; sie machen die Überlegungen sichtbar. Wer anderer Meinung ist, kann auf ein bestimmtes Gewicht oder eine bestimmte Bewertung zeigen, statt über das Ganze zu streiten.

Die Sensitivitätsprüfung ist der aufschlussreichste Schritt: Sie zeigt, von welchen Annahmen das Ergebnis abhängt. Hier hängt das Ergebnis vor allem davon ab, dass dem Team die Kapazität für den Datenbankbetrieb fehlt.

Matrizen lassen sich manipulieren, indem man Kriterien und Gewichte erst wählt, wenn die bevorzugte Option feststeht. Vereinbaren Sie Kriterien und Gewichte vor der Bewertung, idealerweise mit den Stakeholdern, deren Anliegen sie abbilden.

Entscheidungen festhalten: ein Ausblick auf ADRs

Das zweite Gesetz besagt, dass das Warum wichtiger ist als das Wie, also muss das Warum aufgeschrieben werden. Das am weitesten verbreitete schlanke Format ist das Architecture Decision Record (ADR), das Michael Nygard 2011 in einem Blogbeitrag vorgeschlagen hat. Ein ADR ist eine kurze Textdatei, die mit dem Code in der Versionsverwaltung liegt und eine Entscheidung nach der oben beschriebenen Anatomie festhält: Titel, Status, Kontext, Entscheidung und Konsequenzen. ADRs werden nie bearbeitet, um die Geschichte zu ändern; eine spätere Entscheidung löst eine frühere ab und verweist auf sie, sodass das Protokoll erzählt, wie sich die Architektur entwickelt hat.

Kapitel 4 behandelt Vorlagen, Werkzeuge und die Gewohnheiten, die ADRs nützlich halten, in Architecture Decision Records. Das folgende Beispiel hält die Entscheidung aus dem ausgearbeiteten Beispiel fest.

```
ADR 7: Use managed PostgreSQL for the order service

Status: Accepted (2026-10-12)

Context
Orders and payments need multi-row transactions and must never be lost.
The team knows PostgreSQL well and has no database administrator.
Expected growth is about 3x in two years.

Decision
We will use the cloud provider's managed PostgreSQL service.

Consequences
+ Backups, patching and failover are handled by the provider.
+ Standard SQL keeps self-hosting or another provider open as a fallback.
- Monthly cost is about 30% higher than self-hosting at current load.
- Maintenance windows are set by the provider.
Revisit if write volume exceeds what one primary instance can handle.
```

Ein kurzes ADR für die Datenbankentscheidung des Bestellservice, nach Nygards Struktur

ACHTUNG

Häufige Fehler

- Ausführlich über Zweiwegtüren diskutieren und Einwegtüren nebenbei durchschreiten („wir nehmen vorerst einfach die gemeinsame Kundentabelle“).
- Nur eine Option betrachten. Wenn Sie keine glaubwürdige Alternative nennen können, haben Sie die Entscheidung nicht verstanden.
- Die Nachteile verschweigen. Konsequenzen, die nur Vorteile aufzählen, fliegen auf, und das Vertrauen in die Architektur geht mit ihnen verloren.
- Nach Dienstalter oder Lautstärke entscheiden statt nach vorab vereinbarten Kriterien.
- Nie überprüfen. Entscheidungen beruhen auf Annahmen; halten Sie sie fest und überprüfen Sie die Entscheidung, wenn sich eine Annahme ändert.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Was ist der Unterschied zwischen dem ersten und dem zweiten Gesetz von Richards und Ford?

Das erste besagt, dass jede Architekturentscheidung eine Abwägung ist. Das zweite besagt, dass die Begründung einer Entscheidung (warum) wichtiger ist als ihre Umsetzung (wie), weil sich das Warum nicht aus dem Code rekonstruieren lässt.

▼ **Nennen Sie je ein Beispiel für eine Zweiwegtür und eine Einwegtür in einer typischen Webanwendung.**

Zweiwegtür: die Logging-Bibliothek hinter Ihrer eigenen Schnittstelle oder die Ablaufeinstellung eines Caches. Einwegtür: das Datenmodell eines System of Record oder eine öffentliche API, von der bereits viele Partner abhängen.

▼ **Warum kann eine zu frühe Entscheidung ebenso schädlich sein wie eine zu späte?**

Zu früh heißt, mit weniger Informationen zu entscheiden, als Sie hätten haben können, oft durch Raten. Zu spät heißt, dass die Entscheidung faktisch implizit gefallen ist, durch Annahmen, die bereits in Code oder Pläne eingebaut sind, oder dass eine Option verfallen ist.

▼ **Was bringt eine Sensitivitätsprüfung bei einer gewichteten Entscheidungsmatrix?**

Sie zeigt, ob die siegreiche Option weiterhin gewinnt, wenn sich Gewichte oder Bewertungen in plausiblen Bereichen ändern, und macht so sichtbar, von welchen Annahmen die Entscheidung abhängt.

▼ **Probieren Sie es aus: Gewichte: Zuverlässigkeit 5, Teamfähigkeiten 3, Kosten 2. Datenbank A erzielt 4, 5, 3; Datenbank B erzielt 5, 2, 4. Welche gewinnt? Setzen Sie die Gewichte nun auf 5, 2, 3. Was sagt Ihnen das über die Entscheidung?**

Mit 5, 3, 2: A = 20 + 15 + 6 = 41 und B = 25 + 6 + 8 = 39, A gewinnt also knapp. Mit 5, 2, 3: A = 20 + 10 + 9 = 39 und B = 25 + 4 + 12 = 41, also gewinnt B. Das Ergebnis reagiert empfindlich auf die Gewichte, die Matrix allein entscheidet es also nicht. Vereinbaren Sie die Gewichte vor der Bewertung, diskutieren Sie dann, wie wichtig die Fähigkeiten des Teams wirklich sind, und dokumentieren Sie diese Einschätzung.

QUELLEN

- Documenting Architecture Decisions, Michael Nygard (2011): https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions
- Architectural Decision Records (adr.github.io): https://adr.github.io/
- Amazon.com 2015 letter to shareholders (SEC filing): https://www.sec.gov/Archives/edgar/data/1018724/000119312516530910/d168744dex991.htm
- Feature Toggles (aka Feature Flags), Pete Hodgson on martinfowler.com: https://martinfowler.com/articles/feature-toggles.html

Technische Schulden, Architekturerosion und Architekturdrift

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.3 Code organisieren
- 0.8 Wie Software entsteht
- 1.3 Warum sie zählt

Begriffe von früher, die er verwendet: Modul, Abhängigkeit, Änderungskosten, Versionskontrolle

DIE IDEE IN KÜRZE

Stellen Sie sich vor, Sie ziehen in aller Eile um und werfen alles in Kisten ohne Beschriftung. Sie sind pünktlich umgezogen, und darum ging es ja, aber jetzt verschwenden Sie jedes Mal, wenn Sie etwas brauchen, zehn Minuten mit Suchen. Diese verschwendeten Minuten sind die Zinsen. Ordentlich auspacken und beschriften heißt, die Schulden zurückzuzahlen. Software funktioniert genauso: Abkürzungen oder Entwürfe, die sinnvoll waren, bevor Sie das Problem besser verstanden haben, machen jede spätere Änderung langsamer. Ein wenig davon ist in Ordnung und manchmal sogar klug. Gefährlich wird es, wenn es sich so lange anhäuft, bis das Team die meiste Zeit damit verbringt, Kisten zu durchsuchen.

WAS SIE WISSEN MÜSSEN

- Ward Cunningham prägte die Schuldenmetapher 1992, um zu erklären, dass man Code ausliefert, der das aktuelle Verständnis widerspiegelt, und ihn mit wachsendem Verständnis refaktoriert; sie war nie ein Freibrief für nachlässigen Code.
- Der Tilgungsaufwand sind die Kosten, das Problem zu beheben; die Zinsen sind die Zusatzkosten, die bei jeder Änderung anfallen, solange es besteht; Schulden in Code, der sich nie ändert, kosten daher nichts.
- Fowlers Quadrant ordnet Schulden als bewusst oder unbeabsichtigt und als umsichtig oder leichtsinnig ein; umsichtige Schulden sind ein normales Nebenprodukt von Auslieferung und Lernen, leichtsinnige Schulden ein Problem von Können oder Disziplin.
- Architekturschulden, die in der Struktur statt im Code liegen, sind die teuerste Art, weil ihre Tilgung viele Module, Teams und oft auch Daten berührt.
- Perry und Wolf (1992) unterschieden Erosion, verursacht durch Verstöße gegen die Architektur, von Drift, verursacht durch mangelnde Rücksicht auf sie.

- Hotspots (komplexer und häufig geänderter Code), Change Coupling, Abhängigkeitszyklen und Trends in der Auslieferung machen Schulden sichtbar; ein Schuldenregister, geschützte Kapazität und Refactoring parallel zu Features halten sie unter Kontrolle.

Was Ward Cunningham wirklich meinte

Der Begriff technische Schulden stammt aus Ward Cunninghams Erfahrungsbericht über das Portfolio-Management-System WyCash, den er 1992 auf der Konferenz OOPSLA vorstellte. Er schrieb, Code zum ersten Mal auszuliefern sei, als würde man Schulden aufnehmen: Ein wenig Schuld beschleunigt die Entwicklung, solange sie zügig durch eine Überarbeitung zurückgezahlt wird, und gefährlich wird es, wenn sie nicht zurückgezahlt wird: „Jede Minute, die man mit nicht ganz richtigem Code verbringt, zählt als Zins auf diese Schuld." Ganze Entwicklungsorganisationen, warnte er, können unter der Schuldenlast zum Stillstand kommen.

Die Metapher wird oft als Erlaubnis missverstanden, schludrigen Code zu schreiben. Cunningham erklärte später (in einem kurzen Video von 2009), dass er das nie gemeint habe. Es ging ihm um *Verständnis*. Ein Team liefert Code aus, der sein aktuelles, unvollständiges Verständnis eines Problems widerspiegelt, weil man so am schnellsten aus der echten Nutzung lernt. Wenn das Verständnis wächst, passt der Code nicht mehr zu dem, was das Team inzwischen weiß, und jede Änderung muss diese Diskrepanz umgehen. Die Schulden zurückzuzahlen heißt, den Code durch Refactoring so umzubauen, dass er das Gelernte widerspiegelt. Das funktioniert nur, wenn der Code von vornherein sauber genug geschrieben wurde, um refaktoriert werden zu können. So gelesen sind Schulden ein Nebenprodukt des Lernens, nicht der Nachlässigkeit.

Seitdem hat sich der Sprachgebrauch erweitert. Heute umfasst „technische Schulden" jeden Mangel an innerer Qualität, der künftige Änderungen langsamer oder riskanter macht, ob er aus dem Lernen, bewussten Abkürzungen, Unwissen oder schlichter Vernachlässigung entstanden ist. Die weitere Bedeutung ist nützlich, solange Sie im Kopf behalten, dass die verschiedenen Arten unterschiedliche Antworten verlangen.

