Handbuch moderner Softwarearchitektur

Architekturstile, Patterns und Best Practices

KAPITEL 4

Definition architektonischer Eigenschaften

Wir steigen nun in die Details des strukturellen Designs ein, eines der zentralen Themenbereiche in der Softwarearchitektur. Dabei geht es vor allem um zwei Aktivitäten: die *Analyse architektonischer Eigenschaften*, die Thema dieses Kapitels ist, und das *Design logischer Komponenten*, das in Kapitel 8 behandelt wird. Architekten können diese beiden Aktivitäten in beliebiger Reihenfolge ausführen (sogar parallel), aber sie kumulieren gemeinsam an einem kritischen Punkt.

Entscheidet sich ein Unternehmen dazu, ein bestimmtes Problem durch den Einsatz von Software zu lösen, stellt es eine Reihe von Anforderungen für dieses System zusammen (es gibt dafür eine große Vielzahl an Techniken, wie in Kapitel 8 zu sehen ist). Wir werden diese Anforderungen im Buch als *Problemdomäne* (oder einfach *Domäne*) bezeichnen. In Kapitel 1 haben Sie gelernt, dass die architektonischen Eigenschaften die wichtigen Aspekte eines Systems sind – unabhängig von der Problemdomäne und wichtig für den Erfolg des Systems. In diesem Kapitel werden wir uns den Begriff genauer vornehmen und spezifische architektonische Eigenschaften ansprechen.

Architektinnen und Architekten arbeiten oft zusammen, um die Domäne zu definieren, aber sie müssen auch all die Dinge definieren, aufdecken und anderweitig analysieren, die die Software zu erledigen hat, die aber nicht mit der Domänenfunktionalität zusammenhängen – *architektonische Eigenschaften*. Die Rolle der Architekten beim Definieren der architektonischen Eigenschaften ist Teil dessen, was die Softwarearchitektur vom Programmieren und vom Design unterscheidet. Sie müssen bei der Gestaltung einer Softwarelösung noch weitere Faktoren berücksichtigen, wie in Abbildung 4.1 zu sehen ist.

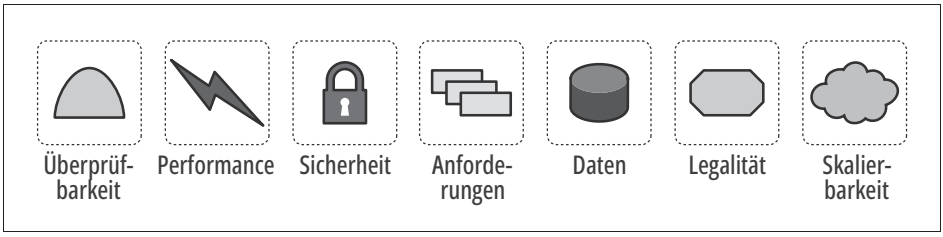

Abbildung 4.1: Eine Softwarelösung besteht aus fachlichen und architektonischen Anforderungen.

Die Zähigkeit des Begriffs der »nicht funktionalen Anforderungen«

Viele Organisationen beschreiben architektonische Eigenschaften mit einer Vielzahl von Begriffen, unter anderem durch *nicht funktionale Anforderungen* – ein Begriff, der geschaffen wurde, um architektonische Eigenschaften von *funktionalen Anforderungen* zu unterscheiden. Wir mögen diesen Begriff nicht, weil er sich aus sprachlicher Sicht selbst verunglimpft und negative Auswirkungen hat: Wie überzeugen Sie Teams davon, etwas Aufmerksamkeit zu schenken, das »nicht funktional« ist? Ein anderer beliebter Begriff sind die *Qualitätsattribute*, den wir ebenfalls nicht mögen, weil er eher eine nachträgliche Qualitätsbewertung statt eines Designs impliziert.

Wir bevorzugen den Begriff *architektonische Eigenschaften*, weil er Überlegungen beschreibt, die für den Erfolg der Architektur – und damit das System als Ganzes – entscheidend sind, ohne die Wichtigkeit dieser Überlegungen abzuwerten. In *Head First Software Architecture* (O'Reilly, 2024, *https://www.oreilly.com/library/view/head-first-software/9781098134341*) beziehen wir uns auf architektonische Eigenschaften als die *Fähigkeiten* des Systems – die Domäne repräsentiert im Gegensatz dazu das *Verhalten*.

Manchmal bleiben Begriffe hängen, und die nicht funktionalen Anforderungen scheinen bei Softwarearchitekten besonders zäh zu sein. Sie sind in vielen Organisationen weiterhin verbreitet. Der Begriff tauchte erstmals in den späten 1970er-Jahren in der Literatur zur Softwareentwicklung auf – ungefähr zur selben Zeit wie das *Function-Point-Verfahren*, eine Schätztechnik, die Systemanforderungen in »Function Points« unterteilt, die jeweils eine gewisse Arbeitseinheit repräsentieren. Theoretisch könnten Teams am Ende des Analyseprozesses all die Function Points aufsummieren und damit Einblicke in das Projekt erhalten. Leider bietet das Ganze aber nur oberflächlich Sicherheit, ist von Subjektivität durchzogen (wie das bei vielen Schemata zum Abschätzen der Fall ist) und daher nicht mehr länger im Einsatz.

Aber eine Erkenntnis aus dieser Zeit ist geblieben: Ein Großteil des Aufwands beim Erstellen eines Systems betrifft die *Fähigkeiten* des Systems und nicht seine Anforderungen. Diese Aufwände wurden als *Non-Function Points* bezeichnet, was dazu führte, dass der Begriff der *nicht funktionalen Anforderungen* Verbreitung fand.

Architektonische Eigenschaften und Systemdesign

Damit eine Anforderung als architektonische Eigenschaft angesehen werden kann, muss sie drei Kriterien erfüllen. Sie muss eine nicht domänenspezifische Designentscheidung spezifizieren, bestimmte strukturelle Aspekte eines Designs beeinflussen und für den Erfolg der Applikation kritisch oder entscheidend sein. Diese miteinander verschränkten Teile unserer Definition sehen Sie zusammen mit ein paar weiteren Faktoren in Abbildung 4.2.

Schauen wir uns diese Komponenten genauer an:

Eine architektonische Eigenschaft bezeichnet eine nicht domänenspezifische Designentscheidung.

