IN DIESEM KAPITEL

Die Leistungen von Software-Architektur

Die Rolle des Software-Architekten

Kapitel 1
Wie wir Software-Systeme bauen

Software wird schnell sehr komplex. Sogar wenn die Aufgabe am Anfang einfach aussieht, wird sie später schnell größer, weil im Produktivbetrieb Sonderfälle auftauchen oder die Benutzer von der laufenden Anwendung inspiriert werden, weitere Arbeitsschritte zu digitalisieren. In vielen Fällen ist die Aufgabenstellung aber schon zu Beginn komplex. So oder so brauchen wir einen strategischen Plan für die Entwicklung.

Software-Architektur beschäftigt sich damit, solche strategischen Pläne bereitzustellen und über die Lebensdauer der Software weiterzuentwickeln. In diesem ersten Kapitel schauen wir darauf, was sie dazu alles leisten muss.

Allerdings soll es dabei in den Beispielen nicht schon zu technisch zugehen, sonst müssten Sie ja dauernd vorausblättern in die späteren Kapitel. Stattdessen nehmen wir Beispiele aus einer Metapher: Die Entwicklung einer Software-Anwendung ist in vielen Aspekten ähnlich dem Bau einer Schule. So wie die meiste Software ist eine Schule vor allem ein Funktionsbau. Das, was dort geleistet werden muss, um eine brauchbare Schule zu erhalten, muss in analoger Form auch für Software geleistet werden. Diese Analogie ist tatsächlich nicht ganz neu: Die Patterns-Community sieht sich bekanntlich in der Tradition des Architekten Christopher Alexander und seiner Ideen, wie man zu guten Gebäuden kommt [7].

Aufgaben und Ziele von Architektur

Wir wollen also eine Schule bauen und dafür einen großen Gesamtplan entwerfen. Dieser Plan muss natürlich viele ganz verschiedene Dinge festlegen, von der Anzahl und Größe der Klassenräume über die Lage und Größe des Lehrerzimmers bis zur Einhaltung von Vorgaben zur Beleuchtung. Auch Details wie die Lage und Anzahl von Steckdosen muss der Architekt schon mitbedenken, denn die hängen oft eng mit der geplanten Raumnutzung zusammen. Es wird also ein sehr vielschichtiger Plan.

So wollen wir arbeiten – funktionale Anforderungen

Im Zentrum der Planung stehen natürlich die *funktionalen Anforderungen* (engl. *functional requirements*): In der Schule soll vor allem effektiv unterrichtet und gelernt werden. Alles, was dazu notwendig ist, muss in der Schule vorhanden sein. Alle Anforderungen aufzulisten, ist allerdings gar nicht so einfach: Zunächst braucht man Klassenräume mit genug Platz je Schüler, klar. Dann braucht man Flure, die die Klassenräume verbinden, klar. Dann sanitäre Einrichtungen, Lehrerzimmer, einen Raum für Kopierer, klar. Und natürlich auch einen Server-Raum und überall Schächte für Netzwerkkabel. Vielleicht Freiarbeitsflächen für spezielle pädagogische Konzepte. Eine Cafeteria für Nachmittagsunterricht und Ganztagsangebote. Labore für Physik und Chemie. Und und und.

In der Software-Architektur ist eine möglichst vollständige Übersicht über die funktionalen Anforderungen deshalb wichtig, weil sie ja darüber bestimmten, welche Aufgaben die Software technisch lösen muss. Und diese Aufgaben werden auf verschiedene Komponenten oder Module der Software verteilt, sodass indirekt auch die Struktur mitgeprägt wird.

Der Teilbereich des *Requirement Engineering* kümmert sich um solche Fragen. Es kommt darauf an, zu Beginn des Projekts wirklich gründlich nach Anforderungen zu suchen und in den einzelnen Iterationen während des Projekts schnell auf neue und geänderte Anforderungen aufmerksam zu werden. Einige mögliche Quellen für funktionale Anforderungen sind:

- ✔ *Use-Cases*: In welchen Situationen und mit welchem Zweck wollen Benutzer mit dem System interagieren? Was geben sie ein? Welche Reaktionen erwarten sie?
- ✔ Anwendungsgebiet (*Application-Domain*): Welche Aufgaben müssen typischerweise von Fachleuten in dem Bereich gelöst werden, in dem die Software eingesetzt werden soll?
- ✔ Unternehmerische Ziele (*Business-Drivers*): Warum schreiben wir die Software überhaupt? Welche wirtschaftlichen Vorteile soll sie bringen?