Tilgung und Zinsen

Das Finanzvokabular macht die Metapher in Gesprächen mit Nicht-Technikern so wirkungsvoll. Der Tilgungsaufwand ist der Aufwand, der nötig ist, um das Problem zu beheben: das Modul umzustrukturieren, die Datenbank aufzutrennen, den Zyklus aufzulösen. Die Zinsen sind der Mehraufwand, der bei jeder Arbeit anfällt, die das Problem berührt, solange es besteht. Martin Fowlers Beispiel: Wenn ein Feature in sauberem Code vier Tage dauern würde, wegen Altlasten im Code aber sechs dauert, sind die zwei zusätzlichen Tage Zinsen.

Daraus folgen drei Konsequenzen. Erstens: **Schulden, die nie berührt werden, kosten nichts**: Ein unordentliches Modul, das niemand ändert, verursacht keine Zinsen, und es zu tilgen kann vergeudete Mühe sein. Zweitens: **Zinsen haben einen Zinseszinseffekt**: Notlösungen werden zum Fundament weiterer Notlösungen, neue Entwickler kopieren die Muster, die sie vorfinden, und die Kosten der späteren Reparatur steigen. Drittens: **Tilgung ist eine Investitionsentscheidung**: Es lohnt sich, den Tilgungsaufwand zu leisten, wenn die Zinsen, die Sie über einen sinnvollen Zeitraum sparen, ihn übersteigen. Damit wird aus einem emotionalen Argument („der Code ist ein Chaos“) ein wirtschaftliches („dieses Modul kostet uns elf Tage pro Quartal“).

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: wann Unordnung Geld kostet und wann nicht

Die Idee, die hier die meiste Sorgfalt verlangt, lautet: Technische Schulden werden nach den Zinsen beurteilt, die sie verursachen, nicht danach, wie unordentlich der Code aussieht. Der Tilgungsaufwand wird einmal bezahlt, um das Problem zu beheben; die Zinsen werden bei jeder Änderung bezahlt, die das Problem umgehen muss. Zwei gleich unordentliche Module zeigen, warum das wichtig ist.

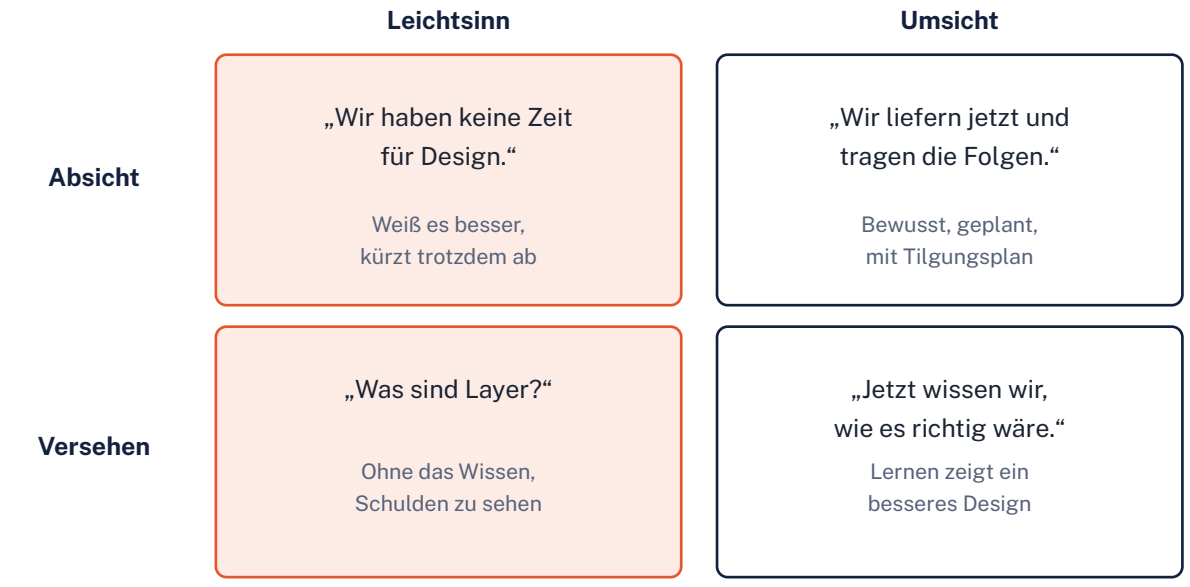
- Nehmen Sie Martin Fowlers Zahlen: Eine Änderung, die in sauberem Code vier Tage dauern würde, dauert in einem unordentlichen Modul sechs, weil die Entwickler das Chaos zuerst verstehen und dann umgehen müssen. Die zwei zusätzlichen Tage sind Zinsen.
- Angenommen, das Aufräumen des Moduls, also der Tilgungsaufwand, würde zehn Entwicklertage dauern und jede Änderung wieder auf etwa vier Tage bringen. Ob sich das lohnt, hängt davon ab, wie oft sich das Modul ändert.
- Wenn das Team es etwa einmal im Monat ändert, zahlt es etwa zwei Tage Zinsen pro Monat; zehn Tage Aufräumen haben sich also nach etwa fünf Monaten bezahlt gemacht. Lässt man das Modul, wie es ist, wachsen die Zinsen eher noch, weil neue Notlösungen auf alten aufbauen.
- Ein zweites, genauso unordentliches Modul wird nur etwa zweimal im Jahr geändert. Es verursacht etwa vier Tage Zinsen pro Jahr, sodass sich dieselben zehn Tage Aufräumen erst nach etwa zweieinhalb Jahren auszahlen würden.
- Ein Modul, das niemand ändert und das keine Ausfälle verursacht, kostet fast keine Zinsen, wie unordentlich es auch aussieht. Es aufzuräumen kann vergeudete Mühe sein.
- **Die Lehre:** Fragen Sie nicht „Wie unordentlich ist dieser Code?“, sondern „Wie viel kostet jede Änderung zusätzlich, und wie oft ändern wir ihn?“ Tilgen Sie zuerst dort, wo die Zinsen am höchsten sind. Das ausgearbeitete Beispiel am Ende dieses Abschnitts stellt dieselbe Rechnung mit Belegen aus Tickets und der Versionskontrolle an.

Fowlers Quadrant der technischen Schulden

Nicht alle Schulden sind gleich. 2009 schlug Martin Fowler vor, Schulden entlang zweier Achsen einzuordnen: ob sie **bewusst** oder **unbeabsichtigt** aufgenommen wurden und ob die Entscheidung **umsichtig** oder **leichtsinnig** war. Die vier Kombinationen verlangen unterschiedliche Antworten.

Umsichtig und bewusst: Das Team kennt das bessere Design, hält es für die Kosten wert, jetzt auszuliefern, und plant die Tilgung. Das ist Cunninghams ursprünglicher Sinn, und es kann geschäftlich klug sein: etwa bei einem Start-up, das erst die Nachfrage nachweist, bevor es in Skalierung investiert. **Leichtsinnig und bewusst**: Das Team weiß es besser, verzichtet aber auf Design, weil es glaubt, keine Zeit zu haben, und bremst sich damit meist innerhalb weniger Wochen selbst aus. **Leichtsinnig und unbeabsichtigt**: Das Team weiß nicht genug, um zu erkennen, dass es Schulden aufbaut; die Abhilfe sind Fähigkeiten, Reviews und Mentoring, nicht Refactoring allein. **Umsichtig und unbeabsichtigt**: Ein fähiges Team leistet gute Arbeit und versteht erst später, wie es hätte strukturiert sein sollen. Fowler argumentiert, dass dies für jedes lernende Team unvermeidlich ist, weshalb kontinuierliches Refactoring zur normalen Arbeit gehört und kein Zeichen des Scheiterns ist.

Der Quadrant der technischen Schulden



Nach Martin Fowler (2009): Die beiden leichtsinnigen Quadranten sind Probleme der Disziplin oder des Könnens; die beiden umsichtigen sind normale Folgen einer Auslieferung unter Unsicherheit.

Architekturschulden und Codeschulden

Schulden gibt es auf jeder Ebene, aber ihr Kostenprofil ändert sich mit der Größenordnung. **Codeschulden** liegen innerhalb eines Moduls: duplizierte Logik, eine Funktion, die fünf Dinge tut, fehlende Tests, schlechte Namen. Sie sind real, aber lokal; ein Entwickler kann sie meist in Stunden oder Tagen tilgen, oft während er an einem benachbarten Feature arbeitet. **Architekturschulden** liegen in der Struktur: Services, die sich eine Datenbank teilen, sodass keiner sein Schema allein ändern kann; eine Domäne, die entlang der falschen Linien aufgeteilt ist, sodass jedes Feature vier Teams berührt; eine synchrone Aufrufkette, durch die ein langsamer Service den Checkout lahmlegt; ein Framework, das mit jeder Schicht verschweißt ist. Ihre Tilgung berührt viele Module, oft viele Teams und manchmal Daten.

Die Forschung stützt die Intuition, dass strukturelle Schulden die teure Art sind. Eine Umfrage unter 1831 Entwicklern und Architekten, die Neil Ernst und Kollegen am Software Engineering Institute 2015 durchführten, ergab, dass Architekturentscheidungen die wichtigste genannte Quelle technischer Schulden waren, noch vor Problemen auf Codeebene. Werkzeuge, die Code Smells melden, finden Codeschulden gut und Architekturschulden schlecht, weil das Problem in den Beziehungen zwischen Teilen liegt und nicht in einer einzelnen Datei.

Architekturschulden sind aus demselben Grund wichtig, aus dem Architektur überhaupt wichtig ist: Wie Warum Architektur zählt gezeigt hat, sind strukturelle Entscheidungen diejenigen mit den höchsten Änderungskosten.

Erosion und Drift

Architekturschulden entstehen oft nicht durch eine schlechte Entscheidung, sondern durch Tausende kleiner Änderungen. Dewayne Perry und Alexander Wolf benannten in ihrem Aufsatz „Foundations for the Study of Software Architecture“ von 1992 zwei Wege, auf denen das geschieht. **Architekturerosion** wird durch *Verstöße* gegen die Architektur verursacht: ein Controller, der die Datenbank direkt abfragt und dabei die Service-Schicht umgeht; ein Modul, das in die internen Tabellen eines anderen greift. Jeder Verstoß macht das System brüchiger, als würde man eine tragende Wand entfernen. **Architekturdrift** wird durch *mangelnde Rücksicht* auf die Architektur verursacht: Änderungen, die gegen keine ausdrückliche Regel verstoßen, aber die beabsichtigten Konzepte ignorieren, sodass das System nach und nach an Stimmigkeit und klarer Form verliert. Drift ist weniger unmittelbar gefährlich als Erosion, verschleiert aber die Architektur, was wiederum Verstöße erleichtert.

Spätere Forscher verwenden „Erosion“ oft für die weitere Lücke zwischen der beabsichtigten Architektur (dem, was die Dokumente und der Kopf des Architekten sagen) und der implementierten Architektur (dem, was der Code tatsächlich tut). Wie auch immer die Begriffe lauten, die Ursachen sind vertraut:

Termindruck, der den kürzesten Weg statt des richtigen belohnt. **Wissensverlust**, wenn die Leute gehen, die das Design verstanden haben, und neue Mitarbeitende kopieren, was sie vorfinden. **Unsichtbare Architektur**: Regeln, die nur in einem Wiki stehen, das niemand liest, ohne dass der Build sie durchsetzt. **Geänderte Anforderungen**, durch die eine einst gute Struktur schlecht zum Problem passt. **Organisatorischer Wandel**: Nach dem Gesetz von Conway erzeugt eine Umorganisation Druck, das System entlang neuer Teamgrenzen umzuformen. **Alternde Technologie**: Frameworks und Plattformen, deren Support ausläuft, werden zu Schulden, ohne dass jemand den Code anfasst.