Strukturelles Design besteht in der Softwarearchitektur aus zwei Aktivitäten eines Architekten: das Verstehen der Problemdomäne und das Ermitteln, welche Fähigkeiten das System unterstützen muss, damit es erfolgreich ist. Die Überlegungen zum Domä-

nendesign drehen sich um das Verhalten des Systems, und architektonische Eigenschaften definieren dessen Fähigkeiten. Zusammen definieren diese beiden Aktivitäten das strukturelle Design.

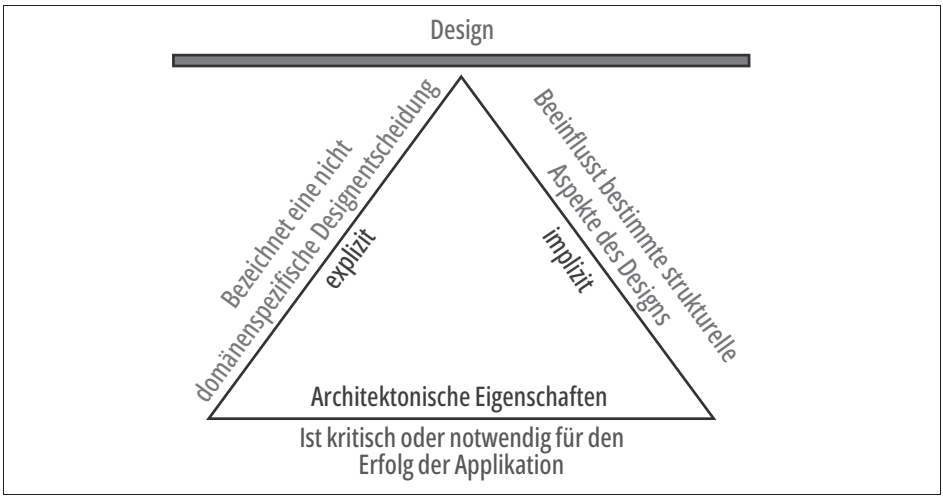

Abbildung 4.2: Unterscheidungsmerkmale architektonischer Eigenschaften

Während die Designanforderungen vorgeben, *was* die Applikation tun soll, definieren architektonische Eigenschaften, *wie* die Anforderungen implementiert werden sollen und *warum* bestimmte Entscheidungen getroffen wurden – kurz gesagt: die Operations- und Designkriterien, die für den Erfolg des Projekts notwendig sind.

So sind beispielsweise bestimmte Performance-Levels eine wichtige architektonische Eigenschaft, die aber oft nicht in Anforderungsdokumenten genannt wird. Oder noch passender: Kein Anforderungsdokument enthält die Weisung, technische Schulden (*Technical Debt*) zu verhindern. Dennoch ist dies eine häufige Designüberlegung. Diese Unterscheidung zwischen expliziten und impliziten Eigenschaften behandeln wir in »Architektonische Eigenschaften aus domänenspezifischen Anforderungen ableiten« auf Seite 83.

Eine architektonische Eigenschaft beeinflusst bestimmte strukturelle Aspekte eines Designs.

Der Hauptgrund, warum Architekten versuchen, architektonische Eigenschaften für Projekte zu beschreiben, hat mit Designüberlegungen zu tun: Kann der Architekt sie über das Design umsetzen, oder sind für diese architektonische Eigenschaft bestimmte strukturelle Überlegungen notwendig, damit sie erfolgreich ist?

Zum Beispiel ist Sicherheit in so gut wie jedem Projekt ein Thema, und alle Systeme müssen beim Design und der Programmierung entsprechende Maßnahmen umsetzen. Muss der Architekt hierfür etwas Besonderes entwickeln, entsteht daraus jedoch eine architektonische Eigenschaft.

Schauen wir uns zwei häufige architektonische Eigenschaften an: Sicherheit und Skalierbarkeit. Architekten können in einem monolithischen System für Sicherheit sorgen, indem auf eine gute Coding-Hygiene geachtet wird, wozu bekannte Techniken

wie Verschlüsselung, Hashing und Salting gehören. (Architektonische Fitnessfunktionen, die auch in diesen Bereich fallen, werden in Kapitel 6 behandelt.) Andererseits würde der Architekt in einer verteilten Architektur wie Microservices einen gehärteteren Service mit strikteren Zugriffsprotokollen bauen – ein struktureller Ansatz. Architekten können die Sicherheit also entweder über das Design oder über die Struktur erreichen. Nun zur Skalierbarkeit: Kein noch so cleveres Design ermöglicht einer monolithischen Architektur, über einen bestimmten Punkt hinaus skalierbar zu sein. Darüber hinaus muss das System zu einem verteilten Architekturstil wechseln.

Architekten achten genau auf betriebsrelevante architektonische Eigenschaften (siehe »Betriebsrelevante architektonische Eigenschaften« auf Seite 75), weil diese oft besondere strukturelle Unterstützung benötigen.

Eine architektonische Eigenschaft muss kritisch oder notwendig für den Erfolg der Applikation sein.

Applikationen *können* eine Unmenge architektonischer Eigenschaften unterstützen ... sollten es aber nicht. Jede unterstützte architektonische Eigenschaft erhöht die Komplexität des Designs. Daher liegt eine wichtige Aufgabe der Architekten darin, mit möglichst wenigen architektonischen Eigenschaften auszukommen.

Wir unterscheiden bei den architektonischen Eigenschaften außerdem zwischen implizit und explizit. *Implizite* Eigenschaften findet man nur selten in den Anforderungsdokumenten, obwohl sie für den Projekterfolg wichtig sind. Hierzu gehören Dinge wie Verfügbarkeit, Verlässlichkeit und der Sicherheitsunterbau fast aller Applikationen. Schon in der Analysephase müssen Architekten ihr Fachwissen einsetzen, um diese architektonischen Eigenschaften zu entdecken. Ein Beispiel ist ein Unternehmen, das sich auf Hochfrequenzhandel mit Wertpapieren spezialisiert hat. Obwohl eine niedrige Latenz nicht für alle Systeme spezifiziert wurde, wissen Architekten, wie wichtig sie für diesen Problembereich ist. Explizite architektonische Eigenschaften können in Anforderungsdokumenten oder anderen spezifischen Anweisungen auftauchen.