Im Fall der Schule wären die entsprechenden Fragen: Wie wird unterrichtet? Welche Aufgaben haben Lehrer und Schüler im Schulbetrieb? Wie gehen wir mit den steigenden Schülerzahlen nach dem Zuzug vieler Familien um?

Richtig rechnen reicht nicht – nicht-funktionale Anforderungen

Wenn man ein Gebäude einen »funktionaler Zweckbau« nennt, ist damit gemeint: Ist schon okay, aber keiner geht gern hin, und hässlich ist es obendrein. Es reicht nicht, eine Schule zu bauen, in der unterrichtet werden kann, wenn die Randbedingungen nicht stimmen: lange Wege zwischen Räumen, hässliche Betonwände, zugige Fenster, häufiger Ausfall der Heizungsanlage, hohe Betriebskosten …Mit so einer Schule kann niemand etwas anfangen.

Mit der Software ist es genauso: Über die funktionalen Anforderungen hinaus gibt es *nicht-funktionale Anforderungen* (engl. *Non-functional Requirements*), die die Software erreichen muss. Weil ihre Bezeichnungen im Englischen fast immer auf »-ility« enden, heißen sie

salopp auch »*die -ilities*«. Wenn es mehr darauf ankommt, die Software selbst zu beschreiben, nutzt man den Begriff *nicht-funktionale Eigenschaften* (*Non-functional Properties*). Teilweise kann man die nicht-funktionalen Anforderungen auch als *Qualitätsmerkmale* (engl. *quality attributes*) sehen, weil sie etwas darüber aussagen, wie gut die Software die funktionalen Anforderungen umsetzt. Ein alternativer Begriff ist *Architectural Characteristics*, denn sie werden hauptsächlich durch die gewählte Software-Architektur bestimmt. Damit ist auch klar: Ein Software-Architekt sollte die »ilities« immer genau im Blick behalten.

Eine auch nur halbwegs vollständige Liste anzugeben, dafür reicht der Platz nicht, und es wäre außerdem recht leserunfreundlich, weil trocken. In der späteren Beschreibung der Architektur-Ansätze werden die jeweils relevanten nicht-funktionalen Eigenschaften besprochen. Deshalb hier nur eine grobe Gruppierung mit einigen Beispielen:

- ✔ Laufzeit-Anforderungen: *Availability*: Welchen Anteil der Zeit ist das System verfügbar? Können Updates ohne Unterbrechung eingespielt werden? *Responsiveness*: Wie lange muss ein Benutzer auf eine Reaktion warten? *Robustness*: Wie gut geht das System mit speziellen Ereignissen wie unerwarteten Eingaben oder Netzwerkausfällen um? *Scalability*: Wie leicht kann die Anwendung für sehr viel mehr Zugriffe ausgelegt werden?

- ✔ Strukturelle Anforderungen: *Maintainability*: Wie leicht können im Nachhinein Anpassungen vorgenommen werden? *Extensibility*: Wie leicht können neue Funktionen ergänzt werden? *Testability*: Wie einfach können einzelne Module oder Komponenten oder das Gesamtsystem getestet werden? *Reusabilty*: Können Teile des Systems in anderen Projekten wiederverwendet werden?

- ✔ Querschnitts-Anforderungen: *Usability*: Wie einfach ist das Erlernen und wie sicher und gern arbeiten die Benutzer mit dem System? *Privacy*: Wie weitgehend können Vorgänge und Daten verborgen werden? *Legal Requirements*: Beachtet die Software alle relevanten rechtlichen Regelungen? *Traceability*: Findet man die funktionalen Anforderungen in der Software-Struktur wieder?

Nicht-funktionale Eigenschaften sind deshalb besonders wichtig, weil sie häufig missionskritisch sind und über den Projekterfolg entscheiden: Eine konzernweite Anwendung, die in Tests perfekt funktioniert, aber nicht für die 20.000 weltweit verteilten Benutzer skalierbar ist, ist wertlos. Gleichzeitig bestimmen sie stark die Architektur und Struktur mit, sodass man sie genauso sorgfältig sammeln muss wie die funktionalen Anforderungen. Weil jede Anforderung aber einen gewissen Mehraufwand mit sich bringt und sich die verschiedenen Anforderungen manchmal auch widersprechen, muss der Architekt außerdem sinnvoll priorisieren.