Lehmans Gesetze: warum das immer wieder passiert

M. M. Lehman begann in den späten 1960er-Jahren mit Arbeiten am OS/360-Prozess von IBM zu untersuchen, wie sich große Programme verändern, und verfeinerte seine Erkenntnisse bis in die 1990er-Jahre zu acht „Gesetzen“. Sie gelten für E-Type-Systeme: Software, die in die reale Welt eingebettet ist und die Welt verändert, der sie dient. Drei sind hier am wichtigsten. **Fortwährender Wandel**: Ein solches System muss laufend angepasst werden, sonst wird es zunehmend unbefriedigender. **Zunehmende Komplexität**: Während es sich weiterentwickelt, steigt seine Komplexität, sofern nicht daran gearbeitet wird, sie zu erhalten oder zu verringern. **Abnehmende Qualität**: Seine wahrgenommene Qualität sinkt, wenn es nicht konsequent gepflegt wird. Das sind empirische Beobachtungen, keine Physik; eine Studie von 1997 stützte einige und ließ andere offen. Die Essenslieferplattform wird verfallen, wenn man sie sich selbst überlässt, selbst ohne schlechte Entscheidungen.

Der Big Ball of Mud

1997 schrieben Brian Foote und Joseph Yoder „Big Ball of Mud“ und benannten damit, was sie die am häufigsten eingesetzte Softwarearchitektur nannten: ein „lässig, ja planlos strukturiertes System“, das mehr von Zweckmäßigkeit als von Entwurf geprägt ist. Ihr Paper ist ungewöhnlich, weil es fragt, warum sich der Matsch hält, statt ihn nur zu verurteilen. Termine belohnen Software, die jetzt funktioniert, Architektur ist weitgehend unsichtbar, das frühe Verständnis der Domäne ist dünn, und Wegwerfcode wird selten weggeworfen. Unordentliche Systeme funktionieren außerdem, und wer sich in ihnen zurechtfindet, wird wertvoll. Deshalb behandeln die Autoren ihn als Muster mit Kräften dahinter: Stückweises Wachstum braucht stetige Reparatur, ein Chaos lässt sich hinter einer sauberen Schnittstelle unter den Teppich kehren, und Abschnitt 11.6 behandelt, was folgt, wenn das nicht geht.

Schulden erkennen

Schulden, die Sie nicht sehen, können Sie nicht steuern, und das Bauchgefühl ist unzuverlässig: Entwickler beklagen sich meist am lautesten über den Code, den sie am häufigsten anfassen, was nützlich ist, aber sie wissen selten, wie viel jedes Problem kostet. Mehrere Techniken machen Schulden sichtbar.

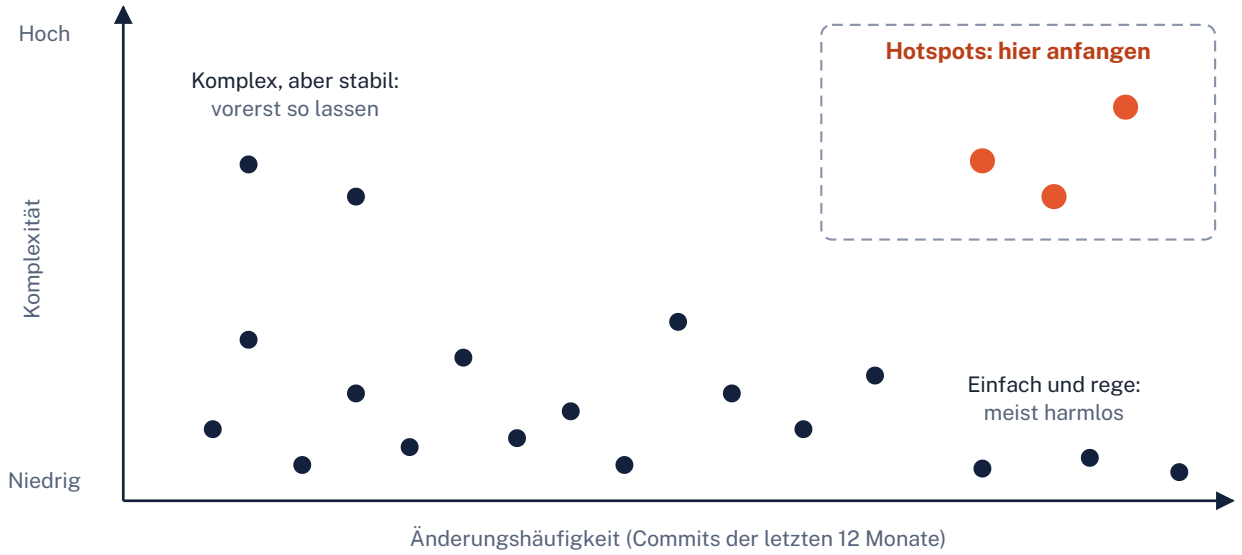
Hotspots. Adam Tornhills *Your Code as a Crime Scene* (2015; zweite Auflage 2024) wendet Ideen der Forensik auf die Historie der Versionskontrolle an. Ein Hotspot ist Code, der sowohl komplex ist als auch häufig geändert wird. Komplexität allein reicht nicht: Eine komplexe Datei, die niemand anfasst, kostet keine Zinsen. Änderungshäufigkeit allein reicht auch nicht: Eine einfache Konfigurationsdatei kann sich täglich ändern, ohne Schaden anzurichten. Wo beides zusammenkommt, werden bei jedem Commit Zinsen gezahlt, und dort zahlt sich Tilgung zuerst aus. In den meisten Codebasen entfällt der Großteil der Änderungsaktivität auf einen kleinen Teil der Dateien.

Change Coupling. Dateien oder Services, die immer wieder in denselben Commits geändert werden, sind gekoppelt, ob der Code eine Abhängigkeit zeigt oder nicht. Wenn jede Änderung am Preisservice mit einer Änderung am Rechnungsservice einhergeht, sind sie eine Sache, die vorgibt, zwei zu sein. Change Coupling über Team- oder Servicegrenzen hinweg ist ein starkes Anzeichen für Architekturschulden.

Abhängigkeitszyklen und Regelverstöße. Statische Analyse kann den tatsächlichen Abhängigkeitsgraphen zeichnen und mit dem beabsichtigten vergleichen. Zyklen (A nutzt B, B nutzt C, C nutzt A) bedeuten, dass keines der drei für sich verstanden, getestet oder ausgeliefert werden kann. Architekturregeln („die Domänenschicht darf nicht von der Webschicht abhängen“) lassen sich bei jedem Build automatisch prüfen; das sind Fitness Functions, die Kapitel 11 behandelt.

Trends in der Auslieferung. Steigende Durchlaufzeiten für Änderungen, mehr durchrutschende Fehler pro Release, längere Einarbeitung, mehr Abstimmungsmeetings pro Feature und wachsende Scheu, bestimmte Bereiche anzufassen, sind nachlaufende Indikatoren für aufgelaufene Zinsen. Die DORA-Metriken für die Auslieferung (etwa Durchlaufzeit von Änderungen und Änderungsfehlerrate) bieten eine einheitliche Möglichkeit, einige davon über die Zeit zu verfolgen.

Hotspots finden



Trägt man jede Datei danach auf, wie oft sie sich ändert und wie komplex sie ist, zeigt sich, wo Schulden am häufigsten bezahlt werden: bei den wenigen Dateien in der Ecke oben rechts.

Schulden steuern

Keine gesunde Codebasis hat null Schulden, und der Versuch, null zu erreichen, ist selbst Verschwendung. Das Ziel ist, Schulden sichtbar, bewusst und tragbar zu halten. Teams, denen das gut gelingt, kombinieren mehrere Praktiken.

Ein Schuldenregister. Halten Sie bedeutende Schuldenposten dort fest, wo das Team seine Arbeit plant: worin das Problem besteht, wo es liegt, wie es entstanden ist, den geschätzten Tilgungsaufwand, die geschätzten Zinsen (mit Belegen wie Hotspot-Daten oder aktuellen Tickets) und den Auslöser für die Tilgung. Ein Register macht aus vagen Klagen eine Liste, die sich zusammen mit Features priorisieren lässt.

Ein fester Anteil der Kapazität. Viele Teams reservieren einen gleichbleibenden Anteil jeder Iteration, meist irgendwo zwischen 10 und 20 Prozent, für die Tilgung von Schulden und andere Verbesserungsarbeit. Für eine magische Zahl gibt es keine Belege; entscheidend ist, dass der Anteil geschützt ist und nicht jedes Mal verschwindet, wenn ein Termin naht.

Refactoring parallel zu Features. Am günstigsten tilgen Sie Schulden, wenn Sie ohnehin in dem Bereich arbeiten, weil Sie den Kontext im Kopf haben und die Tests laufen. Fowler nennt das opportunistisches Refactoring. Seine alltägliche Form ist die Pfadfinderregel, die Robert C. Martin bekannt gemacht hat: Hinterlassen Sie den Code ein wenig sauberer, als Sie ihn vorgefunden haben. Weil Hotspots per Definition dort liegen, wo Features landen, konzentriert sich der Aufwand so ganz von selbst dort, wo die Zinsen am höchsten sind.

Geplante strukturelle Tilgung. Architekturschulden lassen sich selten allein mit kleinen Schritten abbauen. Sie brauchen ein ausdrückliches, finanziertes Arbeitspaket mit einem Ziel, einem Plan für eine schrittweise Migration und der Unterstützung der Stakeholder. Techniken wie das Strangler-Fig-Muster (Kapitel 11, Altsysteme modernisieren) erlauben es einem Team, eine Struktur Stück für Stück zu ersetzen, ohne alles auf einen Schlag neu zu schreiben.

Vorbeugung. Automatisierte Architekturprüfungen im Build, Code-Reviews, die neben dem Stil auch auf die Struktur achten, und aktuelle Entscheidungsdokumente bremsen Erosion und Drift an der Quelle.

KI-gestützte Entwicklung: schnellerer Code, schnellere Erosion?

Programmierassistenten und Agenten (Abschnitt 12.4) können heute schneller Code erzeugen, als ein Team ihn lesen kann. Googles DORA-Bericht von 2025, der auf fast 5000 Technologiefachleuten beruht, ergab, dass 90 % bei der Arbeit KI nutzen. Der Einsatz ging mit höherem Lieferdurchsatz, aber geringerer Lieferstabilität einher, und die wichtigste Schlussfolgerung von DORA lautete, dass KI ein Verstärker ist: Sie vergrößert die vorhandenen Stärken und Schwächen eines Teams, und lose gekoppelte Systeme mit schnellem Feedback profitieren am meisten. Das sind Korrelationen aus einer Umfrage. Kontrollierte Belege sind dünner und ändern sich schnell: METRs Studie mit Tools von Anfang 2025 ergab, dass erfahrene Entwickler 19 % langsamer waren und sich dabei für schneller hielten, und die Folgestudie von 2026 legte nahe, dass spätere Tools wahrscheinlich helfen, bezeichnete ihre eigenen Daten aber als sehr schwache Belege dafür, wie sehr. Messen Sie in Ihrem eigenen Umfeld und prüfen Sie die aktuelle Forschung.