Die Wahl eines Dreiecks in Abbildung 4.2 für die Darstellung ist Absicht: Jedes definierende Element unterstützt die anderen, die wiederum das Gesamtdesign des Systems tragen. Der Drehpunkt des Dreiecks illustriert, wie diese architektonischen Eigenschaften oft zueinander in Beziehung stehen, was dazu führt, dass Architekten den Begriff *Kompromiss* so häufig benutzen.

Architektonische Eigenschaften, eine (unvollständige) Liste

Architektonische Eigenschaften können unterschiedlich komplex sein: von Eigenschaften auf unterer Ebene (wie Modularität) bis hin zu anspruchsvollen Dingen (wie Skalierbarkeit und Elastizität). Obwohl schon viele Anstrengungen unternommen wurden, gibt es hierfür bis heute keinen universellen Standard. Stattdessen erstellt jedes Unternehmen seine eigene Interpretation dieser Begriffe. Hinzu kommt, dass sich die Welt der Software so schnell verändert, dass es ständig neue Konzepte, Begriffe, Vorgehensweisen und Prüfmethoden gibt, die neue Möglichkeiten für die Definition architektonischer Eigenschaften schaffen.

Obwohl es schwierig ist, aufgrund der schieren Menge und Verschiedenheit die architektonischen Eigenschaften zu quantifizieren, werden sie von Architekten grob in verschiedene Kategorien eingeteilt. Die folgenden Abschnitte beschreiben einige davon und geben ein paar Beispiele.

Betriebsrelevante architektonische Eigenschaften

Für den Betrieb eines Systems relevante architektonische Eigenschaften sind beispielsweise Fähigkeiten wie Performance, Skalierbarkeit, Elastizität, Verfügbarkeit und Verlässlichkeit. Tabelle 4.1 zeigt einige betriebsrelevante architektonische Eigenschaften.

Tabelle 4.1: Häufige betriebsrelevante architektonische Eigenschaften

Begriff	Definition
Verfügbarkeit (Availability)	Wie lange muss das System verfügbar sein (für 24/7 werden Maßnahmen gebraucht, mit denen das System bei einem Fehler schnell wieder hochgefahren werden kann)?
Beständigkeit (Continuity)	Die Fähigkeit des Systems, sich von einem Notfall zu erholen.
Performance	Wie gut ist die Leistungsfähigkeit des Systems? Zum Messen können Stresstests, eine Analyse von Leistungsspitzen, die Häufigkeit, mit der bestimmte Funktionen genutzt werden, und Antwortzeiten zum Einsatz kommen.
Wiederherstellbarkeit (Recoverability)	Anforderungen, um den Betrieb sicherzustellen (Wie schnell muss ein System im Notfall wieder verfügbar sein?). Das hat Einfluss auf die Backup-Strategie und den Bedarf an duplizierter Hardware.
Verlässlichkeit/ Sicherheit (Reliability/ Security)	Feststellen, ob das System ausfallsicher sein muss oder ob es so missionskritisch ist, dass Menschenleben auf dem Spiel stehen. Wird ein Ausfall das Unternehmen eine große Menge Geld kosten? Das Ergebnis ist oft eher in einem Spektrum zu finden und nicht nur schwarz oder weiß.
Robustheit (Robustness)	Die Fähigkeit, mit Fehlern und Grenzsituationen umzugehen, wobei das System zum Beispiel weiterläuft, auch wenn es keine Internetverbindung gibt oder es zu Strom- oder Hardwareausfällen kommt.
Skalierbarkeit (Scalability)	Die Fähigkeit eines Systems, auch dann noch zu funktionieren, wenn die Zahl der Nutzer oder Anfragen steigt.

Die betriebsrelevanten architektonischen Eigenschaften haben Überschneidungen mit betrieblichen Belangen und DevOps.

Strukturelle architektonische Eigenschaften

Architekten sind für eine ordentliche Codestruktur verantwortlich. Oftmals hat der Architekt die alleinige, zumindest aber eine Teilverantwortung für die Codequalität. Hierzu gehören Dinge wie gute Modularität, kontrollierte Kopplung zwischen Komponenten, lesbarer Code und eine Vielzahl weiterer interner Maßnahmen zur Qualitätssicherung. Tabelle 4.2 zeigt eine Reihe von strukturellen architektonischen Eigenschaften.

Tabelle 4.2: Strukturelle architektonische Eigenschaften

Begriff	Definition
Konfigurierbarkeit	Wie einfach Endbenutzer Teile der Softwarekonfiguration über Schnittstellen anpassen können.
Erweiterbarkeit	Wie gut die Architektur Änderungen zulässt, die ihre bestehende Funktionalität erweitern.
Installierbarkeit	Die Leichtigkeit, mit der ein System auf allen notwendigen Plattformen installiert werden kann.
Nutzbarkeit/Wiederverwendung	Die Fähigkeit, häufig verwendete Komponenten in mehreren Produkten einzusetzen.
Lokalisierung	Unterstützung einer Mehrsprachigkeit für Eingabe-/Abfrageschnittstellen in Datenfeldern.
Wartbarkeit	Wie einfach es ist, Änderungen vorzunehmen und das System zu verbessern.
Portabilität	Die Fähigkeit des Systems, auf mehr als einer Plattform laufen zu können (zum Beispiel auf Oracle und mit SAP DB).
Aktualisierbarkeit	Die Fähigkeit, einfach und schnell auf Servern und Clients auf eine neue Version umzusteigen.

Cloud-Eigenschaften

Die Welt der Softwareentwicklung unterliegt fortlaufend Änderungen und Weiterentwicklungen – die neueste Errungenschaft ist die Cloud. Als die erste Auflage des Buchs veröffentlicht wurde, gab es zwar schon cloudbasiertes Computing, aber es war nicht allgegenwärtig. Mittlerweile interagieren die meisten Systeme auf die eine oder andere Art und Weise mit cloudbasierten Systemen. Ein paar dieser Überlegungen finden sich in Tabelle 4.3.