Viele Köche …– Stakeholder

Bei einer Schule wollen alle mitreden: der Kreis wegen der Baukosten, die Stadt wegen der Bebauungspläne, der Bürgermeister wegen des Prestiges, die Schulleitung und Lehrer wegen der Pädagogik, der Hausmeister wegen des Pausenverkaufs …Die Liste ist endlos. Am Ende muss es aber eine einzige Schule für alle geben.

Bei der Software-Erstellung ist es nicht anders: Viele verschiedene Interessengruppen (engl. *stakeholder*) wollen Einfluss nehmen und die Architektur muss die unterschiedlichen Anforderungen und Wünsche priorisieren und gegeneinander abwägen. Wie findet man aber die relevanten Stakeholder und ihre Anforderungen? Hier sind einige Ansätze:

- ✔ Verschiedene *Benutzergruppen* (engl. *user groups*) haben jeweils eigene Use-Cases und Anforderungen an das System. Mit einer Liste der Benutzergruppen zu starten, ist sicher keine schlechte Idee.
- ✔ Weil Beschreibungen von Benutzergruppen oft abstrakt bleiben, übersieht man leicht einige ihrer Anforderungen. Eine *Persona* dagegen ist ein prototypischer (engl. *archetypical*) Benutzer, den man sehr detailliert und individualisiert beschreibt. Damit werden implizite Erwartungen besser vorstellbar. Verschiedene Personas, die klassisch zu einer Benutzergruppe gehören würden, werden unterscheidbar.
- ✔ Eine andere Möglichkeit der Konkretisierung sind *Key Users*. Man wählt pro Benutzergruppe einige Mitarbeiter aus, die etwa besondere Fachkompetenzen haben oder gern strategisch denken und handeln.
- ✔ Ein Blick lohnt auch auf die mit der Umsetzung des Projekts beschäftigen Personen: Die Entwicklungsteams und das Projektmanagement haben selbst auch Anforderungen, zum Beispiel zuverlässige und motivierende Fortschritte, interessante Aufgaben oder eine verträgliche Arbeitslast.
- ✔ Die Kunden des Unternehmens, auch wenn sie das System nicht selbst bedienen, sind von seiner Leistung oft indirekt abhängig, zum Beispiel wenn Bestellprozesse mit der Software anders ablaufen oder neue Funktionen bieten.
- ✔ Das allgemeine Management und die Unternehmensführung haben die wirtschaftliche Seite des Projekts im Blick, also Budget, Kosteneinsparungen oder Gewinn durch die Software. Auch Besitzer und Anteilseigner des Unternehmens, die ja letztlich das Kapital für die Entwicklung bereitstellen, haben Interessen.
- ✔ Kooperationspartner in anderen Abteilungen oder anderen Unternehmen sind vielleicht auf Schnittstellen oder bestimmte Zugriffswege angewiesen.
- ✔ Bei einigen Projekten sind Behörden und andere Regulierungsorganisationen beteiligt. Beispielsweise stellt das Finanzamt ziemlich viele Anforderungen an die elektronische Buchhaltung und das Dokumentenmanagement.
- ✔ Mit einer *Empathy Map* kann man schließlich die impliziten und emotionalen Erwartungen von identifizierten Stakeholdern herausarbeiten, zum Beispiel um Usability-Faktoren zu identifizieren.

Sie sehen schon: Es ist nicht einfach, schon allein alle Stakeholder eines Projekts zu identifizieren. Am Ende ist es immer eine Person oder Personengruppe, die direkt oder indirekt vom Fortgang und Erfolg des Projekts und der Einführung der Software betroffen ist. Weil das potenziell sehr viele sind, ist das *Stakeholder Management* oder die *Stakeholder Analysis* eine ganz eigene und wichtige Aufgabe der Software-Architektur. Ein Projekt kann nicht

nur an Technik oder Budget scheitern, sondern auch daran, dass man einen wichtigen Stakeholder oder eines seiner fundamentalen Bedürfnisse übersehen hat.

Wer macht was wann wo? – Software-Strukturen

Wenn wir wissen, welche Aufgaben die Schule übernehmen soll und wer daran welches Interesse hat, geht es an die Gebäudeplanung: Welche Räume sehen wir vor? Wo wird kopiert? Wo werden Chemie-Versuche vorbereitet? Wo läuft der Pausenverkauf? Dabei geht es auch um dynamische Abläufe: Schaffen es die Lehrer in der 5-Minuten-Pause zum Lehrerzimmer und dann in die nächste Klasse? Wie kommen die Schüler trocken zum separaten naturwissenschaftlichen Trakt? Sind die Schulbusse sicher zu erreichen?