Das architektonische Risiko ist Erosion im Eiltempo. Ein Tool arbeitet an der vorliegenden Aufgabe, sieht die beabsichtigte Architektur nur begrenzt und neigt dazu, das nachzuahmen, was in der Nähe steht. Bitten Sie einen Agenten, der Essenslieferplattform einen Rückerstattungsgrund hinzuzufügen, und er fragt womöglich direkt die Zahlungen-Tabellen ab, weil das kompiliert und die Tests bestehen: ein Verstoß im Sinne von Perry und Wolf, in Sekunden erzeugt. GitClear, ein Anbieter von Code-Analysen, berichtete, dass der Anteil geklonter Zeilen an den geänderten Zeilen zwischen 2021 und 2024 von 8,3 auf 12,3 % stieg und der Anteil des Refactorings von 25 auf unter 10 % sank. Das zeigt einen Trend, keinen Beweis für die Ursache. Ein leiserer Preis ist das Verständnis: Code kann in die Produktion gelangen, den niemand im Team erklären könnte.

Die Arbeit des Architekten verlagert sich daher vom Erzeugen von Struktur hin zu ihrer Festlegung und Verteidigung. Was früher in den Köpfen lebte, muss explizit werden: klare Modulgrenzen, Entscheidungsdokumente und Repository-Anweisungen, die Tools und

Menschen lesen können. Was früher auf Wachsamkeit beruhte, muss automatisch werden: Abhängigkeitsregeln und Fitness Functions im Build (Abschnitt 11.3). Gut gesetzte Grenzen begrenzen außerdem den Schaden, den eine einzelne Änderung anrichten kann.

Daraus folgt die Governance für generierten Code. **Verantwortlichkeit**: Eine namentlich benannte Person verantwortet jede Änderung und prüft sie nach dem üblichen Maßstab, und große generierte Änderungen werden aufgeteilt, bis sie sich prüfen lassen. **Automatische Prüfschranken**: Architekturprüfungen, statische Analyse, Secret-Scanning und Tests gelten unabhängig davon, wer oder was den Code geschrieben hat. **Abhängigkeiten**: Eine Studie auf der USENIX Security 2025 mit 576.000 Stichproben ergab, dass Modelle Paketnamen erfinden, im Durchschnitt mindestens 5,2 % bei kommerziellen und 21,7 % bei Open-Source-Modellen, und ein Angreifer kann einen solchen Namen registrieren; beschränken Sie Installationen wie in Abschnitt 8.6. **Sicherheit**: In einer von Stanford geleiteten Studie mit einem frühen Modell schrieben Teilnehmende mit Assistent weniger sicheren Code und hielten ihn eher für sicher. **Berechtigungen**: Agenten erhalten Least Privilege und keine Produktionszugangsdaten. **Richtlinie**: Halten Sie fest, welche Tools zugelassen sind und welcher Code oder welche Daten an sie gesendet werden dürfen, und lassen Sie sich zu Lizenzierung und Vertraulichkeit beraten.

Schulden für nicht technische Stakeholder erklären

Auftraggeber und Product Owner müssen nichts über zyklische Abhängigkeiten hören; sie müssen die Auswirkungen auf das verstehen, was ihnen wichtig ist: Tempo, Kosten, Risiko und Kundenerlebnis. Die Finanzmetapher ist hier ein Geschenk, vorausgesetzt, Sie verwenden sie mit Zahlen.

Sprechen Sie in **Zeit und Geld**: „Jede Änderung am Checkout kostet uns etwa drei zusätzliche Entwicklertage, weil Zahlung und Bestellungen so ineinander verheddert sind.“ Sprechen Sie in **Risiko**: „Zwei unserer letzten vier Incidents begannen in diesem Modul.“ Sprechen Sie in **Optionen**: „Bis wir das trennen, können wir die neuen Zahlungsarten nicht anbieten, die der Vertrieb angefragt hat.“ Und stellen Sie die Tilgung als **Investition mit Amortisationszeit** dar, nicht als Bitte um Entwicklungszeit. Vermeiden Sie zwei Fallen: Schulden als moralisches Versagen der Vorgänger darzustellen (sie sind meist umsichtig oder unbeabsichtigt entstanden) und einen unbefristeten „Refactoring-Sprint“ ohne messbares Ergebnis zu verlangen.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: die Zinsen einer Schuld schätzen

In einem Onlineshop teilen sich Bestell- und Zahlungslogik ein großes Modul und dieselben Datenbanktabellen. Das Team vermutet, dass sie das viel kostet. Es sammelt Belege aus den Tickets und der Versionskontrollhistorie der letzten sechs Monate und schätzt dann.

Änderungen, die das Modul berühren	48 Änderungen in sechs Monaten, etwa 8 pro Monat (es ist der größte Hotspot).
Mehraufwand pro Änderung	Im Vergleich mit ähnlichen Änderungen an anderer Stelle schätzen die Entwickler jeweils etwa 1,5 zusätzliche Tage für das Verstehen, das Umgehen der gemeinsamen Tabellen und zusätzliche Regressionstests.
Aufwandszinsen	8 Änderungen × 1,5 Tage = 12 Entwicklertage pro Monat.
Fehlerzinsen	3 Produktions-Incidents in sechs Monaten, die auf dieses Modul zurückgingen und deren Untersuchung und Behebung jeweils etwa 4 Entwicklertage kostete: 12 Tage ÷ 6 = 2 Tage pro Monat.
Zinsen insgesamt	Etwa 14 Entwicklertage pro Monat. Bei Vollkosten von £600 pro Entwicklertag sind das rund £8400 pro Monat oder etwa £100.000 pro Jahr.
Tilgungsaufwand	Bestellungen und Zahlungen hinter einer Schnittstelle trennen und die Tabellen aufteilen, schrittweise umgesetzt: geschätzt 45 Entwicklertage, etwa £27.000 insgesamt.
Verbleibende Zinsen nach der Tilgung	Die Entwickler rechnen damit, dass etwas Mehraufwand bleibt: geschätzt 3 Entwicklertage pro Monat.
Amortisation	Monatliche Ersparnis etwa 11 Entwicklertage. 45 ÷ 11 ergibt etwa 4 Monate, bis der Tilgungsaufwand wieder hereingeholt ist.
Empfehlung	Tilgen, in Schritten von jeweils etwa zwei Wochen, beginnend mit den Tabellen, die am häufigsten an Incidents beteiligt sind. Die Durchlaufzeit von Änderungen am Checkout verfolgen, um die Ersparnis zu bestätigen.

Jede Zahl ist eine Schätzung mit großer Unsicherheit. Selbst bei doppeltem Tilgungsaufwand von 90 Tagen amortisiert es sich in etwa acht Monaten, die Schlussfolgerung ist also robust; diese Prüfung lohnt es sich, den Stakeholdern zu zeigen.

Dieselbe Methode kann zeigen, dass sich manche Schulden nicht zu tilgen lohnen: Ein unordentliches Modul, das zweimal im Jahr geändert wird, kann eine Amortisationszeit von mehreren Jahren haben.

Die Argumentation erfolgt in den Begriffen des Auftraggebers (Tage, Geld, Incidents, Amortisation), nicht in Begriffen der Codequalität.

ACHTUNG

Verbreitete Irrtümer

- „Technische Schulden bedeuten schlechten Code.“ Viele Schulden stammen aus gutem Code, der mit einem inzwischen gewachsenen Verständnis geschrieben wurde, oder aus Anforderungen, die sich verschoben haben.
- „Alle Schulden müssen getilgt werden.“ Schulden in Code, der sich nie ändert, kosten nichts; tilgen Sie dort, wo die Zinsen anfallen.
- „Wir beheben das bei einer großen Neuentwicklung.“ Neuentwicklungen auf einen Schlag überziehen regelmäßig und verlieren verborgenes Verhalten; schrittweise Tilgung ist meist sicherer.
- „Nur Entwickler kümmern sich um Schulden.“ Ihre Auswirkungen (langsamere Features, mehr Incidents, verbaute Chancen) sind geschäftliche Auswirkungen, und Auftraggeber sollten mitentscheiden, was getilgt wird.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum kann ein sehr unordentliches Modul eine geringe Priorität für die Tilgung haben?

Weil Zinsen nur anfallen, wenn der Code geändert wird oder Ausfälle verursacht. Wenn niemand ihn anfasst und er stabil ist, kosten die Schulden fast nichts.

▼ Warum lassen sich leichtsinnige, unbeabsichtigte Schulden nicht allein durch Refactoring beheben?

Weil das Team nicht wusste, dass es Schulden aufbaute; solange sich seine Fähigkeiten und Review-Praktiken nicht verbessern, baut es dieselbe Art von Schulden so schnell auf, wie es sie tilgt.

▼ Was ist in den Begriffen von Perry und Wolf der Unterschied zwischen Erosion und Drift?

Erosion entsteht durch Verstöße gegen die Regeln der Architektur; Drift entsteht durch Änderungen ohne Rücksicht auf die Architektur, die gegen keine ausdrückliche Regel verstoßen, aber ihre Konzepte verwischen.

▼ Warum verbinden Hotspot-Analysen Änderungshäufigkeit mit Komplexität, statt eines von beiden allein zu nutzen?

Komplexer Code, der sich nie ändert, verursacht kaum Zinsen, und einfacher Code, der sich oft ändert, ist billig zu ändern; erst die Kombination zeigt, wo Schulden immer wieder Aufwand kosten.

▼ Probieren Sie es aus: Ein unordentliches Modul wird 6-mal im Monat geändert, und jede Änderung kostet 1 zusätzlichen Tag zu £600 pro Tag; die Bereinigung dauert 60 Tage. Ein anderes, ebenso unordentliches Modul wurde seit zwei Jahren nicht angefasst. Welche Schulden würden Sie zurückzahlen?

Tilgen Sie die erste. Ihre Zinsen betragen 6 x £600 = £3600 im Monat und ihr Tilgungsaufwand 60 x £600 = £36.000, die Behebung amortisiert sich also in etwa 10 Monaten. Die zweite hat Tilgungsaufwand, aber keine Zinsen: Code, der sich nie ändert, kostet nichts extra, lassen Sie ihn also, und vermerken Sie ihn im Schuldenregister. Wenn er sich häufig zu ändern beginnt, schauen Sie erneut hin. Zeigen Sie dem Geschäft die Zahlen, nicht das Durcheinander.