Tabelle 4.3: Architektonische Eigenschaften im Cloud-Umfeld

Begriff	Definition
On-Demand-Skalierbarkeit	Die Fähigkeit des Cloud-Providers, Ressourcen dynamisch in Abhängigkeit von der Anforderung hochzuskalieren.
On-Demand-Elastizität	Die Flexibilität des Cloud-Providers, wenn es Spitzen bei Ressourcenanforderungen gibt – ähnelt der Skalierbarkeit.
Zonenbasierte Verfügbarkeit	Die Fähigkeit des Cloud-Providers, Ressourcen nach Computing-Zonen zu unterteilen, um Systeme resilienter zu machen.
Regionsbasierter Datenschutz und Sicherheit	Die (rechtliche) Fähigkeit des Cloud-Providers, Daten aus verschiedenen Ländern und Regionen zu verarbeiten. In vielen Staaten gibt es Gesetze zur Speicherung der Daten ihrer Einwohnerinnen und Einwohner (die oft das Speichern in anderen Regionen verbieten).

Für die zweite Auflage dieses Buchs haben wir zu jedem Kapitel zu architektonischen Stilen einen Abschnitt hinzugefügt, der beschreibt, wie dieser Stil Einfluss auf Überlegungen rund um die Cloud hat.

Bereichsübergreifende architektonische Eigenschaften

Viele architektonische Eigenschaften können nur schwer bestimmten Kategorien zugeordnet werden. Trotzdem haben sie großen Einfluss auf das Design. Einige dieser Einschränkungen und Überlegungen sehen Sie in Tabelle 4.4.

Tabelle 4.4: Bereichsübergreifende architektonische Eigenschaften

Begriff	Definition
Zugänglichkeit	Zugriff für alle Nutzer, diejenigen mit Einschränkungen wie Farbenblindheit oder Hörverlust eingeschlossen.
Archivierbarkeit	Die Grenzen des Systems in Bezug auf das Archivieren oder Löschen von Daten nach einem gewissen Zeitraum.
Authentifizierung	Sicherheitsanforderungen, um die Identität der Nutzer festzustellen und zu überprüfen.
Autorisierung	Sicherheitsanforderungen, um sicherzustellen, dass Nutzer nur auf bestimmte Funktionen der Applikation zugreifen können (nach Anwendungsfall, Subsystem, Webseite, Businessregel, Field Level und so weiter).
Rechtliches	Unter welchen rechtlichen Beschränkungen wird das System betrieben (zum Beispiel Datenschutzgesetze wie die DSGVO oder Finanzregulierungen wie Sarbanes-Oxley in den USA)? Gibt es irgendwelche Regelungen, die beeinflussen, wie die Applikation erstellt bzw. bereitgestellt wird? Dazu gehören auch Vorbehaltsrechte durch das Unternehmen?
Privatsphäre	Die Möglichkeit, Transaktionen vor internen Mitarbeitern des Unternehmens zu verstecken (verschlüsselte Transaktionen, sodass nicht einmal DBAs oder Netzwerkarchitekten sie sehen können).
Sicherheit	Müssen die Daten in der Datenbank verschlüsselt werden? Ist eine Verschlüsselung für die Kommunikation zwischen internen Systemen nötig? Welche Art der Authentifizierung wird für den entfernten Benutzerzugriff eingesetzt? Was für Sicherheitsmaßnahmen sind sonst erforderlich?
Supportbarkeit	Welches Level an technischem Support benötigt die Anwendung? Wie weitgehend sind Logging und andere Dinge zum Debuggen erforderlich?
Usability/Erreichbarkeit	Welcher Trainingsaufwand ist nötig, damit Nutzer ihre Ziele mit der Lösung/Applikation erreichen können?

Keine Liste architektonischer Eigenschaften ist absolut vollständig. Basierend auf einmaligen Faktoren kann jede Software wichtige architektonische Eigenschaften dazu erfinden. Viele der gerade genannten Begriffe können aufgrund feiner Unterschiede oder dem Fehlen objektiver Definitionen unpräzise und mehrdeutig sein. Zum Beispiel scheinen die Begriffe

Interoperabilität und *Kompatibilität* gleichbedeutend zu sein, was für einige Systeme auch zutrifft. Dennoch sind sie unterschiedlich, weil *Interoperabilität* eine einfache Integration mit anderen Systemen impliziert, während es bei der *Kompatibilität* eher um Standards der jeweiligen Branche oder des Fachbereichs geht. Ein weiteres Beispiel ist der englische Begriff *Learnability.* Einerseits steht er für *Lernbarkeit*, die angibt, wie leicht Nutzende den Umgang mit einer Software lernen können. Eine weitere Definition ist die *Lernfähigkeit,* die angibt, in welchem Maße ein System seine Umgebung automatisch kennenlernen kann, um sich beispielsweise durch maschinelles Lernen selbst zu konfigurieren oder zu optimieren.

Viele Definitionen überschneiden sich. Nehmen Sie beispielsweise *Verfügbarkeit* (*Availability*) und *Verlässlichkeit* (*Reliability*), die sich in fast allen Punkten zu überschneiden scheinen. Jetzt betrachten Sie sich aber beispielsweise einmal das Internetprotokoll UDP, auf dem TCP basiert. UDP ist über IP verfügbar, aber nicht verlässlich: Die Pakete können in falscher Reihenfolge ankommen, und der Empfänger muss fehlende Pakete eventuell erneut anfordern.

Es gibt keine vollständige Liste mit Standards, die diese Kategorien definieren. Die *International Organization for Standards* (ISO) veröffentlicht eine nach Eigenschaften geordnete Liste (*https://oreil.ly/SKc_Y*), die sich in großen Teilen mit den von uns gezeigten Eigenschaften deckt. In erster Linie definiert sie aber eine unvollständige Liste mit Kategorien. Unten sehen Sie einige der ISO-Definitionen – an moderne Begriffe angepasst und um Kategorien ergänzt, um für die heutige Zeit nutzbar zu sein:

Performance-Effizienz
: Maßeinheit der Performance im Verhältnis zu den unter bekannten Bedingungen eingesetzten Ressourcen. Hierzu gehören *zeitbasiertes Verhalten* (Messung der Antwort- und Verarbeitungszeiten und/oder der Durchsatzraten), *Ressourcennutzung* (Menge und Typ der verwendeten Ressourcen) und *Kapazität* (Grad, zu dem festgelegte Maximalwerte überschritten werden).