Auf der Software-Seite verteilen wir Teilaufgaben auf verschiedene Software-Elemente, wie etwa Komponenten, Module, Services, Klassen oder Methoden. Die gewählte Struktur drückt also ein bestimmtes Problem- und Lösungsverständnis aus, und schon in der Architektur muss man immer mitdenken, wie die Implementierung später aussieht.

Fast noch wichtiger ist aber, dass die Struktur mitbestimmt, welche nicht-funktionalen Anforderungen erfüllt werden. Beispielsweise erhöht sich die Maintainability – also die spätere Änderbarkeit der Software –, wenn wir inhaltlich eng zusammengehörige Funktionselemente in einem Modul zusammenfassen, während unabhängige Elemente auf verschiedene Module aufgeteilt werden. Weil Änderungen häufig nur die Details einzelner Anwendungsfunktionen betreffen, bleiben sie dann auf ein Modul beschränkt. Unter dem Slogan *Low Coupling, High Cohesion* ist diese Idee weit verbreitet.

Architektur-Patterns und *Architektur-Stile* fassen Erfahrungen mit bestimmten Aufteilungen von funktionalen Aspekten zusammen. Besonders heben sie hervor, in welcher Situation und unter welchen Voraussetzungen ein Pattern oder ein Stil sinnvoll angewendet werden kann und welche nicht-funktionalen Eigenschaften man dann erwarten darf. Der größte Teil dieses Buchs beschäftigt sich mit Patterns und Stilen, ihrem Aufbau und ihren Konsequenzen.

Wenn wir bei der Schule bleiben, wäre ein Pattern: Sieh auf jedem Stockwerk am Ende eines Ganges Toiletten-Räume vor. Eine Aufteilung von Funktionen wäre: Der Kopierraum ist neben dem Lehrerzimmer, aber wegen der Geräuschentwicklung mit einer Tür abgetrennt.

Die Software-Struktur hat aber nicht nur technische, sondern auch organisatorische Konsequenzen: Jedes Software-Element wird im Normalfall von einem einzelnen Team betreut. Damit gibt die Software-Struktur gleichzeitig die Arbeitsverteilung vor (engl. *work breakdown structure*). Ein umgekehrtes Phänomen, das Architekten kennen und beachten sollten, ist *Conway's Law*: Organisationen neigen dazu, die technische Funktionsverteilung entlang ihrer vorhandenen Organisationsstrukturen festzulegen. Teams werden also nicht für ein Projekt neu zusammengestellt, sondern die Software muss so strukturiert werden, dass die bestehenden Teams Teile übernehmen können. Diese politische Entscheidung nimmt natürlich keine Rücksicht auf die funktionalen und nicht-funktionalen Anforderungen, was zum Problem werden kann.

Von Silberkugeln und Sonnenseiten – Forces und Trade-offs

Wenn es die eine, ideale Schule gäbe, wären plötzlich viele Architekten arbeitslos. Dann würde man vielleicht 10–20 Typen für unterschiedliche Schülerzahlen einmal ausarbeiten und immer wieder genau so bauen. Aber weil die lokalen Gegebenheiten, vom Bauplatz bis zum pädagogischen Konzept, jeweils variieren, ist das einfach nicht möglich.

In der Software-Architektur gibt es genauso nie die eine richtige Lösung. Jede Entscheidung für eine bestimmte Struktur zieht beispielsweise unterschiedliche nicht-funktionale Eigenschaften nach sich, und je nach Priorisierung im konkreten Projekt kann die eine oder andere Entscheidung sinnvoller sein. Der Begriff *Forces* erfasst die verschiedenen Einflüsse, die die Entscheidung in die eine oder andere Richtung bewegen. Die *Trade-offs* sind die Abwägungen zwischen Einflüssen und Ergebnissen. Manchmal sind die Vorteile einer Lösung besonders stark, dann hat man Glück gehabt und braucht die anderen nur kurz anzudenken – aber das sollte man dann auch tun.

Zwei Schlagworte haben sich für diese Herausforderung eingebürgert:

- ✔ *No Silver Bullet*: Man kann nicht erwarten, dass durch irgendeinen Ansatz oder irgendeine Technologie plötzlich wirklich harte Probleme gelöst werden.
- ✔ *No-Free-Lunch*-Theorem: Wenn man auf einer Seite Gewinne macht, dann muss man auf der anderen Seite irgendetwas dafür bezahlen.