QUELLEN

- The WyCash Portfolio Management System, Ward Cunningham (OOPSLA 1992): https://c2.com/doc/oopsla92.html
- Technical Debt, Martin Fowler: https://martinfowler.com/bliki/TechnicalDebt.html
- Technical Debt Quadrant, Martin Fowler (2009): https://martinfowler.com/bliki/TechnicalDebtQuadrant.html
- Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt (SEI, 2015): https://sei.cmu.edu/library/measure-it-manage-it-ignore-it-software-practitioners-and-technical-debt
- Ward Explains Debt Metaphor (c2 wiki): https://wiki.c2.com/?WardExplainsDebtMetaphor
- Lehman: Laws of Software Evolution Revisited: https://www.cs.kent.edu/~jmaletic/cs63902/Papers/Lehman96.pdf
- Lehman et al.: Metrics and Laws of Software Evolution, The Nineties View: https://www.ifi.uzh.ch/dam/jcr:00000000-5852-3622-0000-00002e60ff8e/lehman-metrics-97.pdf
- Foote and Yoder: Big Ball of Mud: http://laputan.org/mud/
- Google Cloud: Announcing the 2025 DORA report: https://cloud.google.com/blog/products/ai-machine-learning/announcing-the-2025-dora-report
- METR: Measuring the impact of early-2025 AI on experienced open-source developer productivity: https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/

- METR: developer productivity experiment update (February 2026): https://metr.org/blog/2026-02-24-uplift-update/
- GitClear: AI Copilot Code Quality 2025: https://www.gitclear.com/ai_assistant_code_quality_2025_research
- Spracklen et al.: We Have a Package for You!: https://www.usenix.org/conference/usenixsecurity25/presentation/spracklen
- Perry et al.: Do Users Write More Insecure Code with AI Assistants?: https://arxiv.org/abs/2211.03622

Architektur über den Lebenszyklus: vorab geplante, emergente und kontinuierliche Architektur

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.8 Wie Software entsteht
- 1.4 Die Rolle des Architekten
- 1.6 Entscheidungen und Abwägungen

Begriffe von früher, die er verwendet: Agile Softwareentwicklung, Continuous Delivery, Last Responsible Moment, Fitness Function, Qualitätsmerkmal

DIE IDEE IN KÜRZE

Wenn Sie eine lange Reise planen, buchen Sie die Flüge und das Hotel für die erste Nacht lange im Voraus, weil diese teuer umzubuchen sind und ausverkauft sein könnten. Sie planen nicht jede Mahlzeit jedes Tages, bevor Sie losfahren; das entscheiden Sie unterwegs, sobald Sie wissen, wie Ihnen zumute ist und was in der Nähe liegt. Gute Softwarearchitektur funktioniert genauso. Die wenigen großen, schwer änderbaren Entscheidungen werden früh und sorgfältig getroffen. Alles andere wird unterwegs entschieden, während das Team dazulernt, und der Plan wird während der ganzen Reise geprüft und angepasst, statt einmal geschrieben und abgeheftet zu werden.

WAS SIE WISSEN MÜSSEN

- Big Design Up Front passt zu stabilen Anforderungen und sehr teuren Änderungen; rein emergentes Design passt zu kleinen Systemen aus Zweiwegtüren; die meisten Teams brauchen etwas dazwischen.
- Der Grundsatz des Agilen Manifests, dass die besten Architekturen aus selbstorganisierten Teams hervorgehen, betrifft die Frage, wer wie entwirft, und behauptet nicht, Architektur sei unnötig.
- Fairbanks' risikogetriebenes Modell bemisst den Architekturaufwand am Risiko des Scheiterns: Risiken erkennen und priorisieren, Techniken anwenden, die Verringerung bewerten und aufhören, wenn das verbleibende Risiko akzeptabel ist.
- Die Architectural Runway von SAFe und Erders und Pureurs kontinuierliche Architektur behandeln Architektur beide als fortlaufende Arbeit, die den Features, die von ihr abhängen, voraus sein muss.

- Ein Walking Skeleton beweist die Architektur früh von Ende zu Ende; Spikes machen aus unsicheren Entscheidungen belegbasierte.
- Bei Continuous Delivery und DevOps werden Deploybarkeit und Betreibbarkeit zu zentralen Qualitätsmerkmalen, eine lose gekoppelte Architektur ermöglicht Auslieferungsleistung, und das laufende System wird zur wichtigsten Quelle architektonischen Feedbacks.

Der alte Streit: vorab oder emergent?

Den größten Teil der Softwaregeschichte über war Architektur etwas, das man am Anfang erledigte. In einem sequenziellen Prozess („Wasserfall“) wurden Anforderungen gesammelt, dann wurden eine Architektur und ein Detailentwurf erstellt und freigegeben, und dann wurde das System nach diesem Entwurf gebaut. Das ist Big Design Up Front (BDUF). Seine Logik ist in manchen Umgebungen stichhaltig: Wenn Anforderungen stabil und gut verstanden sind, wenn Änderungen nach der Auslieferung extrem teuer sind (Flugsoftware, Medizinprodukte, hardwaregebundene Systeme) oder wenn viele Auftragnehmer Teile bauen müssen, die zusammenpassen, verringert ein gründlicher Entwurf vor dem Bau das Risiko.

Seine Schwächen sind ebenso bekannt. Anforderungen ändern sich, und ein aufwendiger, früh erstellter Entwurf schreibt Annahmen fest, die sich später als falsch erweisen. Feedback aus der echten Nutzung kommt erst am Ende, wenn Änderungen am teuersten sind. Teams entwerfen für eingebildete Bedürfnisse („wir müssen vielleicht fünf Währungen unterstützen“) und bezahlen für Flexibilität, die niemand nutzt. Und Dokumente, deren Freigabe Monate gedauert hat, werden meist noch befolgt, wenn sie längst nicht mehr stimmen.

Agile Methoden waren eine Reaktion darauf. Vor allem Extreme Programming trat für emergentes Design ein: das Einfachste bauen, das für die anstehenden Features funktioniert, den Code mit kontinuierlichem Refactoring und automatisierten Tests sauber halten und das Design mit dem System wachsen lassen. Kent Becks Regeln für einfaches Design und der Grundsatz „you aren’t gonna need it“ (YAGNI) drücken denselben Gedanken aus. Emergentes Design funktioniert gut innerhalb eines Moduls und bei kleinen Systemen, in denen jede Entscheidung eine Zweiwegtür ist. Seine Schwäche zeigt sich im großen Maßstab: Manche Entscheidungen (das Datenmodell eines System of Record, die Grenzen zwischen Teams, das Sicherheitsmodell, die Wahl zwischen synchroner und asynchroner Integration) sind teuer rückgängig zu machen und prägen alles, was darauf aufbaut. Überlässt man sie der Emergenz, werden sie oft zufällig von dem getroffen, der den ersten Code schreibt.

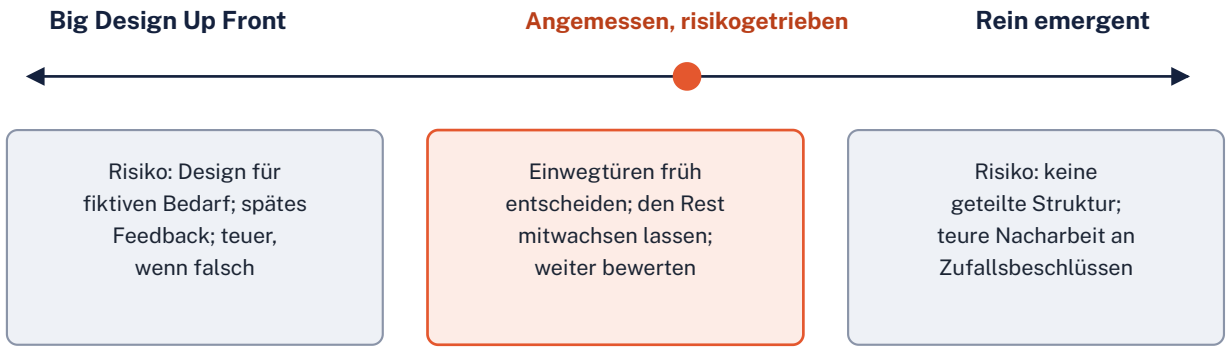
Was das Agile Manifest sagt und was nicht

Einer der zwölf Grundsätze hinter dem Agilen Manifest (2001) lautet: „Die besten Architekturen, Anforderungen und Entwürfe entstehen durch selbstorganisierte Teams.“ Er wird häufig zitiert, als bedeute er, dass agile Teams keine Architektur oder keine Architekten brauchen. Das sagt er nicht.

Genau gelesen trifft der Grundsatz eine Aussage darüber, *wer* die besten Architekturen hervorbringt (die Menschen, die das System bauen, selbst organisiert statt von außen gesteuert), und *wie* (sie entstehen durch die Arbeit, statt fertig vorgegeben zu werden). Er sagt nichts darüber, dass Architektur unwichtig sei, und dasselbe Dokument enthält einen weiteren Grundsatz: Ständiges Augenmerk auf technische Exzellenz und gutes Design fördert Agilität. Ein Team, das strukturelle Entscheidungen ignoriert, folgt nicht dem Manifest; es ignoriert die Hälfte davon.

Was der Grundsatz tatsächlich ausschließt, ist eine Architektur, die aus dem Elfenbeinturm von Leuten verordnet wird, die nicht mit ihren Folgen leben müssen, und eine Phase, in der die Architektur fertiggestellt und eingefroren wird, bevor jemand Code schreibt. In der Praxis sieht der häufigste Ausgleich so aus, dass Architektur zu einer kontinuierlichen, im Team geteilten Tätigkeit wird, bei der ein Architekt (oder architektonisch denkende Entwickler) als Wegweiser, Moderator und Hüter des Gesamtbilds wirkt statt als Torwächter.

Ein Spektrum, keine Entweder-oder-Wahl



Die meisten erfolgreichen Teams liegen zwischen den Extremen: Sie entscheiden die schwer umkehrbaren Fragen früh und lassen den Rest entstehen.

Wie viel Architektur? Das risikogetriebene Modell

Wenn keines der Extreme richtig ist, wie viel Architekturarbeit ist dann genug? George Fairbanks gab eine der klarsten Antworten in *Just Enough Software Architecture: A Risk-Driven Approach* (2010). Sein risikogetriebenes Modell besagt, dass der Aufwand für Architektur dem Risiko des Scheiterns entsprechen sollte. Es hat drei Schritte, die nach Bedarf wiederholt

werden: **Risiken erkennen und priorisieren** (idealerweise als konkrete Ausfallszenarien, etwa „die Buchungsseite läuft in einen Timeout, wenn sich um 8 Uhr 5000 Patienten anmelden“); **Techniken auswählen und anwenden**, die diese Risiken verringern (ein Modell, ein Prototyp, ein Lasttest, eine Analyse, ein Entwurfsmuster); und **die Risikoverringerung bewerten** und aufhören, wenn das verbleibende Risiko akzeptabel ist.

Das Modell ist wirkungsvoll, weil es eine Abbruchregel liefert. Ein einfaches internes Werkzeug auf vertrauter Technologie mit wenigen Nutzern birgt wenige Architekturrisiken, und eine Skizze am Whiteboard kann genügen. Eine Zahlungsplattform, die Millionen von Transaktionen abwickelt und reguliert ist, birgt viele und verdient sorgfältige Modellierung, Prototypen und Reviews. Dasselbe Team kann auf verschiedene Teile eines Systems sehr unterschiedlich viel Architektur verwenden. Fairbanks betont auch, dass Zeit für den Entwurf Zeit ist, die nicht für Bauen oder Testen zur Verfügung steht; zu viel Entwurf ist also echte Kosten und kein sicherer Standard.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: gerade genug Architektur für das Buchungssystem

Der Punkt, an dem hier die meisten stolpern, ist Fairbanks' risikogetriebenes Modell: Es gibt kein festes richtiges Maß an Architektur, nur ein Maß, das zu den Risiken passt. Sie entwerfen weiter, solange das Risiko des Scheiterns zu hoch ist, und hören auf, wenn es akzeptabel ist. Verfolgen Sie das am Buchungssystem des Krankenhauses.