Kompatibilität
: Grad, zu dem ein Produkt, System oder eine Komponente Informationen mit anderen Produkten, Systemen oder Komponenten austauschen und/oder die von ihm geforderten Funktionen auf Basis der gleichen Hardware- oder Softwareumgebung ausführen kann. Hierzu gehören *Koexistenz* (kann die geforderten Funktionen effizient ausführen, während Umgebung und Ressourcen mit anderen Produkten gemeinsam genutzt werden) und *Interoperabilität* (Grad, zu dem zwei oder mehr Systeme Informationen austauschen und verwenden können).

Usability/Benutzbarkeit
: Nutzer können das System für den geplanten Zweck effektiv, effizient und zufriedenstellend verwenden. Hierzu gehören die *Erkennung der Angemessenheit* (*Appropriateness Recognizability*, Nutzer können erkennen, ob die Software für Ihre Bedürfnisse angemessen ist), *Lernbarkeit* (wie leicht Nutzer den Umgang mit der Software lernen können), *Schutz vor Fehlern der Benutzer* und *Zugänglichkeit* (Verfügbarkeit der Software für einen möglichst großen Personenkreis mit der größtmöglichen Anzahl an Eigenschaften und Fähigkeiten).

Verlässlichkeit

Der Grad, zu dem ein System unter bestimmten Bedingungen für eine bestimmte Zeit funktioniert. Zu dieser Eigenschaft gehören Unterkategorien wie *Reife* (ist die Software unter normalen Bedingungen verlässlich genug), *Verfügbarkeit* (die Software funktioniert und ist benutzbar), *Fehlertoleranz* (arbeitet die Software auch bei Hard- oder Softwareproblemen verlässlich) und *Wiederherstellbarkeit* (Kann die Software sich von Fehlern erholen, indem betroffene Daten und der gewünschte Systemzustand wiederhergestellt werden?).

Sicherheit

Grad, zu dem die Software Informationen und Daten schützt, sodass Personen oder andere Produkte bzw. Systeme den für ihre Autorisierungsstufe angemessenen Datenzugriff erhalten. Diese Art der Eigenschaften enthält *Vertrauen* (autorisierte Personen haben nur auf die für sie vorgesehenen Daten Zugriff), *Integrität* (die Software verhindert unautorisierten Zugriff auf oder die Veränderung von Software und/oder Daten), *Unleugbarkeit* (es kann bewiesen werden, dass Aktionen oder Ereignisse stattgefunden haben), *Überwachbarkeit* (Können die Aktionen eines Nutzers nachverfolgt werden?) und *Authentizität* (Beweis der Identität eines Nutzers).

Wartbarkeit

Gibt an, wie effektiv und effizient Entwicklerinnen und Entwickler die Software verändern können, um sie zu verbessern, zu korrigieren oder an veränderte Umgebungen und/oder Anforderungen anzupassen. Zu dieser Eigenschaft gehören *Modularität* (der Grad, zu dem die Software aus eigenständigen Komponenten besteht), *Wiederverwendbarkeit* (Grad, zu dem Entwickler bestimmte Bestandteile in mehr als einem System oder für die Erstellung weiterer Bausteine verwenden können), *Analysierbarkeit* (wie einfach Entwickler konkrete Kennzahlen über die Software gewinnen können), *Modifizierbarkeit* (Grad, zu dem Entwickler die Software verändern können, ohne Fehler zu verursachen oder die vorhandene Produktqualität einzuschränken) und *Testbarkeit* (wie einfach Entwickler und andere die Software testen können).

Portierbarkeit

Grad, zu dem Entwicklerinnen und Entwickler ein System, Produkt oder eine Komponente von einer Hardware, Software oder einer sonstigen betriebs- oder verwendungsrelevanten Umgebung auf eine andere übertragen können. Zu dieser Eigenschaft gehören die Unterkategorien der *Anpassbarkeit* (Können Entwickler die Software effektiv und effizient für andere oder weiterentwickelte Hardware-, Software- oder andere betriebs- oder verwendungsrelevante Umgebungen anpassen?), *Installierbarkeit* (Kann die Software in einer bestimmten Umgebung installiert oder deinstalliert werden?) und *Ersetzbarkeit* (wie einfach Entwickler die Funktionalität durch andere Software ersetzen können).

Der letzte Punkt der ISO-Liste bezieht sich auf die funktionalen Aspekte der Software:

Funktionale Eignung

Diese Eigenschaft steht für den Grad, zu dem ein Produkt oder System Funktionen bereitstellt, die explizite oder implizite Bedürfnisse erfüllen, sofern sie unter den angege-

benen Bedingungen verwendet werden. Diese Eigenschaft besteht aus folgenden Unterkategorien:

- Funktionale Vollständigkeit

  Grad, zu dem ein Satz an Funktionen alle angegebenen Aufgaben und Benutzeranforderungen erfüllt.

- Funktionale Korrektheit

  Grad, zu dem ein Produkt oder System die korrekten Ergebnisse mit der nötigen Präzision bereitstellt.

- Funktionale Angemessenheit

  Grad, zu dem Funktionen die Erfüllung der angegebenen Aufgaben und Ziele ermöglichen.

Wir finden aber nicht, dass die funktionale Eignung in diese Liste gehört. Sie ist keine architektonische Eigenschaft, sondern eher der Beweggrund für die Erstellung der Software. Hier zeigt sich, wie sich die Denkweise über die Beziehung zwischen architektonischen Eigenschaften und Problembereich (*Problem Domain*) weiterentwickelt hat. Diese Entwicklung behandeln wir in Kapitel 7.

Die vielen Widersprüche in der Softwarearchitektur

Eine beständige Quelle der Frustration besteht für Architekten im Fehlen klarer Definitionen für so vieles – inklusive der Softwarearchitektur selbst! Das führt dazu, dass Unternehmen eigene Begriffe für alltägliche Dinge definieren. Das führt wiederum zu einer branchenweiten Verwirrung, weil Architekten entweder intransparente Begriffe oder, noch schlimmer, die gleichen Begriffe für völlig unterschiedliche Dinge verwenden.

So sehr wir es uns wünschen – wir können der Welt der Softwareentwicklung hier keine Standardnomenklatur aufzwingen. Um aber Verwirrung durch Begrifflichkeiten zu vermeiden, empfehlen wir die Verwendung des Domain-Driven Design (dem wir auch selbst folgen), um zumindest im Kollegenkreis eine allgemeingültige und gemeinsame Sprachregelung zu verwenden, um Missverständnisse durch die widersprüchliche Verwendung von Fachbegriffen zu vermeiden.