Ich habe für mich einen dritten Ausdruck gefunden, der mir mehr den Hintergrund ins Gedächtnis ruft:

- ✔ *Komplexität kann man nicht wegzaubern*: Wenn ein komplexes Problem gelöst werden muss, dann muss die Software so oder so Mechanismen enthalten, um es zu lösen. Für diese Mechanismen muss dann eben auf die eine oder andere Art bezahlen, egal für welche Lösung man sich am Ende entscheidet.

Für den Architekten ist es wichtig, die Komplexität und die Herausforderungen im Projekt aktiv zu suchen und früh zu identifizieren. Sonst kann er die Forces nicht in seine Überlegungen einbeziehen. Es funktioniert also langfristig nicht, mit zu viel Optimismus nur auf die Sonnenseiten, die offensichtlichen Aufgaben und die »Low-hanging Fruits« zu schauen.

Luftschlösser sind nutzlos – Architektur, Design und Implementierung

Eine Schule mit modernen Glasfronten, großzügigen Räumen, geschwungenen Treppen und einladender Aula sieht natürlich in den computer-gerenderten Ansichten einfach repräsentativ und attraktiv aus. Wenn auch noch die pädogischen Konzepte in flexiblen und die Kreativität anregenden Lernlandschaften umgesetzt sind und nebenbei die neueste Wärmedämmung niedrige Betriebskosten verspricht, dann steht dem Gewinn der Ausschreibung nichts mehr im Wege. Umso ärgerlicher, wenn nachher bei der Ausführung an allen Ecken und Enden auf Kosten der Qualität gespart werden muss und es bei Regen durch das undichte Deckenlicht ins Atrium nicht tröpfelt, sondern gießt.

Eine Software-Architektur bildet die Grundlage für Design und Implementierung. Sie gibt die großen Strukturen vor, innerhalb derer die Entwickler dann Schnittstellen und Klassen ausarbeiten. Wenn diese Strukturen aber gar nicht technisch umgesetzt werden können, weil der Architekt entscheidende Details übersehen hat, führt das zu Mehraufwand für Umarbeitungen. Außerdem ist der Code langfristig weniger wartbar, weil er nur lokal durchdachte Ad-hoc-Strukturen nutzt, die nicht zur dokumentierten Architektur passen. Es ist also die Aufgabe von Architektur, das Design und an missionskritischen Stellen, wenn beispielsweise viele Nutzer einen hohen Durchsatz an Anfragen erfordern, auch die Implementierung bis hin zur *Mechanical Sympathy* vorauszudenken und zu evaluieren.

Die Rolle des Architekten

Es gibt Architekturbüros, die bauen häufiger Schulen und werden immer angefragt, wenn irgendwo in der Gegend eine neue Schule zu bauen ist. Dann sprechen sie mit allen Beteiligen, bringen ihre eigene Erfahrung ein und legen nacheinander Konzepte, Entwürfe und Werkpläne vor. Auf dieser Grundlage werden dann die einzelnen Gewerke ausgeschrieben und am Ende die Schule gebaut.

Ein Software-Architekt muss natürlich dafür sorgen, dass die Architektur die vorher beschriebenen Aufgaben im Projekt erfüllen kann. Aber wie arbeitet er im Projekt?

Ich bin dann mal weg – die traditionelle Rolle

Im klassischen Wasserfallmodell ist die Rolle des Architekten ganz ähnlich wie beim Schulbau: Er ist dafür verantwortlich, …

- ✔ die Stakeholder zu identifizieren
- ✔ deren Anforderungen und Wünsche aufzunehmen und zu analysieren
- ✔ die umzusetzenden Anforderungen zu definieren und zu priorisieren
- ✔ eine Gesamtstruktur aus Komponenten und Modulen zu schaffen
- ✔ die Verbindungen zwischen den Komponenten vorzugeben
- ✔ die Erreichung der nicht-funktionalen Eigenschaften zu analysieren
- ✔ die Aufteilung der Arbeit auf Teams zu planen
- ✔ das Ergebnis dieser Überlegungen präzise festzuhalten und an die Entwickler für die weitere Arbeit zu übergeben.

In diesem Modell läuft also der Informationsfluss nur in eine Richtung: Vom Architekten zu den Entwicklern. Bestenfalls hat er bei seiner Tätigkeit schon zukünftige Änderungen oder Erweiterungen vorausgedacht. In jedem Fall ist aber die Arbeit des Architekten im Wesentlichen getan, sobald das Architektur-Dokument abgenommen ist.