- Listen Sie die Risiken als konkrete Ausfälle auf, etwa „die Buchungsseite läuft in einen Timeout, wenn sich um 8 Uhr 5000 Patienten anmelden" und „das Patientenverwaltungssystem, das die freien Termine der Ambulanzen liefert, kann keine 50 Anfragen pro Sekunde beantworten". Teile auf vertrauter Technologie für wenige Nutzer bergen kaum ernste Risiken.
- Priorisieren Sie. Der Ansturm um 8 Uhr würde jeden Patienten treffen, der dann bucht, deshalb kommt er ganz nach oben. Die Teile mit geringem Risiko bekommen eine Skizze am Whiteboard und nicht mehr.
- Wenden Sie für jedes Risiko eine passende Technik an. Für den morgendlichen Ansturm baut das Team einen kleinen Prototyp und testet ihn mit 5000 simulierten Patienten, die sich gleichzeitig anmelden. Für das Patientenverwaltungssystem misst ein kurzes, mit dem Lieferanten abgestimmtes Experiment, wie viele Anfragen pro Sekunde es beantworten kann.
- Bewerten Sie, wie weit jedes Risiko geschrumpft ist. Angenommen, der Prototyp bewältigt den Ansturm, aber das Patientenverwaltungssystem wird schon vor 50 Anfragen pro Sekunde stark langsamer. Das erste Risiko ist jetzt akzeptabel, das zweite nicht, also geht das Team eine weitere Runde, vielleicht mit einem Test des kurzlebigen Termin-Caches aus Abschnitt 1.5.
- Hören Sie auf, wenn jedes der wichtigsten Risiken akzeptabel gering ist. Weiteres Entwerfen ist kein sicherer Standard, warnt Fairbanks, weil es Zeit vom Bauen und Testen abzieht.
- **Die Lehre:** Lassen Sie die Risiken den Aufwand bestimmen. Ein Teil eines Systems bekommt vielleicht eine Skizze am Whiteboard, ein anderer wochenlanges Prototyping und Testen: Genau so funktioniert das Modell.

Beabsichtigte Architektur und die Architectural Runway

Große Organisationen haben ein Koordinationsproblem, das ein einzelnes Team nicht hat: Dutzende Teams müssen Teile bauen, die zusammenpassen. Das Scaled Agile Framework (SAFe), ein weit verbreitetes und ebenso umstrittenes Framework zur Skalierung agiler Praktiken, beschreibt Architektur als Gleichgewicht zwischen *emergentem Design* innerhalb der Teams und *beabsichtigter Architektur*: einer bewussten, teamübergreifenden Planung der

Strukturen und Richtlinien, die die Teams gemeinsam nutzen. Das Ergebnis nennt es die Architectural Runway: den vorhandenen Code, die Komponenten und die technische Infrastruktur, die nötig sind, um Features der nahen Zukunft ohne übermäßige Umgestaltung und Verzögerung umzusetzen. Arbeit zur Verlängerung der Runway (etwa die Einführung einer Messaging-Plattform vor den Features, die sie brauchen) wird ausdrücklich neben der Feature-Arbeit geplant, und Features verbrauchen die Runway, indem sie sie nutzen.

Neutral beschrieben erfasst die Runway-Metapher etwas, das für jede Organisation gilt: Wenn die Infrastruktur, die ein Feature braucht, noch nicht existiert, wird dieses Feature langsam sein, und der Aufbau der Infrastruktur braucht Vorlaufzeit. Kritiker von SAFe argumentieren, es könne schwerfällige Planung von oben unter agilem Vokabular wieder einführen; Befürworter halten dagegen, dass im großen Maßstab eine gewisse zentrale Koordination unvermeidlich ist. Sie müssen SAFe nicht übernehmen, um die Idee zu nutzen: „Welche Grundlagen brauchen die Features des nächsten Quartals, und bauen wir sie rechtzeitig?“ ist eine gute Frage für jeden Architekten.

Kontinuierliche Architektur

Murat Erder und Pierre Pureur beschreiben in *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World* (2015) und *Continuous Architecture in Practice* (2021, mit Eoin Woods) Architektur als fortlaufenden Strom von Entscheidungen statt als Phase oder Dokument. Ihre sechs Grundsätze lauten: Produkte architektonisch gestalten, nicht nur Lösungen für Projekte; sich auf Qualitätsmerkmale konzentrieren, nicht auf funktionale Anforderungen; Designentscheidungen aufschieben, bis sie absolut notwendig sind; für Veränderung entwerfen und dabei die Stärke kleiner, lose gekoppelter Teile nutzen; für Bauen, Testen, Deployen und Betreiben entwerfen; und die Organisation der Teams nach dem Design des Systems ausrichten, an dem sie arbeiten.

Zwei dieser Grundsätze greifen frühere Abschnitte dieses Kapitels auf: Entscheidungen aufzuschieben ist der Last Responsible Moment, und Teams nach dem System auszurichten ist eine bewusste Anwendung des Gesetzes von Conway. Der praktische Schwerpunkt liegt darauf, dass die Einheit der Architekturarbeit die *Entscheidung* ist, die festgehalten wird, wenn sie getroffen wird, und auf Feedbackschleifen (automatisierte Tests, Fitness Functions, Monitoring in der Produktion), die dem Team zeigen, ob die Architektur ihre Qualitätsziele noch erfüllt.

Evolutionäre Architekturen bauen

Ein verwandter Strang stammt aus *Building Evolutionary Architectures* von Neal Ford, Rebecca Parsons und Patrick Kua (2017; zweite Auflage 2022 mit Pramod Sadalage). Langlebige Systeme müssen mit ihrem Umfeld Schritt halten, also entwerfen die Autoren für Veränderung,

statt die Zukunft vorherzusagen: viele kleine Schritte in technischen wie geschäftlichen Dimensionen, wobei explizit gemacht und möglichst automatisch geprüft wird, was „passend“ bedeutet. Wenn die Essenslieferplattform verspricht, dass der Bezahlvorgang schnell bleibt, kann ein automatisierter Test dieses Versprechen bei jedem Release absichern, während sich der Code darunter ändert. Abschnitt 11.3 behandelt diese Prüfungen, genannt Fitness Functions, und die ehrlichen Grenzen des Ansatzes.

Walking Skeletons und Spikes

Zwei einfache Techniken verbinden Architektur und Auslieferung schon ab der ersten Woche. Ein Walking Skeleton, ein Begriff, der mit Alistair Cockburn verbunden wird, ist eine winzige Implementierung des Systems, die eine kleine Funktion von Ende zu Ende durch alle wichtigen Architekturkomponenten ausführt. Bei einem Terminbuchungssystem für ein Krankenhaus könnte es einen Testnutzer anmelden, ihm die freien Termine einer fest eingetragenen Ambulanz aus dem Patientenverwaltungssystem anzeigen, einen davon buchen lassen und eine SMS senden, über die echte Delivery-Pipeline in eine produktionsnahe Umgebung deployt. Es tut fast nichts Nützliches, beweist aber, dass die Teile zusammenpassen, dass die Pipeline funktioniert und dass sich die Architektur bauen lässt. Jedes spätere Feature setzt Fleisch an ein Skelett, das bereits läuft.

Ein Spike (ein weiterer Begriff aus Extreme Programming) ist ein kurzes, zeitlich begrenztes Experiment, um eine Frage zu beantworten oder ein Risiko zu verringern: Kann die PAS-Schnittstelle 50 Anfragen pro Sekunde verarbeiten? Wie lange braucht der Identity Provider, um ein Token auszustellen? Kann unsere gewählte Bibliothek barrierefreie PDFs erzeugen? Spikes werden meist weggeworfen; ihr Ergebnis ist Wissen, festgehalten in einer Entscheidung. Sie sind der günstigste Weg, eine Entscheidung vor ihrem Last Responsible Moment von Vermutungen auf Belege zu stellen.

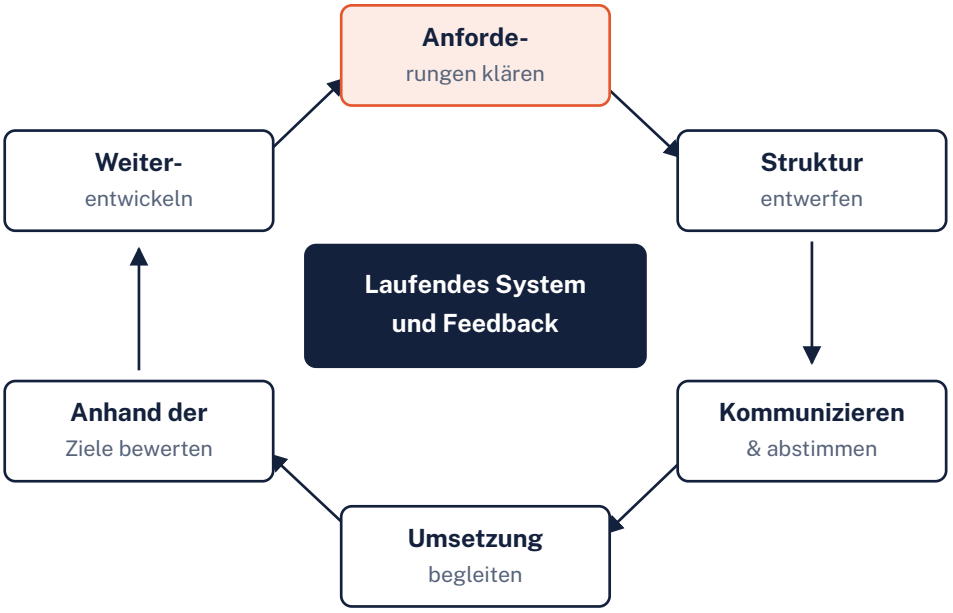
Architekturtätigkeiten in einem iterativen Zyklus

In einem iterativen Prozess findet Architekturarbeit nicht einmal statt; sie wiederholt sich in jedem Zyklus, in einer Tiefe, die den anstehenden Risiken des Teams entspricht. Der iSAQB-Lehrplan beschreibt die Kernaufgaben des Architekten auf eine Weise, die sich natürlich auf einen solchen Zyklus abbilden lässt, und Die Rolle des Architekten hat sie eingeführt. Hier sind sie als Schleife:

Anforderungen klären: die Stakeholder und Anliegen finden, die Qualitätsziele und Randbedingungen schärfen und die Risiken erkennen (siehe Stakeholder und Anliegen). **Strukturen entwerfen**: die bedeutsamen Entscheidungen treffen oder überarbeiten, Stile und Muster wählen und Grenzen und Schnittstellen festlegen. **Kommunizieren und abstimmen**: die Entscheidungen den Menschen erklären, die das System bauen, betreiben und bezahlen, in

Sichten, die zu jeder Zielgruppe passen, und sie festhalten. **Umsetzung begleiten**: mit den Teams arbeiten, während sie bauen, Fragen beantworten, Code reviewen und das Design anpassen, wenn die Realität sich widersetzt. **Bewerten**: prüfen, ob das System seine Qualitätsziele erfüllt, durch Reviews, Tests, Messungen in der Produktion und automatisierte Fitness Functions. **Weiterentwickeln**: auf das reagieren, was die Bewertung ergeben hat, Schulden tilgen und sich an geänderte Anforderungen anpassen, was Sie wieder zum Klären der Anforderungen zurückführt.