Kompromisse und am wenigsten schlechte Architektur

Wir haben weiter oben geschrieben, dass Architekten nur die architektonischen Eigenschaften unterstützen sollten, die für den Erfolg des Systems kritisch oder wichtig sind. Systeme können aus verschiedenen Gründen nur ein paar der aufgeführten Eigenschaften abdecken. Für jede unterstützte Eigenschaft sind ein gewisser Designaufwand durch den Architekten, Aufwände durch die Entwicklung beim Implementieren und Warten und möglicherweise eine strukturelle Unterstützung nötig.

Das zweite Problem liegt darin, dass jede architektonische Eigenschaft oftmals andere Eigenschaften *und* die Problemdomäne beeinflusst. So sehr wir das gern anders sehen würden: Je-

des Designelement interagiert mit allen anderen. Will ein Architekt beispielsweise die *Sicherheit* erhöhen, leidet darunter ganz bestimmt die *Performance*: Die Applikation muss mehr Daten zeitnah verschlüsseln, Umwege gehen, um Geheimnisse zu wahren, und andere Aktivitäten ausführen, was sich mit großer Wahrscheinlichkeit negativ auf die Performance auswirkt.

Piloten haben oft Schwierigkeiten, einen Helikopter fliegen zu lernen, weil jede Hand und jeder Fuß einzeln kontrolliert werden muss und Änderungen an einer Stelle Einfluss auf die anderen haben. Das Fliegen eines Helikopters ist eine Balanceübung, die gut beschreibt, wie die Vor- und Nachteile bei der Entscheidung über architektonische Eigenschaften gegeneinander und gegen die Problemdomäne abgewogen werden müssen. Wie unsere Hubschrauberpiloten müssen Architekten lernen, mit untereinander verbundenen Elementen zu jonglieren.

Das dritte Problem ist (wie schon erwähnt) das Fehlen von Standarddefinitionen für architektonische Eigenschaften, weshalb die Organisation mit Mehrdeutigkeiten zu kämpfen hat. Die Branche wird es zwar nie schaffen, eine abschließende Liste mit architektonischen Eigenschaften zu erstellen (weil fortlaufend neue die Bühne betreten), aber jede Organisation kann ihre eigene Liste (als *omnipräsente Sprache*) mit objektiven Definitionen erschaffen.

Schließlich wächst nicht nur die Anzahl architektonischer Eigenschaften immer weiter, auch die Menge an *Kategorien* ist im letzten Jahrzehnt angestiegen. So haben sich beispielsweise Architekten vor ein paar Jahrzehnten nur wenig Gedanken um betriebsrelevante Aspekte gemacht, weil diese als eigenständige »Black Box« angesehen wurden. Mit der zunehmenden Verbreitung von Architekturen wie Microservices müssen nun aber Architekten und Operations-Mitarbeitende intensiver und häufiger zusammenarbeiten. Durch die komplexer werdende Softwarearchitektur wird diese immer mehr mit anderen Teilen der Organisation verflochten.

Daher kommen Architekten nur selten in Situationen, in denen sie ein System entwickeln und jede einzelne architektonische Eigenschaft maximieren können. Viel öfter kommen die Entscheidungen in Form von Kompromissen zwischen verschiedenen miteinander konkurrierenden Überlegungen.

Strebe niemals nach der *besten* Architektur, sondern eher nach der am *wenigsten schlechten*.

Zu viele architektonische Eigenschaften führen zu Allgemeinlösungen, die versuchen, alle Probleme auf einmal zu lösen. Diese Architekturen funktionieren allerdings nur selten, weil das Design schnell zu umständlich wird.

Ein Architekt sollte nach einer möglichst iterativen Designarchitektur streben. Je einfacher das Umsetzen von Änderungen an der Architektur ist, desto weniger müssen Sie darauf bauen, dass gleich beim ersten Versuch alles klappt. Eine der wichtigsten Lektionen der agilen Softwareentwicklung ist der Wert der Iteration. Das ist für alle Ebenen der Softwareentwicklung gültig – inklusive der Architektur.

KAPITEL 5

Architektonische Eigenschaften ermitteln

Um eine Architektur zu erstellen oder die Gültigkeit einer bereits vorhandenen Architektur zu bestimmen, müssen Architektinnen und Architekten zwei Dinge analysieren: die architektonischen Eigenschaften und den Problembereich. Wie Sie in Kapitel 4 gelernt haben, müssen Sie, um die korrekten architektonischen Eigenschaften (Fähigkeiten) für ein bestimmtes Problem oder eine Applikation zu ermitteln, nicht nur den jeweiligen Problembereich verstehen, sondern auch mit den Stakeholdern für diesen Bereich zusammenarbeiten, um herauszufinden, was aus Domänensicht wirklich wichtig ist.

Ein Architekt findet an mindestens drei Stellen heraus, welche architektonischen Eigenschaften für ein Projekt erforderlich sind: in den domänenspezifischen Problemen, den Projektanforderungen und mithilfe seines impliziten Domänenwissens. Wir haben festgestellt, dass neben den generischen impliziten architektonischen Eigenschaften, wie zum Beispiel Sicherheit und Modularität, manche Domänen ebenfalls implizite architektonische Eigenschaften mitbringen. So sollte sich beispielsweise ein Architekt, der an einer Medizinsoftware arbeitet, die Diagnosegeräte ausliest, immer der Wichtigkeit der Datenintegrität und der potenziellen Konsequenzen beim Verlieren von Nachrichten bewusst sein. Architekten, die in dieser Domäne arbeiten, internalisieren die Datenintegrität, daher ist diese implizit in jeder von ihnen entworfenen Lösung enthalten.

Die impliziten Eigenschaften haben wir bereits besprochen. Daher gehen wir hier nur auf die anderen beiden ein.