Ich begleite euch bis zum Ziel – die moderne Rolle

Tatsächlich stimmt es nicht, dass der Schularchitekt so vorgeht wie im vorherigen Abschnitt angedeutet. Von einer befreundeten Architektin, die öfter Schulen baut, weiß ich, wie oft sie auf den Baustellen unterwegs ist und dort mit den Handwerkern die Details der Ausführung durchspricht und mit den Plänen abgleicht. Falls möglich führt sie auch noch Änderungen an den Plänen durch, wenn der Bauherr es sich in letzter Minute anders überlegt.

Auch der Software-Architekt bleibt heute im Projekt und hat bis zum Ende eine aktive Rolle.

- ✔ Er coacht die Teams und überwacht die Umsetzung der Architektur, besonders die Verbindung mit dem Design.
- ✔ Er nimmt Feedback der Entwickler in die Architektur auf, vor allem, wenn sich durch Design-Entscheidungen der Entwickler Änderungen an der Architektur ergeben.
- ✔ Er überarbeitet und erweitert die Architektur in agilen Projekten von Iteration zu Iteration und passt sie den neuen Zielen und Priorisierungen an.
- ✔ Er behält immer ein Gefühl für die Implementierung, um die Auswirkungen seiner Entscheidungen und möglicher Trade-offs zu verstehen.

Der letzte Punkt ist besonders wichtig, weil eine Architektur natürlich auch effektiv umsetzbar sein muss, um Laufzeit und Budget des Projekts einzuhalten. Am einfachsten ist es natürlich, wenn der Architekt Teile des Codes selbst schreibt. Wenn er allerdings zu sehr Mit-Entwickler wird, kann das problematisch sein, denn seine eigentliche Aufgabe, für die er auch den Großteil seiner Zeit aufwendet, ist ja die Architektur und die Kommunikation mit allen Stakeholdern in vielen Meetings. Um eine Balance zu finden, helfen folgende Strategien:

- ✔ Der Architekt schreibt keinen Code im kritischen Pfad des Projekts. Dieser Code gehört dem Team, damit er weiterentwickelt werden kann, auch wenn der Architekt gerade nicht greifbar ist. Besonders betrifft das zentrale Frameworks, bei denen allerdings die Versuchung besonders groß ist, weil der Architekt sie ja erdacht hat und zu Beginn am besten versteht. Da hilft nur Coaching.
- ✔ Der Architekt schreibt Proof-of-Concept-Prototypen, mit denen er Architektur- und Design-Entscheidungen validiert. Auch wenn der Code selbst nicht produktiv verwendet wird, sollte er trotzdem eine hohe Qualität haben, weil Entwickler ihn wahrscheinlich als Blaupause nutzen.
- ✔ Der Architekt schreibt Teile der Business-Logik. Sie ist das tägliche Brot der Entwickler und bestimmt damit den Aufwand und den Projekterfolg maßgeblich mit. Der Architekt sollte daher ein gutes Gefühl dafür haben, wie effektiv sie geschrieben werden kann.
- ✔ Der Architekt geht in Code-Reviews zusammen mit den Entwicklern größere Teile der Implementierung durch und hört aufmerksam auf ihr Feedback. So bekommt er wenigstens passiv einen Eindruck von der täglichen Arbeit.

Am Ende geht es darum, dass der Architekt Teil des Teams bleibt und einen starken Eindruck davon gewinnt, wie sich Architektur-Entscheidungen auf die Entwicklung tatsächlich auswirken und wie Änderungen der Architektur gefühlte Schmerzpunkte beheben können. Und natürlich ist konkreter Code auch eine sehr direkte Art des Coachings, das zu verbessertem Design und einer klareren Implementierung führt.

Wo Sie mehr lesen können

Einen sehr vollständigen Überblick über die Konzepte, Begriffe und Methoden der Software-Architektur finden Sie in [10]. Das Buch war in den 1990ern eines der Ersten zum Thema und wird seitdem fortlaufend aktualisiert. Einen Hands-on- Überblick aus Sicht eines erfahrenen Architekten gibt [89], sowohl über Konzepte als auch über aktuelle Stile und Patterns. Eine moderne Sicht auf Strategien für Architektur und Design finden Sie in [73]; weil weniger konkrete Software-Strukturen besprochen werden, braucht man ein wenig eigene Entwicklungserfahrung.