Der Architekturzyklus



Jede Iteration durchläuft alle sechs Tätigkeiten erneut, mit einer Tiefe, die den anstehenden Risiken des Teams entspricht, rund um ein laufendes System, das Feedback liefert.

Architektur in DevOps und Continuous Delivery

Continuous Delivery, bei der jede Änderung gebaut, getestet und releasefähig gehalten wird, und DevOps, das die Verantwortung für Entwicklung und Betrieb zusammenführt, verändern sowohl die Ziele der Architektur als auch ihre Feedbackschleifen.

Sie verändern die Ziele, weil **Deploybarkeit, Testbarkeit und Betreibbarkeit zu vollwertigen Qualitätsmerkmalen werden**. Ein System lässt sich nicht mehrmals am Tag releasen, wenn Komponenten gemeinsam in fester Reihenfolge deployt werden müssen, wenn Tests eine vollständige Kopie der Produktion brauchen oder wenn niemand am Monitoring erkennen kann, ob ein Release die Lage verschlechtert hat. Die in *Accelerate* (Nicole Forsgren, Jez Humble und Gene Kim, 2018) zusammengefasste Forschung ergab, dass eine lose gekoppelte Architektur,

die es Teams erlaubt, ihre Teile unabhängig zu testen und zu deployen, zu den stärksten Prädiktoren für die Leistung bei Continuous Delivery gehörte. Architektur wird also nicht nur von guter Auslieferungspraxis unterstützt; sie ist eine ihrer Voraussetzungen.

Sie verändern die Feedbackschleifen, weil das laufende System zum ehrlichsten Architektur-Review wird. Produktionsmetriken, Traces und Incident-Reviews zeigen weit zuverlässiger als Diagramme, wo die Architektur schwach ist. Architekturregeln lassen sich als automatisierte Fitness Functions in der Pipeline durchsetzen (Kapitel 11), Infrastruktur wird als Code definiert und wie Code reviewt (Kapitel 10), und Techniken wie Feature-Flags und Canary Releases machen viele Entscheidungen günstig umkehrbar. Die Rolle des Architekten verschiebt sich entsprechend: weniger Freigabe von Dokumenten, mehr Gestaltung der Plattform, der Pipeline und der Guardrails, mit denen Teams schnell gute Entscheidungen treffen können.

Checkliste: Ist Ihre Architekturarbeit angemessen?

Gehen Sie diese Liste zu Beginn eines Projekts und bei jedem wichtigen Meilenstein durch. Jeder Punkt ist eine Frage danach, ob der Aufwand zum Risiko passt.

- [ ] **Sie können die fünf größten Architekturrisiken benennen**
  Als konkrete Ausfallszenarien formuliert, priorisiert und jeweils mit einer verantwortlichen Person und einem Plan zur Verringerung.
- [ ] **Ihr Entwurfsaufwand konzentriert sich auf diese Risiken**
  Bereiche mit geringem Risiko und vertrauter Technologie bekommen eine Skizze; Bereiche mit hohem Risiko bekommen Prototypen, Messungen und Reviews.
- [ ] **Einwegtüren sind erkannt und werden bewusst entschieden**
  Schwer umkehrbare Entscheidungen sind bekannt, werden zum Last Responsible Moment getroffen und festgehalten; Zweiwegtüren bleiben den Teams überlassen.
- [ ] **Es gibt früh ein Walking Skeleton**
  Ein Pfad von Ende zu Ende durch die wichtigsten Komponenten läuft innerhalb der ersten Iterationen in einer produktionsnahen Umgebung.
- [ ] **Entscheidungen werden kurz festgehalten, wenn sie getroffen werden**
  Für jede bedeutsame Entscheidung gibt es ein kurzes Entscheidungsdokument; niemand muss die Begründung aus dem Gedächtnis rekonstruieren.
- [ ] **Qualitätsziele werden laufend geprüft**
  Zentrale Qualitätsmerkmale haben automatisierte Tests, Fitness Functions oder Messungen in der Produktion, nicht nur ein einmaliges Review.
- [ ] **Architektur blockiert die Auslieferung nicht**
  Teams warten selten auf eine Architekturentscheidung oder Freigabe; tun sie es doch, ist der Prozess für das Risiko zu schwerfällig.
- [ ] **Die Auslieferung untergräbt die Architektur nicht**
  Hotspots, Change Coupling und Regelverstöße werden verfolgt; Erosion fällt innerhalb von Wochen auf, nicht erst nach Jahren.
- [ ] **Der Umfang der Dokumentation passt zu ihren Lesern**
  Jedes Dokument und jede Sicht beantwortet das Anliegen eines echten Stakeholders und wird aktuell gehalten; ungelesene Dokumente werden ausgemustert.

ACHTUNG

Zwei entgegengesetzte Fehlentwicklungen

Architektur-Astronautik: monatelang Diagramme, Frameworks für Bedürfnisse, die nie eintreten, und ein Design, das schon vor dem ersten Release veraltet ist. **Zufällige Architektur**: Niemand befasst sich mit der Struktur, bis das System groß, schwer zu ändern und nicht mehr zu durchschauen ist, und eine teure Rettungsaktion unvermeidlich wird. Die Abhilfe für das Erste ist Feedback aus laufendem Code; die Abhilfe für das Zweite ist bewusste Aufmerksamkeit für die wenigen Entscheidungen, die am meisten zählen. Das risikogetriebene Modell ist ein Weg, zwischen beidem hindurchzusteuern.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Wann ist Big Design Up Front eine sinnvolle Wahl?**

Wenn Anforderungen stabil und gut verstanden sind, wenn Änderungen nach der Auslieferung extrem teuer sind oder wenn viele getrennte Parteien Teile bauen müssen, die genau zusammenpassen.

▼ **Welche Abbruchregel liefert das risikogetriebene Modell für Architekturarbeit?**

Aufhören, wenn die verbleibenden Risiken akzeptabel gering sind; der Aufwand sollte dem Risiko des Scheiterns entsprechen.

▼ **Was ist der Unterschied zwischen einem Walking Skeleton und einem Spike?**

Ein Walking Skeleton ist eine minimale Implementierung von Ende zu Ende durch die echte Architektur, die behalten und weiter ausgebaut wird. Ein Spike ist ein kurzes Experiment, um eine Frage zu beantworten, und wird meist weggeworfen, sobald es Wissen geliefert hat.

▼ **Warum macht Continuous Delivery Architektur wichtiger statt weniger wichtig?**

Häufige unabhängige Releases erfordern eine lose gekoppelte, testbare und deploybare Struktur; ohne sie kann die Delivery-Pipeline nicht schnell arbeiten.

▼ **Probieren Sie es aus: Für ein neues Ticketing-System steht der Zahlungsanbieter fest, aber niemand weiß, ob Platzreservierungen 2500 Besucher pro Sekunde überstehen. Wie viel Vorabentwurf würden Sie betreiben, und was würden Sie zuerst bauen?**

Folgen Sie dem Risiko. Verwenden Sie leichtes Design, wo die Dinge gut verstanden sind, etwa bei der Zahlungsintegration, und stecken Sie den Aufwand in das größte Risiko, die Platzreservierungen unter Last. Bauen Sie früh ein Walking Skeleton durch den gesamten Kaufpfad, und führen Sie einen Spike durch, der den Reservierungsansatz unter Last testet, damit die Entscheidung auf Belegen beruht. Hören Sie mit der Analyse auf, wenn das verbleibende Risiko akzeptabel ist. Weder ein vollständiges Design für alles noch gar keines würde gut passen.

QUELLEN

- Principles behind the Agile Manifesto: https://agilemanifesto.org/principles.html
- A Risk-Driven Model for Agile Software Architecture, George Fairbanks (Methods and Tools): https://www.methodsandtools.com/archive/agilearchi.html
- Is Design Dead?, Martin Fowler: https://martinfowler.com/articles/designDead.html
- Architectural Runway (SAFe): https://framework.scaledagile.com/architectural-runway
- Continuous Architecture (Erder, Pureur and Woods): https://continuousarchitecture.com/
- Evolutionary Architecture (authors' site): https://evolutionaryarchitecture.com/
- O'Reilly: Building Evolutionary Architectures, 2nd edition: https://www.oreilly.com/library/view/building-evolutionary-architectures/9781492097532/

Weiterführende Literatur

- *Software Architecture in Practice, 4th edition*, Len Bass, Paul Clements and Rick Kazman (2021), Addison-Wesley. Das Standardlehrbuch des SEI: Strukturen, Qualitätsmerkmale und Taktiken sowie die Definition, die in diesem Kurs durchgehend verwendet wird.
  https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- *Fundamentals of Software Architecture, 2nd edition*, Mark Richards and Neal Ford (2025), O'Reilly Media. Eine praxisnahe, gut lesbare Einführung in die Rolle des Architekten, das Denken in Abwägungen und Architekturstile.
- *ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description*, ISO/IEC JTC 1/SC 7 and IEEE (2022), ISO. Der internationale Standard für Architekturbeschreibungen, Stakeholder, Anliegen und Sichten.
  https://www.iso.org/standard/74393.html
- *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 Quelle der Strukturkategorien Modul, Komponente und Konnektor sowie Zuordnung.
  https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/
- *Who Needs an Architect?*, Martin Fowler (2003), IEEE Software. Ein kurzer, einflussreicher Essay darüber, was Architektur ist und welche Art von Architekt ein Team braucht.
  https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- *Just Enough Software Architecture: A Risk-Driven Approach*, George Fairbanks (2010), Marshall & Brainerd. Erklärt, wie viel Architekturarbeit nötig ist, ausgerichtet an den Risiken, denen ein Projekt tatsächlich gegenübersteht.
  https://www.georgefairbanks.com/book/
- *The Software Architect Elevator*, Gregor Hohpe (2020), O'Reilly Media. Über die Rolle des Architekten in großen Organisationen, als Bindeglied zwischen Geschäftsstrategie und technischer Umsetzung.
- *The WyCash Portfolio Management System (OOPSLA '92 experience report)*, Ward Cunningham (1992), OOPSLA 1992. Der kurze Bericht, in dem die Metapher der technischen Schulden zum ersten Mal auftauchte.
  https://c2.com/doc/oopsla92.html

Glossar

Abhängigkeitszyklus
Eine Schleife, in der Module direkt oder indirekt voneinander abhängen, sodass keines für sich allein verstanden, getestet oder ausgeliefert werden kann.

Abwägung (Trade-off)
Eine Wahl, bei der man für einen Vorteil Kosten in Kauf nimmt oder auf einen anderen Vorteil verzichtet, etwa schnellere Auslieferung um den Preis höherer Betriebskosten.