Architektonische Eigenschaften aus domänenspezifischen Anforderungen ableiten

Die aus Domänensicht wichtigsten architektonischen Eigenschaften erhält man, indem man den Hauptentscheidern einer Domäne zuhört und mit ihnen zusammenarbeitet. Und obwohl dieses Vorgehen auf den ersten Blick recht leicht erscheint, besteht das Problem dabei oft darin, dass Architekten und Domänenentscheider unterschiedliche Sprachen sprechen. Architekten reden von Skalierbarkeit, Interoperabilität, Fehlertoleranz, Lernfähigkeit und Verfügbarkeit. Stakeholder sprechen von Fusionen, Akquisitionen, Benutzerzufriedenheit, *Time-to-Market* und Wettbewerbsvorteilen. Dadurch kommt es zu einem »Lost in

Translation«-Problem, bei dem Architekt und Entscheider sich gegenseitig missverstehen. Architekten haben keine Ahnung, wie sie eine Architektur erstellen sollen, die für Kundenzufriedenheit sorgt, und die Entscheider verstehen nicht, warum es so wichtig ist, über Verfügbarkeit, Interoperabilität, Lernbarkeit und Fehlertoleranz in der Applikation zu sprechen.

Zum Glück ist es jedoch möglich, die domänenspezifischen Bedürfnisse in architektonische Eigenschaften zu übersetzen. Tabelle 5.1 zeigt einige der häufigsten domänenspezifischen Bedürfnisse und die entsprechenden architektonischen Eigenschaften (Fähigkeiten). Wenn ein Architekt die zentralen Domänenziele versteht, kann er diese in Fähigkeiten übersetzen, die dann die Basis für korrekte und begründbare architektonische Entscheidungen bilden. Ist das domänenspezifische Bedürfnis *Time-to-Market* beispielsweise wichtiger als Skalierbarkeit, oder sollte sich der Architekt auf Fehlertoleranz, Sicherheit oder Performance konzentrieren? Vielleicht müssen für das System all diese Eigenschaften kombiniert werden.

Tabelle 5.1: Übersetzung domänenspezifischer Bedürfnisse in architektonische Eigenschaften

Bedürfnisse	Architektonische Eigenschaften
Fusionen und Akquisitionen	Interoperabilität, Skalierbarkeit, Anpassungsfähigkeit, Erweiterbarkeit
Zeit bis zur Markteinführung (Time-to-Market)	Agilität, Testbarkeit, Bereitstellungsfähigkeit (*Deployability*)
Benutzerzufriedenheit	Leistung, Verfügbarkeit, Fehlertoleranz, Testbarkeit, Bereitstellungsfähigkeit, Agilität, Sicherheit
Wettbewerbsvorteil	Agilität, Testbarkeit, Bereitstellungsfähigkeit, Skalierbarkeit, Verfügbarkeit, Fehlertoleranz
Zeit und Budget	Einfachheit, Machbarkeit

Zusammengesetzte architektonische Eigenschaften

Beim Übersetzen von domänenspezifischen Bedürfnissen in architektonische Eigenschaften wird man leicht dazu verleitet, von falschen Äquivalenzen auszugehen – zum Beispiel *Agilität* einfach mit *Time-to-Market* gleichzusetzen. Agilität ist ein Beispiel für eine *zusammengesetzte* architektonische Eigenschaft – eine, die keine einzelne objektive Definition besitzt, sondern aus anderen messbaren Elementen kombiniert ist. Es gibt keinen Messwert für Agilität, daher müssen Architekten fragen, woraus Agilität entsteht. Sie enthält Dinge wie *Deploybarkeit*, *Modularität* und *Testbarkeit*, die jeweils alle messbar sind.

Ein verbreitetes Antipattern entsteht, wenn sich Architekten zu sehr auf nur einen Teil einer zusammengesetzten Eigenschaft konzentrieren – oft aus Bequemlichkeit. Das ist in etwa so, als würde man vergessen, Mehl in den Kuchenteig zu rühren. Ein Domänenentscheider könnte beispielsweise sagen: »Aufgrund der regulatorischen Anforderungen ist es

absolut obligatorisch, dass wir die Preisfestsetzung des Tagesendgeschäfts rechtzeitig abschließen.«

Ein ineffektiver Architekt würde sich dann möglicherweise nur auf die Leistung konzentrieren, weil dies der wichtigste Punkt für dieses domänenspezifische Bedürfnis zu sein scheint. Aus verschiedenen Gründen wird dieser Ansatz jedoch scheitern:

- Die Geschwindigkeit eines Systems spielt keine Rolle, wenn es bei Bedarf nicht verfügbar ist.
- Wenn die Domäne wächst und mehr Umsatz erzeugt, muss das System skalieren können, um die Preisfestsetzung des Tagesendgeschäfts rechtzeitig beenden zu können.
- Ein System muss nicht nur zur Verfügung stehen, sondern auch verlässlich sein, damit es nicht abstürzt, wenn die Festsetzung der Tagesendpreise durchgeführt wird.
- Was passiert, wenn die Festsetzung der Tagesendpreise zu 85 % vollständig ist, dann aber das System abstürzt? Es muss sich wiederherstellen und mit dem Festsetzen der Preise dort fortfahren können, wo es abgebrochen wurde.
- Abschließend kann ein System zwar schnell sein, aber werden die Preise wirklich korrekt berechnet?

Zusätzlich zur Leistungsfähigkeit muss der Architekt also auch auf Verfügbarkeit, Skalierbarkeit, Verlässlichkeit, Wiederherstellbarkeit und Auditierbarkeit achten.

Viele Unternehmensziele beginnen als zusammengesetzte architektonische Eigenschaften. Es ist Teil des Jobs eines Architekten, sie auseinanderzunehmen und den sich so ergebenden Eigenschaften objektive Definitionen zu verpassen. (Wir werden in Kapitel 6 sehen, wie wichtig das ist.)

Architektonische Eigenschaften ableiten

Die meisten architektonischen Eigenschaften entstammen expliziten Aussagen in den Anforderungsdokumenten. So werden in der Domäne oder den domänenspezifischen Bedürfnissen häufig eine bestimmte Nutzerzahl und eine Größe benannt. Andere Eigenschaften stammen aus bereits vorhandenem Domänenwissen der Architekten. (Dies ist übrigens einer der vielen Gründe, warum Fachkenntnis bzw. Domänenwissen für Architekten immer von Vorteil ist.)

Angenommen, Sie entwerfen als Architektin ein System, das bei der Immatrikulation von Studenten helfen soll. Um leichter rechnen zu können, gehen wir davon aus, dass die Uni 10 Stunden Zeit für die Registrierung von 1.000 Studentinnen und Studenten hat. Sollten Sie das System mit einer gleichbleibenden Skalierung entwerfen, also implizit davon ausgehen, dass sich die Studenten gleichmäßig über den gesamten Zeitraum verteilen? Oder sollte er ein System entwerfen, dass alle 1.000 Studenten gleichzeitig handhaben kann, weil er weiß, dass diese bestimmte Angewohnheiten und Vorlieben haben?

Jeder, der weiß, wie gerne Studentinnen und Studenten Dinge bis auf den letzten Moment aufschieben, kennt die Antwort auf diese Frage! Nur selten tauchen solche Details in Anforderungsdokumenten auf, und trotzdem beeinflussen sie die Designentscheidungen.

Der Ursprung der Architektur-Katas

Vor über zehn Jahren entwickelte Ted Neward, ein bekannter Architekt, eine Reihe von *Architektur-Katas*, eine praktische Methode, mit der angehende Architekten das Ableiten architektonischer Eigenschaften aus domänenspezifischen Beschreibungen üben konnten. Aus Japan und den Kampfkünsten stammend, ist ein *Kata* eine individuelle Trainingsübung, bei der die Betonung auf ordentlicher Form und Technik liegt.

> Wie wird man ein großartiger Designer? Großartige Designer designen natürlich.
>
> *– Fred Brooks*

Große architektonische Projekte brauchen Zeit, und Architekten designen im Laufe ihrer Karriere durchaus ein halbes Dutzend Systeme. Wie sollen wir dann also großartige Architekten werden? Wichtig ist, angehenden Architektinnen und Architekten die Gelegenheit zum Üben zu geben. Ted erstellte die erste Website mit Architektur-Katas, um angehenden Architekten ein Curriculum zu bieten. Diese Website wurde von den Autoren dieses Buchs, Neal und Mark, übernommen, aktualisiert und auf der zum Buch gehörenden Website *https://fundamentalsofsoftwarearchitecture.com* eingebaut. Entsprechend ihrem ursprünglichen Zweck bieten die Architektur-Katas angehenden Architekten einen nützlichen Ansatz zum Experimentieren.

Mit Katas arbeiten

Die grundlegende Prämisse ist, dass jede Kata-Übung ein Problem in der Domänensprache, einen Satz Anforderungen und zusätzlichen Kontext bereitstellt (Dinge, die eventuell nicht in den Anforderungen zu finden sind, aber das Design beeinflussen können). Kleine Teams arbeiten für eine festgelegte Zeit an einem Design (bestehend aus einer Analyse der architektonischen Eigenschaften und Diagrammen). Dann stellen sich die Gruppen gegenseitig ihre Ergebnisse vor und stimmen darüber ab, wer die beste Architektur erstellt hat.

Jedes Kata besteht aus bestimmten Abschnitten:

Beschreibung
: Das allgemeine domänenspezifische Problem, das das System zu lösen versucht.

Nutzer
: Die erwartete Anzahl und die erwarteten Arten der Benutzenden des Systems.

Anforderungen
: Domäne und domänenspezifische Anforderungen, wie ein Architekt sie von Domänennutzern und -experten erwarten kann.

Zusätzlicher Kontext
: Die Autoren haben das Kata-Format auf der weiter oben erwähnten Website um einen zusätzlichen Kontext erweitert, der weitere Überlegungen enthält und die Übungen noch realistischer macht.

Wir wollen die Leserinnen und Leser dazu ermuntern, die Website zu nutzen, um ihre eigenen Kata-Übungen durchzuführen. Jeder kann Gastgeber eines informellen Treffens zum Beispiel in der Mittagspause (eines »Brown-Bag Lunch«) sein. Hierbei löst ein Team aus

angehenden Architekten ein Problem, und ein erfahrener Architekt wertet das Design und die Kompromissanalyse aus. Das kann entweder direkt vor Ort oder als kurze Analyse im Nachhinein geschehen, und das Team erhält Feedback zu übersehenen Vor- und Nachteilen und alternativen Designs. Das Design wird nicht besonders ausgefeilt sein, weil die Übung zeitlich begrenzt ist.

Als Nächstes werden wir ein Architektur-Kata nutzen, um zu zeigen, wie Architekten architektonische Eigenschaften aus den Anforderungen ableiten. Willkommen zum Kata *Silicon Sandwiches*.

Kata: Silicon Sandwiches

Beschreibung
: Ein landesweiter Sandwich-Hersteller möchte die Möglichkeit schaffen, Bestellungen online aufzugeben (zusätzlich zur bereits vorhandenen telefonischen Bestellung).

Nutzer und Nutzerinnen
: Tausende, später vielleicht Millionen.

Anforderungen

- Nutzer geben ihre Bestellung auf. Bietet der Shop einen Lieferservice an, können sie zwischen Abholung und Lieferung wählen.
- Abholkunden erhalten eine Uhrzeit, zu der sie ihr Sandwich abholen können, sowie eine Wegbeschreibung zur Filiale (die mit verschiedenen Kartendiensten zusammenarbeiten soll, die Verkehrsinformationen bereitstellen).
- Wenn die Filiale einen Lieferdienst anbietet, soll der Fahrer mit dem Sandwich zum Nutzer geschickt werden.
- Zugänglichkeit für Mobilgeräte.
- Landesweite tägliche Werbeaktionen und Sonderangebote.
- Lokale tägliche Werbeaktionen und Sonderangebote.
- Zahlungsmöglichkeiten: online, im Shop und bei Lieferung.

Zusätzlicher Kontext

- Die einzelnen Sandwich-Läden basieren auf einem Franchise-Modell und haben jeweils verschiedene Eigentümer.
- Das Mutterunternehmen hat für die nächste Zukunft Pläne, in weitere Länder zu expandieren.
- Ein Unternehmensziel besteht darin, billige Arbeitskräfte anzustellen, um den Profit zu maximieren.

Wie würden Sie anhand dieses Szenarios die architektonischen Eigenschaften ableiten? Jeder Teil der Anforderungen kann einen oder mehrere Teile der Architektur beeinflussen (und viele eben auch nicht). Der Architekt entwirft hier nicht das gesamte System – es ist immer noch erheblicher Aufwand nötig, den Code zu erstellen, um das Gesamtproblem zu