Änderungskosten
Der Aufwand, das Geld, die Zeit und das Risiko, die nötig sind, um eine Entscheidung zu ändern, nachdem bereits darauf aufgebaut wurde.

Anliegen
Ein Interesse, das ein Stakeholder an einem System hat, etwa seine Kosten, Performance, Sicherheit, Änderbarkeit oder die Einhaltung rechtlicher Vorgaben.

Architectural Runway
Im Scaled Agile Framework der vorhandene Code, die Komponenten und die Infrastruktur, die nötig sind, um kurzfristig geplante Features ohne übermäßiges Umgestalten und Verzögerung umzusetzen.

Architecture Decision Record (ADR)
Ein kurzes Dokument, das eine wesentliche Entscheidung festhält: ihren Kontext, die erwogenen Optionen, die Entscheidung, ihren Status und ihre Konsequenzen.

Architekturbeschreibung
Ein Arbeitsergebnis (Dokumente, Diagramme, Modelle, Entscheidungsdokumente), das eine Architektur ausdrückt. Die Architektur existiert unabhängig davon, ob sie beschrieben ist; die Beschreibung ist das Mittel, mit dem Menschen sie teilen.

Architekturdrift
Änderungen, die ohne Kenntnis der beabsichtigten Architektur vorgenommen werden und das System, ohne ausdrückliche Regeln zu verletzen, von ihr wegbewegen und ihre Konzepte verwischen.

Architekturerosion
Der allmähliche Verfall eines Systems, wenn Änderungen seine beabsichtigte Architektur verletzen, zum Beispiel durch das Umgehen von Schichten oder das Hinzufügen verbotener Abhängigkeiten.

Architekturrelevant
Bezeichnet eine Entscheidung oder Anforderung, die später teuer zu ändern ist, viele Teile eines Systems betrifft oder ein zentrales Qualitätsmerkmal bestimmt.

Architekturschulden
Technische Schulden in der Struktur eines Systems, etwa falsche Grenzen, eine gemeinsam genutzte Datenbank oder zyklische Abhängigkeiten; sie sind teuer zu tilgen, weil sie viele Teile berühren.

Beziehung
Eine Verbindung zwischen Elementen, etwa dass ein Modul ein anderes nutzt, ein Service einen anderen aufruft oder ein Service auf einem Server deployt ist.

Big Design Up Front (BDUF)
Ein Ansatz, bei dem Architektur und Design vollständig im Detail ausgearbeitet werden, bevor die Implementierung beginnt.

Blickwinkel (Viewpoint)
Eine Festlegung, wie eine bestimmte Art von Architektursicht erstellt und gelesen wird: welchen Stakeholdern und Anliegen sie dient und welche Modellierungskonventionen sie verwendet.

Change Coupling
Zwei oder mehr Dateien oder Komponenten, die sich tendenziell in denselben Commits ändern, was eine Abhängigkeit offenlegt, die im Code möglicherweise nicht sichtbar ist.

Element
Ein Teil eines Systems, den ein Architekt oder eine Architektin als Einheit betrachtet, etwa ein Modul, ein Service, eine Datenbank, eine Queue oder ein Server.

Emergentes Design
Ein Ansatz, bei dem das Design schrittweise wächst, geprägt von den Anforderungen jeder neuen Funktionalität und kontinuierlichem Refactoring.

Feature-Flag
Ein Konfigurationsschalter, der ein Verhalten zur Laufzeit ein- oder ausschaltet, ohne neuen Code zu deployen.

Feinentwurf
Entscheidungen über das Innere eines Teils eines Systems (seine Klassen, Funktionen, Algorithmen und lokalen Datenstrukturen), die sich ändern lassen, ohne den Rest zu beeinflussen.

Fitness Function
Eine automatisierte Prüfung, die Ihnen sagt, ob ein System eine architektonische Eigenschaft noch besitzt, etwa ein Test, der fehlschlägt, wenn ein Modul von einem anderen abhängt, das es nicht verwenden darf.

Gesetz von Conway
Melvin Conways Beobachtung von 1968, dass Organisationen Systeme entwerfen, deren Struktur die Kommunikationsstruktur der Organisation selbst widerspiegelt.

Gewichtete Entscheidungsmatrix
Eine Tabelle, die jede Option anhand gewichteter Kriterien bewertet, damit die Begründung einer Wahl offenliegt und hinterfragt werden kann.

Hotspot
Ein Teil des Codes, der zugleich komplex ist und häufig geändert wird, gefunden durch die Kombination der Versionskontrollhistorie mit einem Komplexitätsmaß; ein naheliegender Ort, um Schulden zu tilgen.

Kognitive Verzerrung
Ein systematischer Denkfehler, etwa der Ankereffekt oder der Sunk-Cost-Effekt, der dazu führt, dass Menschen Optionen unfair beurteilen.

Komponente
Eine Einheit, die zur Laufzeit des Systems existiert, etwa ein Service, ein Prozess oder eine Datenbank, und über Konnektoren mit anderen Komponenten interagiert.

Konnektor
Der Laufzeitmechanismus, über den Komponenten interagieren: ein HTTP-Aufruf, eine Message Queue, eine gemeinsame Datenbank, ein Event-Stream oder ein Funktionsaufruf.

Last Responsible Moment
Der Zeitpunkt, ab dem ein weiteres Aufschieben einer Entscheidung eine wichtige Option zunichtemachen würde; wer dann entscheidet, nutzt die meisten Informationen, ohne Schaden zu verursachen.

Lösungsarchitektur
Die Architektur einer Lösung für ein einzelnes Geschäftsproblem, die oft mehrere Anwendungen, Anbieter und Teams umfasst, etwa ein neuer Online-Schadensmeldeprozess für einen Versicherer.

Macht-Interesse-Matrix
Eine Vier-Felder-Matrix, die Stakeholder nach ihrer Macht über ein Projekt und ihrem Interesse daran einordnet, um zu planen, wie eng jede Gruppe eingebunden wird.

Monolith
Ein System, das als eine einzige Einheit gebaut und deployt wird, selbst wenn sein Code intern in sauber getrennte Module aufgeteilt ist.

Proof of Concept
Ein kleines, zeitlich begrenztes Stück funktionierenden Codes, das gebaut wird, um zu prüfen, ob eine Idee, Technologie oder Entwurfsentscheidung funktioniert, bevor man sich darauf festlegt.

Qualitätsmerkmal
Eine messbare Eigenschaft eines Systems, die angibt, wie gut es seine Aufgabe erfüllt, etwa Performance, Verfügbarkeit, Sicherheit oder Änderbarkeit. Kapitel 2 behandelt sie ausführlich.

Querschnittsanliegen (Cross-Cutting Concern)
Ein Anliegen, das viele Teile eines Systems zugleich betrifft, etwa Logging, Sicherheit, Fehlerbehandlung oder Persistenz, und daher meist einmal für das ganze System entschieden wird.

Refactoring
Die Änderung der inneren Struktur von Code, ohne sein äußeres Verhalten zu ändern, um ihn leichter verständlich und änderbar zu machen.

Risikogetriebenes Modell
George Fairbanks' Ansatz, bei dem sich der Umfang der Architekturarbeit nach den Risiken richtet, denen ein Projekt gegenübersteht: Risiken erkennen, Techniken anwenden, die Risikominderung bewerten.

Serverless
Ein Cloud-Modell, bei dem der Anbieter Ihre Funktionen oder Workflows bei Bedarf ausführt und nach Nutzung abrechnet, sodass Sie selbst keine Server verwalten.

Sicht
Eine Darstellung einer oder mehrerer Strukturen eines Systems, ausgerichtet auf bestimmte Anliegen bestimmter Stakeholder, etwa ein Deployment-Diagramm für das Betriebspersonal.

Softwarearchitektur
Die grundlegenden Strukturen eines Softwaresystems und die wesentlichen Entscheidungen, die sie geprägt haben: seine Hauptbestandteile, wie sie zusammenhängen, ihre wichtigen Eigenschaften und die Prinzipien, nach denen es gebaut und weiterentwickelt wird.

Spike
Eine kurze, zeitlich begrenzte Untersuchung oder ein Prototyp, um eine technische Frage zu beantworten oder ein Risiko zu verringern, und der danach normalerweise verworfen wird.

Staff Engineer
Eine Senior-Rolle als Individual Contributor oberhalb des Senior Engineer, die teamübergreifende technische Führung ohne Personalverantwortung umfasst; Principal Engineer ist meist die nächsthöhere Stufe.

Stakeholder
Eine Person, ein Team oder eine Organisation mit einem Interesse an einem System, etwa Nutzer, Auftraggeber, Entwickler, Betrieb, Sicherheitsverantwortliche oder Aufsichtsbehörden.

Struktur
Eine Menge von Elementen und den Beziehungen zwischen ihnen, aus einem bestimmten Blickwinkel betrachtet: zum Beispiel die Codemodule und ihre Abhängigkeiten oder die laufenden Services und wie sie kommunizieren.

Technische Breite
Das Wissen, dass es viele Technologien und Ansätze gibt, was sie gut und schlecht können und wann man sie einsetzt, auch ohne tiefe praktische Erfahrung mit jedem einzelnen.

Technische Schulden
Die zusätzlichen künftigen Änderungskosten, verursacht durch Abkürzungen oder durch einen Entwurf, der nicht mehr passt, bezahlt in Form langsamerer und riskanterer Arbeit, bis sie durch eine Verbesserung des Entwurfs getilgt werden.

Tilgungsaufwand (technischer Schulden)
Der Aufwand, der nötig ist, um ein Stück technischer Schulden zu beheben, also Code oder Struktur in den Zustand zu bringen, in dem sie sein sollten.

TOGAF
Das Open Group Architecture Framework, ein weit verbreitetes Framework und eine Methode für Unternehmensarchitektur, herausgegeben von The Open Group; die aktuelle Version ist der TOGAF Standard, 10th Edition (2022).

Toter Code
Code, der nicht mehr ausgeführt werden soll, aber im deployten System noch vorhanden ist, wo er versehentlich ausgelöst werden kann.

Umkehrbarkeit
Wie kostengünstig und sicher sich eine Entscheidung rückgängig machen lässt. Gut umkehrbare Entscheidungen können schnell getroffen werden; irreversible verdienen mehr Analyse.

Unternehmensarchitektur
Architektur auf der Ebene einer ganzen Organisation: wie ihre Geschäftsfähigkeiten, Informationen, Anwendungen und Technologien zusammenpassen und sich im Laufe der Zeit verändern.

Walking Skeleton
Eine winzige Implementierung des Systems, die eine kleine End-to-End-Funktion über alle wichtigen Architekturkomponenten hinweg ausführt und meist zuerst gebaut wird.

Wissenspyramide
Ein Modell mit drei Ebenen: was Sie wissen, wovon Sie wissen, dass Sie es nicht wissen, und wovon Sie nicht wissen, dass Sie es nicht wissen. Architekten erweitern gezielt die mittlere Ebene.

Zinsen (technischer Schulden)
Der zusätzliche Aufwand, die Verzögerung oder das Risiko, die bei jeder Änderung anfallen, weil die Schulden nicht getilgt wurden.